Under consideration for publication in J. Functional Programming

Fair Enumeration Combinators

Max S. New[1] Burke Fetscher[2] Jay McCarthy[3] Robert Bruce Findler[2]
[1]Northeastern University [2]Northwestern University [3]UMass Lowell

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. 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 better 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 some combinations of fair combinators are not fair.

We also report on an evaluation of fairness for the purpose of finding bugs in programming-language models. We show that fair enumeration combinators have complementary strengths to an existing, well-tuned ad hoc random generator (better on short time scales and worse on long time scales) and that 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 to support property-based testing. First was Feat for Haskell in 2012[1], then SciFe for Scala in 2014[2], and we released `data/enumerate` as part of Racket in 2015.[3]

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

[1] `https://hackage.haskell.org/package/testing-feat`
[2] `https://kaptoxic.github.io/SciFe/`
[3] `https://docs.racket-lang.org/data/Enumerations.html`

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 used our existing benchmark suite of 50 bugs 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.

In the next two sections, we discuss how testing and enumeration fit together and then introduce enumeration libraries, using the Racket-based library to make the introduction concrete. In section 4, we give an intuition-based definition of fairness and in section 5 discuss our *n*-ary combinators, whose designs are motivated by fairness. Section 6 has a formal definition of fairness and proofs showing 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 7. The next two sections discuss related work and future work; section section 10 concludes. Several places in the paper mention supplementary material; it is included available at `https://www.eecs.northwestern.edu/~robby/jfp-enum/`

2 Enumeration in Property-based Testing

Our interest in enumeration is motivated by property-based testing, as popularized by Quickcheck (Classen and Hughes 2000). Property-based testing enables programmers to simply and effectively test their software by supplying a property that a program should have and then generating a large supply of examples and testing to see if any of them falsify that property.

To see how bijective enumerations can help with property-based testing, consider this snippet of Racket (Flatt and PLT 2010) code that checks to see if a given binary tree is a binary search tree:

```
(struct node (n l r) #:transparent)
(struct leaf () #:transparent)

(define (bst? t)
  (match t
    [(leaf) #true]
    [(node n l r)
     (and (check-all l (λ (i) (<= i n)))
          (check-all r (λ (i) (>= i n)))
          (bst? l)
          (bst? r))]))

(define (check-all t p?)
  (match t
    [(leaf) #true]
    [(node n l r)
     (and (p? n)
          (check-all l p?)
          (check-all r p?))]))
```

The first line defines a node struct with three fields (`n` for the value in the node, `l` for the left subtree, and `r` for the right subtree). The second line defines a nullary struct to represent leaf nodes. The `bst?` function recursively checks the tree, ensuring that the value in each node is larger than all of the values to its left and smaller that those to its right.

While this function is correct, the algorithm it uses is inefficient, because it repeatedly processes subtrees as it recurs over the structure of the tree, running in $O(n^2)$. A better algorithm would make only a single pass over the tree. The basis for a naive and incorrect function that makes such a single pass is the false observation that, for each node, if the root of the left subtree is smaller than the value in the node and the root of the right subtree is larger, then the tree is a binary search tree.

We can easily write this (incorrect) code, too:

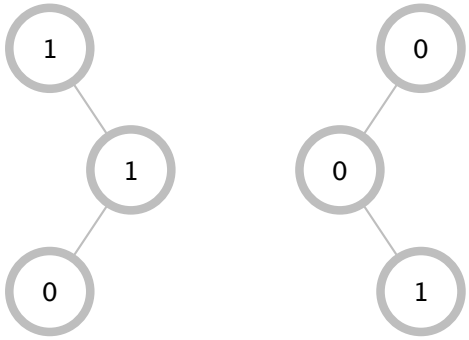
```
(define (not-quite-bst? t)
  (match t
    [(leaf) #true]
    [(node n l r)
     (and (<= (or (root-n l) -inf.0)
              n
              (or (root-n r) +inf.0))
          (not-quite-bst? l)
          (not-quite-bst? r))]))

(define (root-n t)
  (match t
    [(leaf) #false]
    [(node n l r) n]))
```

To use property-based testing to uncover the difference between these two functions, we need a source of binary trees and then we can simply compare the results of the two functions. This is where enumerations come in. They allow us to describe a mapping between the natural numbers and arbitrary data-structures. Then we can simply choose some natural numbers, map them to binary trees and see if we can find a difference between the two predicates.

Our library allows us to describe binary trees using combinators in the usual way, namely that a binary tree is either a leaf or a triple of a natural number and two more binary trees. With that description in hand, we can simply count, supplying each natural number in turn and checking to see if the corresponding tree differentiates the two predicates.

If we use our fair combinators, we find that the smallest natural that demonstrates the difference is tiny, namely 345, and it takes only about 1/100th of a second to search from 0 to the counterexample. If we swap out the fair pairing combinator for an unfair one based on the bijection discussed in the introduction, then that same tree appears at a position with 1,234 digits. The smallest index that we know has a counter example is this 78 digit number: 115,792,089,237,316,195,423,570,985,008,687,907,853,269,984, 665,640,564,039,457,584,007,913,129,639,936. That might not be the first counterexample, but we do know that there are no counterexamples in the first 10 billion naturals. These are the two trees; the one on the left is the counterexample at position 345 in the fair enumerator and the one of the right is the smallest known counterexample when using the unfair combinators.



3 Enumeration Combinators

This section introduces the basics of enumeration via a tour of our Racket-based enumeration library. Each enumeration in our library 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, and a contract that captures exactly the values in the enumeration. For the purposes of this paper, it is sufficient to think of the contracts as predicates on values; they are more general, but that generality is not needed to understand our enumeration library.

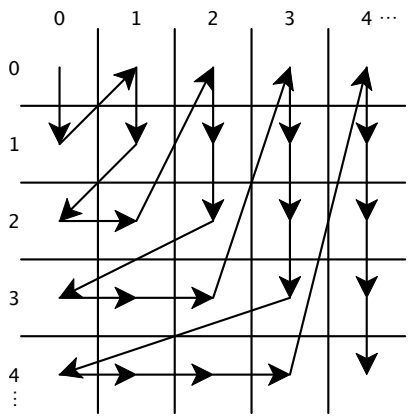


Figure 1: Pairing Order

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] Our most basic enumeration is `below/e` (by convention, the names of our enumeration library functions end with `/e`; the slash is a legal character in Racket identifiers). The `below/e` combinator accepts a natural number or `+inf.0` and returns an enumeration of that size. Its `to-nat` and `from-nat` functions are both the identity function.

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"
```

The `or/e` enumeration insists that contracts for its arguments be disjoint so that it can compute the reverse direction of the bijection. Specifically, given a value, it tests the value to see which argument enumeration it comes from, and then it finds the position in that enumeration in order to find the position in the union enumeration.

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 input enumerations is finite, the result enumeration loops through the finite enumeration, first pairing all of the elements of the finite enumeration with the first element from the infinite enumeration. Then it continues by pairing the second element of the infinite enumeration with each of the elements of the finite one, etc. If both are finite, it loops through the one with lesser cardinality. This process corresponds to taking the quotient and remainder of the index with the size of the smaller enumeration.

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

Pairing infinite enumerations requires 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. Here is the formula showing the coordinates for the zth element of the enumeration.

$$\begin{cases} \left\langle z - \lfloor\sqrt{z}\rfloor^2, \lfloor\sqrt{z}\rfloor \right\rangle & \text{if } z - \lfloor\sqrt{z}\rfloor^2 < \lfloor\sqrt{z}\rfloor \\ \left\langle \lfloor\sqrt{z}\rfloor, z - \lfloor\sqrt{z}\rfloor^2 - \lfloor\sqrt{z}\rfloor \right\rangle & \text{if } z - \lfloor\sqrt{z}\rfloor^2 \geqslant \lfloor\sqrt{z}\rfloor \end{cases}$$

The n-ary `list/e` generalizes the binary `cons/e`. We discuss this in detail in section 4.

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:

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

This code says that the `lon/e` enumeration is a disjoint union of the singleton enumeration `(fin/e null)` (an enumeration that contains only `null`, the empty list) and an enumeration of pairs, where the first component of the pair is a natural number and the second component of the pair is again `lon/e`. The `delay/e` around the latter `lon/e` is what makes the fixed point work. It simply delays the construction of the enumeration until the first time something is indexed from the enumeration. 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. Indeed, switching the order of the arguments to `or/e` above also produces an enumeration that fails to terminate. Here are the first 12 elements of the correct `lon/e`:

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

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 `map/e` we can, for example, construct enumerations of natural numbers that start at some natural `i` beyond zero. The function `naturals-above/e` accepts a natural `i` and returns such an enumeration.

```
(define (naturals-above/e i)
  (map/e (λ (x) (+ x i))
         (λ (x) (- x i))
         (below/e +inf.0)
         #:contract (and/c natural? (>=/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, in this case saying that the contract accepts natural numbers larger than or equal to `i`). 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 `from-nat` function.

One important point about the combinators presented 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)])
```

It is important to note that `cons/de` is not a function (like the earlier combinators). It is a special expression form with 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`; putting the identifiers the other way around would allow the head position to depend on the tail instead). 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 remaining case is when the dependent enumerations are all finite (our `cons/de` enumeration does not support mixed finite and infinite dependent enumerations), and it is troublesome. 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, calling `from-nat` on the result of `cons/de` can take time proportional to the input index (or possibly even worse if computing the dependent enumerations themselves are costly). We use memoization to avoid repeatedly paying this cost, but even with memoization this case for `cons/de` is observably worse in practice.

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 rest are described here: `https://docs.racket-lang.org/data/Enumerations.html`.

4 Fairness, Informally

This section introduces our definition of fairness in a precise but informal way, giving a rationale for our definitions and some examples to clarify them.

A fair enumeration combinator is one that indexes into its argument enumerations in equal proportions, 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 upper row of histograms correspond to the fair enumerators and the lower row corresponds to the unfair enumerators. Each of the four columns of histograms corresponds to a particular position in the four-tuple. The x-coordinates of each plot corresponds 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.

For example, the relative height of the leftmost bar in the two leftmost histograms says that zero appears much less frequently in the first component of the four tuples when using the unfair enumerator than using the fair one. Similarly, the relative height of the leftmost

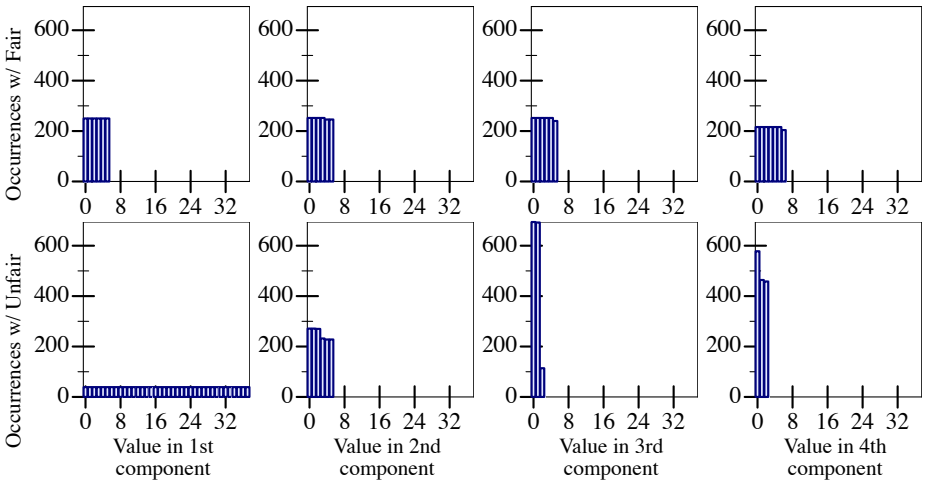


Figure 2: Histograms of the occurrences of each natural number in fair and unfair tuples.

bar in the two rightmost histograms says that zero appears much more frequently in the fourth component of the four tuples when using the unfair enumerator than it does when using the fair one.

More generally, 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.

It is tempting to think of fairness as simply a notion of size, perhaps the number of bits required to represent the result elements of the enumeration. This is not a helpful perspective, however, because it leaves open the representation choice and how to count the number of bits required. Indeed, one could say that the representation of a pair is the binary representation of the index into some (possibly unfair) enumeration, thus depriving us of a way to distinguish fair from unfair enumerations.

Also, we cannot simply 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 fair pairing would be impossible.

We can take that basic idea and weaken a little bit, however. Instead of insisting they use their arguments completely in lock-step, we insist that there are infinitely many places in the result enumeration where they have used their input enumerators equally. We call these special places *equilibrium points*. For example, consider the `(list/e (below/e +inf.0) (below/e +inf.0))` enumeration; here are the its first 9 elements:

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

At this point in the enumeration, we have seen all of the numbers in the interval $[0,2]$ in both elements of the pair and we have not seen anything outside that interval. That makes 9 an equilibrium point. The 10th element of the enumeration is `'(1 3)`, and thus 10 is not an equilibrium point because we have seen the number 3 in one component of the pair, but not in the other component. In general, `(list/e (below/e +inf.0) (below/e +inf.0))` has an equilibrium point at every perfect square. Similarly, here are the first 8 elements of `(list/e (below/e +inf.0) (below/e +inf.0) (below/e +inf.0)) 8`:

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

This is an equilibrium point because we have seen 0 and 1 in every component of the pair, but no numbers larger than that. In general the that enumeration has equilibrium points at every perfect cube.

An unfair combinator is one where there are only a finite number of equilibrium points (or, equivalently, there is a point in the result enumeration after which there are no more equilibrium points). As an example consider `triple/e`:

```
(define (triple/e e_1 e_2 e_3)
  (list/e e_1 (list/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 another equilibrium point after `0`.

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.

5 Fair Combinators

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 4, we cannot simply nest the pairing operation because the outermost pair evenly divides the input between its two argument enumerations, even if there is a nested pair on one side, but not on the other side.

Our approach to fair n dimensional tuples is to build a biased pairing operation that does not divide the input evenly, but instead divides it in the ratio $1 : n$, in expectation that left-hand side of the pair will have one sub-enumeration and the right-hand side of the pair will have n.

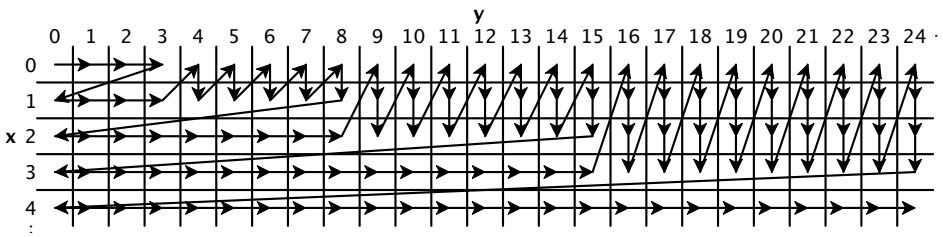


Figure 3: Ordering for Pair with Bias of 1:2

In other words, we can view Szudzik (2006)'s enumeration function as enumerating all (i, j) whose maximum is 0, then 1, then 2, etc. This is what gives it a square-like pattern. Since we want to bias the right argument by n, we can enumerate the pairs in a similar manner, but considering the n-th root of the right coordinate, not just its plain value. We call this the "biased maximum". More precisely, the biased maximum of the pair (i, j) is $\max(i+1, \lceil (j+1)^{1/n} \rceil)$ and we first enumerate all pairs where the biased maximum is 1, then 2, then 3, etc.

Figure 3 shows the first few entries of the enumeration order for pairs that has a $1:2$ bias. The diagram is reversed (the y-coordinate is horizontal and the x-coordinate is vertical) so it fits more easily on the page. The first point $(0,0)$ is the only point where biased maximum is 1; the next seven points are those where the biased maximum is always 2, etc. With a pair that has a $1:2$ bias, the biased maximum will be the same in the interval $[3^k, 3^{k+1})$, for any value of k. In general, with a $1:n$ biased pair enumerator, any pair in the interval $[(n+1)^k, (n+1)^{k+1})$ has the same biased maximum, namely $k+1$.

This is the formula for the z-th tuple in the enumeration of pairs with bias of $1:n$:

$$\begin{array}{ll|l} \left\langle r \bmod q, q^n + \left\lfloor \frac{r}{q} \right\rfloor \right\rangle & \text{if } r < s & \text{where} \quad q = \lfloor z^{1/(n+1)} \rfloor \\ \langle q, r-s \rangle & \text{if } r \geqslant s & \qquad\quad r = z - q^{n+1} \\ & & \qquad\quad s = ((q+1)^n - q^n) \cdot q \end{array}$$

To define a fair n-dimensional tupling function, we can systematically exploit the bias. Once we have a fair n-dimensional tuple enumeration, we can make a $n+1$-dimensional fair tuple enumeration by pairing the n-dimensional tuple enumeration with the new enumeration for the nth enumeration using a biased $1:n$ pairing.

The combinatorially-inclined reader may wonder why our tupling operation is based on Szudzik (2006)'s pairing function and not the classic Cantor pairing function. The two bijections are similar; they are both quadratic functions with geometric interpretations. Szudzik (2006)'s traces out the edges of squares and Cantor's traces out the bottoms of triangles. Importantly, they are both fair (but 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:

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

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.

Furthermore, Szudzik (2006)'s pairing function lends itself quite easily to a biased formulation, since enumerating rectangles is a simple modification from enumerating squares. We leave it to future work to find a biased formulation of the Cantor bijection.

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. 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.

6 Enumeration Semantics

Figure 4 shows a formal model of our enumerations. The model differs from our implementation in the way it handles unions (forcing them to be disjoint via inl and inr) and by having a type system instead of using contracts to describe the sets of values that an enumeration produces.

The relation @ defines the semantics of the enumerations. It 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; we explain it after introducing the basics of the model. 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 of the figure has the simplest rule, the one for (below/e n+); it is just the identity. Below the below/e rule is the fix/e rule. The fix/e combinator in the model is like `delay/e` from the implementation, except it provides an explicit name for the enumeration. The rule uses substitution (the definition of substitution we use is the standard one).

The next two rules, reading straight 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 in-

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

$$\frac{n < n+}{(\text{below/e } n+) \ @\ n = n \mid \varnothing}\ [\text{below/e}]$$

$$\frac{e\{x := (\text{fix/e } x\ e)\} \ @\ n = v \mid T}{(\text{fix/e } x\ e) \ @\ n = v \mid T}\ [\text{fix}]$$

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

$$\frac{\begin{array}{c}e \ @\ n = v_1 \mid T \\ v_2 = (f_1\ v_1) \qquad v_1 = (f_2\ v_2)\end{array}}{(\text{map/e } f_1\ f_2\ e) \ @\ n = v_2 \mid T}\ [\text{map}]$$

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

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

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

$$\frac{\begin{array}{c}\langle n_1, n_2 \rangle = \text{unpair}(\|e_1\|, \|e_2\|, n) \\ e_1 \ @\ n_1 = v_1 \mid T_1 \qquad e_2 \ @\ n_2 = v_2 \mid T_2\end{array}}{(\text{cons/e } e_1\ e_2) \ @\ n = (\text{cons } v_1\ v_2) \mid \lambda \text{x}.\ T_1(\text{x}) \cup T_2(\text{x})}\ [\text{cons}] \qquad \frac{\begin{array}{c}e \ @\ n_1 = v_1 \mid T_2 \\ e \ @\ n_2 = v_2 \mid T \qquad n_2 < n_1\end{array}}{(\text{except/e } e\ v_1) \ @\ n_2 = v_2 \mid T}\ [\text{ex<}]$$

$$\frac{e \ @\ n_2 = v \mid T}{(\text{trace/e } n_1\ e) \ @\ n_2 = v \mid \lambda \text{x}.\ n_1 = \text{x}\ ?\ \{n_2\} : \varnothing}\ [\text{trace}] \qquad \frac{\begin{array}{c}e \ @\ n_1 = v_1 \mid T_2 \\ e \ @\ n_2 + 1 = v_2 \mid T \qquad n_2 \ge n_1\end{array}}{(\text{except/e } e\ v_1) \ @\ n_2 = v_2 \mid T}\ [\text{ex}\ge]$$

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

Figure 4: Semantics of Enumeration Combinators

dices when it deals with infinite enumerations and uses sum_up_to for finite enumerations (defined at the bottom of figure 5).

$$\mathsf{ty}[\![e]\!] = \mathsf{tye}[\![\varnothing, e]\!]$$

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

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

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

$$\frac{n < n+}{n : n+} \qquad \frac{v_1 : \tau_1 \qquad v_2 : \tau_2}{(\mathsf{cons}\ v_1\ v_2) : \tau_1 \wedge \tau_2} \qquad \frac{v : \tau\{x := \mu x.\ \tau\}}{v : \mu x.\ \tau}$$

$$\frac{v : \tau_2}{(\mathsf{inr}\ v) : \tau_1 \vee \tau_2} \qquad \frac{v : \tau_1}{(\mathsf{inl}\ v) : \tau_1 \vee \tau_2} \qquad \frac{v_1 : \tau_1 \qquad v_1 \neq v_2}{v_1 : (\text{- } \tau_1\ v_2)}$$

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

Figure 5: Semantics of Enumeration Combinators, Continued

The first rule underneath the boxed grammar is the map/e rule, showing how its bijection is used. The next four rules govern the or/e combinator. These rules work by alternating between the two enumerations until one runs out (in the case that it is finite), and then they just use the other enumeration. The upper two or/e rules cover the case where neither has yet run out. The lower two cover the situation where of the arguments was finite and the enumeration has already produced all of those elements. 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.

The cons/e rule uses the unpair function, shown in figure 5. The unpair function accepts the sizes of the two enumerations, computed by the size function in the middle of figure 5 (written using double vertical bars), and the index. The function maps indices as discussed in section 3.

To the right of the cons/e rule are the rules for the except/e combinator, which behaves as discussed in section 3, one rule for the situation where the value is below the excepted value and one for where it is above.

We return to the rule for trace/e shortly, and the last rule is an unfair pairing operation using the bijection from the introduction.

The Coq model is simpler than the model presented here and the model presented here is simpler than our implementation. The primary difference between the three 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 implementation does not have a type system; the role of types is played by the contract system instead. Contracts give us additional flexibility that ordinary type systems do not have, allowing us to maintain the invariant that the contract describes the precise set of values that can be enumerated, even for enumerations of only positive numbers, or non-empty lists, etc. Having these precise contracts has proven helpful in practice as we debug programs that use the enumeration library.

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. There are no other differences between the Coq model and the model in the paper. In general, the Coq model is designed to be just enough for us to state and prove some results about fairness whereas the model presented in the paper is designed to provide a precise explanation of our enumerations.

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.

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 before, 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 formula inverses. □

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 (i.e., a runtime error).

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:nat | 0 ⩽ x ⩽ 15}, meaning that the two arguments were explored the same amount, at least for the first 256 elements. The complete trace of

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

up to 256, however, maps 0 to {x:nat | 0 ⩽ x ⩽ 127} and 1 to {x:nat | 0 ⩽ x ⩽ 8}, where unfair-cons/e is the unfair pairing combinator from the introduction. This shows that the first argument (traced with the 0) is explored more than the second.

We say that an enumeration combinator c^k : *enum*... → *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 $enum_1$) ⋯ (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. In other words, a fair combinator is one where the traces of its arguments are explored the same amount at an infinite number of points, namely the values of M. As such, we call the values of M the equilibrium points.

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. □

Concretely, this means that the equilibrium points of or/e are 2, 4, 6, 8, etc. Tracing `or/e` up to those points produces the sets {0}, {0, 1}, {0, 1, 2}, and {0, 1, 2, 3}, etc.

Theorem 4
Using nested `or/e` to construct a three-way enumeration 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 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 4 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 `Unfair-PairUnfair` in the Coq model. □

7 Empirical Evaluation

As our motivation for studying enumerations is test case generation, we performed an empirical evaluation of fair and unfair enumerations described earlier in the paper 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 our evaluation and its results.

7.1 Setup

We conducted the evaluation in the context of Redex (Felleisen et al. 2009; Matthews et al. 2004), a domain-specific 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 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 3 to build two generators based on enumeration,

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

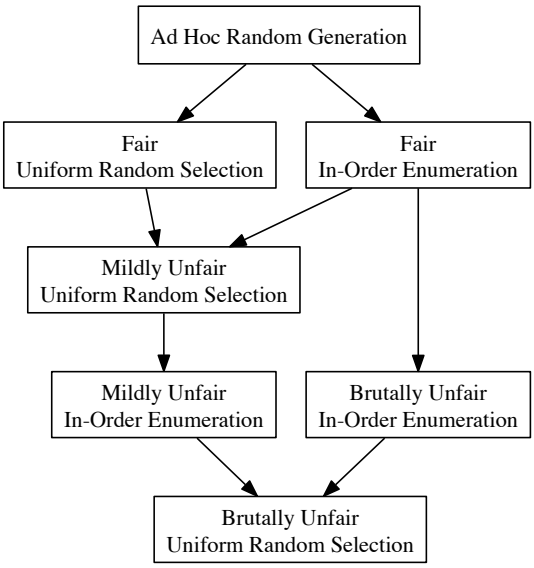


Figure 6: Partial Order Between Generators Indicating Which Find More Bugs

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 generates expressions matching a particular non-terminal by randomly choosing a production, expanding the non-terminal based on the production chosen, and then repeating the process until a depth bound is reached. At that point, it limits the random choice to productions that do not require recursive unfoldings (such productions are guaranteed to exist or else the original Redex program would have been syntactically ill-formed).

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.

For our evaluation, we use the default value of 5 for this depth since that is what Redex users see without customization. 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 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.

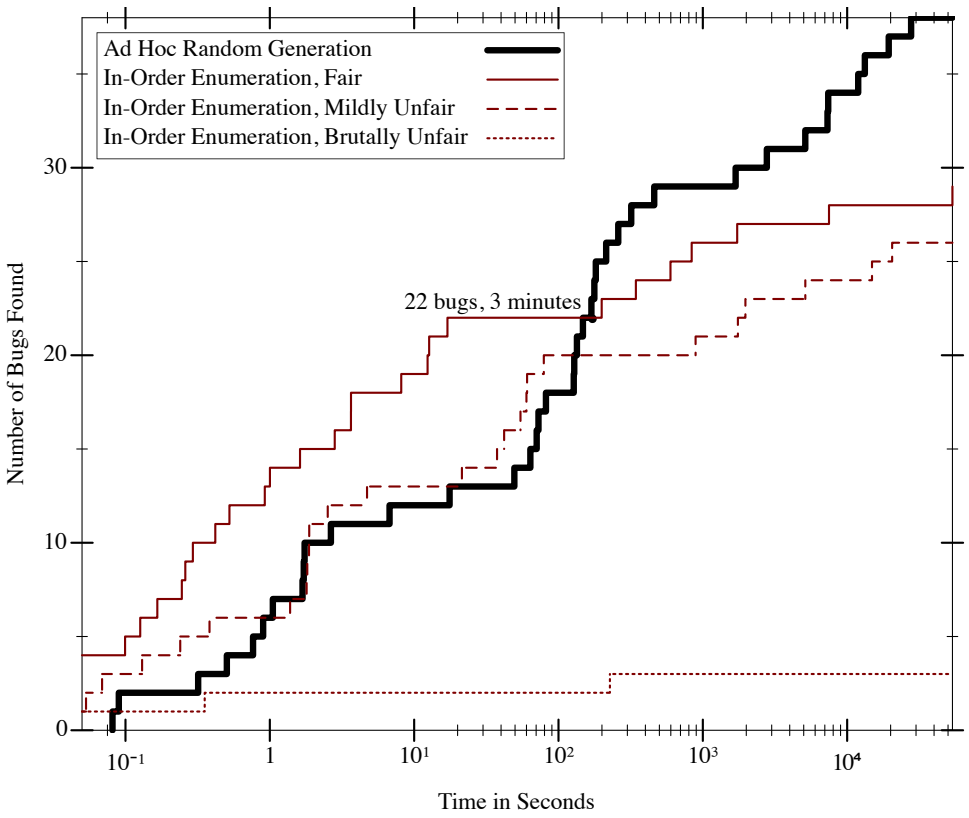


Figure 7: Overview of random testing performance of ad hoc generation and in-order enumeration

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 others.

We used three variations on the enumeration combinators. The first is the fair combinators described in section 4. 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

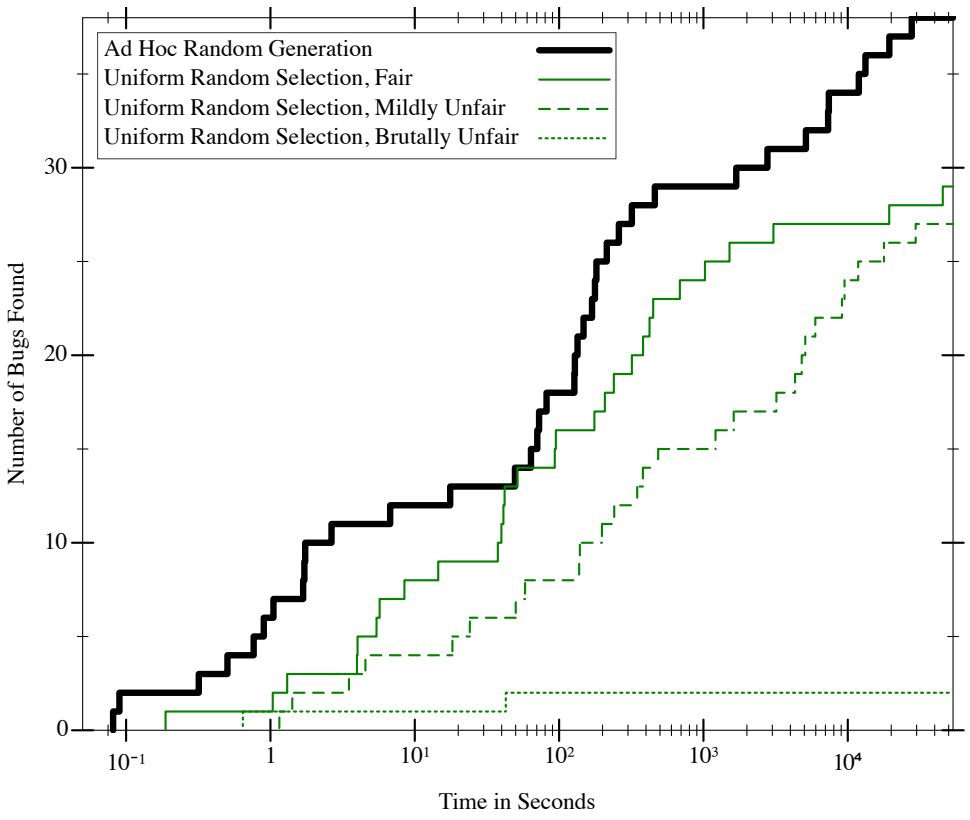


Figure 8: Overview of random testing performance of ad hoc generation and random indexing into an enumeration

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.

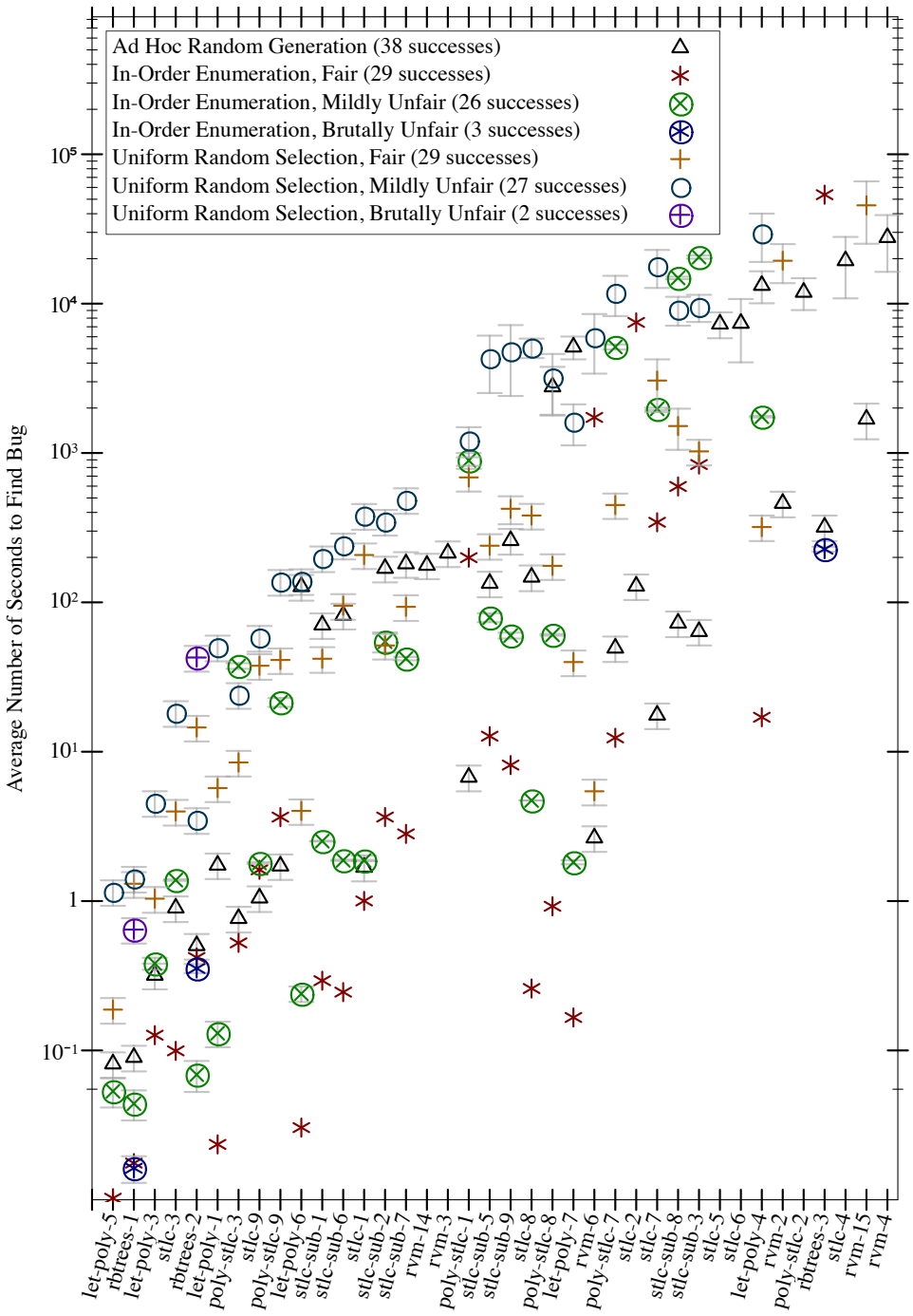
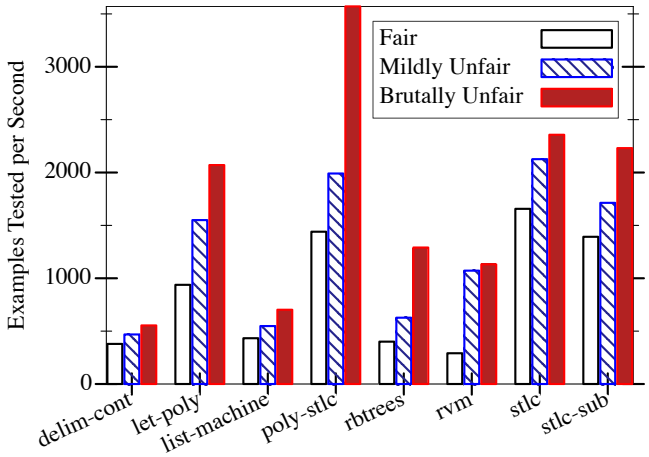
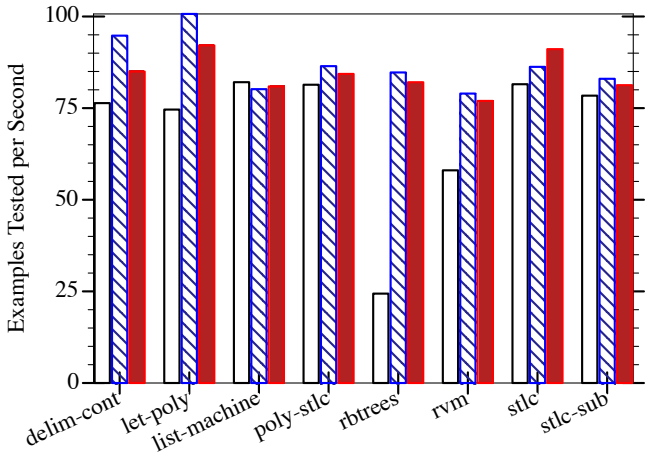


Figure 9: Time taken to find each bug for each generator



In-Order Enumeration



Uniform Random Selection

Figure 10: Examples tested per second for each benchmark model and enumeration-based generator

7.2 Results

The graph in figure 6 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; it does not take into account how long it took for each generator to find the bugs that it found. The plot in figure 7 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 a few hours. 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 shows the number of counterexamples found by the ad hoc 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, all when running in order. This plot shows that the mildly unfair combinators are worse than the fair ones and the brutally unfair combinators are much worse than either.

That plot also reveals that the ad hoc generator is only better than the best enumeration strategy after 3 minutes. Before that time, the fair in-order enumeration strategy is the best approach.

Figure 8 has a similar plot that uses the same set of combinators, but randomly picks natural numbers (as described above) and uses those to generate candidates. This plots shows that that approach is never the best approach, on any time scale.

No strategy was able to find more than 38 of the 50 bugs in the benchmark.

Figure 9 shows a plot of every generator's performance on each bug. The x axis has one entry for each different bug (for which a counter-example was found) and the y axis shows the average number of seconds required to find that bug. The chart confirms the conclusion from figure 7 showing that the unfair combinators are never significantly below their fair counterparts and often significantly above.

Our data 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 6 bugs and the fair in-order enumerator was significantly better on (a different) 6 bugs. The unfair enumerators were never significantly better on any bug.

We believe the reason that the fair enumerators are better than the unfair ones is that their more balanced exploration of the space leads to a wider variety of interesting examples being tested. Figure 10 provides some evidence for this theory. It shows the number of examples tested per second for each model (the Redex bug benchmark 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 upper plot shows the in-order generators and the lower 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 have other uninteresting variations.

8 Related Work

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

8.1 Bijective Enumeration Methods

The SciFe library for Scala (Kuraj and Kuncak 2014; Kuraj et al. 2015) is most similar to our library, 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 7. 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 4. These differences and the lack of fairness aside, the technical details of the implementation are very similar and our library shares all of the strengths and weakness of their library.

Tarau (2013)'s work on bijective encoding schemes for Prolog terms is also similar to ours. 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 combinator is fair in the sense that its output reaches equilibrium infinitely often. Indeed, it reaches equilibrium at the end of each of the parts in the result. The code given in the paper for the pairing and alternation combinators are f-fair with equilibrium points that have the same asympototic complexity as our binary combinators. In the implementation, however, they use a binary representation, not a unary representation of naturals, which makes the distance between consecutive equilibrium points double at each step, making the equilibrium points exponentially far apart.

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 depen-

dent 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.

8.2 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. One is 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 detailed 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 found by enumeration that were not found randomly. 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.

9 Future Work

There are two ways in which we we believe more work with our definitions and use of fairness would be productive.

9.1 Enumerations of Recursive Structures

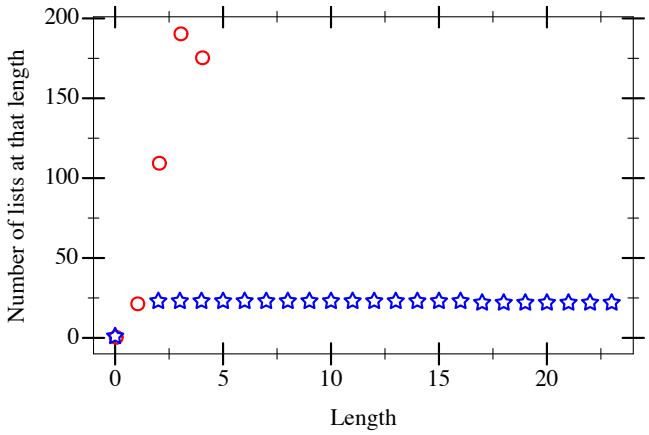
Our definition of fairness requires (at least) binary combinators. That is, a definition of enumerations of, for example, lists is not a candidate for our fairness definition because it accepts only one argument (the enumeration of the elements). There is, however, more than one way to define an enumeration of lists and some ways seem to more fair than others. More precisely, the `lon/e` enumeration from section 3 tends to bias towards shorter

lists that have bigger numbers at the front of the list in an unfair way. For example, the 10,000,000,000th element is `'(99999 142 17 2 0 0)`, which seems to suggest that `lon/e` is exploring the first element at the expense of exploring longer lists.

Instead, we can build 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, i.e.:

```
(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, at least intuitively, 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 the difference between the two approaches to defining data-structure enumerators.

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).

9.2 Intentional Unfairness

The second way in which our notion of fairness is incomplete has to do with real-world testing. Consider, for example, this definition of a grammar for the lambda calculus (written in Redex's notation):

```
(define-language L
  (e ::=
     x
     (e e)
     (λ (x) e))
  (x ::= variable))
```

Our implementation translates the `e` non-terminal into uses of the `or/e` enumeration combinator for the productions and the `list/e` combinator for each production to combine the pieces, as you might expect. Looking at a prefix of the enumeration, however, clearly suggests that it is not-optimal for most bug-finding tasks. In particular, every third expression generated is simply just a free variable!

We think that, while fairness is a good guide for combinators, it is important to be able to selectively bias the enumerations away from fairness (much like the bias used to obtain fairness, as discussed in section 5). We have not explored how to specify these biases in Redex nor their impact on testing, but believe that a domain-specific language for tuning the enumerations is worthy of more study.

10 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 fair in-order enumeration for 10 seconds, then alternates between fair 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.

Acknowledgments. Thanks to Neil Toronto for helping us find a way to select from the natural numbers at random. Thanks to Ben Lerner for proving a square root property that gave us fits. Thanks to Hai Zhou, Li Li, Yuankai Chen, and Peng Kang for graciously sharing their compute servers with us. Thanks to William H. Temps, Matthias Felleisen, and Ben Greenman for helpful comments on the writing. Thanks to one of the anonymous reviewers at ICFP 2015 for suggesting that we refine our definition of fairness with a function.

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.

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.

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.

Matthew Flatt and PLT. Reference: Racket. PLT Design Inc., PLT-TR-2010-1, 2010. `https://racket-lang.org/tr1/`

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.

Ivan Kuraj, Viktor Kuncak, and Daniel Jackson. Programming with Enumerable Sets of Structures. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

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.