Ffeat: Fair Functional Enumeration of Algebraic Types

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

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

Jay McCarthy
Vassar College
jay.mccarthy@gmail.com

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

Abstract

This paper reports on the design of combinators for building efficient bijections between the naturals numbers (or a prefix of them) and algebraic datatypes constructed by sums, recursion, and (possibly dependent) pairs.

Our enumeration combinators support a new property we call fairness. Intuitively the result of fair combinator indexes into its argument combinators 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 you build a three-tuple out of nested pairs. The paper develops the theory of fairness and contains proofs establishing fairness of our combinators and a proof that some combinations of fair combinators are not fair.

The design of our combinators is driven by our primary application, property-based testing in Redex. The paper also reports on how our combinators can support enumeration for arbitrary Redex models and an empirical comparison between enumeration-based property generation and ad hoc random generation, showing that the strategies are complementary.

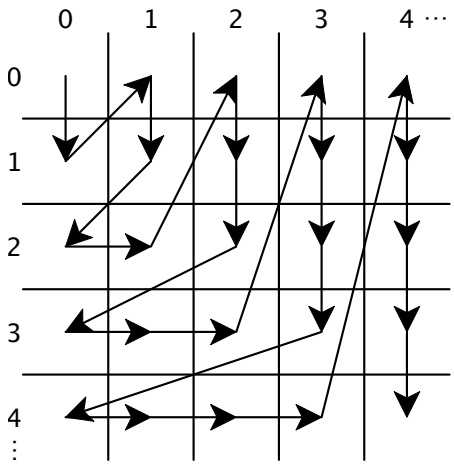


Figure 1: Pairing Order

1. Introduction to Enumeration

This section gives a tour of our enumeration library and introduces the basics of enumeration. Section 2 explains the concept of fairness and section 3 gives our formal model and an overview of the proofs. The rest of the paper discusses our evaluation, related work, and concludes.

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 all of the values in the enumeration precisely. 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 are being enumerated.[1]

The most basic enumeration is `natural/e`. Its `to-nat` and `from-nat` functions are simply the identity function and its size is `+inf.0`. The combinator `fin/e` builds a finite enumeration from its arguments, so `(fin/e 1 2 3 "a" "b" "c")` is an enumeration with the six given elements, where the elements are put in correspondence with the naturals in order they are given.

The 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"
```


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

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

The contracts associated with the enumerations are used to determine which enumeration a value came from when going from a value back to a natural bnumber.

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

Pairing infinite enumerations require more care. If we imagine our sets as being laid out in an infinite two dimensional table, `cons/e` walks along the edge of ever-widening squares to enumerate all pairs (using Szudzik (2006)'s bijection), as shown in figure 1. The first 12 elements of `(cons/e natural/e natural/e)` 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 2.

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 natural/e
                        (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 encoding or decoding happens, the value of the expression is computed (and cached). This means that a use of `fix/e` that is too eager, e.g.: `(letrec ([e (fix/e e)]) e)` will cause `from-nat` to fail to terminate. Indeed, switching the order of the arguments to `or/e` in the above example also produces an enumeration that fails to terminate when decoding or encoding happens.

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 any two sets with the bijection in an enumeration, so 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))
         natural/e
         #: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 (the `#: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 that doesn't have the given element. For example, the first 8 elements of `(except/e natural/e 4)` are

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

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

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

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 natural/e]
         [tl (hd) (naturals-above/e hd)])
```

A `cons/de` has two subexpressions (`natural/e` and `(naturals-above/e i)` in this example), each of which is named (`hd` and `tl` in this example). 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 `dep/e` 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, requiring only linear time in the number of bits of the representation of the number for decoding.

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.

2. Fairness

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 natural/e (cons/e natural/e
 (cons/e natural/e (cons/e natural/e
  (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 `natural/e` than the others. In contrast, if we balance the `cons/e` expressions like this:

```
(cons/e
 (cons/e natural/e natural/e)
 (cons/e natural/e natural/e))
```

(and then use `map/e` to adjust the elements of the enumeration to be lists), then the 1,000,000,000 element 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 plot correspond to the different values that appear in the tuples and the height of each bar is the number of times that particular number appeared when enumerating the first 1,500 tuples. As you can see, all four components have 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.

A subtle point about fairness is that we cannot restrict the combinators to work completely in lock-step on their argument enumerations, or else we would not admit *any* pairing operation as fair. After all, a combinator that builds the pair of `natural/e` with itself we 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 that 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.

As an example, consider the fair nested `cons/e` from the beginning of the section. As we saw, at the point 1,000,000,000, it was not at equilibrium. But at 999,999,999,999, it produces `'(999 999 999 999)`, and indeed it has indexed into each of the four `natural/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 natural/e natural/e)` 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)
```

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.

Okay, so now 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 1, the fair pair 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.

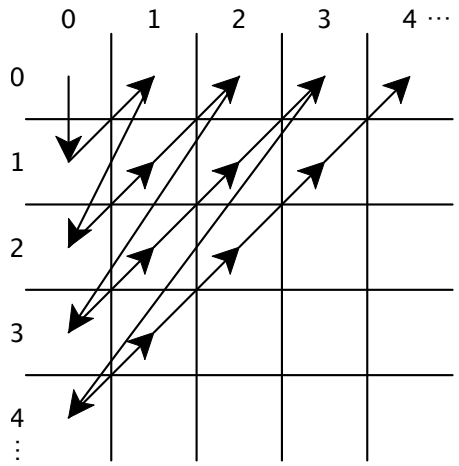
So, we need to generalize the pairing function to compute which face of which hypercube we are on at each point in the enumeration. Returning to two dimensions, say we want the 44th element of the enumeration, we first compute the integer square root, 6, and then compute the remainder (i.e. $44\text{-}6^2$), 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)`.

To perform the inverse, we take `'(6 . 2)` and identify that its max is 6. Then we pass `'(6 . 2)` into the opposite direction of the enumeration of pairs with max of 6, giving us 8. 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 with 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 1. 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:



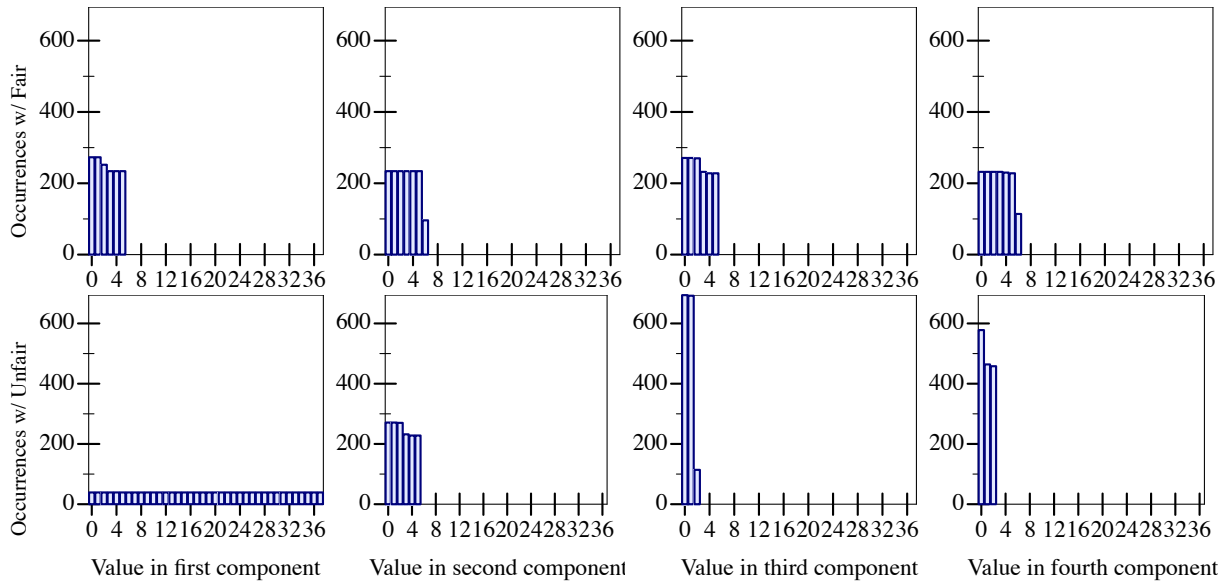


Figure 2: Histograms of the occurrences of each natural number in fair and unfair tuples

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 a 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 uses 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.

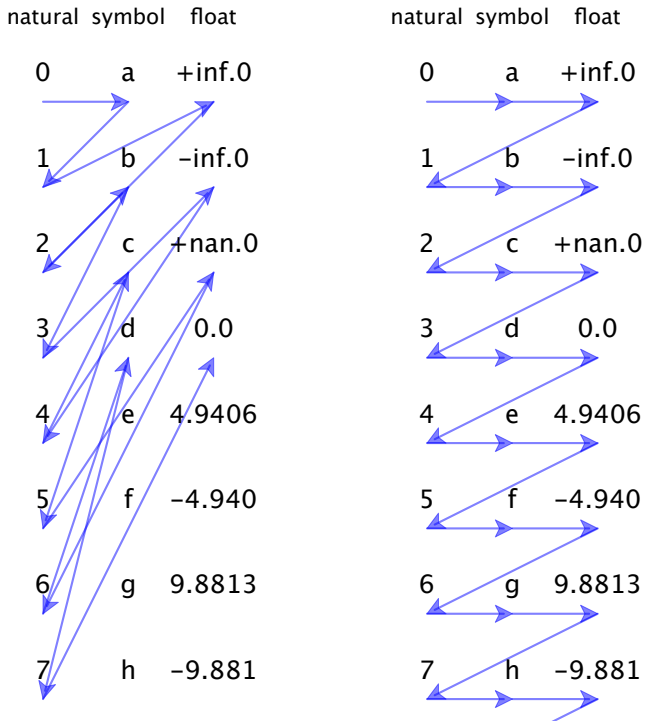


Figure 3: Unfair (left) and fair (right) disjoint union enumerations

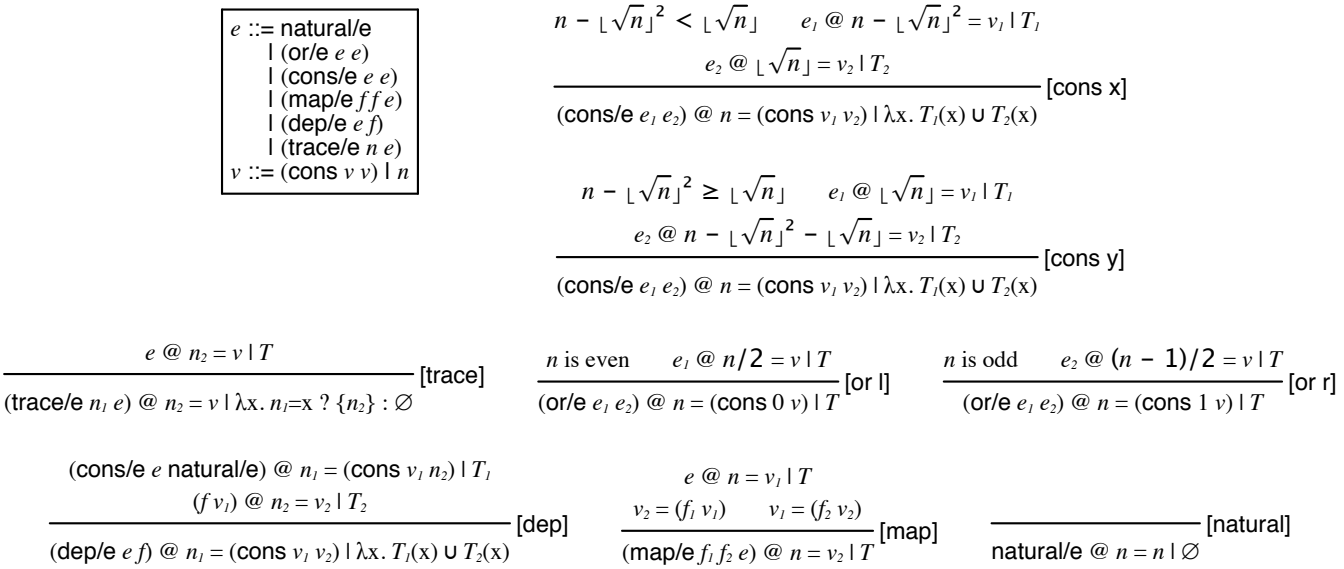


Figure 4: Semantics of Enumeration Combinators

3. Enumeration Semantics

Figure 4 shows a formal model of a subset of our enumerations. It defines of 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 build a Coq model of this semantics. All of the theorems stated in this section are proven with respect to the Coq model and the Redex model, Coq model, and our implementation are all tested against each other.

The simplest rule is for natural/e, in the bottom right; it is just the identity. The two rules just above it show how or/e works; if the number is even we use the left enumeration and if it is odd, we use the right one. The two cons/e rules at the top of the figure are the most complex. They enumerate in the order discussed in section 1, walking in ever larger squares starting at the origin. The "x" rule walks horizontally and the "y" rule walks vertically. The condition in the first premise controls which rule applies. The map/e rule shows how the bijection is used. The dep/e rule exploits cons/e to get two indicies. We return to the rule for trace/e shortly.

The model simplifies our implementation in three ways. First, it covers only some of the combinators and only infinite enumerations. Second, the enumerations do not have contracts. Third, or/e in our implementation allows user-specified predicates instead of forcing disjointness by construction like or/e in the model. Nevertheless, it is enough for us to state and and prove some results about fairness.

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

THEOREM 1. *For all e, n, there exists a unique v and T such that e @ n = v | T, 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, v, there exists a unique T and n such that e @ n = v | T, or there are no n or T such that e @ n = v | T, and we can compute either the existential witness of n or its absence.*

Proof As with the previous theorem, we recursively process the rules to compute n, but 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.

Although we don't prove it formally, the situation when there is no n 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 specific 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 numbers traced of an enumeration for all naturals up to some given n. So, for some enumeration expression e, the complete trace up to

n is the union of all of the T components for e @ i = v ! T, for all values v and i strictly less than n.

We say that an enumeration combinator c^k : *enum*... $\rightarrow$ *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 of c^k applied to (trace/e 1 $enum_1$) $\cdots$ (trace/e k $enum_k$), for any enumerations $enum_l$ 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. The Coq model contains this definition only for $k \in \{2,3,4\}$, called `Fair2`, `Fair3`, and `Fair4`.

THEOREM 3. `or/e` *is fair.*

Proof The equilibrium points are at $2 * n$ for each n and this can be proved by induction on n. The full proof is `SumFair` in the Coq model.

THEOREM 4. `or-three/e` *from section 2 is unfair.*

Proof We show that after a certain point, there are no equilibria. For $n \geq 8$, there exist natural numbers m, p such that $2m \leq n < 4p$ 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 fair*

Proof Our equilibria points are n^2 for every natural number n. First, it can be shown 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 $\bigcup_{m=0}^{n}\{0,\ldots,n-1\}$ and the result then holds by induction on n. The full proof is `PairFair` in the Coq model.

THEOREM 6. `triple/e` *from section 2 is unfair*

Proof For any natural $n \geq 16$, there exist natural numbers m, p such that $m^2 \leq n < p^4$ 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.

4. Enumerating Redex Patterns

The inspiration for our enumeration library is bringing the success of Lazy Small Check (Runciman et al. 2008) and FEAT (Duregård et al. 2012), to property-based testing for Redex programs. This section gives an informal overview of how Redex programs are compiled into calls to the enumeration library in preparation for the evaluation in section 5.

Redex programmers write down a high-level specification of a grammar, reduction rules, type system, etc., and properties that should hold for programs in these languages that relate, say, the reduction semantics to the type system. Redex can then generate example programs and try to falsify the properties.

To give a flavor for the new capability in Redex, consider the language in figure 5, which contains a Redex program that defines the grammar of a simply-typed calculus, plus numeric constants. With only this much written down, a Redex programmer can ask for the first nine terms:

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

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

Figure 5: Simply-Typed λ-Calculus

or the 100,000,000th term:

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

which takes only 10 or 20 milliseconds to compute.

Our extensions to Redex map each different pattern that can appear in a grammar definition into an enumeration. At a high-level, the correspondence between Redex patterns and our combinators is clear, namely recursive non-terminals map into uses of `dep/e`, alternatives map into `or/e` and sequences map into `list/e`. We also take care to exploit our library's fairness. In particular, when enumerating the pattern, `(λ (x : τ) e)`, we do not generate list and pair patterns following the precise structure, which would lead to an unfair nesting. Instead, we generate the pattern `(list/e x/e τ/e e/e)`, where `x/e`, `τ/e` and `e/e` correspond to the enumerations for those non-terminals, and use `map/e` to construct the actual term.

Redex's pattern language is more general, however, and there are four issues in Redex patterns that require special care when enumerating.

Patterns with repeated names If the same meta-variable is used twice when defining a metafunction, reduction relation, or judgment form in Redex, then the same term must appear in both places. For example, a substitution function will have a case with a pattern like this:

```
(subst (λ (x : τ) e_1) x e_2)
```

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

To handle such patterns the enumeration makes a pass over the entire term and collects all of the variables. It then enumerates a pair where the first component is an environment mapping the found variables to terms and the second component is the rest of the term where the variables are replaced with constant enumerations that serve as placeholders. Once a term has been enumerated, Redex makes a pass over the term, replacing the placeholders with the appropriate entry in the environment. This strategy also ensures that we generate a fair enumeration of each pattern, rather than introducing unwanted nesting.

Patterns with inequalities Second, in addition to patterns that insists on duplicate terms, Redex also has a facility for insisting

that two terms are different from each other. For example, if we write a subscript with `_!_` in it, like this:

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

then the two `x`s must be different from each other.

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

Generating a list without duplicates requires the `dep/e` combinator and the `except/e` combinator. For example, to generate lists of distinct naturals, we first define a helper function that takes as an argument a list of numbers to exclude

```
(define (no-dups-without eles)
  (or/e (fin/e null)
        (dep/e
         (except/e* natural/e eles)
         (λ (new)
           (delay/e
            (no-dups-without
             (cons new eles)))))))
```

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

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

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

List patterns with length constraints The third complex aspect of Redex patterns is Redex's variation on Kleene star that requires that two distinct sub-sequences in a term have the same length.

To explain these kinds of patterns, first consider the Redex pattern

```
((λ (x ...) e) v ...)
```

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

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

```
((λ (x ..._1) e) v ..._1)
```

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

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

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

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

Ambiguous patterns And finally, there is one relatively uncommon use of Redex's patterns that we cannot enumerate. It is a bit technical and explaining it requires first explaining ambiguity in matching Redex patterns. There are several different ways that a Redex grammar definition can be ambiguous. The simplest one is when a single non-terminal has overlapping productions, but it can occur due to multiple uses of ellipses in a single sequence or when matching `in-hole`. Because of the way our enumeration compilation works, we are not technically building a bijection between the naturals and terms in a Redex pattern; it is more accurate to say we are building a bijection between the naturals and the ways one might parse a Redex pattern. In our implementation, of course, we construct a concrete term from the parse, but when a pattern is ambiguous there may be a single term that corresponds to multiple parses and thus our enumeration is not bijective. Usually, this is not a problem, as we ultimately need only the mapping from naturals to Redex patterns, not the inverse. There is one situation, however, where we need the inverse, namely to handle patterns with inequalities, as discussed above. Determining if a pattern is ambiguous in the general case is a computationally difficult task, so we approximate it in a way that works well for Redex models we encounter in practice (since few Redex languages are intentionally ambiguous), but if we cannot determine that a pattern is unambiguous and it is combined with a `_!_` pattern, Redex will signal an error instead of enumerating the pattern.

4.1 Fairness and Redex

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

Of course, the flip-side of this coin is that using unfair combinators can improve the quality of the search in some cases, even over fair enumeration. For example, when we are enumerating expressions that consist of a choice between variables and other kinds of expressions, we do not want to spend lots of time trying out different variables because most models are sensitive only to when variables differ from each other, not their exact spelling. Accordingly unfairly biasing our enumerations away from different variables can be a win for finding bugs. Overall, however, it is important that we do have a fair set of combinators that correspond to the way that Redex programs are constructed and then when Redex programs are compiled into the combinators, the compilation can use domain knowledge about Redex patterns to selectively choose

targeted unfairness, but still use fair combinators when it has no special knowledge.

5. Empirical Evaluation

We compared three types of test-case generation using a set of buggy models. Each model and bug is equipped with a property that should hold for every term (but does not, due to the bug) and three functions that generate terms, each with a different strategy: in-order enumeration, random selection from an enumeration, and ad hoc random generation. The full benchmark consists of a number of models, including the Racket virtual machine model (Klein et al. 2013), a polymorphic λ-calculus used for random testing in other work (Pałka 2012; Pałka et al. 2011), the list machine benchmark (Appel et al. 2012), and a delimited continuation contract model (Takikawa et al. 2013), as well as a few models we built ourselves based on our experience with random generation and to cover typical Redex models.[2]

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

We used two identical 64 core AMD machines with Opteron 6274s running at 2,200 MHz with a 2 MB L2 cache to run the benchmarks. Each machine has 64 gigabytes of memory. Our script runs each model/bug combination sequentially, although we ran multiple different combinations at once in parallel. We used the unreleased version 6.1.1.8 of Racket (of which Redex is a part); more precisely a version of Racket with all libraries at their latest versions on February 23, 2015 .

For the in-order enumeration, we simply indexed into the decode functions (as described in section 1), starting at zero and incrementing by one each time.

For the random selection from an enumeration, we need a mechanism to pick a natural number. To do this, we first pick an exponent i in base 2 from the geometric distribution and then pick uniformly at random an integer that is between 2^{i-1} and 2^i. We repeat this process three times and then take the largest – this helps make sure that the numbers are not always small.

We choose this distribution because it does not have a fixed mean. That is, if you take the mean of some number of samples and then add more samples and take the mean again, the mean of the new numbers is larger than from the mean of the old. We believe this is a good property to have when indexing into our enumerations so as to avoid biasing our indices towards a small size.

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

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

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

Figure 6 shows a high-level view of our results. Along its x-axis is time in seconds in a log scale, and along the y-axis is the total number of counterexamples found for each point in time. There are three lines on the plot showing how the total number of counterexamples found changes as time passes.

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

There are three crossover points marked on the graph with black dots. After 3 minutes, with 23 of the bugs found, the in-order enumeration starts to lose and the ad hoc generator starts to win and it gives up the lead only briefly, between the 33 and 39 minute marks (with one bug).

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

Although there are 50 bugs in the benchmark, no strategy was able to find more than 37 of them in a 24 hour period.

Figure 7 also shows that, for the most part, bugs that were easy (could be found in less than a few seconds) for either the ad hoc generator or the generator that selected at random from the enumerations were easy for all three generators. The in-order enumeration, however, was able to find several bugs (such as bug #8 in poly-stlc and #7 in let-poly) in much shorter times than the other approaches.

6. Related Work

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

6.1 Enumeration Methods

Tarau (2011)'s work on bijective encoding schemes for Prolog terms is most similar to ours. However, we differ in three main ways. First, our n-ary 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. We also use the "function" perspective. While our approach is closer to Tarau's, we share support for finite sets with Feat, but are distinct from Feat in our support for dependent pairing and fairness. Also, Feat has only one half of the bijection and thus cannot support `except/e` (and thus cannot easily support identifiers that contain all strings, except without a small set of keywords like we need for Redex).

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

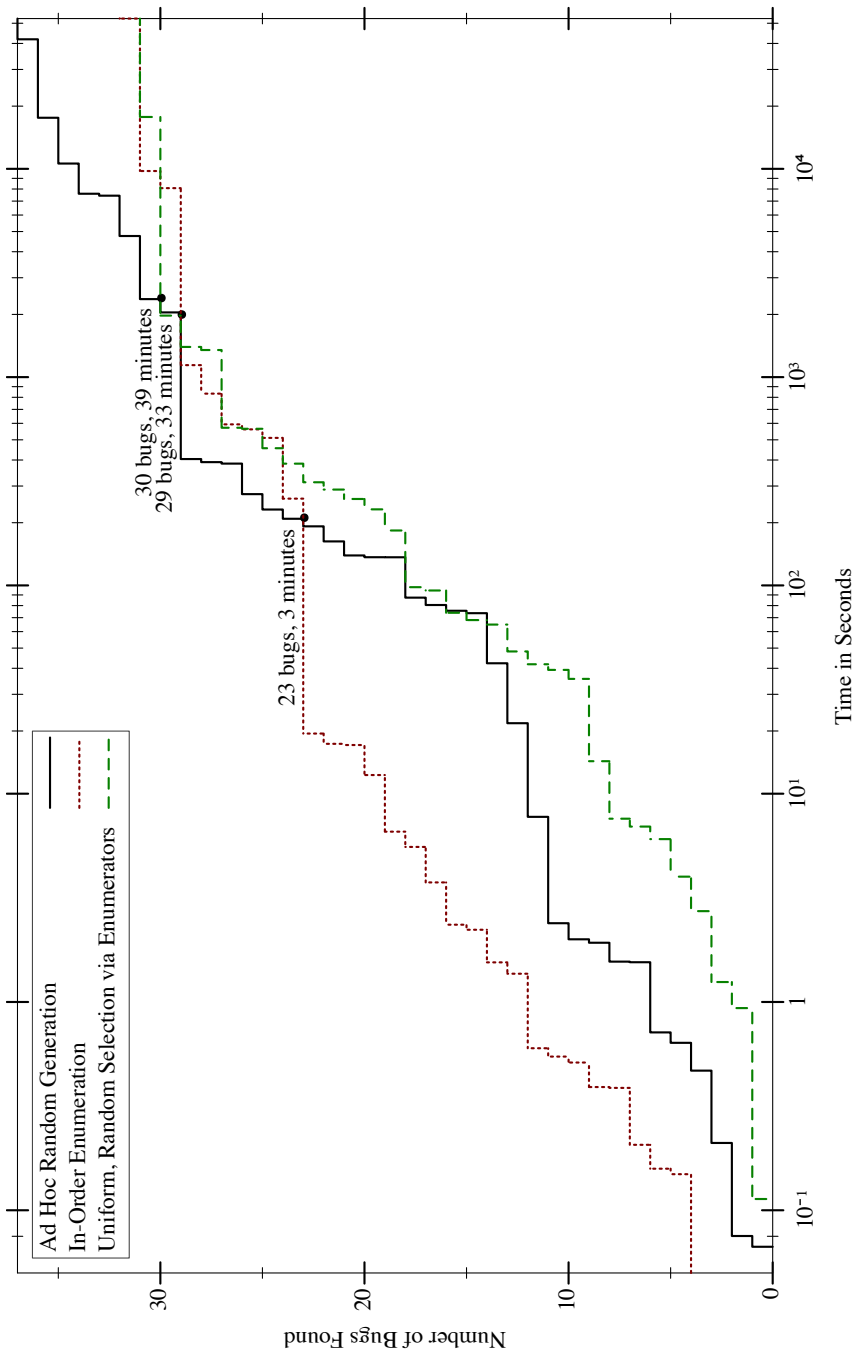


Figure 6: Overview of random testing performance of ad hoc generation, in-order enumeration, and random indexing into an enumeration, on a benchmark of Redex models.

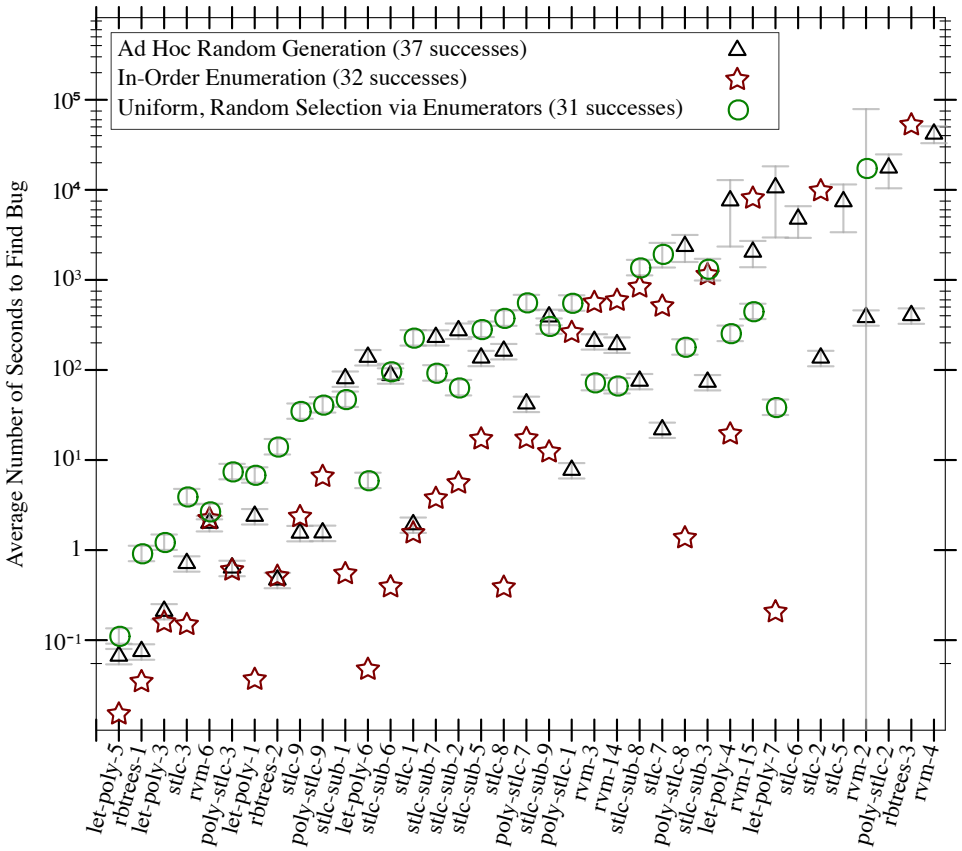


Figure 7: The mean time each generator takes to find the bugs, for each bug that some generator found; bars indicate 90% confidence intervals

It is tempting to think of Feat as supporting fairness because of the way it partitions the values to be enumerated into finite subsets, in such a way that the element of each finite subset has a fixed number of constructors. Unfortunately, this is not the same as fairness because Feat inserts "dummy" constructors in a way that makes all pairing operations be binary pairs. Put another way, it is not possible to fairly enumerate a three-tuple of natural numbers using Feat. Feat can enumerate such tuples, but it effectively gives you an enumeration of nested pairs which is, as discussed in section 2 and section 3, unfair.

Kuraj and Kuncak (2014) is similar to Feat, but with the addition of a dependent combinator. Like Feat, it does not support `except/e` or fairness.

Kennedy and Vytiniotis (2010) take a different approach to something like enumeration, viewing the bits of an encoding as a sequence of messages responding to an interactive question-and-answer game. This method also allows them to define an 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.

Kiselyov et al. (2005) explicitly discuss fairness in the context of logic programming, but talk about it in the specific cases of fair disjunction and fair conjunction, but they do not have a unification of these two different types of fairness, nor do they have a concept of n-ary fair operators for $n > 2$.

6.2 Testing Studies

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

This paper presents a new concept for enumeration, 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.

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.

Acknowledgements. 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 Matthias Felleisen and Ben Greenman for helpful comments on the writing.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8. Appendix

This section shows the precise code we used to get the numbers discussed in section 6. This is the Quickcheck code for the first conjecture:

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

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

The corresponding Redex code:

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

The Quickcheck code for the second conjecture:

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

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

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

The corresponding Redex code:

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