Valid Generators via Parsing*

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

When generating random data structures, it is often important that the generated values satisfy some validity condition. We tackle this problem by observing that, perhaps surprisingly, random data generators are closely related to *parsers*. In particular, both structures make a sequence of choices that are used to build a data structure.

This insight leads us to a rich theory of *free generators* that makes the connection between generators and parsers explicit; free generators can be interpreted as either generators or parsers as needed. Free generators admit an efficiently computable notion of *Brzozowski derivative* that can be used to preview the effect of a particular choice in the generator. We use this derivative operation in a novel algorithm that aims to solve the problem of generating valid data structures in certain domains.

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

1 Introduction

Various uses for random data are subject to the *valid generation problem*: given a type τ and a validity condition φ, efficiently produce random $x{:}\tau$ such that $\varphi(x)$. If we take τ to be the type of binary trees and φ to be "is a binary search tree (BST)", the challenge is to randomly generate a variety of BSTs as quickly possible.

This problem is especially common when using tools like *QuickCheck* [3] for *property-based testing*. Property-based testing checks that a program obeys a functional specification using hundreds or thousands of random inputs. This runs into the valid generation problem when the random inputs are expected to satisfy some precondition. For example, in order to test that a BST insert procedure preserves the BST invariants, we need to generate a lots of valid BSTs.

The valid generation problem is computationally hard in general, but various approaches have shown promise [2, 5, 9]. We choose to focus on *generator optimization* methods, that start with an existing generator (which does not produce exclusively valid inputs) and tune or modify the generator to produce valid results with higher-than-normal probability [11, 12]. In particular, we explore new avenues for generator optimization that hinge on a simple observation: generators operate like *parsers* of random choices.

A random generator is program that makes a sequence of random choices—essentially coin-flips—to decide how some data structure should be constructed. As the program runs, each choice is made in real-time by querying some kind of random number generator. Of course, there nothing stopping us from pre-computing 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 subtle shift in perspective turns our generator into a parser of sequences of choices!

We make this intuition concrete with structures that we call *free generators*, which re-imagine standard random generators as data structures that can be interpreted and manipulated at will. Free generators can be viewed as normal generators, which make real-time choices, but they can also be interpreted as deterministic parsers of sequences of choices in the way we observed above. This gives a concrete way to understand the close connection between generation and parsing, clarifying existing folklore into a useful formalism.

As practical tools, free generators give much-needed flexibility. For example, free generators are not tied to a particular output distribution—we can modify the average size and shape of output values without modifying the actual generator. This works by taking some user-specified distribution over choices, which may be carefully tuned by hand or even constructed automatically via some learning algorithm, and using the parser interpretation of the free generator to parse those sequences of choices into data. This is important, but it does not yet give us a way to solve the valid generation problem.

For that, we rely on the fact that free generators are represented as data structures that can be manipulated by programs. In particular, we can define a *derivative* operation on free generators that computes a new generator that is identical to the first, except with a particular choice already fixed. In the case of our binary search tree example, this might fix a particular value for a node, or fix a stopping point for a particular sub-tree. We can use derivatives to evaluate how likely a given choice is to produce valid results—this yields a new approach to the valid generation problem that has done remarkably well in our tests.

In the next section, we describe the high-level motivation behind our free generator formalism, and we present promising

*Placeholder title

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

preliminary results that justify the theory in the remainder of the paper (§2). After that, we make the following contributions:

- We present a novel theory of *free generators* that gives a formal account of the connections between random generators and parsers (§3).
- We define a *derivative* operation on free generators that computes the state of a generator after making a particular choice and prove this operation consistent with related notions in formal languages (§4).
- We give the details behind the results presented in §2, including an algorithm for tackling the *valid generation problem* that relies on free generator derivatives (§5).

We conclude with related work (§6) and ideas for future research (§7).

2 The High-Level Story

This section gives a bird's-eye view of our motivation, theory, and preliminary experimental results.

2.1 Generators and Parsers

The goal of this work is to optimize an existing generator by observing that generators are like parsers. 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

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

by making a series of random choices. (For the sake of this example, and most of this paper, assume those choices are made uniformly.) 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.

Now, consider parseTree (also in Figure 1), which parses a tree from a string containing the characters `n`, `l`, `t`, and `f`. For the sake of simplicity, assume we have some built-in way to consume the next character in the string. The parser turns

$$\texttt{ntll} \text{ into Node True Leaf Leaf} \quad \text{and}$$
$$\texttt{ntlnfll} \text{ into Node True Leaf (Node False Leaf Leaf).}$$

The parser operates character by character, sometimes changing the way 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 were 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. Ideally, we should always be able to re-interpret a generator as a parser of sequences of choices, or a parser as a generator over its output set. Next, we make that intuition concrete.

```
genTree height =
  1. If height = 0, then return Leaf.
  2. Else, randomly choose:
     • return Leaf.
     • Do the following:
        a. Randomly choose:
           – Let x = True.
           – Let x = False.
        b. Let l = genTree (height − 1).
        c. Let r = genTree (height − 1).
        d. return (Node x l r).

parseTree height =
  1. If height = 0, then return Leaf.
  2. Else, consume a character c:
     • If c = l, then return Leaf.
     • If c = n, then do the following:
        a. Consume a character c:
           – If c = t, then let x = True.
           – If c = f, then let x = False.
           – Else, fail.
        b. Let l = parseTree (height − 1).
        c. Let r = parseTree (height − 1).
        d. 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.[1] Free generators are not traditional programs like genTree or parseTree; instead, they are data structures that can be *interpreted* as we see fit.

In our Haskell development 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 in order to make it clear that these are data structures—not normal programs—we give an example here that is explicitly built from data constructors. This makes everything a bit harder to interpret, but the details are not critical; instead, look at Figure 2 and notice

[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.

```
mkfTree height =
  1. If height = 0, then Pure Leaf.
  2. Else:
     Select
       [ (l, Pure Leaf),
         (n, MapR
           (Pair (Select
                    [ (t, Pure True),
                      (f, Pure False) ])
                 (Pair (mkfTree (height - 1))
                       (mkfTree (height - 1))))
           (λ (x, (l, r)) → Node x l r)) ]
```

Figure 2. A free generator for Boolean binary trees.

the structural similarities between that free generator and the examples in Figure 1.

We can use mkfTree to construct a free generator for Boolean binary trees of a given height. When reading the data structure, think of Pure 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 the result that is generated/parsed. (In §3 we define Map instead, which is the same, but the arguments are reversed.) 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 and checking c against the first elements of the pairs.

The almost line-to-line correspondence between mkfTree 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,

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

These functions mean that we can write one free generator that can be used in different ways depending on what we need at the moment.

We also define $\mathcal{C}[\![g]\!]$ to be the choice distribution of a free generator. Intuitively, the choice distribution should produce the sequences of choices that the generator interpretation can make, or equivalently the sequences that the parser interpretation can parse. To make this relationship formal, we prove Theorem 3.4, which says that for any free generator g,

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

(where ⟨\$⟩ is defined as a kind of "mapping" operation over distributions). Another way to read this theorem is that generators can be factored into two pieces: a distribution over choice sequences, and a parser of those sequences.

We can work with those pieces independently if we wish. For example, we can replace $\mathcal{C}[\![g]\!]$ with some other distribution over choice sequences to obtain a generator with a totally different output distribution but the same overall structure.

Besides swapping out a generator's distribution, there are other reasons to prefer a free generator to a normal generator. In the next section, we see how free generators can be syntactically manipulated.

2.3 Derivatives of Free Generators

Every parser defines a formal language of strings based on the strings that it can successfully parse. Similarly, we define the language interpretation of a free generator $\mathcal{L}[\![g]\!]$ to be the set of choice sequences parsed (or made) by the generator. Viewing free generators this way suggests some interesting ways that free generators might be manipulated.

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

$$\delta_{\mathtt{c}} L = \{\mathtt{s} \mid \mathtt{cs} \in L\}$$

We can illustrate this derivative using parsers—conceptually, the derivative of a parser with respect to a character c is whatever parser remains assuming c has just been parsed. Parsers like parseTree cannot literally be modified in this way, but we can show the conceptual result of the transformation. For example, if we fix height = 5 for simplicity, we can think of the derivative of parseTree 5 with respect to n as:

```
δ_n(parseTree 5) ≈
  1. Consume a character c:
     • If c = t, then let $x$ = True.
     • If c = f, then let $x$ = False.
     • Else, fail.
  2. Let $l$ = parseTree 4.
  3. Let $r$ = parseTree 4.
  4. return (Node $x$ $l$ $r$).
```

This parser is "one step" simpler than parseTree. 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 imagine another step and look at the derivative of $\delta_{\mathtt{n}}$(parseTree 5) with respect to t:

```
δ_t δ_n(parseTree 5) ≈
  1. Let $l$ = parseTree 4.
  2. Let $r$ = parseTree 4.
  3. 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.

Excitingly, our free generators also have a closely related notion of derivative! We can take derivatives of the free generator produced by mkfTree that look almost identical to the ones that we saw for parseTree:

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

and

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

Even better, these derivatives are easily computable! 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:

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

Intuitively, the derivative of a free generator can be thought of as the generator that remains after parsing *or making* a choice. This means that we can essentially preview the result of making a choice without actually interpreting the generator. In the next section, we show the practical applications of this technique.

2.4 Preliminary Results

Starting from the idea of previewing a choice via a derivative, we develop an algorithm for generating valid data relative to a precondition φ, given a simple free generator g of values that may or may not satisfy φ.

The algorithm is shown graphically 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.

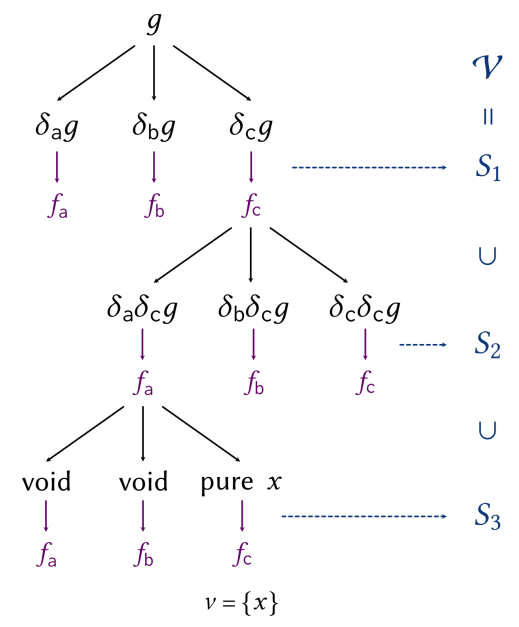

Figure 3. Generating valid values with free generator derivatives.

There are more algorithmic details that we elide until §5. For now, it suffices to understand that this approach uses free generators and their derivatives to find "good" choices and produce valid data.

We compared this algorithm to simple rejection sampling (i.e., generating values with uniform choices and ignoring any values that do not satisfy φ) in two small benchmarks. The first benchmark (BST) uses a free generator similar the ones produced by mkfTree, except with integers between 0 and 9 in each node instead of Booleans. Our validity condition ensures that the tree is a valid binary search tree. The second benchmark (STLC) uses a generator for simply-typed lambda calculus terms, where valid terms are those that are well-typed. We ran both algorithms ("QuickCheck" doing normal rejection sampling, and "Gradient" doing our gradient search) for two minutes, and tracked the number of unique valid values that each produced.

	QuickCheck	Gradient
BST	12,344	26,527
STLC	147,712	427,005

Table 1. Unique valid values generated in 120 seconds.

The numbers after two minutes are shown in Table 1. Clearly this is promising! Our algorithm consistently beat rejection sampling by more than a factor of two. Looking at plots of these values over time (Figure 4) gives even more

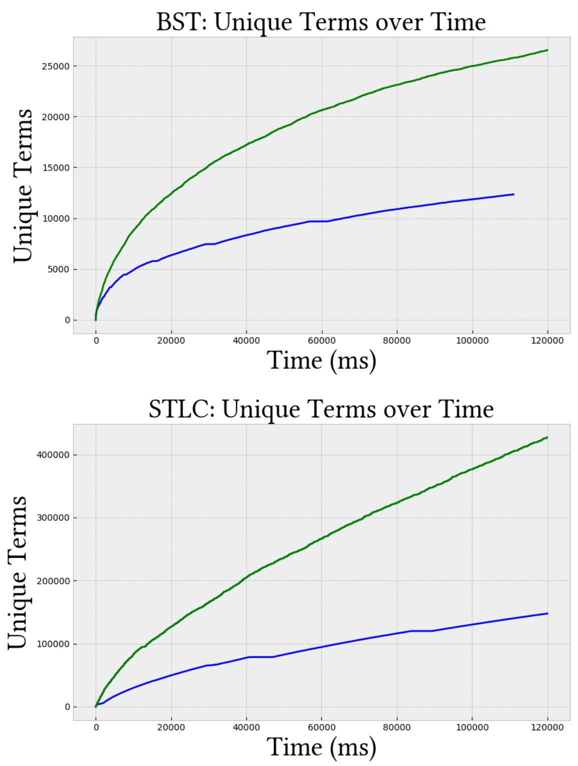

Figure 4. Unique valid values generated over 120 seconds. (QuickCheck in blue, Gradient in green.)

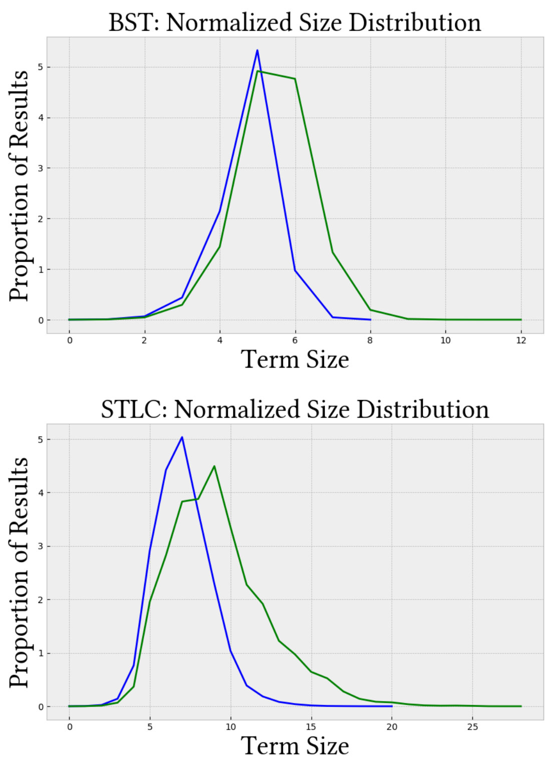

Figure 5. Normalized size distributions of valid values generated in 120 seconds. (QuickCheck in blue, Gradient in green.)

reason for optimism: for STLC especially, neither plot seems to be leveling off, and the Gradient line is increasing quite quickly.

Finally, we compared the relative sizes of the generated results. For BST, the Gradient algorithm increased the average term size by around 2, and for STLC the average size almost doubled from 5 to 10. The results are in Figure 5.

Overall we are encouraged by these preliminary results. They represent a strong proof of concept that indicates that free generators and their derivatives are practically useful. For more details on our gradient algorithm and our experimental design, see §5.

3 Free Generators in Detail

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 our new framing of generators that makes choices explicit and enables interesting syntactic manipulations.

3.1 Background: Applicative Parsers and Generators

In §2 we represent generators and parsers with pseudo-code, but moving forward we should to make our abstractions more concrete.

We represent both generators and parsers using *applicative functors*. In Haskell, we define an applicative functor as a type constructor f with an operation,

```
(⟨$⟩) :: (a → b) → f a → f b
```

also known as "fmap" and operations,

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

the latter of which is often called "TIE". At first they may seem fairly esoteric, but these operations really just define a convenient way to apply functions over some data structure. For example, the idiom

```
f ⟨$⟩ x ⟨∗⟩ y ⟨∗⟩ z
```

applies a pure function f "over" three structures x, y, and z.

We can use these operations to define genTree like we would in *QuickCheck* [3]:

```
genTree 0 = pure Leaf
genTree n =
  oneof [ pure Leaf,
          Node ⟨$⟩ genInt
               ⟨∗⟩ genTree (n − 1)
               ⟨∗⟩ genTree (n − 1) ]
```

In this context, pure is the trivial generator that always generates the same value, and Node ⟨$⟩ g1 ⟨∗⟩ g2 ⟨∗⟩ g3 means apply the function Node over three generators. 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 *Parsec* [10]:

```
parseTree 0 = pure Leaf
parseTree n =
  choice [ (l, pure Leaf),
           (n, Node ⟨$⟩ parseInt
                    ⟨∗⟩ parseTree (n - 1)
                    ⟨∗⟩ parseTree (n - 1)) ]
```

Parsers of this form represent functions of type

```
String → Maybe (a, String),
```

where a is the result type of the parser. The String in the input and output is the state of the parser, which changes as characters are consumed. The Maybe is there to handle possible failures. In this context, pure consumes no characters and never fails, it just produces the value passed to it. We can interpret Node ⟨$⟩ p1 ⟨∗⟩ p2 ⟨∗⟩ p3 as parsing 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 behavior is the same: choose between sub-parsers.[2] The only difference is that now the choice is based on an input string instead of a pseudo-random number generator, and we might fail if the consumed character is not in the list.

3.2 Representing Generators

With the applicative interface in mind, we can now give the formal definition of a *free generator*. We represent free generators as a data type in Haskell:

3.2.1 Type Definition.

The type FGen is defined as follows:

```
data FGen a where
  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 each correspond to some function from the interfaces in the previous section. 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 ⟨$⟩. Finally, Select subsumes both oneof and choice—it might mean either, depending on the interpretation.

[2]For clarity, we have slightly modified the interface for applicative parsers; *Parsec* has different way of expressing choice.

We call this structure "free" because of similarities to the free applicative functor. In essence, all this means is that, since this generator is represented syntactically, we can write a transformation from FGen a → f a for any f with similar structure. This is important later on in 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 recursively, by cases:

$$
\begin{aligned}
\mathcal{L}[\![\text{Void}]\!] &= \varnothing \\
\mathcal{L}[\![\text{Pure a}]\!] &= \varepsilon \\
\mathcal{L}[\![\text{Map f x}]\!] &= \mathcal{L}[\![\text{x}]\!] \\
\mathcal{L}[\![\text{Pair x y}]\!] &= \mathcal{L}[\![\text{x}]\!] \cdot \mathcal{L}[\![\text{y}]\!] \\
\mathcal{L}[\![\text{Select xs}]\!] &= \{\text{cs} \mid (\text{c}, \text{x}) \in \text{xs} \land \text{s} \in \mathcal{L}[\![\text{x}]\!]\}
\end{aligned}
$$

(Recall that $S \cdot T = \{st \mid s \in S \land t \in T\}$, the standard definition of language concatenation.)

3.2.3 Smart Constructors and Normal Forms.

Free generators admit a useful *normal form*. We ensure this property by requiring that free generators are built with *smart constructors*. Instead of Pair, users should use ⊗:

```
(⊗) :: 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. 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)
```

Finally, we define a smart constructor for Select:

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

Unlike the other smart constructors, select can fail. This helps us ensure that generators make sense later on—for example, our interpretation of choices means that it is not clear what it would mean the same choice label to appear twice in a given call to Select. When generators are built with only smart constructors, we say they are in normal form.

3.2.4 Examples. Before we continue, here are a couple of examples of free generators that illustrate how they should be used; these examples are used later in our evaluation. First, we can generalize our definitions from earlier in this section to get a single free generator fgenTree that subsumes genTree and parseTree.

```
fgenTree  ::  Depth ⟶ FGen Tree
fgenTree  0  =  pure  Leaf
fgenTree  n  =
    select  [  (l, pure  Leaf ),
               (n, Node ⟨$⟩ fgenInt
                       ⟨∗⟩  fgenTree  (n  –  1)
                       ⟨∗⟩  fgenTree  (n  –  1))  ]
```

This free generator looks almost exactly like genTree, but remember that whereas genTree is a program that makes choices as it runs, fgenTree is just a data structure.

Another example of a generator that we use later produces random terms of a simply-typed lambda-calculus. We discuss the details of the language in Appendix A. For now, we can just look at the generator:

```
data Type = TInt  |  TFun Type Type
data Expr
  =  Lit  Int  |  Plus  Expr  Expr
  |  Lam Type Expr |  Var  Int  |  App Expr Expr

fgenExpr  ::  Depth ⟶ FGen Expr
fgenExpr  0  =
    select  [  (i, Lit  ⟨$⟩  fgenInt ),
               (v, Var  ⟨$⟩  fgenVar)  ]
fgenExpr  n  =
    select  [  (i, Lit  ⟨$⟩  fgenInt ),
               (p, Plus  ⟨$⟩  fgenExpr  (n  –  1)
                      ⟨∗⟩  genExpr (n  –  1)),
               (l, Lam ⟨$⟩ fgenType
                      ⟨∗⟩  genExpr (n  –  1)),
               (a, App ⟨$⟩ fgenExpr  (n  –  1)
                      ⟨∗⟩  fgenExpr  (n  –  1)),
               (v, Var  ⟨$⟩  fgenVar)  ]
```

Structurally this is very similar to the previous generator, it just has more cases and more choices.

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 discussed 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 some data structure. We define the *generator interpretation* of a generator to be:

```
𝒢⟦Void⟧      = ⊥
𝒢⟦Pure v⟧    = pure  v
𝒢⟦Map f x⟧   = f  ⟨$⟩  𝒢⟦x⟧
𝒢⟦Pair x  y⟧  =
   (λx  y  ⟶  (x,  y))  ⟨$⟩  𝒢⟦x⟧ ⟨∗⟩ 𝒢⟦y⟧
𝒢⟦Select  xs⟧  =
   oneof  (map (λ (_,  x)  ⟶  𝒢⟦x⟧) xs)
```

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 normalized,*

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

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 normal form and not Void, $\mathcal{G}[\![g]\!]$ is defined.

Example 3.2. As desired, 𝒢⟦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:

```
𝒫⟦Void⟧      = λ  s ⟶Nothing
𝒫⟦Pure a⟧    = pure  a
𝒫⟦Map f x⟧   = f  ⟨$⟩  𝒫⟦x⟧
𝒫⟦Pair x  y⟧  =
   (λx  y  ⟶  (x,  y))  ⟨$⟩  𝒫⟦x⟧ ⟨∗⟩ 𝒫⟦y⟧
𝒫⟦Select  xs⟧  =
   choice  (map (λ (c,  x)  ⟶  (c,  𝒫⟦x⟧)) xs)
```

This definition uses the representation of parsers as functions of type String ⟶ Maybe (a, String) that we saw earlier. Error handling happens under the hood for operations like Pair.

Example 3.3. As desired, 𝒫⟦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 generator to be:

```
𝒞⟦Void⟧      = ⊥
𝒞⟦Pure a⟧    = Gen.pure ε
𝒞⟦Map f x⟧   = 𝒞⟦x⟧
𝒞⟦Pair x  y⟧  =
  (λ s t →st) Gen.⟨$⟩ 𝒞⟦x⟧ Gen.⟨∗⟩ 𝒞⟦y⟧
𝒞⟦Select  xs⟧ =
  Gen.oneof
    (map
      (λ (c, x) → (λ s →cs) Gen.⟨$⟩ 𝒞⟦x⟧) xs)
```

Note that we can equivalently think of this as a distribution over $\mathcal{L}[\![g]\!]$.

3.3.4 Coherence.

We would like for our three interpretations of free generators to make sense together. We call this property coherence, and prove it now:

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

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

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

$$\mathcal{P}[\![\text{Pair x y}]\!]\ \langle\$\rangle\ \mathcal{C}[\![\text{Pair x y}]\!]$$

should be a function of

$$\mathcal{P}[\![\text{x}]\!]\ \langle\$\rangle\ \mathcal{C}[\![\text{x}]\!] \quad \text{and} \quad \mathcal{P}[\![\text{y}]\!]\ \langle\$\rangle\ \mathcal{C}[\![\text{y}]\!].$$

Showing the correct relationship between these requires a lemma that says that for any sequence s of choices generated by $\mathcal{C}[\![\text{x}]\!]$ and an arbitrary sequence t, $\mathcal{P}[\![\text{x}]\!]$ st returns Just (a, t) for some a.

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. □

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 to a generator of next characters:

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

We interpret these functions as specifying the distribution over the a next choice, given a history of choices. (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 $\mathcal{C}[\![g]\!]$?

The solution is actually quite elegant: we just replace $\mathcal{C}[\![g]\!]$ with a distribution to yield the *definition*:

```
𝒢̅⟦((h, d), g)⟧ = 𝒫⟦g⟧ ⟨$⟩ genDist h
  where
    genDist h = d h ≫= λx → case x of
      Nothing → Gen.pure h
      Just c →genDist hc
```

Whereas before we proved an equivalence between $\mathcal{G}[\![\text{g}]\!]$ and $\mathcal{P}[\![\text{g}]\!]\langle\$\rangle\ \mathcal{C}[\![\text{g}]\!]$, 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 theoretically have a different distribution for each use-case of the free generator. In a property-based testing scenario, this might mean a finely-tuned distribution for each property, carefully optimized to find bugs as quickly as possible.

4 Brzozowski Derivatives

So far we have shown how to construct a free generator that factors into a distribution over choice sequences and a parser. This is useful on its own, but it would be a shame to stop there. In particular, free generators, like many other representations of languages, can be manipulated in interesting and useful ways. In this section, we recall the idea of a Brzozowski derivative for languages and regular expressions, and we show that a similar notion can be applied to free generators.

4.1 Background: Derivatives of Languages

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

$$\delta_{\text{c}} L = \{\text{s} \mid \text{cs} \in L\}.$$

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

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

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

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

where *nullability* (the ν operator in the "$\cdot$" rule) is defined as:

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

As one would hope, if r has language L, it is always the case that $\delta_{\mathsf{c}}r$ has language $\delta_{\mathsf{c}}L$.

It is worth noting that the derivative operation is called a "derivative" because of the "$\cdot$" rule, which is closely related to the product rule for derivatives in other contexts. The only difference is the nullability check, which enforces that that a character can only be removed from r_2 if it is possible for r_1 to accept ε.

4.2 Derivatives of Free Generators

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

4.2.1 Definitions.

We define the *derivative of a free generator* in the following way:

$$\begin{aligned}
&\delta_{\mathsf{c}}\mathsf{Void} &&= \mathsf{Void} \\
&\delta_{\mathsf{c}}(\mathsf{Pure\ v}) &&= \mathsf{Void} \\
&\delta_{\mathsf{c}}(\mathsf{Map\ f\ x}) &&= \mathsf{f}\ \langle\$\rangle\ \delta_{\mathsf{c}}\mathsf{x} \\
&\delta_{\mathsf{c}}(\mathsf{Pair\ x\ y}) &&= \delta_{\mathsf{c}}\mathsf{x} \otimes \mathsf{y} \\
&\delta_{\mathsf{c}}(\mathsf{Select\ xs}) &&= \mathsf{if}\ \ (\mathsf{c}, \mathsf{x}) \in \mathsf{xs\ then\ x\ \ else\ \ Void}
\end{aligned}$$

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. Map does not affect the derivative, since it is concerned with choices, not the final result. The derivative of Select 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 a derivative to the first generator in the pair, but what about when the first generator's language is nullable? One might imagine that the rule for Pair should look more like the one for "$\cdot$" in the definition for the derivative of a regular expression. Luckily, our normal form clears up the confusion. We can prove a lemma:

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

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 $\nu\mathcal{L}[\![\mathsf{x}]\!] = \varnothing$.
Since the smart constructor $\otimes$ never constructs a Pair with Pure on the left, it must be that $\nu\mathcal{L}[\![\mathsf{x}]\!] = \varnothing$.
Therefore, it must be the case that $\nu\mathcal{L}[\![\mathsf{Pair\ x\ y}]\!] = \varnothing$.

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

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

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

A simple corollary of this lemma is that if Pair x y is in normal form, $\mathcal{L}[\![\mathsf{x}]\!]$ is not nullable. In addition, the lemma means we can define *nullability* for free generators simply as:

$$\begin{aligned}
&\nu(\mathsf{Pure\ v}) &&= \{\mathsf{v}\} \\
&\nu\mathsf{g} &&= \varnothing
\end{aligned}$$

4.2.2 Consistency.

Before we prove our main theorem, one quick aside.

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

This is mostly a technical detail, but it means that anything we know about normalized generators also applies to derivatives of normalized generators.

With that in mind, 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 normalized free generators g and characters* c,

$$\delta_{\mathsf{c}}\mathcal{L}[\![g]\!] = \mathcal{L}[\![\delta_{\mathsf{c}}g]\!].$$

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

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

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 normalized, 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,

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

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

5 Generating Valid Results with Gradients

So far we have shown that factoring generators through the sequences of choices that they make leads to the natural abstraction of free generators. We have also established that free generators admit derivatives. This gives a symbolic way to compute a view of a generator after making a particular choice. We now put these theoretical ideas into practice—we present an algorithm that helps to solve the *valid generation problem* by guiding a generator to outputs that satisfy a computable precondition.

The basic intuition behind our algorithm 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 generation.

5.1 The Algorithm

With this intuition in mind, we present an algorithm that searches for valid results using repeated free generator *gradients* (i.e., derivatives with respect to every available choice). Given a free generator G in normal form and a validity predicate φ, the following algorithm produces a set of outputs O such that $\forall x \in O.\ \varphi(x)$.

Let $N \in \mathbb{N}$ be the *sample rate constant*. The algorithm proceeds as follows:

```
1. Set $g \leftarrow G$.
2. Set $\mathcal{V} \leftarrow \varnothing$.
3. If $\nu g \neq \varnothing$, then **return** $\nu g \cup \mathcal{V}$.
4. If $g = \text{Void}$, set $g \leftarrow G$.
5. Let $\nabla g = \{\delta_c g \mid c \in C\}$ be the gradient of $g$.
6. For each generator, $\delta_c g$, in $\nabla g$:
    a. If $\delta_c g = \text{Void}$, let $V = \varnothing$ and skip to Step c.
    b. Sample $x_1 \dots x_N$ from $\mathcal{G}[\![\delta_c g]\!]$ and let $V = \{x_j \mid \varphi(x_j)\}$.
    c. Let $f_c = |V|$, in other words, the number of valid samples.
    d. Set $\mathcal{V} \leftarrow \mathcal{V} \cup V$.
7. Randomly choose a derivative $\delta_c g$ with weight $f_c$. (If all weights $f$ are zero, choose uniformly.)
8. Set $g \leftarrow \delta_c g$.
9. Go to Step 3.
```

The intuition from earlier plays out in steps 5–7. 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.

One critical thing that makes this process work is the 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.

5.2 Experiments

In §2 we showed the results of our experiments using the gradient search algorithm. This section goes into more detail on the experimental setup.

Overall our goal was not to prove that this algorithm should be implemented as-is in any concrete tools, though we do believe a similar algorithm could be used in practice. Instead, our goal was to see how our implementation does in comparison to the baseline, and gain confidence that free generators and their derivatives are useful.

We ran one main experiment that uses two different benchmarks. The benchmarks are based on the example generators given in §3.2.4: the first benchmark, "BST," uses fgenTree to generate valid binary search trees; and the second, "STLC," uses fgenExpr to generate well-typed terms in a simply-typed lambda calculus. We assume readers are familiar with binary search trees and their invariants, but we provide a brief explanation of our STLC example in Appendix A.

In our experiment, we compared our gradient algorithm to simple rejection sampling. Concretely, we took our free generator g, interpreted it as $\mathcal{G}[\![g]\!]$, sampled from that generator, and ignored any results that were invalid. Admittedly, comparing against rejection sampling is not a terribly high bar, but our goals with these experiments were modest: we aimed to understand the pros and possible cons of our approach in a controlled environment.

We ran each of the two algorithms, "QuickCheck" (rejection sampling) and "Gradient" (our algorithm), for two minutes on a Macbook Pro with an M1 processor. We felt two minutes was appropriate for a small-scale experiment like this, as it is within the amount of time that a tester might reasonably spend on a particular property. For BST we used $N = 50$, and for STLC we used $N = 400$—in general, higher sample rates yield better performance more complex data structures, although this must be balanced by the added cost of sampling. As the algorithms ran, we kept track of every valid value that each generated and counted the number of *unique* values. We cared about unique values specifically, since there is no value in testing the same input more than once in a property-based testing scenario.

We do not repeat our charts here (Figures 4 and 5), but we can briefly review the results from Table 1:

	QuickCheck	Gradient
BST	12, 344	26, 527
STLC	147, 712	427, 005

It is clear that the Gradient algorithm produces significantly more unique, valid values than QuickCheck in the same amount of time, which is very promising. We expected to see this effect: the gradient algorithm is able to rule out choices that are unlikely to result in valid results and make choices that tend to work out well. The high incidence of *unique* values is a natural side-effect.

5.3 Modified Distributions

In §3.4, 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 the above algorithm to work for distribution-modified generators!

The algorithm requires that our generator structure admit three basic operations: δ_c so we can compute the gradients, $\mathcal{G}[\![\cdot]\!]$ so we can sample from the 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 just need definitions of δ_c and ν.

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

$$\delta_c(h, d) = (hc, d).$$

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 (d, g) of a Dist and a free generator to be:

$$\delta_c(d, g) = (\delta_c d, \delta_c g)$$

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:

$$\nu(d, g) = \nu g$$

With these definitions in hand, we can adapt our algorithm with essentially no changes!

6 Related Work

The valid generation problem need not be solved automatically. Libraries like *QuickCheck* [7] provide domain specific languages that make it easier to easier to write manual generators that produce valid inputs by construction.

Languages like *Luck* [8] provide a sort of 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. We ultimately still want a more automated solution.

When validity predicates are expressed as inductive relations, approaches like the one in *Generating Good Generators for Inductive Relations* [9] are extremely powerful. This is a sufficiently automatic solution to the valid generation problem—we simply choose to tackle situations with less structured validity predicates.

Some approaches have tried to use machine learning to generate valid inputs. *Learn&Fuzz* [5] generates valid PDF documents 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* [12] 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.

Claessen et al. present a structure that is superficially similar to our free generator structure, but with a different underlying approach. 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. 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.

Finally, while it does not explicitly aim to solve the valid generation problem, *Clotho* [4] is an interesting point in the generator design space.

7 Future Directions

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

7.1 Distribution Optimization

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 expand merge the theory of free generators with the emerging theory of *ungenerators* [6]. 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 one 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 §5.3, this would give a better starting point for generation with little extra work from the user.

7.2 Making Choices with Neural Networks

As our gradient algorithm 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 the gradient algorithm, this information is used once (to make the next choice) and then forgotten—what if there was a way to learn from it?

In theory, it should be possible to use pairs like this to train a recurrent neural network (RNN) to make choices that are similar to the ones made by the gradient algorithm. If this went well, we could run the gradient algorithm for a while, then train the model, and after that point only use the model to generate valid data. Staging things this way could potentially save a lot of time that is currently wasted by the constant sampling of derivative generators.

Of course, there are a number of hurdles that one would need to overcome to make this work properly. For a start, the right network architecture and hyper-parameters would need to be chosen, which might need to be different from one context to the next. If this is the case, the RNN approach would be significantly less useful from the perspective of the valid generation problem. In addition, the training process will be lossy, so extra care will need to be taken to make sure the network is still able to produce mostly valid data.

Still, if we can overcome these challenges, this approach could be extremely useful. 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] 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

[3] 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

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

[5] 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.

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

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

[8] 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

[9] 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.

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

[11] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching processes for QuickCheck generators. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 1–13.

https://doi.org/10.1145/3242744.3242747

[12] 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 STLC Details

$$\tau ::= \mathsf{int} \mid \tau_1 \to \tau_2$$
$$e ::= n \mid e_1 + e_2 \mid \lambda\tau.e \mid e_1\ e_2 \mid v$$

```
typeOf :: Expr → Maybe Type
typeOf expr = runReaderT (aux expr) []
  where
    aux (Int _) = return TInt
    aux (Plus e1 e2) = do
      TInt ← aux e1
      TInt ← aux e2
      return TInt
    aux (Lam t e) = do
      u ← local (t :) (aux e)
      return (Fun t u)
    aux (App e1 e2) = do
      (Fun t1 t2) ← aux e1
      u1 ← aux e2
      guard (t1 == u1)
      return t2
    aux (Var n) = do
      ctx ← ask
      if length ctx <= n then lift Nothing else return (ctx !! n)

hasType :: Expr → Bool
hasType = isJust . typeOf
```

B Proof of Theorem 3.4

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

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

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

Case g = Pure a.

$$\begin{aligned}\mathcal{P}[\![\text{Pure a}]\!]\ \langle\$\rangle\ \mathcal{C}[\![\text{Pure a}]\!] &= \text{pure (Just (a}, \varepsilon)) && \text{(by defn)}\\ &= (\lambda x \to \text{Just } (x, \varepsilon))\ \langle\$\rangle\ \mathcal{G}[\![\text{Pure a}]\!] && \text{(by defn)}\end{aligned}$$

Case g = Pair x y.

$$\begin{aligned}\mathcal{P}[\![\text{Pair x y}]\!]\ \langle\$\rangle\ \mathcal{C}[\![\text{Pair x y}]\!] &= \cdots && \text{(by defn)}\\ &= (\lambda x \to \text{Just } (x, \varepsilon))\ \langle\$\rangle\ \mathcal{G}[\![\text{Pair x y}]\!] && \text{(by defn)}\end{aligned}$$

Case ...

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