Parsing Randomness

Reinterpreting and Differentiating Random Generators

Harrison Goldstein
University of Pennsylvania
Philadelphia, PA, USA
hgo@seas.upenn.edu

Benjamin C. Pierce
University of Pennsylvania
Philadelphia, PA, USA
bcpierce@cis.upenn.edu

Abstract

"A generator is a parser of randomness." This perspective on random generation is well established as folklore, but to our knowledge has not been formalized, nor have its consequences been deeply explored.

We present *free generators*, which unify parsing and generation in a common structure that makes the relationship between the two concepts concrete. Free generators naturally lead us to a proof that many generators can be factored into a parser and a distribution over choice sequences.

We also put this new abstraction to work by observing that free generators have a notion of *derivative* that "previews" the effect of a particular generator choice. This derivative is compatible with related ideas in formal languages, and we use it in a novel algorithm for generating data satisfying preconditions.

Keywords: Random generation, Parsing, Property-based testing, Formal languages

1 Introduction

"A generator is a parser of randomness." It's one of those observations that's confusing until it's obvious.

A random generator processes a series of random choices into a data structure, just as a parser processes a series of characters into a data structure. But few tend to think of generators this way. Indeed, to our knowledge, using parsing in generator implementations [14], the framing of generators as parsers is folklore.

But the relationship between random generation and parsing deserves more than passing mentions. Both generators and parsers have rich theories, and formally connecting them is valuable for transporting results between the two areas. Our exploration down this path has already borne fruit.

How, exactly, is a random generator like a parser? A generator is a program that makes a sequence of random choices to guide the construction of some data structure. Typically, these choices are made by calling `rand()` or some similar function. But there is nothing stopping us from precomputing choices for the generator to use; we could provide a list of choices ahead of time and tell the generator to make those choices one after the other. This mode of operation is exactly parsing!

This intuition needs some fleshing out. For example, is there a representation of generators that makes this connection concrete? Where do does the probability distribution go when viewing generators as parsers? To answer these questions, we present a data structure called a *free generator* that can be interpreted as either a generator or as a parser. We establish a rich theory of free generators and use them to prove that a subset of generator programs can be factored into a parser and a distribution over sequences of choices. This gives a precise foundation for understanding generators as parsers.

Besides clarifying folklore, free generators admit transformations that make them useful as a generation representation. The most exciting transformation is a notion of *derivative* which modifies a generator by asking the question: "what would this generator look like after it makes choice c?" The derivative gives a way of previewing a particular choice to determine how good or bad that choice is.

We use free generator derivatives to implement a novel algorithm for tackling a well-documented problem, which we refer to as the *valid generation problem*. The valid generation problem requires generating a large number of the random values that satisfy some specified validity condition. This comes up often in property-based testing, where the validity condition is the precondition of some functional specification. Since generator derivatives give a way of previewing the effects of a particular choice, we can use *gradients* (derivatives of many choices at once) to preview all possible choices and pick the most promising one.

In the next section, we describe the high-level motivation behind free generators and give an overview of potential applications that justify the theory in the remainder of the paper (§2). After that, we present our main contributions:

- We formalize a folklore correspondence between parsers and generators using *free generators* and prove that every *"applicative"* generator can factored into a parser and a probability distribution (§3).
- We exploit the structure of free generators to define a *derivative* operation that computes a preview of a particular generator choice, and we prove that this operation is consistent with related notions in formal languages (§4).

PLDI'22, June 20–24, 2022, San Diego, CA, USA
2022.

- We use generator derivatives to give an algorithm that solves the valid generation problem for simple Boolean predicates (§5). Our algorithm generates significantly more valid values than rejection sampling and exemplifies the practical value of viewing generators as parsers (§6).

We conclude with related work (§7) and ideas for future research (§8).

2 The High-Level Story

Moving straight into the technical details would risk missing the forest for the trees. Accordingly, this section gives a bird's-eye view of our motivation, theory, and exploratory evaluation.

2.1 Generators and Parsers

Before trying to formalize the connection between generation and parsing, we should see some examples. Let us start by looking at a specific generator and understanding what it has in common with a specific parser.

Consider genTree, defined in Figure 1. The program genTree is a generator that produces random binary trees of Booleans like

$$\text{Node True Leaf Leaf} \quad \text{and}$$
$$\text{Node True Leaf (Node False Leaf Leaf)}$$

guided by a series of random coin flips. The choices lead the program to generate trees up to a given height via a simple recursive procedure. Note that the generator does not always make the same number of choices: sometimes it chooses to return a Leaf and terminate, and other times it chooses to build a Node, which requires more choices.[1]

Now, consider parseTree (also in Figure 1), which parses a tree from a string containing the characters `n`, `l`, `t`, and `f`. The parser turns

`ntll` into Node True Leaf Leaf and
`ntlnfll` into Node True Leaf (Node False Leaf Leaf).

The program operates character by character, consuming characters of the input string with consume and sometimes changing the way it handles the next character based on the previous character.

At this point the superficial similarities between genTree and parseTree should be clear, but what is really going on? It all comes down to *choices*. In genTree, choices are made randomly during the execution of the program, while in parseTree the choices are made ahead of time and manifest as the characters in the input string. The programs have the same structure, and only differ in their expectation of when choices should be made and the "labels" for those choices.

[1] Program synthesis experts might wonder why we use programs like this, rather than PCFGs or other related representations. Our work may very well translate to those domains, but we chose to target "applicative" generators because they are slightly more expressive.

```
genTree h =
  if h = 0 then
    return Leaf
  else
    c ← flip()
    if c == Head then return Leaf
    if c == Tail then
      c ← flip()
      if c == Head then x ← True
      if c == Tail then x ← False
      l ← genTree (h − 1)
      r ← genTree (h − 1)
      return Node x l r

parseTree h =
  if h = 0 then
    return Leaf
  else
    c ← consume()
    if c == l then return Leaf
    if c == n then
      c ← consume()
      if c == t then x ← True
      if c == f then x ← False
      else fail
      l ← parseTree (h − 1)
      r ← parseTree (h − 1)
      return Node x l r
    else fail
```

Figure 1. A generator and a parser for Boolean binary trees.

2.2 Free Generators

We can unify random generation with parsing by abstracting both ideas into a single data structure. For this, we introduce free generators.[2] Free generators are not traditional programs like genTree or parseTree, they are more like abstract syntax trees. Free generators can be *interpreted* as programs that either generate or parse.

In §3 we give a domain-specific language for constructing free generators that is almost identical to the normal way users would construct generators with QuickCheck. But we give an example here that is explicitly built from data constructors order to make it clear that these are data structures, not normal programs. When reading Figure 2, focus on the structural similarities between fgenTree and the examples in Figure 1.

The free generator fgenTree h produces binary trees of Booleans. When reading the data structure, think of Pure

[2] This document uses the `knowledge` package in LaTeX to make definitions interactive. Readers viewing the PDF electronically can click on technical terms and symbols to see where they are defined in the document.

```
fgenTree h =
  if h = 0 then
    Pure Leaf
  else
     Select
      [ (l, Pure Leaf),
        (n, MapR
          (Pair (Select
                  [ (t, Pure True),
                    (f, Pure False) ])
                (Pair (fgenTree (h − 1))
                      (fgenTree (h − 1))))
          (λ (x, (l, r)) → Node x l r)) ]
```

Figure 2. A free generator for binary trees of Booleans.

in almost the same way as **return** from the previous examples, returning a pure value without making any choices or parsing any characters. MapR takes two arguments, a free generator and a function that can be applied to the result that is generated/parsed. The Pair constructor does a sort of sequencing: it generates/parses using its first argument, then it does the same with its second argument, and finally it pairs the results together. Finally, the real magic is in the way we interpret the Select structure. When we want a generator, we treat it as making a uniform random choice, and when we want a parser we treat it as consuming a character, c, and checking it against the first elements of the pairs.

The almost line-to-line correspondence between fgenTree and genTree (or parseTree) is no accident! In §3 we give formal definitions of free generators, along with a number of interpretation functions. We use $\mathcal{G}[\![\cdot]\!]$ to mean the generator interpretation of a free generator and $\mathcal{P}[\![\cdot]\!]$ to mean the parser interpretation of a free generator. In other words,

$$\mathcal{G}[\![\text{fgenTree } 5]\!] \approx \text{genTree } 5 \quad \text{and} \quad \mathcal{P}[\![\text{fgenTree } 5]\!] \approx \text{parseTree } 5.$$

These functions allow us to write one free generator that can be used in different ways.

With the generator and parser interpretations in hand, we can ask about the formal relationship between them. The key is one final interpretation, $\mathcal{C}[\![\cdot]\!]$, the choice distribution. Intuitively, the choice distribution produces the sequences of choices that the generator interpretation can make, or equivalently the sequences that the parser interpretation can parse.

The choice distribution is used in Theorem 3.4 to connect the dots between parsing and generation. The theorem says that for any free generator g,

$$\mathcal{P}[\![g]\!] \langle\$\rangle\, \mathcal{C}[\![g]\!] \approx \mathcal{G}[\![g]\!]$$

(where $\langle\$\rangle$ is defined as a kind of "mapping" operation that applies a function to samples from a distribution). Since many normal QuickCheck generators can be written as free generators, another way to read this theorem is that generators can often be factored into two pieces: a distribution over choice sequences (given by $\mathcal{C}[\![\cdot]\!]$), and a parser of those sequences (given by $\mathcal{P}[\![\cdot]\!]$). This formalizes the intuition from §1!

2.3 Derivatives of Free Generators

Free generators do more than formalize the folklore about generators as parsers. Since a free generator defines a parser, it defines a formal language—we write the language interpretation of a free generator as $\mathcal{L}[\![\cdot]\!]$. The language of a free generator is the set of choice sequences that it can parse (or make). Viewing free generators this way suggests some interesting ways that free generators might be manipulated.

Formal languages have a notion of *derivative* due to Brzozowski [1]. For a formal language L, the Brzozowski derivative is defined as:

$$\delta_c L = \{s \mid c \cdot s \in L\}$$

In other words, the derivative of L with respect to c is all strings in L that start with c, with the first c removed.

Some models of languages, including regular expressions and context-free grammars, have syntactic transformations that correspond to derivatives. While parser programs like parseTree cannot be automatically modified in this way, we can write out what such a transformation would look like by hand. Conceptually, the derivative of a parser with respect to a character c is whatever parser remains assuming c has just been parsed.

For example, we can think of the derivative of parseTree 5 with respect to n as:

```
δn(parseTree 5) ≈
  c ← consume()
  if c == t then x ← True
  if c == f then x ← False
  else fail
  l ← parseTree 4
  r ← parseTree 4
  return Node x l r
```

This parser is "one step" simpler than parseTree 5. After parsing the character n, the next step is to parse either t or f and then construct a Node, so the derivative does just that.

We can look at another derivative of δ_{n}(parseTree 5), this time with respect to t:

```
δt δn(parseTree 5) ≈
  l ← parseTree 4
  r ← parseTree 4
  return Node True l r
```

Now we have fixed the value True for x, and we can continue by making the recursive calls and constructing the final tree.

Our free generators also have a closely related notion of derivative, and these derivatives can be computed easily! We can take derivatives of the free generator produced by fgenTree that look almost identical to the ones that we saw for parseTree:

```
δn(fgenTree 5) ≈
    MapR
      (Pair (Select
                [ (t, Pure True),
                  (f, Pure False) ])
              (Pair (fgenTree 4)
                    (fgenTree 4)))
      (λ (x, (l, r)) → Node x l r)
```

```
δtδn(fgenTree 5) ≈
    MapR
      (Pair (fgenTree 4)
            (fgenTree 4))
      (λ (l, r) → Node True l r)
```

In §4 we define a simple procedure for computing the derivative of a free generator and show that the definition behaves the way we want. Formally, we prove Theorem 4.2 which says that for all free generators g,

$$\delta_c \mathcal{L}[\![g]\!] = \mathcal{L}[\![\delta_c g]\!].$$

In other words, the derivative of the language of g is equal to the language of the derivative of g.

We can think of the derivative of a free generator as the generator that remains after a particular choice. This means that we can preview the result of making a choice without actually interpreting the free generator as a generator or parser. In the next section, we show the practical applications of this technique.

2.4 The Choice Gradient Sampling Algorithm

This section introduces Choice Gradient Sampling (CGS), an algorithm for generating data that satisfies a validity condition. Given a simple free generator, CGS builds on the ideas in the previous section: it previews choices using derivatives. In fact, it previews all possible choices, essentially taking the *gradient* of the free generator. (This is akin to the gradient in calculus, which is a vector of partial derivatives with respect to each variable.) We can write

$$\nabla g = \langle \delta_a g,\ \delta_b g,\ \delta_c g \rangle$$

for the gradient of g with respect to alphabet $\{a, b, c\}$. Each derivative in the gradient can be sampled to get a sense of how good or bad the respective choice was. This provides a metric that guides the algorithm to valid inputs.

An example execution of CGS shown pictorially in Figure 3. Starting from g we take the gradient of g by taking the derivative with respect to each possible character, in this case a, b, and c. Then we evaluate each of the derivatives by (1) interpreting the generator with $\mathcal{G}[\![\cdot]\!]$, (2) sampling values from the resulting generator, and (3) counting how many of those results are valid with respect to φ. The result of (3) is values f_a, f_b, and f_c, which we can think of as relative weights for how likely each choice is to lead us to a valid result. We then pick a choice randomly, weighted based on the values for f, and continue until our choices produce an output.

Critically, we avoid wasting effort by saving the samples (V_1 and V_2) that we use to evaluate the gradients. Many of those samples will be valid results that we can use, so there is no reason to throw them away. The final return value is the union of all of the valid samples, plus the final valid result (given by the ν function that we define in §4). We elide the finer details of the algorithm until §5.

2.5 Exploratory Evaluation

To validate the efficacy of Choice Gradient Sampling, we evaluate it on four small benchmarks, all of which are common in the property-based testing literature. Our main point of comparison is rejection sampling—sampling from a naïve generator repeatedly and discarding invalid results—which is a simple but useful baseline for understanding how well or algorithm performs. In our experiments, CGS does remarkably well on all but one benchmark, generating more than twice as many valid values as rejection sampling in the same period of time.

We discuss our experiments in detail in §6.

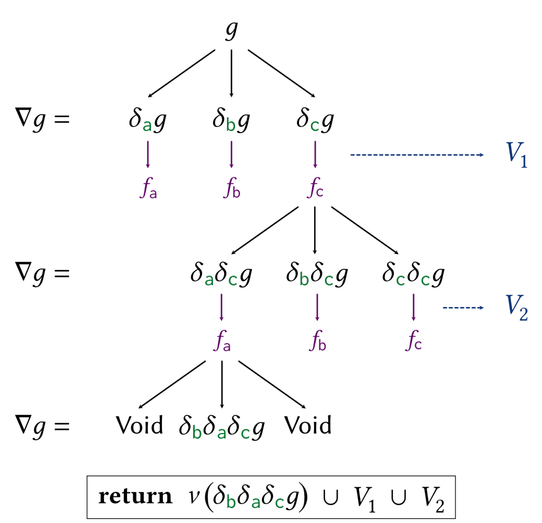


Figure 3. Choice Gradient Sampling: Generating valid values with free generator derivatives.

3 Free Generators

In this section, we develop the theory of *free generators*. We start with some background information on applicative abstractions for parsing and random generation and then present a new framing of generators that makes choices explicit and enables syntactic manipulations.

3.1 Background: Applicative Parsers and Generators

In §2 we represented generators and parsers with pseudo-code. Here we bridge the gap between the code in that section and the code that users of our theory would actually work with.

We represent both generators and parsers using *applicative functors* [15][3]—fair warning, things are about to get a bit Haskell-ey. At a high level, an applicative functor is a type constructor f with an infix operation:

```
(⟨$⟩) :: (a → b) → f a → f b
```

and operations:

```
pure :: a → f a
(⟨∗⟩) :: f (a → b) → f a → f b
```

These operations are mainly useful as a way to apply functions to values inside the type constructor f's structure. For example, the idiom "g ⟨$⟩ x ⟨∗⟩ y ⟨∗⟩ z" applies a pure function g to three structures x, y, and z.

We can use these operations to define genTree like we would in QuickCheck [4], since the type constructor Gen representing generators is an applicative functor:

```
genTree :: Int → Gen Tree
genTree 0 = pure Leaf
genTree h =
  oneof [ pure Leaf,
          Node ⟨$⟩ genInt
               ⟨∗⟩ genTree (h − 1)
               ⟨∗⟩ genTree (h − 1) ]
```

Here, pure is the trivial generator that always generates the same value, and Node ⟨$⟩ g1 ⟨∗⟩ g2 ⟨∗⟩ g3 means apply the constructor Node to three sub-generators to produce a new generator. (Operationally, this means sampling x1 from g1, x2 from g2, and x3 from g3, and then constructing Node x1 x2 x3.) Notice that we need one extra function beyond the applicative interface: oneof makes a uniform choice between generators, just as we saw in the pseudo-code.

We can do the same thing for parseTree, using combinators inspired by libraries like Parsec [12]:

```
parseTree :: Int → Parser Tree
parseTree 0 = pure Leaf
parseTree h =
  choice [ (l, pure Leaf),
           (n, Node ⟨$⟩ parseInt
                    ⟨∗⟩ parseTree (h − 1)
                    ⟨∗⟩ parseTree (h − 1)) ]
```

In this context, pure is a parser that consumes no characters and never fails. It just produces the value passed to it. We can interpret Node ⟨$⟩ p1 ⟨∗⟩ p2 ⟨∗⟩ p3 as running each sub-parser in sequence (failing if any of them fail) and then wrapping the results in the Node constructor. Finally, we have replaced oneof with choice, but the idea is the same: choose between sub-parsers.

Parsers of this form have type String → Maybe (a, String). They can be applied to a string to obtain either Nothing or Just (a, s), where a is the parse result and s contains any extra characters.

3.2 Representing Generators

With the applicative interface in mind, we can now give the formal definition of a *free generator*.[4]

3.2.1 Type Definition.

We represent free generators as an inductive data type, FGen, defined as:

```
data FGen a where
  Void :: FGen a
  Pure :: a → FGen a
  Pair :: FGen a → FGen b → FGen (a, b)
  Map :: (a → b) → FGen a → FGen b
  Select :: [(Char, FGen a)] → FGen a
```

These constructors form an abstract syntax tree with nodes that roughly correspond to the functions in the applicative interface. Clearly Pure represents pure. Pair is a slightly different form of ⟨∗⟩—one is definable from the other, but this version makes more sense as a data constructor. Map corresponds to ⟨$⟩.[5] Finally, Select subsumes both oneof and choice—it might mean either, depending on the interpretation. We need Void for technical reasons; it represents failure.

Free generators draw inspiration from *free applicative functors* [2]. Accordingly, we can write transformations from FGen a → f a for any f with similar structure. This fact motivates the rest of this section.

3.2.2 Language of a Free Generator.

We say that the *language of a free generator* is the set of choice sequences that it might make or parse. We define a generator's language

[3]For Haskell experts: we choose to focus on applicatives, not monads, to clarify our development and avoid some efficiency issues in §4 and §5. We suspect that much of our approach would work for monadic generators as well.

[4]For algebraists: free generators are "free," in the sense that they admit unique structure-preserving maps to other "generator-like" structures. In particular, the $\mathcal{G}[\![\cdot]\!]$ and $\mathcal{P}[\![\cdot]\!]$ maps are canonical. For the sake of space, we do not explore these ideas rigorously.

[5]Note that the arguments to Map flipped relative to MapR from §2.

recursively, by cases:

```
𝓛⟦·⟧ :: FGen a → Set String
𝓛⟦Void⟧       = ∅
𝓛⟦Pure a⟧     = ε
𝓛⟦Map f x⟧    = 𝓛⟦x⟧
𝓛⟦Pair x y⟧   = {s · t | s ∈ 𝓛⟦x⟧ ∧ t ∈ 𝓛⟦y⟧}
𝓛⟦Select xs⟧ = {c · s | (c, x) ∈ xs ∧ s ∈ 𝓛⟦x⟧}
```

3.2.3 Smart Constructors and Simplified Forms.

Free generators admit a useful *simplified form*. We ensure that generators are simplified by requiring that free generators are built with *smart constructors*.

Instead of Pair, users should pair free generators with ⊗:

```
(⊗) :: FGen a → FGen b → FGen (a, b)
Void   ⊗ _      = Void
_      ⊗ Void   = Void
Pure a ⊗ y      = (λb → (a, b)) ⟨$⟩ y
x      ⊗ Pure b = (λa → (a, b)) ⟨$⟩ x
x      ⊗ y      = Pair x y
```

which makes sure that Void and Pure are simplified as much as possible with respect to Pair. For example, Pure a ⊗ Pure b will simplify to Pure (a, b).

Next, ⟨$⟩ is a version of Map that does similar simplifications:

```
(⟨$⟩) :: (a → b) → FGen a → FGen b
f ⟨$⟩ Void   = Void
f ⟨$⟩ Pure a = Pure (f a)
f ⟨$⟩ x      = Map f x
```

We define pure and ⟨∗⟩ to make FGen an applicative functor:

```
pure :: a → FGen a
pure = Pure
(⟨∗⟩) :: FGen (a → b) → FGen a → FGen b
f ⟨∗⟩ x = (λ (f, x) → f x) ⟨$⟩ (f ⊗ x)
```

The smart constructor Select looks like:

```
select :: [(Char, FGen a)] → FGen a
select xs =
  case filter (λ (_, p) → p ≠ Void) xs of
    xs | xs == [] || duplicates (map fst xs) → ⊥
    xs → Select xs
```

Unlike the other smart constructors, select can return ⊥ and fail. This ensures that the operations on generators defined later in this section will be well-formed.

Finally, we define:

```
void :: FGen a
void = Void
```

for consistency.

When a generator constructed using only these smart constructors, we say it is in simplified form.

3.2.4 Examples.

We can generalize our definitions from earlier in this section to get a single free generator fgenTree that subsumes genTree and parseTree:

```
fgenTree :: Int → FGen Tree
fgenTree 0 = pure Leaf
fgenTree h =
  select [ (l, pure Leaf),
           (n, Node ⟨$⟩ fgenInt
                    ⟨∗⟩ fgenTree (h − 1)
                    ⟨∗⟩ fgenTree (h − 1)) ]
```

Excitingly, even though we are building a data structure we can write it like a program with exactly the same abstractions that are used for constructing generators and parsers.

Remark. One might be concerned that these free generators can be quite large, growing exponentially in *h*. We are able to avoid most size-related issues in Haskell due to laziness—the parts of the structure that are not yet needed are left uninterpreted—but we recognize that relying on laziness is a bit unsatisfying. Luckily it is straightforward to share the recursive calls to fgenExpr (h − 1) between all of the branches of the Select node (and the Pairs below that) in languages with pointers or references. This avoids any blowup.

Another example of a free generator produces random terms of a simply-typed lambda-calculus:

```
fgenExpr :: Int → FGen Expr
fgenExpr 0 =
  select [ (i, Lit ⟨$⟩ fgenInt),
           (v, Var ⟨$⟩ fgenVar) ]
fgenExpr h =
  select [ (i, Lit ⟨$⟩ fgenInt),
           (p, Plus ⟨$⟩ fgenExpr (h − 1)
                    ⟨∗⟩ fgenExpr (h − 1)),
           (l, Lam ⟨$⟩ fgenType
                   ⟨∗⟩ fgenExpr (h − 1)),
           (a, App ⟨$⟩ fgenExpr (h − 1)
                   ⟨∗⟩ fgenExpr (h − 1)),
           (v, Var ⟨$⟩ fgenVar) ]
```

Structurally this is very similar to the previous generator, it just has more cases and more choices. Our lambda calculus is constructed with de Bruijn indices for variables and has integers and functions as values. The "raw" untyped language is exceedingly simple, but generating well-typed terms is still fairly difficult. We use this example as one of our case studies in §6.

3.3 Interpreting Free Generators

A free generator does not *do* anything on its own—it is simply a data structure. In this section, we see the formal definitions of the interpretation functions that we mentioned in §2 and prove a theorem that links those interpretations together.

3.3.1 As a Generator of Values.

The most natural way to interpret a free generator is as a QuickCheck generator—that is, as a distribution over data structures. We define the *generator interpretation* of a free generator to be:

$$
\begin{aligned}
&\mathcal{G}[\![\cdot]\!] :: \mathsf{FGen\ a} \to \mathsf{Gen\ a}\\
&\mathcal{G}[\![\mathsf{Void}]\!] \quad = \bot\\
&\mathcal{G}[\![\mathsf{Pure\ v}]\!] \quad = \mathsf{pure\ v}\\
&\mathcal{G}[\![\mathsf{Map\ f\ x}]\!] \ = \mathsf{f}\ \langle\$\rangle\ \mathcal{G}[\![\mathsf{x}]\!]\\
&\mathcal{G}[\![\mathsf{Pair\ x\ y}]\!] \ =\\
&\quad (\lambda \mathsf{x\ y} \to (\mathsf{x},\ \mathsf{y}))\ \langle\$\rangle\ \mathcal{G}[\![\mathsf{x}]\!]\ \langle * \rangle\ \mathcal{G}[\![\mathsf{y}]\!]\\
&\mathcal{G}[\![\mathsf{Select\ xs}]\!] \ =\\
&\quad \mathsf{oneof}\ (\mathsf{map}\ (\lambda\ (\_,\ \mathsf{x}) \to \mathcal{G}[\![\mathsf{x}]\!])\ \mathsf{xs})
\end{aligned}
$$

Notice that the implementation of this interpretation is quite straightforward: in most cases, it simply maps the "AST node" version of an applicative operation to the concrete version implemented by Gen.

One detail worth noting is that the interpretation behaves poorly on Void, but we can prove a lemma to show that this does not cause problems in practice:

Lemma 3.1. *If a generator g is simplified,*

$$g \text{ contains } Void \iff g = Void.$$

Proof. By induction on the structure of g and inspection of the smart constructors. □

Thus we can conclude that as long as g is in simplified form and not Void, $\mathcal{G}[\![g]\!]$ is defined.

Example 3.2. $\mathcal{G}[\![\mathsf{fgenTree\ 5}]\!]$ is equivalent to genTree 5.

3.3.2 As Parser of Choice Sequences.

Of course, there would be no point in defining free generators if we were only going to interpret them as QuickCheck generators. We can make use of the choice labels using the free generator's *parser interpretation*—in other words, viewing it as a parser of choices as we originally wanted. The translation looks like:

$$
\begin{aligned}
&\mathcal{P}[\![\cdot]\!] :: \mathsf{FGen\ a} \to \mathsf{Parser\ a}\\
&\mathcal{P}[\![\mathsf{Void}]\!] \quad = \lambda \mathsf{s} \to \mathsf{Nothing}\\
&\mathcal{P}[\![\mathsf{Pure\ a}]\!] \quad = \mathsf{pure\ a}\\
&\mathcal{P}[\![\mathsf{Map\ f\ x}]\!] \ = \mathsf{f}\ \langle\$\rangle\ \mathcal{P}[\![\mathsf{x}]\!]\\
&\mathcal{P}[\![\mathsf{Pair\ x\ y}]\!] \ =\\
&\quad (\lambda \mathsf{x\ y} \to (\mathsf{x},\ \mathsf{y}))\ \langle\$\rangle\ \mathcal{P}[\![\mathsf{x}]\!]\ \langle * \rangle\ \mathcal{P}[\![\mathsf{y}]\!]\\
&\mathcal{P}[\![\mathsf{Select\ xs}]\!] \ =\\
&\quad \mathsf{choice}\ (\mathsf{map}\ (\lambda\ (\mathsf{c},\ \mathsf{x}) \to (\mathsf{c},\ \mathcal{P}[\![\mathsf{x}]\!]))\ \mathsf{xs})
\end{aligned}
$$

This definition uses the representation of parsers as functions of type String → Maybe (a, String) that we saw earlier, and, just like $\mathcal{G}[\![\cdot]\!]$, mostly just maps between applicative operations.

Example 3.3. $\mathcal{P}[\![\mathsf{fgenTree\ 5}]\!]$ is equivalent to parseTree 5.

3.3.3 As a Generator of Choice Sequences.

Our final interpretation of free generators is a sort of dual to the previous one. Instead of throwing away randomness in favor of deterministic parsing, we can extract only the random distribution and ignore everything about how result values are constructed. We define the *choice distribution* of a free generator to be:

$$
\begin{aligned}
&\mathcal{C}[\![\cdot]\!] :: \mathsf{FGen\ a} \to \mathsf{Gen\ String}\\
&\mathcal{C}[\![\mathsf{Void}]\!] \quad = \bot\\
&\mathcal{C}[\![\mathsf{Pure\ a}]\!] \quad = \mathsf{pure}\ \varepsilon\\
&\mathcal{C}[\![\mathsf{Map\ f\ x}]\!] \ = \mathcal{C}[\![\mathsf{x}]\!]\\
&\mathcal{C}[\![\mathsf{Pair\ x\ y}]\!] \ =\\
&\quad (\lambda \mathsf{s\ t} \to \mathsf{s} \cdot \mathsf{t})\ \langle\$\rangle\ \mathcal{C}[\![\mathsf{x}]\!]\ \langle * \rangle\ \mathcal{C}[\![\mathsf{y}]\!]\\
&\mathcal{C}[\![\mathsf{Select\ xs}]\!] \ =\\
&\quad \mathsf{oneof}\ (\mathsf{map}\ (\lambda\ (\mathsf{c},\ \mathsf{x}) \to (\lambda \mathsf{s} \to \mathsf{c} \cdot \mathsf{s})\ \langle\$\rangle\ \mathcal{C}[\![\mathsf{x}]\!])\ \mathsf{xs})
\end{aligned}
$$

This definition is not quite as obvious as the previous two, but it is still natural. We can think of the result of this interpretation as a distribution over $\mathcal{L}[\![g]\!]$. The language of a free generator is exactly those choice sequences that the generator interpretation can make and the parser interpretation can parse.

3.3.4 Factoring Generators.

These different interpretations of free generators are closely related to one another, and in particular we can reconstruct $\mathcal{G}[\![\cdot]\!]$ from $\mathcal{P}[\![\cdot]\!]$ and $\mathcal{C}[\![\cdot]\!]$. In essence, this means that a free generator's generator interpretation can be factored into a distribution over choice sequences and a parser of those sequences.

To make this more precise, we need a notion of equality for generators like the ones produced via $\mathcal{G}[\![\cdot]\!]$. We say two QuickCheck generators are *equivalent*, written $g_1 \equiv g_2$, if and only if the generators represent the same distribution over values. This is coarser notion than program equality, since two generators might produce the same distribution of values in different ways.

With this in mind, we can state and prove the relationship between different interpretations of free generators:

Theorem 3.4 (Factoring)**.** *Every simplified free generator can be factored into a parser and a distribution over choice sequences. In other words, for all simplified free generators $g \neq Void$,*

$$\mathcal{P}[\![g]\!]\ \langle\$\rangle\ \mathcal{C}[\![g]\!] \equiv (\lambda x \to Just\ (x, \varepsilon))\ \langle\$\rangle\ \mathcal{G}[\![g]\!].$$

Proof sketch. We proceed by induction on the structure of g.

Case g = Pure a. Straightforward.
Case g = Map f x. Straightforward.
Case g = Pair x y. This case is the most interesting one. The difficulty is that it is not immediately obvious why $\mathcal{P}[\![\mathsf{Pair\ x\ y}]\!]\ \langle\$\rangle\ \mathcal{C}[\![\mathsf{Pair\ x\ y}]\!]$ should be a function of $\mathcal{P}[\![\mathsf{x}]\!]\ \langle\$\rangle\ \mathcal{C}[\![\mathsf{x}]\!]$ and $\mathcal{P}[\![\mathsf{y}]\!]\ \langle\$\rangle\ \mathcal{C}[\![\mathsf{y}]\!]$. Showing the correct relationship requires a lemma that says that for any sequence s generated by $\mathcal{C}[\![\mathsf{x}]\!]$ and an arbitrary sequence t, there is some a such that $\mathcal{P}[\![\mathsf{x}]\!]\ (s \cdot t) =$ Just (a, t).

Case g = Select xs. The reasoning in this case is a bit subtle, since it requires certain operations to commute with Select, but the details are not particularly instructive.

See Appendix B for the full proof. □

A natural corollary of Theorem 3.4 is the following:

Corollary 3.5. *Any applicative generator, γ, written in terms of pure functions,* ⟨\$⟩*,* pure*,* ⟨∗⟩*, and* oneof*, can be factored into a parser and distribution over choice sequences.*

Proof. Translate γ into a free generator, g, by replacing operations with the equivalent smart constructor. (For oneof, draw unique labels for each choice and use select.) By induction, $\mathcal{G}[\![g]\!] = \gamma$.

The resulting free generator can be factored into a parser and a choice distribution via Theorem 3.4. Thus,

$$(\lambda x \to \mathsf{Just}\ (x, \varepsilon))\ \langle\$\rangle\ \gamma \equiv \mathcal{P}[\![g]\!]\ \langle\$\rangle\ \mathcal{C}[\![g]\!],$$

and γ can be factored as desired. □

This corollary gives a concrete way to view the connection between applicative generators and parsers.

3.4 Replacing a Generator's Distribution

Since a generator g can be factored using $\mathcal{C}[\![\cdot]\!]$ and $\mathcal{P}[\![\cdot]\!]$, we can explore what it would look like to modify a generator's distribution (i.e., change or replace $\mathcal{C}[\![\cdot]\!]$) without having to modify the entire generator.

Suppose we have some other distribution that we want our choices to follow, represented as a function from a history of choices to a generator of next characters:

```
type DistF = String → Gen (Maybe Char)
```

(If the next choice is Nothing, then generation stops.) We can be a bit more general and pair a distribution function with a "current" history, to get our formal definition of a custom choice distribution:

```
type Dist = (String, DistF)
```

A Dist may be arbitrarily complex—in particular, it might contain information obtained from a machine learning model, example-based tuning, or some other automated tuning process. How would we use such a distribution in place of the standard distribution given by $\mathcal{C}[\![\cdot]\!]$?

The solution is to replace $\mathcal{C}[\![\cdot]\!]$ with a distribution to yield the *definition*:

```
G̅[[·]] :: (Dist, FGen a) → Gen (Maybe a)
G̅[[((h, d), g)]] = P[[g]] ⟨$⟩ genDist h
  where
    genDist h = d h >>= λx → case x of
      Nothing → pure h
      Just c → genDist (h · c)
```

Whereas before we proved an equivalence between $\mathcal{G}[\![\mathsf{g}]\!]$ and $\mathcal{P}[\![\mathsf{g}]\!]\langle\$\rangle\ \mathcal{C}[\![\mathsf{g}]\!]$, we can now use that relationship as a definition of what it means to interpret a generator under a new distribution.

Since replacing a free generator's distribution does not actually change the structure of the generator, we can have a different distribution for each use-case of the free generator. In a property-based testing scenario, one could imagine the tester finely-tuning a distribution for each property that is carefully optimized to find bugs as quickly as possible.

4 Derivatives of Free Generators

In this section, we review the notion of Brzozowski derivative in formal language theory and we show that a similar operation exists for free generators. Free generator derivatives highlight the advantages of taking the correspondence between generators and parsers seriously.

4.1 Background: Derivatives of Languages

The *Brzozowski derivative* [1] of a formal language L with respect to some character c is defined as

$$\delta_c L = \{s \mid c \cdot s \in L\}.$$

In other words, the derivative is the set of strings in L with c removed from the front. For example,

$$\delta_{\mathsf{a}}\{\mathsf{abc}, \mathsf{aaa}, \mathsf{bba}\} = \{\mathsf{bc}, \mathsf{aa}\}.$$

Many language representations have syntactic transformations that correspond to Brzozowski derivatives. For example, we can take the derivative of a regular expression:

$$\begin{aligned}
\delta_c \varnothing &= \varnothing \\
\delta_c \varepsilon &= \varnothing \\
\delta_c \mathsf{c} &= \varepsilon \quad (c = \mathsf{c}) \\
\delta_c \mathsf{d} &= \varnothing \quad (c \neq \mathsf{d}) \\
\delta_c (r_1 + r_2) &= \delta_c r_1 + \delta_c r_2 \\
\delta_c (r_1 \cdot r_2) &= \delta_c r_1 \cdot r_2 + \nu r_1 \cdot \delta_c r_2 \\
\delta_c (r^*) &= \delta_c r \cdot r^*
\end{aligned}
\qquad
\begin{aligned}
\nu \varnothing &= \varnothing \\
\nu \varepsilon &= \varepsilon \\
\nu \mathsf{c} &= \varnothing \\
\nu (r_1 + r_2) &= \nu r_1 + \nu r_2 \\
\nu (r_1 \cdot r_2) &= \nu r_1 \cdot \nu r_2 \\
\nu (r^*) &= \varepsilon
\end{aligned}$$

The ν operator, used in the "·" rule and defined on the right, determines the *nullability* of an expression (whether or not it accepts ε). As one would hope, if r has language L, it is always the case that $\delta_c r$ has language $\delta_c L$.

4.2 The Free Generator Derivative

Since free generators define a language (given by $\mathcal{L}[\![\cdot]\!]$), it makes sense to ask: can we take their derivatives? Yes! We define the *derivative of a free generator* to be:

$$
\begin{aligned}
&\delta :: \mathsf{Char} \to \mathsf{FGen\ a} \to \mathsf{FGen\ a}\\
&\delta_c \mathsf{Void} &&= \mathsf{void}\\
&\delta_c(\mathsf{Pure\ v}) &&= \mathsf{void}\\
&\delta_c(\mathsf{Map\ f\ x}) &&= \mathsf{f}\ \langle\$\rangle\ \delta_c \mathsf{x}\\
&\delta_c(\mathsf{Pair\ x\ y}) &&= \delta_c \mathsf{x} \otimes \mathsf{y}\\
&\delta_c(\mathsf{Select\ xs}) &&= \mathsf{if}\ \ (c, \mathsf{x}) \in \mathsf{xs\ then\ x\ \ else\ \ void}
\end{aligned}
$$

These definitions should be mostly intuitive. The derivative of a generator that does not make a choice (i.e., Void and Pure) is void, since the corresponding language would be empty. The derivative commutes with Map since the transformation affects choices, not the final result. Select's derivative is just the argument generator corresponding to the appropriate choice.

The one potentially confusing case is the one for Pair. We have defined the derivative of a pair of generators by taking the derivative of the first generator in the pair and leaving the second unchanged, but this is inconsistent with the case for "·" in the regular expression derivative. What happens when the first generator's language is nullable? Luckily, our simplified form clears up the confusion: if Pair x y is in simplified form, x is not nullable. This is a simple corollary of Lemma 4.1.

Lemma 4.1. *If g is in simplified form, then either g = Pure a or $\nu(\mathcal{L}[\![g]\!]) = \varnothing$.*

Proof sketch. See Appendix A. □

Remark. The derivative of a simplified generator is simplified. This follows simply from the definition, since we only use smart constructors and parts of the original generator to build the derivative generators. By induction, this also means that repeated derivatives preserve simplification.

Besides clearing up the issue with Pair, Lemma 4.1 also says that we can define *nullability* for free generators simply as:

$$
\begin{aligned}
&\nu :: \mathsf{FGen\ a} \to \mathsf{Set\ \ a}\\
&\nu(\mathsf{Pure\ v}) = \{\mathsf{v}\}\\
&\nu \mathsf{g} \qquad\ \ = \varnothing \qquad (\mathsf{g} \neq \mathsf{Pure\ v})
\end{aligned}
$$

A generator is nullable if and only if it can produce a result without making any more choices.

With the derivative operation defined, we can prove a concrete theorem that says our definition of derivative acts the way we expect:

Theorem 4.2 (Generator Derivative Consistency)**.** *For all simplified free generators g and characters c,*

$$\delta_c \mathcal{L}[\![g]\!] = \mathcal{L}[\![\delta_c g]\!].$$

Proof sketch. The proof proceeds by mostly straightforward induction, but it is interesting enough to be worth writing out. See Appendix 4.2. □

5 Generating Valid Results with Gradients

In this section, we put the theory of free generators and their derivatives into practice. We present Choice Gradient Sampling (CGS), which tackles the *valid generation problem*: CGS guides a free generator to generate values that satisfy a Boolean precondition. This is critical in contexts like property-based testing.

The basic intuition behind CGS is that a generator's derivative with respect to a choice c contains information about how "good" or "bad" the choice c would be—in other words, how likely that choice is to lead the generator to a valid result. In order to access this information, we can sample from $\mathcal{G}[\![\delta_c g]\!]$ a number of times and count up how many of the samples are valid. The more valid samples we get, the more likely c will lead us to a valid result at the end of our search.

5.1 The Algorithm

With this intuition in mind, we present Choice Gradient Sampling (shown in Figure 5), which searches for valid results using repeated free generator *gradients* (i.e., derivatives with respect to every available choice). Given a free generator G in simplified form and a validity predicate φ, the CGS produces a set of outputs $\mathcal{O}$ such that $\forall x \in \mathcal{O}.\ \varphi(x)$.

1: $g \leftarrow G$
2: $\mathcal{V} \leftarrow \varnothing$
3: **while true do**
4: **if** $\nu g \neq \varnothing$ **then return** $\nu g \cup \mathcal{V}$
5: **if** g = Void **then** $g \leftarrow G$
6: $\nabla g \leftarrow \langle \delta_c g \mid c \in C \rangle$ ▹ ∇g is the gradient of g
7: **for** $\delta_c g \in \nabla g$ **do**
8: **if** $\delta_c g$ = Void **then**
9: $V \leftarrow \varnothing$
10: **else**
11: $x_1, \ldots, x_N \leftsquigarrow \mathcal{G}[\![\delta_c g]\!]$ ▹ Sample $\mathcal{G}[\![\delta_c g]\!]$
12: $V \leftarrow \{x_j \mid \varphi(x_j)\}$
13: $f_c \leftarrow |V|$ ▹ f_c is the *fitness* of c
14: $\mathcal{V} \leftarrow \mathcal{V} \cup V$
15: **if** $\max_{c \in C} f_c = 0$ **then**
16: **for** $c \in C$ **do** $f_c \leftarrow 1$
17: $g \leftsquigarrow$ weightedChoice $\{(f_c, \delta_c g) \mid c \in C\}$

(Where $N \in \mathbb{N}$ is the *sample rate constant*.)

Figure 5. Choice Gradient Sampling

The intuition from earlier plays out in lines 7–14. We take the derivative of our current generator with respect to each choice, sample values from each derivative, and then choose one of the derivatives with weights proportional to the number of valid samples from each. Since each choice is made based on the "fitness" of the associated derivative, we are likely to keep making choices that lead us to valid results.

As discussed in §2, it is important to track set $\mathcal{V}$, which keeps track of all of the valid samples that we generate while evaluating derivatives. Storing the valid samples that we find along the way ensures that no effort is wasted by the sampling process.

5.2 Modified Distributions

Before moving on, here an interesting extension to CGS based on the theory in §3.4. In that section we show that replacing a generator's distribution is as simple as pairing it with a distribution over choices (represented by the type Dist). It turns out that there is a straightforward way to adapt CGS to work for distribution-modified generators!

CGS requires that our generator structure admit three basic operations: δ_c so we can compute gradients, $\mathcal{G}[\![\cdot]\!]$ so we can sample from gradients, and ν so we know when to stop. We already showed how to define $\overline{\mathcal{G}}[\![\cdot]\!]$ for generators with modified distributions, so we only need definitions of δ_c and ν.

As a starting point, we can observe that Dist admits a simple kind of derivative:

$$\delta_c(\mathsf{h}, \mathsf{d}) = (\mathsf{h} \cdot \mathsf{c},\ \mathsf{d})$$

Intuitively, the derivative just internalizes c into the history, so future queries of the distribution take that character into account. Given that, we can further define the derivative of a pair $(\mathsf{d},\ \mathsf{g})$ of a Dist and a free generator to be:

$$\delta_c(\mathsf{d}, \mathsf{g}) = (\delta_c \mathsf{d}, \delta_c \mathsf{g})$$

Finally, to complete the construction, we can say that a modified generator's nullable set is the same as the nullable set of the underlying generator. With these definitions in hand, we can adapt our algorithm with essentially no changes!

6 Exploratory Evaluation

This paper is primarily about the theory of free generators and their derivatives, but we were curious to see how well Choice Gradient Sampling would perform on a few property-based testing benchmarks. This section describes our experiments in detail and shows that, with a few caveats, CGS presents a promising approach to solving the valid generation problem.

6.1 Experimental Setup

Our experiments compare CGS to the default way QuickCheck handles properties with preconditions (assuming there is no bespoke generator available), rejection sampling. Rejection sampling takes a generic generator, samples from it, and simply discards any results that are not valid. We chose this over more state-of-the-art comparisons [3, 16], because our primary goal was to validate our theory, not to produce a production-ready tool.

We use four simple free generators in order to test four different benchmarks: **BST**, **SORTED**, **AVL**, and **STLC**. Information about each of these benchmarks is given in Table 1.

Each of these benchmarks requires a free generator as a starting point, and the exact implementation of that initial generator can have a significant impact on the final performance of generation. In order to avoid any generator cleverness complicating the final results, we chose to implement generators that followed the our inductive data types as closely as possible. For example, fgenTree, shown in §3 and used in the **BST** benchmark, follows the structure of Tree exactly. In fact, all of the generators that we use in benchmarks could be reconstructed from only type information, given appropriate meta-programming effort. For each benchmark we also chose an appropriate sample-rate constant N, based on the complexity of the particular task at hand. In general, a higher value for N means that estimates of choice fitness will be better, at the cost of increased time spent sampling. In practice this could be determined by the programmer or via heuristics based on the data type.

6.2 Results

With a generator and a sample rate constant chosen, we ran CGS and Rejection for one minute each and recorded the unique valid values produced. The totals are presented in Table 2.

These numbers are quite promising—CGS is always able to generate more unique values than Rejection in the same amount of time, and it often generates *significantly* more. (Later in this section we explore the **AVL** benchmark to understand why it does not perform as well.)

In addition to the raw numbers, we tracked some other metrics. The plots in Figure 6 give some deeper insights. The first plot ("Unique Terms over Time") shows that after one minute, CGS has not yet "run out" of unique terms to generate. The "Normalized Size Distribution" chart shows that CGS also generates larger terms on average. This is great from the perspective of property-based testing, where test size is often positively correlated with bug-finding power since larger test inputs tend to exercise more of the implementation code. Finally, "Normalized Constructor Frequency" demonstrates some room for improvement: the higher green bars for "Plus" and "Lit$n$" suggest that CGS capitalizes on the fact that integer expressions are easy to generate by slightly biasing generation towards terms containing them. However, this effect is minor, and **STLC** is actually our worst-performing benchmark in that regard.

We give charts for all four benchmarks in Appendix D.

6.3 Measuring Diversity

Constructor diversity is a good starting point for understanding differences between rejection sampling and CGS, but we also sought to measure a more robust notion of value diversity—after all, in a testing scenario more diverse tests

	Free Generator	Validity Condition	N	Depth
BST	Binary trees with values 0–9	Is a valid BST	50	5
SORTED	Lists with values 0–9	Is sorted	50	20
AVL	Binary trees with values and stored heights 0–9	Is a valid AVL tree (balanced)	500	5
STLC	Arbitrary ASTs for λ-terms	Is well-typed	400	5

Table 1. Overview of benchmarks.

	BST	**SORTED**	**AVL**	**STLC**
Rejection	9,762	6,452	156	106,282
CGS	22,286	59,436	221	298,001

Table 2. Unique valid values generated in 60 seconds.

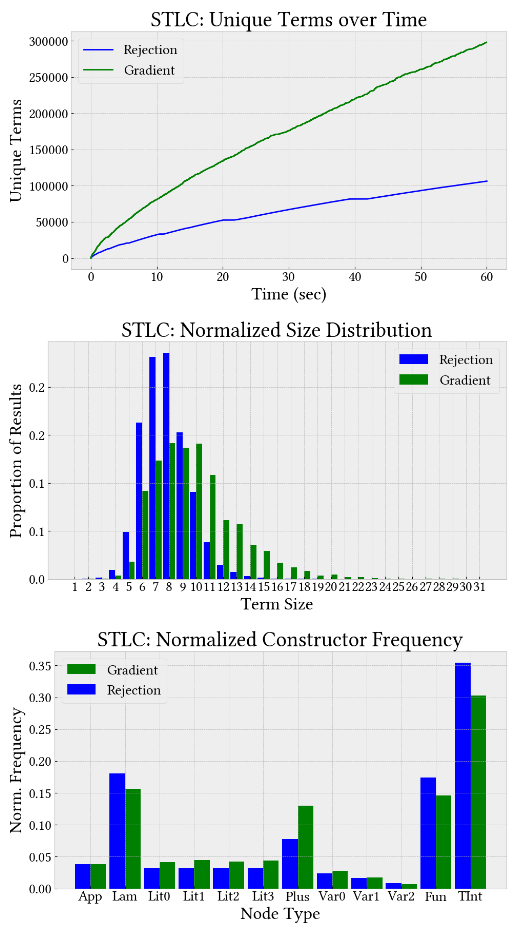


Figure 6. Unique values and term sizes for the **STLC** benchmark.

lead to higher bug-finding potential. To do this, we first note that the values that they generate are roughly isomorphic to the choice sequences that generated them. For example, in the case of **BST**, the sequence `n5l6ll` can be parsed to produce `Node 5 Leaf (Node 6 Leaf Leaf)` and a simple in-order traversal can recover `n5l6ll` again. This means that it is safe to measure choice sequence diversity as a proxy for value diversity.

With this in mind, we compute a *diversity metric* over our generated values by estimating the average Levenshtein distance [13] between pairs of choice sequences in the generated results. Computing the true mean would be computationally infeasible, so we settle for the mean of a sample of 3000 pairs from each set of valid values. The results are summarized in Table 3.

	BST	**SORTED**	**AVL**	**STLC**
Rej.	7.72(1.68)	4.77(1.14)	4.53(2.08)	12.31(4.63)
CGS	8.85(1.95)	7.34(2.02)	4.43(2.00)	13.64(4.83)

Table 3. Average Levenshtein distance between pairs of choice sequences.

These results are not as dramatic as we had hoped, with diversity only increasing by around 10% for **STLC** and around the same for **BST**. One explanation for this effect might come down to the fact that CGS retains so many of its samples. Late in the algorithm, when a large prefix of choices is already fixed, samples from the derivative generators will be quite similar. This likely results in some clusters of inputs that are all valid, but that only explore one particular shape of input.

All of that said, CGS is still promising. To start, it is already common practice to test clusters of similar inputs in certain fuzzing contexts [10], so the fact that CGS does this is not unusual. In fact, this method has been shown to be effective at finding bugs in some cases. Additionally, for most of our benchmarks (again, we return to **AVL** in a moment) CGS does increase diversity of tests; combined with the sheer number of valid inputs available, this means that CGS covers a slightly larger space of tests much more thoroughly. This effect should lead to better bug-finding in testing scenarios.

6.4 The Problem with AVL

Of course, the outlier in all of these results seems to be the **AVL** benchmark. CGS only manages to find a modest number of extra valid AVL trees, and their pairwise diversity is actually slightly worse than that of rejection sampling. Why might this be? We suspect that this is effect arises because AVL trees are so difficult to find randomly. Balanced binary search trees are hard to generate on their own, and AVL trees are even more difficult because the generator must guess the correct height to cache at each node. This is why rejection sampling only finds 156 AVL trees in the time it takes to find 9,762 binary search trees.

The problem with AVL trees being so hard to generate is that CGS unlikely find *any* valid trees while sampling. In particular, the check in line 15 of Figure 3 is often true, meaning that choices made uniformly at random rather than guided by the fitness of the appropriate derivatives. We could mitigate this problem by significantly increasing the sample rate constant N, but then sampling time would likely dominate generation time and result in worse performance overall.

Ultimately this failure is disappointing, but not wholly surprising—CGS approximates generator fitness via sampling, and that approximation cannot always be accurate. For now it seems that especially hard-to-satisfy predicates are out of reach for this specific algorithm. Still, gradient-based algorithms might not all fail in this way. In §8 we discuss ideas for alternative algorithms that may perform better in tricky cases.

7 Related Work

7.1 Similar Approaches

The Python library Hypothesis [14] implements its generators by parsing a stream of random bits. This is another point in favor of thinking of generators as parsers, but it is more of an implementation detail.

In, *Generating constrained random data with uniform distribution* [3], Claessen et al. present a structure that is superficially similar to our free generator structure, but which is used in a very different way. They primarily use the syntactic structure of their generators (they call them "Spaces") to control the size distribution of the outputs; in particular, Spaces do not make choice information explicit in the way free generators do. Claessen et al.'s generation approach uses Haskell's laziness, rather than derivatives and sampling, to prune unhelpful paths in the generation process. It seems plausible that we could incorporate some of these ideas into our work to improve performance and give finer control over size distributions.

The Clotho [5] library is another interesting point in the generator design space, though it does not incorporate parsing.

7.2 The Valid Generation Problem

Many other partial solutions to the valid generation problem exist.

The domain specific generator languages provided by QuickCheck [8] make it easier to write manual generators that produce valid inputs by construction. We avoid manual approaches like this in the hopes of making techniques like property-based testing more accessible to those do not have experience writing their own generators.

When validity predicates are expressed as inductive relations, approaches like the one in *Generating Good Generators for Inductive Relations* [11] are extremely powerful. Unfortunately, most programming languages cannot express inductive relations that capture the kinds of preconditions that we care about.

Some approaches have tried to use machine learning to automatically generate valid inputs. Learn&Fuzz [6] generates valid data using a recurrent neural network. While the results are promising, this solution seems to work best when a large corpus of inputs is already available and the validity condition is more structural than semantic. In the same vein, RLCheck [16] uses reinforcement learning to guide a generator to valid inputs. We hope to incorporate ideas from RLCheck into this work in the future, but we felt that getting the theory of free generators right was a critical first step.

8 Future Directions

There are a number of exciting paths forward from this work: some continue our theoretical exploration, while others look towards algorithmic improvements that would be incredibly useful in practice.

8.1 Bidirectional Free Generators

We believe that we have only scratched the surface of what is possible with free generators. Making choices explicit opens the door for other transformations and interpretations that take advantage of the extra information. One concrete idea would be to merge the theory of free generators with the emerging theory of *ungenerators* [7]. That work expresses generators that can be run both forward (to generate values as usual) and *backward*. In the backward direction, the program takes a value that the generator might have generated and "un-generates" it to give a sequence of choices that the generator might have made when generating that value.

The free generator formalism is quite compatible with these ideas—since free generators make choices explicit. Turning a free generator into a bidirectional generator that can both generate and ungenerate should be fairly straightforward. From there, we can build on the ideas in the ungenerators work and use the backward direction of the generator to learn a distribution of choices that approximates some user-provided samples of "desirable" values. Used in conjunction with the extended algorithm from §5.2, this would give a better starting point for generation with little extra work from the user.

8.2 Algorithmic Optimizations

In §6.4, we uncover some problems with the Choice Gradient Sampling algorithm. Recall that because CGS evaluates derivatives via sampling, it does poorly on when validity conditions are particularly difficult to satisfy. This begs the question: might it be possible to evaluate the fitness of a derivative without naïvely sampling?

One potential angle involves staging the sampling process. Given a free generator with a depth parameter, first evaluate choices on generators of size 1, then evaluate choices with size 2, etc. These intermediate stages will make gradient sampling more successful at larger sizes, and might significantly improve the results on benchmarks like **AVL**. Unfortunately, this kind of approach might perform poorly on benchmarks like **STLC** where the validity condition is not uniform: size-1 generators would avoid generating variables, leading larger generators to avoid variables as well. In any case, we think this design space is worth exploring.

8.3 Making Choices with Neural Networks

Another algorithmic optimization is considerably farther afield: we think it may be possible to use recurrent neural networks (RNNs) to improve our generation procedure.

As Choice Gradient Sampling makes choices, it generates a lot of useful data about the frequencies with which choices should be made. Specifically, every iteration of the algorithm produces a pair of a history and a distribution over next choices that looks something like:

$$\mathsf{abcca} \mapsto \{\mathsf{a} : 0.3, \mathsf{b} : 0.7, \mathsf{c} : 0.0\}$$

In the course of CGS, this information is used once (to make the next choice) and then forgotten—what if there was a way to learn from it? Pairs like this could be used to train an RNN to make choices that are similar to the ones made by CGS.

There are still plenty of details to work out, including network architecture, hyper-parameters, etc., but given sufficient performance, we could run CGS for a while, then train the model, and after that point only use the model to generate valid data. Staging things this way would recover some of the time that is currently wasted by the constant sampling of derivative generators. One could imagine a user writing a definition of a type and a predicate for that type, and then setting the model to train while they work on their algorithm. By the time the algorithm is finished and ready to test, the RNN model would be trained and ready to produce valid test inputs. A workflow like this could significantly increase adoption of property-based testing in industry and give developers more control over the quality of their software.

References

[1] Janusz A Brzozowski. 1964. Derivatives of regular expressions. *Journal of the ACM (JACM)* 11, 4 (1964), 481–494.

[2] Paolo Capriotti and Ambrus Kaposi. 2014. Free applicative functors. *arXiv preprint arXiv:1403.0749* (2014).

[3] Koen Claessen, Jonas Duregård, and Michal H. Palka. 2015. Generating constrained random data with uniform distribution. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000143

[4] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, Montreal, Canada, 268–279. https://doi.org/10.1145/351240.351266

[5] Pierce Darragh, William Gallard Hatch, and Eric Eide. 2021. Clotho: A Racket Library for Parametric Randomness. In *Functional Programming Workshop*. 3.

[6] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59.

[7] Harrison Goldstein. 2021. Ungenerators. In *ICFP Student Research Competition*. https://harrisongoldste.in/papers/icfpsrc21.pdf

[8] John Hughes. 2007. QuickCheck testing for fun and profit. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 1–32.

[9] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 114–129. http://dl.acm.org/citation.cfm?id=3009868

[10] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *PACMPL* 3, OOPSLA (2019), 181:1–181:29. https://doi.org/10.1145/3360607

[11] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2017. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[12] Daan Leijen and Erik Meijer. 2001. Parsec: Direct style monadic parser combinators for the real world. (2001).

[13] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.

[14] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891.

[15] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 1 (2008), 1–13.

[16] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly generating diverse valid test inputs with reinforcement learning. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1410–1421. https://doi.org/10.1145/3377811.3380399

Appendix

A Proof of Lemma 4.1

Lemma 4.1. *If g is in simplified form, then either g = Pure a or $\nu\mathcal{L}[\![g]\!] = \emptyset$.*

Proof. We proceed by induction on the structure of g.

Case g = Void. Trivial.

Case g = Pure a. Trivial.

Case g = Pair x y. By our inductive hypothesis, x = Pure a or $\nu\mathcal{L}[\![\text{x}]\!] = \emptyset$.
Since the smart constructor $\otimes$ never constructs a Pair with Pure on the left, it must be that $\nu\mathcal{L}[\![\text{x}]\!] = \emptyset$.
Therefore, it must be the case that $\nu\mathcal{L}[\![\text{Pair x y}]\!] = \emptyset$.

Case g = Map f x.
Similarly to the previous case, our inductive hypothesis and simplification assumptions imply that $\nu\mathcal{L}[\![\text{x}]\!] = \emptyset$.
Therefore, $\nu\mathcal{L}[\![\text{Map f y}]\!] = \emptyset$.

Case g = Select xs.
It is always the case that $\nu\mathcal{L}[\![\text{Select xs}]\!] = \emptyset$.

Thus, we have shown that every simplified free generator is either Pure a or has an empty nullable set. ◻

B Proof of Theorem 3.4

Lemma B.1. *Pairing two parsers and mapping over the concatenation of the associated choice distributions is equal to a function of the two parsers mapped over the distributions individually. Specifically, for all simplified free generators* x *and* y*,*

$$
\begin{aligned}
((\lambda x\ y \to (x, y)) \langle\$\rangle\, \mathcal{P}\llbracket x \rrbracket \langle * \rangle\, \mathcal{P}\llbracket y \rrbracket) \langle\$\rangle ((\cdot) \langle\$\rangle\, \mathcal{C}\llbracket x \rrbracket \langle * \rangle\, \mathcal{C}\llbracket y \rrbracket) \equiv & (\lambda a_\perp\ b_\perp \to \textbf{\textit{case}}\ (a_\perp, b_\perp)\ \textbf{\textit{of}} \\
& \quad (Just\ (a, \_), Just\ (b, \_)) \to Just\ ((a, b), \varepsilon) \\
& \quad \_ \to Nothing) \\
& \langle\$\rangle (\mathcal{P}\llbracket x \rrbracket \langle\$\rangle\, \mathcal{C}\llbracket x \rrbracket) \langle * \rangle (\mathcal{P}\llbracket y \rrbracket \langle\$\rangle\, \mathcal{C}\llbracket y \rrbracket)
\end{aligned}
$$

Proof. First, note that for any simplified generator, g, if $\mathcal{C}\llbracket g \rrbracket$ generates a string s, for any other string t $\mathcal{P}\llbracket g \rrbracket\ (s \cdot t) =$ Just (a, t) for some value a. This can be shown by induction on the structure of g.
Now, assume $\mathcal{C}\llbracket \mathrm{x} \rrbracket$ generates a string s, and $\mathcal{C}\llbracket \mathrm{y} \rrbracket$ generates t. This means that $(\cdot) \langle\$\rangle\, \mathcal{C}\llbracket \mathrm{x} \rrbracket \langle * \rangle\, \mathcal{C}\llbracket \mathrm{y} \rrbracket$ generates $s \cdot t$.
By the above fact, it is simple to show that both sides of the above equation simplify to Just $((a, b), \varepsilon)$ for some values a and b that depend on the particular interpretations of x and y.
Since this is true for any s and t that the choice distributions generate, the desired fact holds. □

Theorem 3.4. *Every simplified free generator can be factored coherently into a parser and a distribution over choice sequences. In other words, for all simplified free generators $g \neq Void$,*

$$
\mathcal{P}\llbracket g \rrbracket \langle\$\rangle\, \mathcal{C}\llbracket g \rrbracket \equiv (\lambda x \to Just\ (x, \varepsilon)) \langle\$\rangle\, \mathcal{G}\llbracket g \rrbracket.
$$

Proof. We proceed by induction on the structure of g.

Case $g =$ Pure a.

$$
\begin{aligned}
\mathcal{P}\llbracket \mathrm{Pure\ a} \rrbracket \langle\$\rangle\, \mathcal{C}\llbracket \mathrm{Pure\ a} \rrbracket & \equiv \mathrm{pure\ (Just\ (a}, \varepsilon)) && \text{(by defn)} \\
& \equiv (\lambda x \to \mathrm{Just}\ (x, \varepsilon)) \langle\$\rangle\, \mathcal{G}\llbracket \mathrm{Pure\ a} \rrbracket && \text{(by defn)}
\end{aligned}
$$

Case $g =$ Pair x y.

$$
\begin{aligned}
\mathcal{P}\llbracket \mathrm{Pair\ x\ y} \rrbracket \langle\$\rangle\, \mathcal{C}\llbracket \mathrm{Pair\ x\ y} \rrbracket & \equiv ((\lambda x\ y \to (x, y)) \langle\$\rangle\, \mathcal{P}\llbracket \mathrm{x} \rrbracket \langle * \rangle\, \mathcal{P}\llbracket \mathrm{y} \rrbracket) \langle\$\rangle ((\cdot) \langle\$\rangle\, \mathcal{C}\llbracket \mathrm{x} \rrbracket \langle * \rangle\, \mathcal{C}\llbracket \mathrm{y} \rrbracket) && \text{(by defn)} \\
& \equiv (\lambda a_\perp\ b_\perp \to \textbf{case}\ (a_\perp, b_\perp)\ \textbf{of} \\
& \qquad (\mathrm{Just}\ (a, \_), \mathrm{Just}\ (b, \_)) \to \mathrm{Just}\ ((a, b), \varepsilon) \\
& \qquad \_ \to \mathrm{Nothing}) \\
& \quad \langle\$\rangle (\mathcal{P}\llbracket \mathrm{x} \rrbracket \langle\$\rangle\, \mathcal{C}\llbracket \mathrm{x} \rrbracket) \langle * \rangle (\mathcal{P}\llbracket \mathrm{y} \rrbracket \langle\$\rangle\, \mathcal{C}\llbracket \mathrm{y} \rrbracket) && \text{(by Lemma B.1)} \\
& \equiv (\lambda a_\perp\ b_\perp \to \textbf{case}\ (a_\perp, b_\perp)\ \textbf{of} \\
& \qquad (\mathrm{Just}\ (a, \_), \mathrm{Just}\ (b, \_)) \to \mathrm{Just}\ ((a, b), \varepsilon) \\
& \qquad \_ \to \mathrm{Nothing}) \\
& \quad \langle\$\rangle ((\lambda x \to \mathrm{Just}\ (x, \varepsilon)) \langle\$\rangle\, \mathcal{G}\llbracket \mathrm{x} \rrbracket) \langle * \rangle ((\lambda x \to \mathrm{Just}\ (x, \varepsilon)) \langle\$\rangle\, \mathcal{G}\llbracket \mathrm{y} \rrbracket) && \text{(by IH)} \\
& \equiv (\lambda x \to \mathrm{Just}\ (x, \varepsilon)) \langle\$\rangle ((\lambda x\ y \to (x, y)) \langle\$\rangle\, \mathcal{G}\llbracket \mathrm{x} \rrbracket \langle * \rangle\, \mathcal{G}\llbracket \mathrm{y} \rrbracket) && \text{(by app. properties)} \\
& \equiv (\lambda x \to \mathrm{Just}\ (x, \varepsilon)) \langle\$\rangle\, \mathcal{G}\llbracket \mathrm{Pair\ x\ y} \rrbracket && \text{(by defn)}
\end{aligned}
$$

Case $g =$ Map f x.

$$
\begin{aligned}
\mathcal{P}\llbracket \mathrm{Map\ f\ x} \rrbracket \langle\$\rangle\, \mathcal{C}\llbracket \mathrm{Map\ f\ x} \rrbracket & \equiv (\mathrm{f} \langle\$\rangle\, \mathcal{P}\llbracket \mathrm{x} \rrbracket) \langle\$\rangle\, \mathcal{C}\llbracket x \rrbracket && \text{(by defn)} \\
& \equiv \mathrm{f} \langle\$\rangle (\mathcal{P}\llbracket \mathrm{x} \rrbracket \langle\$\rangle\, \mathcal{C}\llbracket x \rrbracket) && \text{(by functor properties)} \\
& \equiv \mathrm{f} \langle\$\rangle ((\lambda x \to \mathrm{Just}\ (x, \varepsilon)) \langle\$\rangle\, \mathcal{G}\llbracket \mathrm{x} \rrbracket) && \text{(by IH)} \\
& \equiv (\lambda x \to \mathrm{Just}\ (x, \varepsilon)) \langle\$\rangle (\mathrm{f} \langle\$\rangle\, \mathcal{G}\llbracket \mathrm{x} \rrbracket) && \text{(by functor properties)} \\
& \equiv (\lambda x \to \mathrm{Just}\ (x, \varepsilon)) \langle\$\rangle\, \mathcal{G}\llbracket \mathrm{Map\ f\ x} \rrbracket && \text{(by defn)}
\end{aligned}
$$

Case $g = \text{Select xs}$.

$$
\begin{aligned}
\mathcal{P}[\![\text{Select xs}]\!] \mathbin{\langle\$\rangle} \mathcal{C}[\![\text{Select xs}]\!] \equiv{} & (\text{choice}\ (\text{map}\ (\lambda(c, \text{x}) \rightarrow (c, \mathcal{P}[\![\text{x}]\!]))\ \text{xs})) \\
& \langle\$\rangle\ \text{oneof}\ (\text{map}\ (\lambda(c, \text{x}) \rightarrow (c\cdot)\ \langle\$\rangle\ \mathcal{C}[\![\text{x}]\!])\ \text{xs}) && \text{(by defn)} \\
\equiv{} & \text{oneof}\ (\text{map}\ (\lambda(c, \text{x}) \rightarrow \\
& \quad (\text{choice}\ (\text{map}\ (\lambda(c, \text{x}) \rightarrow (c, \mathcal{P}[\![\text{x}]\!]))\ \text{xs})) \circ (c\cdot)\ \langle\$\rangle\ \mathcal{C}[\![\text{x}]\!] \\
&)\ \text{xs}) && \text{(by generator properties)} \\
\equiv{} & \text{oneof}\ (\text{map}\ (\lambda(\_, \text{x}) \rightarrow \mathcal{P}[\![\text{x}]\!]\ \langle\$\rangle\ \mathcal{C}[\![\text{x}]\!])\ \text{xs}) && \text{(by parser properties)} \\
\equiv{} & \text{oneof}\ (\text{map}\ (\lambda(\_, \text{x}) \rightarrow (\lambda x \rightarrow \text{Just}\ (x, \varepsilon))\ \langle\$\rangle\ \mathcal{G}[\![\text{x}]\!])\ \text{xs}) && \text{(by IH)} \\
\equiv{} & (\lambda x \rightarrow \text{Just}\ (x, \varepsilon))\ \langle\$\rangle\ \text{oneof}\ (\text{map}\ (\lambda(\_, \text{x}) \rightarrow \mathcal{G}[\![\text{x}]\!])\ \text{xs}) && \text{(by generator properties)} \\
\equiv{} & (\lambda x \rightarrow \text{Just}\ (x, \varepsilon))\ \langle\$\rangle\ \mathcal{G}[\![\text{Select xs}]\!] && \text{(by defn)}
\end{aligned}
$$

Thus, generators can be coherently factored into a parser and a distribution. □

C Proof of Theorem 4.2

Theorem 4.2. *For all simplified free generators g and characters c,*

$$\delta_c \mathcal{L}[\![g]\!] = \mathcal{L}[\![\delta_c g]\!].$$

Proof. We again proceed by induction on the structure of g.

Case $g = \text{Void}$. $\varnothing = \varnothing$.

Case $g = \text{Pure a}$. $\varnothing = \varnothing$.

Case $g = \text{Pair x y}$.

$$\begin{aligned}
\delta_c \mathcal{L}[\![\text{Pair x y}]\!] &= \delta_c(\mathcal{L}[\![\text{x}]\!] \cdot \mathcal{L}[\![\text{y}]\!]) && \text{(by defn)} \\
&= \delta_c(\mathcal{L}[\![\text{x}]\!]) \cdot \mathcal{L}[\![\text{y}]\!] + \nu \mathcal{L}[\![\text{x}]\!] \cdot \delta_c \mathcal{L}[\![\text{y}]\!] && \text{(by defn)} \\
&= \delta_c(\mathcal{L}[\![\text{x}]\!]) \cdot \mathcal{L}[\![\text{y}]\!] && \text{(by Lemma 4.1)} \\
&= \mathcal{L}[\![\delta_c \text{x}]\!] \cdot \mathcal{L}[\![\text{y}]\!] && \text{(by IH)} \\
&= \mathcal{L}[\![\text{Pair } (\delta_c \text{x}) \text{ y}]\!] && \text{(by defn)} \\
&= \mathcal{L}[\![\delta_c \text{Pair x y}]\!] && \text{(by defn)}
\end{aligned}$$

Case $g = \text{Map f x}$.

$$\begin{aligned}
\delta_c \mathcal{L}[\![\text{Map f x}]\!] &= \delta_c \mathcal{L}[\![\text{x}]\!] && \text{(by defn)} \\
&= \mathcal{L}[\![\delta_c \text{x}]\!] && \text{(by IH)} \\
&= \mathcal{L}[\![\text{Map f } (\delta_c \text{x})]\!] && \text{(by defn)} \\
&= \mathcal{L}[\![\delta_c (\text{Map f x})]\!] && \text{(by defn*)}
\end{aligned}$$

*Note that the last step follows because Map f x is assumed to be simplified, so $x \neq$ Pure a. This means that f ⟨\$⟩ x is equivalent to Map f x.

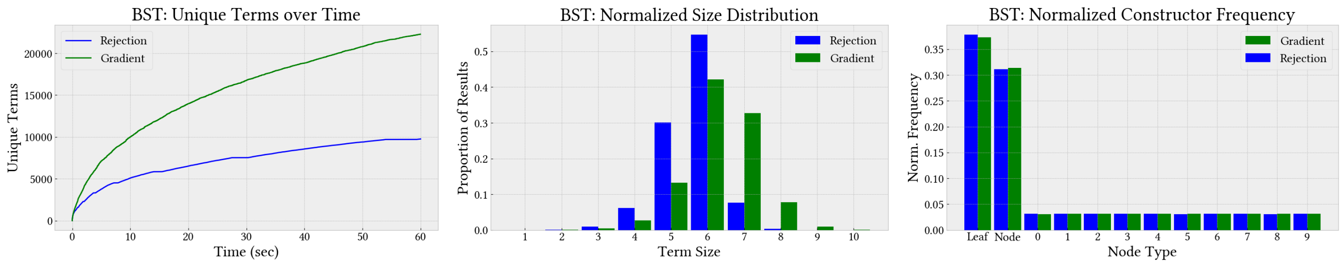
Case $g = \text{Select xs}$. If there is no pair (c, x) in xs, then $\varnothing = \varnothing$. Otherwise,

$$\begin{aligned}
\delta_c \mathcal{L}[\![\text{Select xs}]\!] &= \delta_c \{c \cdot s \mid s \in \mathcal{L}[\![\text{x}]\!]\} && \text{(by defn)} \\
&= \mathcal{L}[\![\text{x}]\!] && \text{(by defn)} \\
&= \mathcal{L}[\![\delta_c (\text{Select xs})]\!] && \text{(by defn)}
\end{aligned}$$

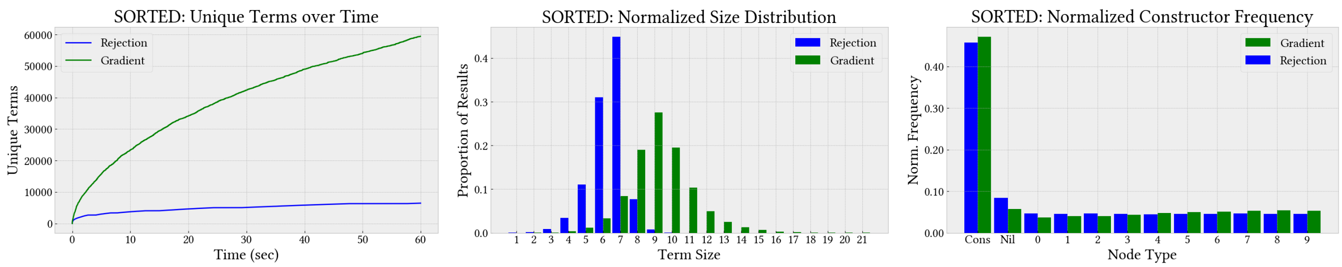
Thus we have shown that the symbolic derivative of free generators is compatible with the derivative of the generator's language. □

There is another proof of this theorem, suggested by Alexandra Silva, which uses the fact that 2^{Σ^*} is the final coalgebra, along with the observation that FGen has a $2 \times (-)^{\Sigma}$ coalgebraic structure. This approach is certainly more elegant, but it abstracts away some helpful operational intuition.

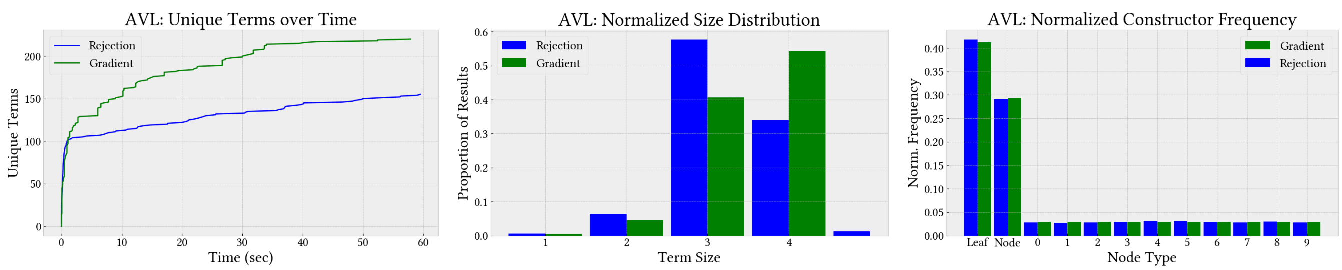
D Full Experimental Results



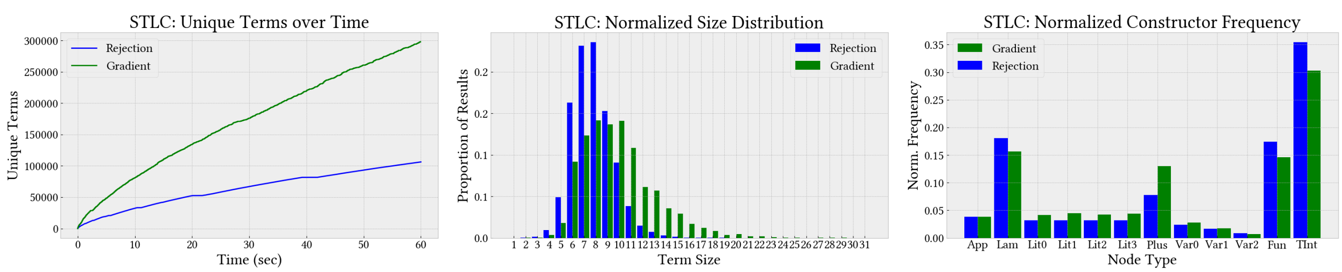
BST Charts



SORTED Charts



AVL Charts



STLC Charts