# Generators as Parsers

A View of Randomness

HARRISON GOLDSTEIN, University of Pennsylvania, Philadelphia, PA, USA
BENJAMIN C. PIERCE, University of Pennsylvania, Philadelphia, PA, USA

"A generator is a parser of randomness." This perspective on generators for random data structures is established folklore in the programming languages community, but it has never been formalized, nor have its consequences been deeply explored.

We build on the idea of *freer monads* from functional programming to develop *free generators*, which unify parsing and generation using a common structure that makes the relationship between the two concepts precise. Free generators lead naturally to a proof monadic generator can be factored into a parser plus a distribution over choice sequences. Free generators also support a notion of *derivative*, analogous to familiar Brzozowski derivatives of formal languages, allowing analysis tools to "preview" the effect of a particular generator choice. This, gives rise to a novel algorithm for generating data structures satisfying user-specified preconditions.

## 1 INTRODUCTION

"A generator is a parser of randomness. . . " It's one of those observations that's totally puzzling right up to the moment it becomes totally obvious: a random generator—such as might be found in a property-based testing tool like QuickCheck [Claessen and Hughes 2000]—is a transformer from a series of random choices into a data structure, just as a parser transforms a series of characters into a data structure.

Although this connection may be obvious once it is pointed out, few actually think of generators this way. Indeed, to our knowledge the framing of random generators as parsers has never been explored formally. But this is a shame! The relationship between these fundamental concepts deserves a deeper look.

A generator is a program that builds a data structure by making a sequence of random choices—those choices are the key. A "traditional" generator makes decisions using a stored source of randomness (e.g., a seed) that it consults and updates whenever it must make a choice. Equivalently, if we like, we can pre-compute a list of choices and pass it in to the generator, which gradually walks down the list whenever it needs to make random decisions. In this mode of operation, the generator is effectively *parsing* the sequence of choices into a data structure!

To connect generators and parsers, we introduce a data structure called a *free generator* that can be interpreted as *either* a generator or as a parser. Free generators have a rich theory; in particular, we can use them to prove that a large subset of generator programs can be factored into a parser and a distribution over sequences of choices.

Besides clarifying folklore, free generators admit transformations that cannot be implemented for standard generators and parsers. A particularly exciting one 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 likely it is to lead us to useful values.

We use derivatives of free generators to tackle a well-known problem—we call it the *valid generation problem*. The challenge is to generate a large number of random values that satisfy some validity condition. This problem comes up often in property-based testing, where the validity condition is the precondition of some functional specification. Since generator derivatives give a

Authors' addresses: 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.

way of previewing the effects of a particular choice, we can use *gradients* (derivatives with respect to a vector of choices) to preview all possible choices and pick a promising one. This leads us to an elegant algorithm for turning a naïve free generator into one that only generates valid values.

In §2 below, we introduce the ideas behind free generators and the operations that can be defined on them. We then present our main contributions:

- We formalize the folklore analogy between parsers and generators using *free generators*, a novel class of structures that make choices explicit and support syntactic transformations (§3). We use free generators to prove that any *monadic* generator can factored into a parser and a distribution over strings.
- We exploit free generators to to transport an idea from formal languages—the *Brzozowski derivative*—to the context of generators (§4).
- To illustrate the potential applications of these formal results, we present an algorithm that uses derivatives to turn a naïve generator into one that produces only values satisfying a Boolean precondition (§5). Our algorithm performs well on simple benchmarks, in most cases producing more than twice as many valid values as a naïve "rejection sampling" generator in the same amount of time (§6).

We conclude with related and future work (§8 and §9).

## 2 THE HIGH-LEVEL STORY

Let's take a walk in the forest before we dissect the trees.

**Generators and Parsers.** To start, we can clarify the kinds of generators and parsers that we want to focus on. Consider the following programs:

```
genTree h =
  if h = 0 then
    return Leaf
  else
    c ← flip 0.75
    if c == Tails then return Leaf
    if c == Heads then
      x ← genInt ()
      l ← genTree (h − 1)
      r ← genTree (h − 1)
      return Node l x r
```

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

The generator program genTree produces random binary trees of integers like

Node Leaf 5 Leaf  and  Node Leaf 5 (Node Leaf 8 Leaf),

up to a given height $h$, guided by a series of weighted random coin flips. Each time the program is run it produces a random tree, and the program as a whole denotes a distribution over trees. Generators like these can produce arbitrary discrete distributions of values.

The parser program parseTree parses a string into a tree. The parser turns

`n5ll` into Node Leaf 5 Leaf  and  `n5ln8ll` into Node Leaf 5 (Node Leaf 8 Leaf).

It consumes the input string character by character with consume and uses the characters to decide what to do next. This program is deterministic, but its execution (and thus the final tree it produces) is guided by a string of characters it is passed as input. Parsers like these can parse arbitrary computable languages.

Obviously, there is considerable similarity between genTree and parseTree. The programs' structure is almost identical, and both programs produce the same set of values. The main difference lies in how program branches execute: in genTree branches are taken at random, whereas in parseTree they are controlled by the input string. This is the key observation that links generators and parsers!

To make this more concrete, let us imagine how to recover the distribution of genTree $h$ using parseTree $h$. We can do this by choosing a string at random and then parsing it—if we choose strings with the correct distribution, then the result of parsing those strings into values will be the same as if we had run genTree in the first place.

In the case of the programs above, the distribution over strings that we parse with parseTree should satisfy the weighting of the flip ped coin in genTree. Since the coin in genTree is weighted 75% towards Heads, n should appear three times more often than l. The coin weights in genInt will yield further constraints. We can capture these relative weights in an informal equation:

$$\text{genTree } h = \text{parseTree } h + \{\mathtt{l} \mapsto 1, \mathtt{n} \mapsto 3, \dots\}$$

In English, this says that a generator is just a parser of a source of random strings with the correct weights.

**Free Generators.** With the intuition nailed down, we need a formal way to connect parsing and generation. First, we unify random generation with parsing by abstracting both into a single data structure, and then we show that a structure of this form can be viewed equivalently as a generator or as a parser and a source of randomness.

The unifying data structure we introduce is called a free generator.[1] Free generators are syntactic structures that can be *interpreted* as programs that either generate or parse. Here is an example of a free generator:

```
fgenTree h =
    if h = 0 then
        return Leaf
    else
        c ← pick [(1, l, return Heads), (3, n, return Tails)]
        if c == Tails then return Leaf
        if c == Heads then
            x ← fgenInt ()
            l ← fgenTree (h − 1)
            r ← fgenTree (h − 1)
            return Node l x r
```

This program has a very similar structure to that of genTree and parseTree, but its meaning is very different. For present purposes, think of fgenTree as having *no interpretation at all.* Instead, the program constructs in fgenTree are purely symtactic, and the result of running fgenTree 10 is simply the abstract syntax tree (AST) of the program as-written. This is not exactly how free generators work (§3 refines the notion considerably), but it is the right intuition.

The syntactic, uninterpreted nature of free generaotrs means that they can simultaneously represent generation, parsing, and more. In §3 we give several ways to interpret free generators.

[1]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.

We write $\mathcal{G}[\![\cdot]\!]$ for the random generator interpretation of a free generator and $\mathcal{P}[\![\cdot]\!]$ for the parser interpretation. In other words,

$$\mathcal{G}[\![\text{fgenTree } 10]\!] \approx \text{genTree } 10 \quad \text{and } \mathcal{P}[\![\text{fgenTree } 10]\!] \approx \text{parseTree } 10.$$

The interpretation functions walk the AST and recover the behavior of the generator and parser programs.

These two interpretations can be related by our previous observation about parsing and generation. This requires one final interpretation function, $\mathcal{R}[\![\cdot]\!]$, the randomness interpretation of the free generator. Intuitively, the randomness interpretation produces the set of sequences of choices that the random generator interpretation can make, or equivalently the set of sequences that the parser interpretation can parse.

The randomness interpretation is used in Theorem 3.1 to connect parsing and generation. The theorem says that for any free generator $g$,

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

where $\langle\$\rangle$ is a "mapping" operation that applies a function to samples from a distribution. Since most QuickCheck generators can also be written as free generators, another way to read this theorem is that such generators can be factored into two pieces: a distribution over choice sequences (given by $\mathcal{R}[\![\cdot]\!]$), and a parser of those sequences (given by $\mathcal{P}[\![\cdot]\!]$). This precisely formalizes the intuition that "A generator is a parser of randomness."

**Derivatives of Free Generators.** But wait, there's more! Since a free generator defines a parser, it also defines a formal language: we write $\mathcal{L}[\![\cdot]\!]$ for this language interpretation of a free generator. 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. In particular, formal languages come with a notion of *derivative*, due to Brzozowski [Brzozowski 1964]. Given a language $L$, the Brzozowski derivative of $L$ is

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

That is, the derivative of $L$ with respect to $c$ is all the strings in $L$ that start with $c$, with the first $c$ removed.

We can visualize this derivative process in the context of a parser, by considering the derivative of a parser with respect to $c$ to be whatever parser remains after $c$ has been parsed. Each consecutive derivative fixes certain choices within the parser, simplifying the program:

| parseTree 10 = | $\delta_{\mathsf{n}}$(parseTree 10) ≈ | $\delta_5\delta_{\mathsf{n}}$(parseTree 10) ≈ |
|---|---|---|
|   $c$ ← consume() | | |
|   **if** $c$ == l **then** | | |
|     **return** Leaf | | |
|   **if** $c$ == n **then** | | |
|     $x$ ← parseInt() | $x$ ← parseInt() | |
|     $l$ ← parseTree 9 | $l$ ← parseTree 9 | $l$ ← parseTree 9 |
|     $r$ ← parseTree 9 | $r$ ← parseTree 9 | $r$ ← parseTree 9 |
|     **return** Node $l$ $x$ $r$ | **return** Node $l$ $x$ $r$ | **return** Node $l$ 5 $r$ |
|   **else fail** | | |

The first derivative fixes the character n, ensuring that the parser will produce a Node. Then the next derivative fixes the character 5, which in turn fixes the value 5 in the final Node.

Free generators have a closely related notion of derivative, as illustrated by an almost identical set of transformations:

```
fgenTree 10 =                   δn(fgenTree 10) ≈               δ5δn(fgenTree 10) ≈
  c ← pick [...]
  if c == Heads then
    return Leaf
  if c == Tails then
    x ← fgenInt()                 x ← fgenInt()
    l ← fgenTree 9                l ← fgenTree 9                  l ← fgenTree 9
    r ← fgenTree 9                r ← fgenTree 9                  r ← fgenTree 9
    return Node l x r             return Node l x r               return Node l 5 r
  else fail
```

But there is a critical difference between this series of derivatives and the ones for parseTree. Whereas the parser derivatives were merely illustrative, these are readily computable! Just as one can compute the derivative of a regular expression or a context-free grammar, one can compute the derivative of a free generator via an extremely simple and efficient syntactic transformation.

In §4 we define a procedure for computing the derivative of a free generator and prove it correct, in the sense 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 $g$. (See Theorem 4.2.)

**Putting Free Generators to Work.** The derivative of a free generator is intuitively *the generator that remains after a particular choice*. This gives us a way of "previewing" the effect of making a choice by looking at the generator after fixing that choice.

In §5 and §6 we present and evaluate an algorithm called Choice Gradient Sampling that uses free generators to address the *valid generation problem*. Given a validity predicate on a data structure, the goal is to generate as many unique, valid structures as possible in a given amount of time. Given a simple free generator, our algorithm uses derivatives to evaluate choices and search for valid values.

We evaluate our algorithm on four small benchmarks, all standard in the property-based testing literature. We compare our algorithm to rejection sampling—sampling from a naïve generator and discarding invalid results—as a simple but useful baseline for understanding how well or algorithm performs. Our algorithm 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.

## 3 FREE GENERATORS

We now turn to developing the theory of free generators, beginning with some background on monadic abstractions for parsing and random generation.

**Background: Monadic Parsers and Generators.** In §2 we represented generators and parsers with pseudo-code. Here we flesh out the details. We present all definitions as Haskell programs, both for the sake of concreteness and also because Haskell's abstraction features (e.g., typeclasses) allow us to focus on the key concepts. Haskell is a lazy functional language, but, as we focus our attention on finite programs, our results are also applicable to eager functional languages. Indeed, with appropriate domain knowledge, it should also be possible to translate these ideas to idiomatic constructs in popular imperative languages [Petříček 2009].

We represent both generators and parsers using *monads* [Moggi 1991]. A monad is a type constructor (e.g., List, Maybe, etc.) M with two operations:

```
return :: a → M a
```

and

```
(>>=) :: M a → (a → M b) → M b
```

(pronounced "bind"). Conceptually, return is the simplest way to put some value into the monad, and bind gives a way to sequence operations that produce monadic values.

All of this is a bit abstract, so let us examine two examples. We can use these operations to define genTree like we would in QuickCheck [Claessen and Hughes 2000] and parseTree like we would using combinators inspired by libraries like Parsec [Leijen and Meijer 2001]:

```
genTree :: Int → Gen Tree
genTree 0 = return Leaf
genTree h = do
  c ← frequency [(1, Tails), (3, Heads)]
  case c of
    Tails → return Leaf
    Heads → do
      x ← genInt
      l ← genTree (h − 1)
      r ← genTree (h − 1)
      return (Node l x r)
```

```
parseTree :: Int → Parser Tree
parseTree 0 = return Leaf
parseTree h = do
  c ← consume
  case c of
    'l' → return Leaf
    'n' → do
      x ← parseInt
      l ← parseTree (h − 1)
      r ← parseTree (h − 1)
      return (Node l x r)
    _ → fail
```

In the first program, genTree, we use monadic operations (along with a combinator for weighted random choice, called frequency) to generate a random tree of integers. The expression return Leaf is a degenerate generator that always produces the value Leaf—this is what we mean by the "simplest way to put a value into the monad." Rather than use bind explicitly, we use Haskell's **do**-notation; the expression

```
do
  a ← x
  f a
```

is simply syntactic sugar for x >>= f. In the context of the Gen type, this operation samples from x and then passes the sampled result to f for further processing—this is what we mean by "sequencing operations."

We can see these same combinators (used with a different monad) in parseTree. There, return a means "parse nothing and produce a", and x >>= f means "run the parser x to get a value a and then run the parser f a". Under the hood,

```
type Parser a = String → Maybe (a, String)
```

so a Parser 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. The consume function pulls the first character off of the string for inspection.

Monadic parsers and generators are maximally expressive in their respective areas. Monadic parsers can parse arbitrary computable languages, subsuming more restricted parser descriptions like context-free grammars and regular expressions. Likewise, monadic generators can generate values satisfying arbitrary computable constraints (e.g., presents a generator for well-typed System F terms), subsuming less powerful representations like probabilistic context-free grammars.

**Representing Free Generators.** With the monad interface in mind, we can now give the formal definition of a *free generator*.[2]

*Type Definition.* The actual type of free generators is based on a structure called a *freer monad* [Kiselyov and Ishii 2015]:

```
data Freer f a where
  Return :: a → Freer f a
  Bind :: f a → (a → Freer f b) → Freer f b
```

This type looks complicated, but it is essentially just a representation a monadic syntax tree. The constructors of Freer align almost exactly with the monadic operations return and (>>=). The only difference is the type constructor f that appears throughout the type.

We can think of f a as a type of *specialized operations* returning a. In the cases of generation and parsing, those operations were flip and consume respectively. These operations are important to the way programs are written in a particular monad, but not part of the monad interface itself.

One might expect the type of Bind to be

```
Bind :: Freer f a → (a → Freer f b) → Freer f b
```

(with Freer f a as the first argument), but it turns out that keeping a bare operation at the "front" of the Bind is incredibly convenient. We will see in a moment that syntax trees in a freer monad are automatically normalized by construction.

For free generators specifically, the specialized operation we need is called pick (which we introduced in §2). Intuitively, pick subsumes both flip and consume. We define the pick operation with data type (since free generators are syntactic objects) along with FGen, the type of free generators:

```
data Pick a where
  Pick :: [(Weight, Choice, Freer Pick a)] → Pick a

type FGen a = Freer Pick a
```

The Pick operation takes a list of triples. The first element of type Weight represents the weight given to a particular choice; weights are represented by signed integers for efficiency, but for theoretical purposes we treat them as strictly positive. The type Choice can theoretically be any type that admits equality, but for the purposes of this paper we take choices to be single characters. This makes the analogy with parsing clearer. Finally, Freer Pick a is actually just the type FGen a! The third element in the triple is a *nested* free generator that is run iff a specific choice is made.

Together the elements of these triples represent both kinds of choices that we have seen so far, subsuming both the weighted random choices of generators and the input-directed choices of parsers. Depending on our needs, we can interpret Pick as either kind of choice. In the rest of the

[2]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 further here.

paper, we sometimes speak of free generators "making" or "parsing" a choice, but remember that this really just an analogy—a free generator is simply syntax, and the interpretation of that syntax comes later.

By defining FGen as Freer Pick, we are really saying that "FGen is a monad with operation Pick."

*Recovering Monadic Syntax.* The FGen structure is exactly what we need to unify monadic generation and parsing, but writing down instances with Return and Bind explicitly is rather clunky. Luckily this is easy to fix, because FGen, like Gen and Parser, is a monad! We can define return and (>>=) for FGen as follows, allowing us to use **do**-notation to write free generators:

```
return :: a → FGen a
return = Return

(>>=) :: FGen a → (a → FGen b) → FGen b
Return a >>= f = f a
Bind p g >>= f = Bind p (λa → g a >>= f)
```

The return operator maps directly to a Return syntax node, but there is a bit more going on in the definition of (>>=). Specifically, (>>=) normalizes the structure of the computation, ensuring that there is always an operation at the "front." The advantage of this is that it is always $O(1)$ to check if a free generator has a choice to make. There is no need to dig through the syntax tree to determine what the next step is.

Another convenient way to manipulate free generators is via an operation called called "fmap," written f ⟨$⟩ x. Like return and (>>=), (⟨$⟩) is a syntactic transformation, but intuitively f ⟨$⟩ x to means "apply the function f to the result of generating/parsing with x". We define it as:

```
(⟨$⟩) :: (a → b) → FGen a → FGen b
f ⟨$⟩ Return a = Return (f a)
g ⟨$⟩ (Bind p f) = Bind p ((g ⟨$⟩) . f)
```

(Note that all monads have an analogous operation; this will come in handy later.)

*Representing Failure.* For reasons that will become clear in §4, it is useful to be able to represent a free generator that can "fail." We call the always-failing free generator *void*, and define it like this:

```
void :: FGen a
void = Bind (Pick []) Return
```

Any reasonable interpretation of this free generator must fail (by either diverging or signaling failure); with no choices in the Pick list, there is no way to get a value of type a to pass to the second argument of Bind.

Additionally, the use of Return as the second argument is irrelevant, since all free generators with no choices to make get equally stuck. This suggests that there is an easy way to check if a free generator is equivalent to void—just match on an empty list of choices! In Haskell this is easy to do with a pattern synonym:

```
pattern Void :: FGen a
pattern Void ← Bind (Pick []) _
```

This declaration means that pattern-matching on Void is equivalent to matching a Bind with no choices to make and ignoring the second argument. It is simple to define a function that uses this new pattern to check if a particular free generator is void:

```
isVoid  :: FGen a → Bool
isVoid  Void  = True
isVoid  _     = False
```

While void is useful as an error case for algorithms that build free generators, it would be incorrect for a user to use void in a hand-written free generator. To enforce this constraint, we define a wrapper around Pick (called *pick*) that does a few coherence checks to make sure that the generator is constructed properly:

```
pick :: [(Weight, Choice, FGen a)] → FGen a
pick xs =
  case filter (λ (w, _, x) → not (isVoid x)) xs of
    ys | hasDuplicates (map fst ys) → undefined
    [] → undefined
    ys → Bind (Pick ys) Return
```

This function is partial: it yields **undefined** if the list passed to pick is invalid.[3] The first line filters out any choices that are equivalent to void, since making those choices would lead to failure. The second line checks that the user has not duplicated any of the choice labels; this would introduce a nondeterministic choice that would complicate the interpretation considerably. (We discuss this further in §7.) Finally, the third line ensures that the generator we construct is not itself void. In practice, these checks ensure that the various interpretations of free generators presented in the remainder of this section work as intended.

*Examples.* Now that we've seen the building blocks of free generators, let's look at a couple of concrete examples. We saw a version of fgenTree in §2 that was written out explicitly as an AST. Here's how looks in our framework:

```
fgenTree :: Int → FGen Tree
fgenTree 0 = return Leaf
fgenTree h = do
  c ← pick [(1, l, return Tails), (3, n, return Heads)]
  case c of
    Tails → return Leaf
    Heads → do
      x ← fgenInt
      l ← fgenTree (h − 1)
      r ← fgenTree (h − 1)
      return (Node l x r)
```

Recall that fgenTree is designed to subsume both genTree and parseTree. The height parameter $h$ controls the depth of the trees that the interpretations can produce.

Here is another example of a free generator that produces random terms of the simply-typed lambda-calculus:

```
fgenExpr :: Int → FGen Expr
fgenExpr 0 = pick [ (1, i, Lit ⟨$⟩ fgenInt), (1, v, Var ⟨$⟩ fgenVar) ]
fgenExpr h =
```

[3]This is analogous to raising an exception in a conventional imperative language; however, because Haskell uses a lazy evaluation strategy, no exception will be raised if the result of pick is never needed.

```
pick [ (1, i, Lit ⟨$⟩ fgenInt ),
       (1, p, do {e1 ← fgenExpr (h - 1); e2 ← fgenExpr (h - 1); return (Plus e1 e2 )}),
       (1, l, do {t ← fgenType e ← fgenExpr (h - 1) return (Lam t e )}),
       (1, a, do {e1 ← fgenExpr (h - 1) e2 ← fgenExpr (h - 1) return (App e1 e2 )}),
       (1, v, Var ⟨$⟩ fgenVar) ]
```

Structurally this is similar to the previous generator; it just has more cases and more choices. One stylistic difference between fgenExpr and fgenTree is that fgenExpr does not pick a coin and use it to decide what should be generated next; instead, it picks among a list of free generators directly. These styles of writing free generators are equivalent.

This verson of the lambda calculus uses de Bruijn indices for variables and has integers and functions as values. This is a useful example because, while syntactically valid terms in this language are easy to generate (as we just did), it is more difficult to generate only well-typed terms. We will return to this problem in §6.

**Interpreting Free Generators.** A free generator does not do anything on its own—it is just a data structure. To actually use these structures, we next define the interpretation functions that we mentioned in §2 and prove a theorem linking those interpretations together.

*Free Generators as Generators of Values.* The first and most natural way to interpret a free generator is as a QuickCheck generator—that is, as a distribution over data structures. Plain QuickCheck generators ignore failure cases like void (they throw an error if there are no valid choices to make), but to make things a bit more explicit for our theory we use a modified generator monad: $\text{Gen}_\bot$.

We define the *random generator interpretation* of a free generator to be:

```
𝒢⟦·⟧ :: FGen a → Gen⊥ a
𝒢⟦Void⟧             = ⊥
𝒢⟦Return v⟧         = return v
𝒢⟦Bind (Pick xs) f⟧ = do
  x ← frequency (map (λ (w, _, x) → (w, return x)) xs)
  a ← 𝒢⟦x⟧
  𝒢⟦f a⟧
```

Note that the operations on the right-hand side of this definition do *not* build a free generator; they are $\text{Gen}_\bot$ operations. This translation turns the syntactic form Return v into the semantic action "always generate the value v" and the syntactic form Bind into the generator sequencing operation we saw previously.

Note that $\mathcal{G}[\![\text{fgenTree } h]\!]$ has the same distribution as genTree $h$.

*Free Generators as Parsers of Random Sequences.* The *parser interpretation* of a free generator views it as a parser of sequences of choices. The translation looks like this:

```
𝒫⟦·⟧ :: FGen a → Parser a
𝒫⟦Void⟧             = λs → Nothing
𝒫⟦Return a⟧         = return a
𝒫⟦Bind (Pick xs) f⟧ = do
  c ← consume
  x ← case find ((== c) . snd) xs of
    Just (_, _, x) → return x
    Nothing → fail
```

```
  a ← 𝒫⟦x⟧
  𝒫⟦f a⟧
```

This time the **do**-notation on the right hand side is interpreted using the Parser monad (as before, defined as String → Maybe (a, String)). In the case for Bind, the parser consumes a character and attempts to make the corresponding choice from the list provided by Pick. If it succeeds, it runs the corresponding sub-parser and continues with f. If it fails, the whole parser fails.

Note that $\mathcal{P}[\![\text{fgenTree } h]\!]$ has the same parsing behavior as parseTree $h$.

*Free Generators as Generators of Random Sequences.* Our final interpretation of free generators represents the distribution with which the generator makes choices and ignores how those choices are used to produce values. That is, it captures exactly the parts of the structure that the parser interpretation discards. We define the *randomness interpretation* of a free generator to be:

```
ℛ⟦·⟧ :: FGen a → Gen⊥ String
ℛ⟦Void⟧               = ⊥
ℛ⟦Return a⟧           = return ε
ℛ⟦Bind (Pick xs) f⟧ = do
  (c, x) ← frequency (map (λ (w, c, x) → (w, return (c, x))) xs)
  s ← ℛ⟦x >>= f⟧
  return (c : s)
```

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 random generator interpretation can make and the parser interpretation can parse. Again, we use $\text{Gen}_\perp$ and frequency to capture randomness and potential failure.

*Factoring Generators.* These different interpretations of free generators are closely related to one another; in particular, we can reconstruct $\mathcal{G}[\![\cdot]\!]$ from $\mathcal{P}[\![\cdot]\!]$ and $\mathcal{R}[\![\cdot]\!]$. That is, a free generator's random generator interpretation can be factored into a distribution over choice sequences plus 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 $g_1 \equiv g_2$, iff 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 3.1 (Factoring). *Every free generator can be factored into a parser and a distribution over choice sequences that is equivalent to its interpretation as a generator. In other words, for all free generators g,*

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

Proof sketch. By induction on the structure of $g$; see Appendix A for the full proof. □

Corollary 3.2. *Any monadic generator, $\gamma$, written using return, (>>=), and flip, can be factored into a parser plus a distribution over choice sequences.*

Proof. Translate $\gamma$ into a free generator, $g$, by replacing return and (>>=) with the equivalent free generator constructs, and flip with the following free generator:

```
flip  :: ℚ → FGen Coin
flip (p / q) = pick [(p, h, return Heads), (q - p, t, return Tails )]
```

When interpreted as a generator, this function takes a rational number represented by $\frac{p}{q}$ and produces Heads with probability

$$\frac{p}{p+q-p} = \frac{p}{q}$$

and Tails with probability

$$\frac{q-p}{p+q-p} = \frac{q-p}{q} = 1 - \frac{p}{q}.$$

By construction, $\gamma = \mathcal{G}[\![g]\!]$.

Additionally, $g$ can be factored into a parser and a source of randomness via Theorem 3.1. Thus,

$$(\lambda x \longrightarrow (x, \varepsilon)) \langle\$\rangle\, \gamma = (\lambda x \longrightarrow (x, \varepsilon)) \langle\$\rangle\, \mathcal{G}[\![g]\!] \equiv \mathcal{P}[\![g]\!] \langle\$\rangle\, \mathcal{R}[\![g]\!],$$

and $\gamma$ can be factored as desired. □

This corollary is what we wanted to show all along. Monadic generators are parsers of randomness.

*Free Generators as Formal Language Syntax.* One final interpretation will prove useful. The *language of a free generator* is the set of choice sequences that it can make or parse. It is defined recursively, by cases:

$$
\begin{aligned}
&\mathcal{L}[\![\cdot]\!] :: \text{FGen a} \longrightarrow \text{Set String}\\
&\mathcal{L}[\![\text{Void}]\!] = \varnothing\\
&\mathcal{L}[\![\text{Return a}]\!] = \varepsilon\\
&\mathcal{L}[\![\text{Bind (Pick xs) f}]\!] = [\ c : s \mid (w, c, x) \leftarrow xs,\ s \leftarrow \mathcal{L}[\![x \ggg\!= f]\!]\ ]
\end{aligned}
$$

This definition uses Haskell's list comprehensions to iterate through the large space of choices sequences in the language of a free generator. To determine the language of a Bind node, we look at each possible choice and then at each possible string in the language $\mathcal{L}[\![x \ggg\!= f]\!]$ obtained by continuing with that choice. For each of these strings, we attach the appropriate choice label to the front. The end result is a massive list of all of the sequences of choices that, if made in order, would result in a valid output.

## 4 DERIVATIVES OF FREE GENERATORS

Next, we review the notion of Brzozowski derivative from formal language theory and show that a similar operation exists for free generators. The way these derivatives fall out from the structure of free generators highlights the advantages of taking the correspondence between generators and parsers seriously.

**Background: Derivatives of Languages.** The *Brzozowski derivative* [Brzozowski 1964] of a formal language $L$ with respect to some choice $c$ is defined as

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

In other words, the derivative is the set of strings in $L$ that begin with $c$, with the initial $c$ removed. For example,

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

Many formalisms for defining languages support syntactic transformations that correspond to Brzozowski derivatives. For example, we can take the derivative of a regular expression like this:

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

The $\nu$ operator, used in the "$\cdot$" rule and defined on the right, determines the *nullability* of an expression—whether or not it accepts $\varepsilon$. If $r$ accepts $\varepsilon$ then $\nu r = \varepsilon$, otherwise $\nu r = \varnothing$.

As one would hope, if $r$ has language $L$, it is always the case that $\delta_c r$ has language $\delta_c L$.

**The Free Generator Derivative.** First, we define *nullability* for free generators:

```
ν :: FGen a → Set a
ν(Return v) = {v}
νg          = ∅      (g ≠ Return v)
```

Note that this behaves a bit differently than the $\nu$ operation on regular expressions. For a regular expression $r$, the expression $\nu r$ is either $\varnothing$ or $\varepsilon$. Here, the null check returns either $\varnothing$ or the singleton set containing the value in the Return node. That is, $\nu$ for free generators extracts a value that can be obtained by making no further choices. Another difference is that, for free generators, "can accept the empty string" and "accepts only the empty string" are equivalent statements; this greatly simplifies the definition of $\nu$.

To see the derivative operation might look like, we can write down some equations that a *derivative* operation should satisfy, based on the equations satisfied by regular expressions:

$$
\begin{aligned}
\delta_c\ \mathsf{void} &\equiv \mathsf{void} && && (1)\\
\delta_c(\mathsf{return}\ v) &\equiv \mathsf{void} && && (2)\\
\delta_c(\mathsf{pick}\ xs) &\equiv x && \text{if } (c,\ x) \in xs && (3)\\
\delta_c(\mathsf{pick}\ xs) &\equiv \mathsf{void} && \text{if } (c,\ x) \notin xs && \\
\delta_c(x \ggg f) &\equiv \delta_c(f\ a) && \text{if } \nu x = \{a\} && (4)\\
\delta_c(x \ggg f) &\equiv \delta_c x \ggg f && \text{if } \nu\ x = \varnothing &&
\end{aligned}
$$

The derivative of an empty generator, or of one that immediately returns a value without looking at any input, should be void. The derivative of pick depends on whether or not $c$ is present in the list of possible choices—if it is, we simply make the choice; if not, the result is void. Finally, the equations for $(\ggg)$ are based on the equation for concatenation of regular expressions, using $\nu$ to check to see if the left hand side of the expression is out of choices to make.

Of course, these equations are not definitions, and in fact the actual definition of the derivative for a free generator $g$ is quite simple:

```
δ :: Char → FGen a → FGen a
δc(Return v)          = void
δc(Bind (Pick xs) f)  =
  case find ((== c) . snd) xs of
    Just (_, _, x) → x >>= f
```

Nothing $\longrightarrow$ void

Since freer monads are pre-normalized, there is no need to check nullability explicitly in this definition. It is always apparent from the top-level constructor (Return or Bind) whether or not there is a choice available to be made.

Lemma 4.1. *$\delta_c$ satisfies equations (1), (2), (3), and (4). In other words, the free generator derivative behaves similarly to the regular expression derivative.*

Proof sketch. See Appendix B for the proofs. Most are immediate. □

Another way to ensure that the derivative operation acts as expected is to see how it behaves in relation to the free generator's language interpretation. The following theorem makes this concrete:

Theorem 4.2. *For all free generators $g$ and choices $c$,*

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

Proof sketch. Straightforward induction (see Appendix C). □

That is, the derivative of a free generator's language is the same as the language of its derivative.

Since derivatives behave as expected, we can use them to simulate the behavior of a free generator. Just as we can check if a regular expression matches a string by taking derivatives with respect to each character in the string, we can simulate a free generator's parser interpretation by taking repeated derivatives. Each derivative fixes a particular choice, so a sequence of derivatives fixes a choice sequence.

## 5 GENERATING VALID RESULTS WITH GRADIENTS

We now put the theory of free generators and their derivatives into practice. We introduce Choice Gradient Sampling (CGS), a novel algorithm for generating data that satisfies some given validity condition.

The *Choice Gradient Sampling* uses derivatives to step a free generator through choices one at a time, guiding it towards values that are valid with respect to a given validity condition. At each step, the algorithm looks at all available choices and takes the free generator's derivative with respect to each one. Since this is, in a sense, a vector of all possible derivatives, we call this the *gradient* of the free generator, by analogy with calculus. We write

$$\nabla g = \langle \delta_{\mathsf{a}} g,\ \delta_{\mathsf{b}} g,\ \delta_{\mathsf{c}} g \rangle$$

for the gradient of $g$ with respect to the available choices $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$.

Since each derivative in the gradient is itself a free generator, the derivatives can be interpreted as value generators and sampled. If the derivative with respect to c produces lots of valid samples, then c is a good choice. If it produces mostly invalid samples, maybe other choices would be better. This process provides a metric that guides the algorithm toward a series of "good" choices, leading, on average, to lots of valid inputs.

We present the CGS algorithm in detail in Figure 2. Lines 7–14 are the core of the algorithm; their execution is shown pictorially in Figure 3. We take the gradient of $g$ by taking the derivative with respect to each possible choice, in this case a, b, and c. Then we evaluate each of the derivatives by interpreting the free generator with $\mathcal{G}[\![\cdot]\!]$, sampling values from the resulting value generator, and counting how many of those results are valid with respect to $\varphi$. The precise number of samples is controlled by $N$, the sample rate constant; this is up to the user, but in general higher values for $N$ will give better information about each derivative at the expense of time spent sampling. At the

$g \leftarrow G$
$\mathcal{V} \leftarrow \varnothing$
**while true do**
  **if** $\nu g \neq \varnothing$ **then return** $\nu g \cup \mathcal{V}$
  **if** isVoid g **then** $g \leftarrow G$
  $\nabla g \leftarrow \langle \delta_c g \mid c \in C \rangle$ ▷ $\nabla g$ is the gradient of $g$
  **for** $\delta_c g \in \nabla g$ **do**
    **if** isVoid $\delta_c g$ **then**
      $v \leftarrow \varnothing$
    **else**
      $x_1, \ldots, x_N \leftsquigarrow \mathcal{G}[\![\delta_c g]\!]$ ▷ Sample $\mathcal{G}[\![\delta_c g]\!]$
      $v \leftarrow \{x_j \mid \varphi(x_j)\}$
    $f_c \leftarrow |v|$ ▷ $f_c$ is the *fitness* of c
    $\mathcal{V} \leftarrow \mathcal{V} \cup v$
  **if** $\max_{c \in C} f_c = 0$ **then**
    **for** $c \in C$ **do** $f_c \leftarrow 1$
  $g \leftsquigarrow$ weightedChoice $\{(f_c, \delta_c g) \mid c \in C\}$

Fig. 2. Choice Gradient Sampling: Given a free generator $G$ in simplified form, a sample rate constant $N$, and a validity predicate $\varphi$, this algorithm produces a set of outputs that all satisfy $\varphi(x)$.

end of sampling, we have values $f_a$, $f_b$, and $f_c$, which we can think of as the "fitness" of each choice. We then pick a choice randomly, weighted based on fitness, and continue until our choices produce a valid output.

Critically, we avoid wasting effort by saving the samples ($\mathcal{V}$) 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.

We have implemented our Choice Gradient Sampling algorithm in Haskell, along with all of the definitions and proofs[4] presented throughout the paper. This code will be submitted for review as an artifact.

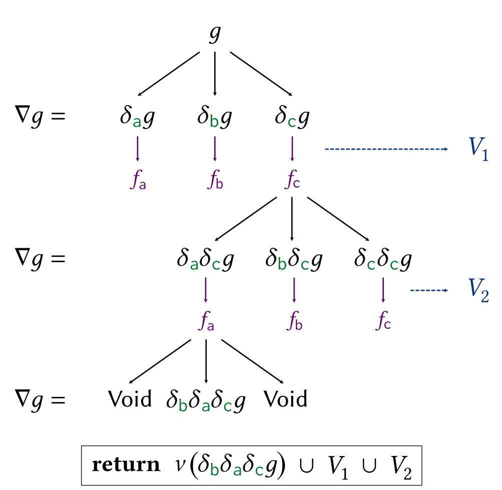



Fig. 3. The main loop of Choice Gradient Sampling.

## 6 EXPLORATORY EVALUATION

The Choice Gradient Sampling algorithm is not a tightly optimized production algorithm: it is a proof of concept. Primarily, CGS exists to illustrate the theory of free generators and their derivatives. Still, there is much to learn by exploring how well CGS is able to guide realistic generators to valid outputs.

We set out to answer two basic research questions:

[4]The proofs are not machine checked, they are implemented as unchecked equational reasoning in Haskell.

**RQ1** Does CGS produce more useful test inputs than standard sampling procedures, in the same period of time?

**RQ2** Are the test inputs obtained from CGS well-distributed in shape and size?

Our experimental results suggest that, with a few (interesting) caveats, these questions are both answered in the affirmative. We find that CGS generally produces at least twice as many valid values as *rejection sampling* (explained in the next section) in the same period of time, and we also find that CGS's values tend to be at least as diverse as the ones from rejection sampling. This indicates that using derivatives to guide generation is a promising approach to the valid generation problem.

**Experimental Setup.** Our experiments explore how well CGS improves on a canonical generation strategy. We compare our algorithm to the standard rejection sampling approach used by default in frameworks like QuickCheck, which takes a naïve generator, samples from it, and discards any results that are not valid. Rejection sampling is an interesting point of comparison because, like our approach, it requires no extra effort from the user.

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

| | 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** | AVL trees with values 0–9 | Is a balanced AVL tree | 500 | 5 |
| **STLC** | Arbitrary ASTs for $\lambda$-terms | Is well-typed | 400 | 5 |

Table 1. Overview of benchmarks.

Each of our benchmarks requires a simple free generator to act as a baseline and as a starting point for CGS. For consistency, and to avoid potential biases, our generators follow the respective inductive data types as closely as possible. For example, fgenTree, shown in §3 and used in the **BST** benchmark, follows the structure of the definition of the Tree type exactly.

The parameter $N$, used by CGS to decide how many samples to use for each iteration, was chosen via trial and error in order to balance fitness accuracy with sampling time. It is possible that some of our best-case results might improve with a more careful choice of $N$.

**Results.** We ran CGS and Rejection on each benchmark for one minute (on a MacBook Pro with an M1 processor and 16GB RAM) and recorded the unique valid values produced. We counted unique values because duplicate tests are generally less useful than fresh ones (if the system under test is pure, duplicate tests add no value). The totals, averaged over 10 trials, are presented in Table 2.

| | **BST** | **SORTED** | **AVL** | **STLC** |
|---|---|---|---|---|
| Rejection | 9,729 (103) | 6,587 (125) | 156 (5) | 105,602 (2,501) |
| CGS | 22,349 (416) | 58,656 (881) | 220 (1) | 297,703 (11,726) |

Table 2. Unique valid values generated in 60 seconds ($n$ = 10 trials). Standard deviation in parentheses.

These measurements show that CGS is always able to generate more unique values than Rejection in the same amount of time, often significantly more. The exception is the **AVL** benchmark; we discuss this below.

Besides unique values, we measured some other metrics; the charts in Figure 4 show the results for the **STLC** benchmark. The first plot ("Unique Terms over Time") shows how CGS behaves over time. Not only does CGS find more unique terms than Rejection overall, but its lead continues to grow over time. Additionally "Normalized Size Distribution" chart shows that CGS generates larger terms on average; this is good 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. Analogous charts for the remaining benchmarks can be found in Appendix D.

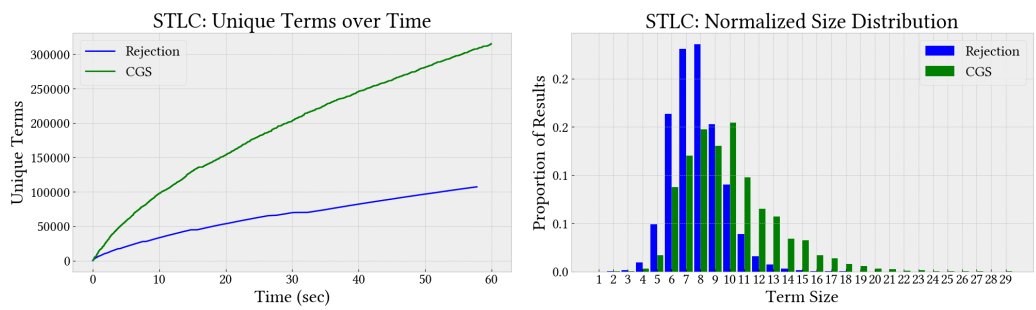



Fig. 4. Unique values and term sizes for the **STLC** benchmark, averaged over values in a single trial.

**Measuring Diversity.** For effective testing, we care about more than just the number of valid test inputs generated in a period of time—we care about the *diversity* of those inputs, since a more diverse test suite will find more bugs more quickly.

Our diversity metric relies on the fact that each value is roughly isomorphic to the choice sequence that generated it. 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. Thus, choice sequence diversity is a reasonable proxy for value diversity.

We estimated the average Levenshtein distance [Levenshtein et al. 1966] (the number of edits needed to turn one string into another) between pairs of choice sequences in the values generated by each of our algorithms. We chose this metric for sequence distance because it is fairly standard and implementations were readily available. Computing an exact mean distance between all pairs in such a large set would be very expensive, so we settled for the mean of a random sample of 3000 pairs from each set of valid values. The results are summarized in Table 3.

| | **BST** | **SORTED** | **AVL** | **STLC** |
|---|---|---|---|---|
| Rejection | 7.70(1.71) | 4.80(1.15) | 4.42(2.01) | 12.24(4.55) |
| CGS | 8.89(1.95) | 7.28(1.92) | 4.35(1.98) | 13.62(4.72) |

Table 3. Average Levenshtein distance between pairs of choice sequences, averaged over values in a single trial. Standard deviation in parentheses.

While **SORTED** does see significantly improved diversity, the effect is less dramatic **STLC** and **BST**, and diversity for **AVL** actually gets slightly worse.

One explanation for these lackluster results rests on the way CGS retains intermediate samples. While the first few samples will be mostly uncorrelated, the samples drawn later on in the generation process (once a number of choices have been fixed) will tend to be similar to one another. This likely results in clusters of inputs that are all valid but that only explore one particular shape of input.

Of course, some testing techniques actively seek out clusters of nearby inputs. Adaptive fuzzing algorithms [Lampropoulos et al. 2019; Zalewski 2018] intentionally explore clusters of inputs to ensure that they have adequately exercised a particular set of code paths, and that has shown to be useful for finding bugs. Additionally, for most of our benchmarks (again, we return to **AVL** in a moment), CGS does increase the overall 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.

**The Problem with AVL: Very Sparse Validity Conditions.** The **AVL** benchmark is an outlier in most of our measurements: 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. Understanding this phenomenon provides insight into a critical assumption the CGS algorithm, namely that it is not too difficult to find valid values randomly.

It is clear that AVL trees *are* quite 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.

In domains like this, CGS is unlikely find *any* valid trees while sampling. In particular, the check in line 15 of Figure 3 will often be true, meaning that choices will be made uniformly at random rather than guided by the fitness of the appropriate derivatives. We could reduce this effect by significantly increasing the sample rate constant $N$, but then sampling time would likely dominate generation time, resulting in worse performance overall.

The lesson here seems to be that the CGS algorithm does not work well with especially hard-to-satisfy validity conditions. In §9, we present an idea that would do some of the hard work ahead of time and help with this issue.

## 7 LIMITATIONS

Our free generator abstraction is extremely general and demonstrably useful, but a few technical weaknesses are worth discussing.

The biggest limitation has to do with the kinds of distributions our free generators can represent. In our exposition, we use a weighted coin ( flip ) as our randomness primitive, but QuickCheck is technically built using a primitive like:

```
choose :: Random r ⇒ (r, r) → Gen r
```

Intuitively, choose (x, y) uniformly picks a value in the *range* from x to y, and this range can technically be infinite (e.g., if r = Rational). This cannot be replicated with flip , nor can it be replicated with one of the higher-order combinators like frequency or pick. Thus, our results only apply to generators whose distributions are finitely supported.

Another small issue is that we have intentionally neglected one common element of monadic generators in the style of QuickCheck: size. Generators in standard QuickCheck track size bounds dynamically, allowing the testing framework to externally control the size distribution of the inputs that it generates. This does not impact our theoretical results (sizes can always be passed around manually, as we do in the examples in this paper), and sizes should be relatively easy to add to the free generator language in practice.

Finally, a note on the class of languages that free generators can parse (when interpreted with $\mathcal{P}[\![\cdot]\!]$). Free generators are limited in their nondeterminism (by the definition of pick, and by assumptions made in the definition of $\mathcal{P}[\![\cdot]\!]$); choices in a free generator are always unambiguous. This means that the parser interpretation of a free generator cannot parse arbitrary languages of choices, even though monadic parsers in general can parse arbitrary languages. Ultimately this is not a practical concern, as free generators parse sequences of choices, not realistic languages, but it is aesthetically disappointing. We believe it would be straightforward to add an operator for explicit nondeterminism and extend the interpretations accordingly.

## 8 RELATED WORK

We discuss a variety of publications that relate to the present work via connections either to free generators or to our Choice Gradient Sampling algorithm.

**Parsing and Generation.** The connection between parsers and generators has been employed implicitly in some generator implementations. Two popular property-based testing libraries, Hypothesis [MacIver et al. 2019] and Crowbar [Dolan and Preston 2017], implement generators by parsing a stream of random choices. While these libraries do not make a formal connection between parsing and generation, they further illustrate the value in understanding the relationship between these ideas.

**Free Generators.** Garnock-Jones et al. present a formalism based on parsing expression grammars (PEGs) with some of the same goals as ours. They give a derivative-based algorithm that somewhat resembles CGS, which constructs sentences that match a particular PEG. Their work does not attempt to solve the valid generation problem for complex validity conditions like the ones we tackle, but it does provide further evidence that connecting parsing and generation is advantageous.

Claessen et al. [2015] present a generator representation that is related to our free generator structure, but 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 generated 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. This pruning procedure performs well when validity conditions are written to take advantage of laziness, but it is highly dependent on evaluation order and avoids generator choices that immediately lead to exclusively invalid results. In contrast, CGS does not require that predicates be written in a specific way and has a much more nuanced notion of "unhelpful" choices.

**The Valid Generation Problem.** Many other approaches to the valid generation problem have been explored.

The domain-specific language for generators provided by the QuickCheck library [Hughes 2007] makes it easier to write manual generators that produce valid inputs by construction. This approach is extremely general, but it can be labor intensive. In the present work, we aim to avoid manual techniques like this in the hopes of making property-based testing more accessible to programmers that do not have the time or expertise to write their own custom generators.

The Luck language [Lampropoulos et al. 2017a] provides a middle-ground solution; users are still required to put in some effort, but they are able to define generators and validity predicates at the same time. Luck provides a satisfying solution if users are starting from scratch and willing to learn a domain-specific language, but if validity predicates have already been written or users do not want to learn a new language, a more automated solution such as ours may be preferable.

When validity predicates are expressed as inductive relations, approaches like the one in *Generating Good Generators for Inductive Relations* [Lampropoulos et al. 2017b] are extremely powerful.

In the QuickChick framework, users can extract generators from the inductive relations that they likely already have for their proofs. This is incredibly convenient for testing lemmas that will eventually be proved, but that the user wants more confidence in before attemptiting the proof. Unfortunately, the kinds of inductive relations that QuickChick works from generally require dependent types to express, so this approach does not work in most mainstream programming languages.

Target [Löscher and Sagonas 2017] uses search strategies like hill climbing and simulated annealing to supplement random generation and significantly streamline property-based testing. Löscher and Sagonas's approach works extremely well when inputs have a sensible notion of "utility," but in the case of valid generation the utility is often degenerate—0 if the input is invalid, and 1 if it is valid—with no good way to say if an input is "better" or "worse." In these cases, derivative-based searches may make more sense.

Some approaches use machine learning to automatically generate valid inputs. Learn&Fuzz [Godefroid et al. 2017] generates valid data using a recurrent neural network. 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 [Reddy et al. 2020] uses reinforcement learning to guide a generator to valid inputs. This approach served as early inspiration for our work, and we think that the theoretical advance of generator derivatives may lead improved learning algorithms in the future (see §9).

## 9 CONCLUSION

Free generators and their derivatives are powerful structures that give a flexible perspective on random generation. This formalism yields a useful algorithm for addressing the valid generation problem, and it clarifies the folklore that a generator is a parser of randomness. Moving forward, there are a number of paths to explore, some continuing our theoretical exploration and others looking towards algorithmic improvements.

**Bidirectional Free Generators.** We have only scratched the surface of what seems possible with free generators. One concrete next step is to merge the theory of free generators with the emerging theory of *ungenerators* [Goldstein 2021]. This work describes 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.

Free generators are quite compatible with these ideas, and 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.

**Algorithmic Optimizations.** In §6, we saw some problems with the Choice Gradient Sampling algorithm: because CGS evaluates derivatives via sampling, it does poorly when validity conditions are very difficult to satisfy. This begs the question: might it be possible to evaluate the fitness of a derivative without naïvely sampling?

One potential approach involves staging the sampling process. Given a free generator with a depth parameter, we can first evaluate choices on generators for size 1, then evaluate choices for size 2, etc. These intermediate stages would make gradient sampling more successful at larger sizes, and might significantly improve the results on benchmarks like **AVL**. Unfortunately, this 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. Nevertheless, this design space seems well worth exploring.

**Making Choices with Neural Networks.** Another algorithmic optimization is a bit farther afield: using recurrent neural networks (RNNs) to improve our generation procedure.

As Choice Gradient Sampling makes choices, it generates 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 this:

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

In the course of CGS, this information is used once (to make the next choice) and then forgotten—but 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 details to work out, including network architecture, hyper-parameters, etc., but in theory we could run CGS for a while, train an RNN, and after that point only use the RNN to generate valid data. Setting things up this way would recover some of the time that is currently spent 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 might help increase adoption of property-based testing in industry.

## REFERENCES

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

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

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

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

Stephen Dolan and Mindy Preston. 2017. Testing with crowbar. In *OCaml Workshop*.

Tony Garnock-Jones, Mahdi Eslamimehr, and Alessandro Warth. 2018. Recognising and generating terms using derivatives of parsing expression grammars. *arXiv preprint arXiv:1801.10490* (2018).

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.

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

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

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. *ACM SIGPLAN Notices* 50, 12 (2015), 94–105.

Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017a. 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

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

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

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

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.

Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-Based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 46–56. https://doi.org/10.1145/3092703.3092711

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.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

Tomáš Petříček. 2009. Encoding monadic computations in C# using iterators. *Proceedings of ITAT* (2009).

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

Michal Zalewski. 2018. AFL quick start guide. http://lcamtuf.coredump.cx/afl/QuickStartGuide.txt.

# Appendix

## A PROOF OF THEOREM 3.1

Lemma A.1.

$$\mathcal{P}[\![x \ggg f]\!] \langle\$\rangle \mathcal{R}[\![x \ggg f]\!] \equiv (\mathcal{P}[\![x]\!] \langle\$\rangle \mathcal{R}[\![x]\!]) \ggg \lambda(a, []) \to (\mathcal{P}[\![f\,a]\!] \langle\$\rangle \mathcal{R}[\![f\,a]\!])$$

Proof. By induction on the structure of $x$.

- Case x = Return a

```
𝒫⟦Return a >>= f⟧ ⟨$⟩ ℛ⟦Return a >>= f⟧
  -- By definition (>>=).
  ≡ 𝒫⟦f a⟧ ⟨$⟩ ℛ⟦f a⟧
  -- By η-expansion and definitions of 𝒫⟦·⟧ and ℛ⟦·⟧.
  ≡ 𝒫⟦Return a⟧ ⟨$⟩ ℛ⟦Return a⟧ >>= λ (a, []) → 𝒫⟦f a⟧ ⟨$⟩ ℛ⟦f a⟧
```

- Case x = Bind (Pick xs) k

```
𝒫⟦Bind (Pick xs) k >>= f⟧ ⟨$⟩ ℛ⟦Bind (Pick xs) k >>= f⟧
  -- By definition (>>=).
  ≡ 𝒫⟦Bind (Pick xs) (λa → k a >>= f)⟧ ⟨$⟩ ℛ⟦Bind (Pick xs) (λa → k >>= f))⟧
  -- By definition (𝒫⟦·⟧ and ℛ⟦·⟧).
  ≡ (do
      c ← consume
      x ← case find ((== c) . snd) xs of
        Just (_, _, x) → return x
        Nothing → fail
      𝒫⟦x >>= λa → k a >>= f⟧) ⟨$⟩ (do
        (c, x) ← frequency (map (λ (w, c, y) → (w, return (c, y))) xs)
        s ← ℛ⟦x >>= (λa → k a >>= f)⟧
        pure (c : s))
  -- By simplification .
  ≡ do
      (_, x) ← frequency (map (λ (w, c, y) → (w, return (c, y))) xs)
      𝒫⟦x >>= λa → k a >>= f⟧ ⟨$⟩ ℛ⟦x >>= λa → k a >>= f⟧
  -- By monad laws.
  ≡ do
      (_, x) ← frequency (map (λ (w, c, y) → (w, return (c, y))) xs)
      𝒫⟦(x >>= k) >>= f⟧ ⟨$⟩ ℛ⟦(x >>= k) >>= f⟧
  -- By IH.
  ≡ do
      (_, x) ← frequency (map (λ (w, c, y) → (w, return (c, y))) xs)
      (𝒫⟦x >>= k⟧ ⟨$⟩ ℛ⟦x >>= k⟧) >>= (λa → 𝒫⟦f a⟧ ⟨$⟩ ℛ⟦f a⟧)
  -- By expansion and definitions (𝒫⟦·⟧ and ℛ⟦·⟧).
  ≡ (𝒫⟦Bind (Pick xs) k⟧ ⟨$⟩ ℛ⟦Bind (Pick xs) k⟧) >>= (λa → 𝒫⟦f a⟧ ⟨$⟩ ℛ⟦f a⟧)
```

□

Theorem 3.1 (Factoring). *Every free generator can be factored into a parser and a distribution over choice sequences that is equivalent to its interpretation as a generator. In other words, for all free generators g,*

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

Proof. By induction on the structure of $x$.

- Case x = Return a

```
𝒫⟦Return a⟧ ⟨$⟩ ℛ⟦Return a⟧
  -- By definition (𝒫⟦·⟧ and ℛ⟦·⟧).
  ≡ return (a, [])
  -- By definition (𝒢⟦·⟧).
  ≡ (λa → (a, [])) ⟨$⟩ 𝒢⟦Return a⟧
```

- Case x = Bind (Pick xs) k

```
𝒫⟦Bind (Pick xs) k⟧ ⟨$⟩ ℛ⟦Bind (Pick xs) k⟧
  -- By definition (𝒫⟦·⟧ and ℛ⟦·⟧).
  ≡ (do c ← consume
        x ← case find ((== c) . snd) xs of
          Just (_, _, x) → return x
          Nothing → fail
        𝒫⟦x ≫= k⟧) ⟨$⟩ (do
          (c, x) ← frequency (map (λ (w, c, x) → (w, return (c, x))) xs)
          s ← ℛ⟦x ≫= k⟧
          pure (c : s))
  -- By simplification .
  ≡ do x ← frequency (map (λ (w, _, x) → (w, return x)) xs)
       s ← ℛ⟦x ≫= k⟧
       return 𝒫⟦x ≫= k⟧ s
  -- By monad laws.
  ≡ do x ← frequency (map (λ (w, _, x) → (w, return x)) xs)
       𝒫⟦x ≫= k⟧ ⟨$⟩ ℛ⟦x ≫= k⟧
  -- By Lemma A.1.
  ≡ do x ← frequency (map (λ (w, _, x) → (w, return x)) xs)
       (𝒫⟦x⟧ ⟨$⟩ ℛ⟦x⟧) ≫= λ (a, []) → 𝒫⟦k a⟧ ⟨$⟩ ℛ⟦k a⟧
  -- By IH.
  ≡ do x ← frequency (map (λ (w, _, x) → (w, return x)) xs)
       ((λa → (a, [])) ⟨$⟩ 𝒢⟦x⟧) ≫= λ (a, []) → (λa → (a, [])) ⟨$⟩ 𝒢⟦k a⟧
  -- By simplification .
  ≡ (λa → (a, [])) ⟨$⟩ do x ← frequency (map (λ (w, _, x) → (w, return x)) xs)
                          a ← 𝒢⟦x⟧
                          𝒢⟦k a⟧
  -- By definition (𝒢⟦·⟧)
  ≡ (λa → (a, [])) ⟨$⟩ 𝒢⟦Bind (Pick xs) k⟧
```

Thus the decomposition of a free generator into a parser and a source of randomness is equivalent to interpreting it as a generator. □

## B PROOF OF LEMMA 4.1

LEMMA 4.1. *$\delta_c$ satisfies equations (1), (2), (3), and (4). In other words, the free generator derivative behaves similarly to the regular expression derivative.*

PROOF. We prove each equation individually.

- Equation 1: $\delta_c\,\text{void} \equiv \text{void}$
  By evaluation.
- Equation 2: $\delta_c(\text{return}\ v) \equiv \text{void}$
  By definition.
- Equation 3:

$$\begin{aligned} \delta_c(\text{pick}\ xs) &\equiv x && \text{if } (c,\ x) \in xs \\ \delta_c(\text{pick}\ xs) &\equiv \text{void} && \text{if } (c,\ x) \notin xs \end{aligned}$$

  Unfold the definition of pick, by evaluation.
- Equation 4:

$$\begin{aligned} \delta_c(x \ggg f) &\equiv \delta_c(f\ a) && \text{if } \nu x = \{a\} \\ \delta_c(x \ggg f) &\equiv \delta_c x \ggg f && \text{if } \nu x = \varnothing \end{aligned}$$

  – Case $x = \text{Return}\ a$. By definition, $\nu x = \{a\}$.

$$\begin{aligned} \delta_c(\text{x} \ggg \text{f}) &\equiv \delta_c(\text{Return a} \ggg \text{f}) && \textit{-- By assumption.} \\ &\equiv \delta_c(\text{f a}) && \textit{-- By definition } (\ggg). \end{aligned}$$

  – Case $x = \text{Bind}\ (\text{Pick}\ xs)\ g$. By definition, $\nu x = \varnothing$.

$$\begin{aligned} \delta_c(\text{x} \ggg \text{f}) &\equiv \delta_c(\text{Bind (Pick xs) g} \ggg \text{f}) && \textit{-- By assumption.} \\ &\equiv \delta_c(\text{Bind (Pick xs)}\ (\lambda \text{a} \rightarrow \text{g a} \ggg \text{f})) && \textit{-- By definition } (\ggg). \\ &\equiv \textbf{case}\ \text{find ((== c) . snd) xs}\ \textbf{of} && \textit{-- By definition } (\delta). \\ &\qquad \text{Just (\_, \_, x)} \rightarrow \text{x} \ggg (\lambda \text{a} \rightarrow \text{g a} \ggg \text{f}) \\ &\qquad \text{Nothing} \rightarrow \text{void} \\ &\equiv \textbf{case}\ \text{find ((== c) . snd) xs}\ \textbf{of} && \textit{-- By monad laws.} \\ &\qquad \text{Just (\_, \_, x)} \rightarrow (\text{x} \ggg \text{g}) \ggg \text{f} \\ &\qquad \text{Nothing} \rightarrow \text{void} \\ &\equiv \delta_c \text{x} \ggg \text{f} && \textit{-- By definition } (\delta). \end{aligned}$$

Thus all four equations hold. □

## C PROOF OF THEOREM 4.2

THEOREM 4.2. *For all free generators g and choices c,*

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

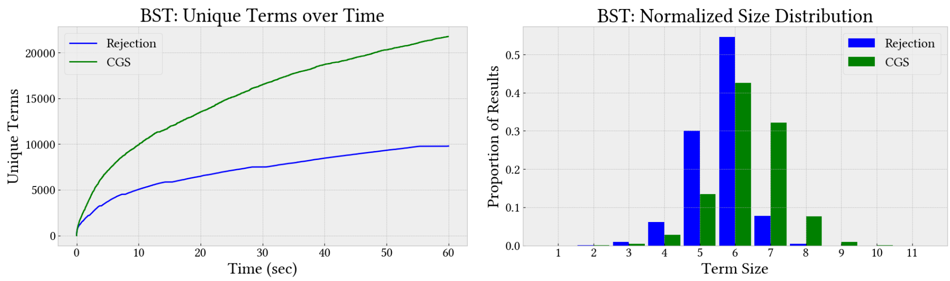
PROOF.

```
ℒ⟦δ_c x⟧ = ℒ⟦case x of                                       -- By definition (δ).
               Return _ → void
               Bind (Pick xs) k → case find ((== c) . snd) xs of
                                    Just (_, _, y) → ℒ⟦y >>= k⟧
                                    Nothing → [] ⟧
         = case x of                                          -- By definition (ℒ).
             Return _ → []
             Bind (Pick xs) k → o
               (_, d, y) ← xs
               cs ← ℒ⟦y >>= k⟧
               guard (c == d)
               pure cs
         = do                                                 -- By Haskell identities .
             (d : cs) ← case x of
               Return _ → [ [] ]
               Bind (Pick xs) k → do
                 (_, d, y) ← xs
                 s ← ℒ⟦y >>= k⟧
                 pure (d : s)
             guard (c == d)
             pure cs
         = do                                                 -- By definition (ℒ).
             (d : cs) ← ℒ⟦x⟧
             guard (c == d)
             pure cs
         = δ_c(ℒ⟦x⟧)                                          -- By definition (δ)
```

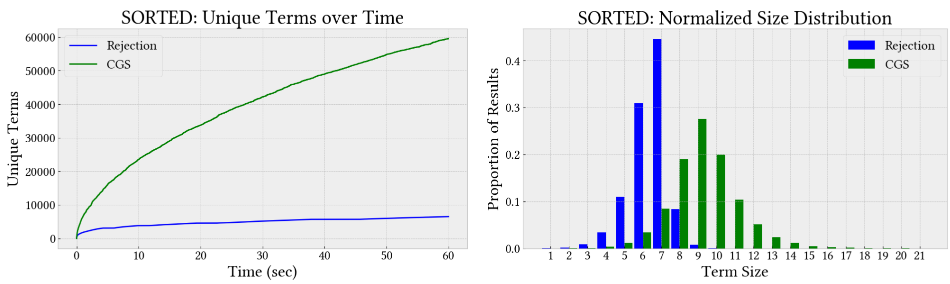
□

There is another proof of this theorem, suggested by Alexandra Silva, which uses the fact that $2^{\Sigma^*}$ 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.

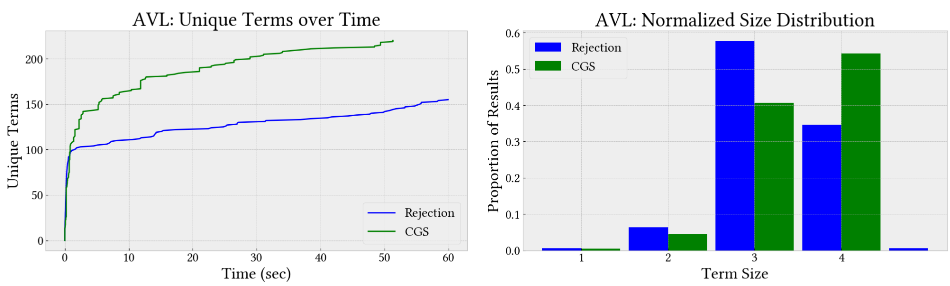
# D FULL EXPERIMENTAL RESULTS




**BST** Charts




**SORTED** Charts




**AVL** Charts