Compositional Reasoning for Generator-Based Shrinking (Extended Version)

ANONYMOUS AUTHOR(S)

Property-based testing (PBT) is a powerful tool for identifying software bugs. Users provide *generators*, which compute random test cases, and *shrinkers*, which compute a set of "smaller" test cases from an existing test case. Under *generator-based shrinking*, shrinkers are not specified separately, but derived from annotated generator specifications. Generator behaviour is not compositional in many frameworks that support generator-based shrinking (such as Hedgehog [20], falsify [6], RapidCheck [7], and jqwik [12]). In this paper, we address the problem of designing a language with generator-based shrinking that supports compositional reasoning. We propose an eDSL, halcheck, where users must manually annotate random choices with unique labels. In return, halcheck generators can be reasoned about compositionally. We justify our design by (1) showing that existing language designs do not — and cannot — support compositional reasoning and (2) the labelling requirement has negligible overhead in practice.

ACM Reference Format:
Anonymous Author(s). 2025. Compositional Reasoning for Generator-Based Shrinking (Extended Version). 1, 1 (March 2025), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Property-based testing (PBT) [3] is a powerful tool for identifying software bugs. Users provide an executable specification — or *property* — to a PBT framework, which automatically checks the specification against a system under test (SUT) over a wide range of randomly generated inputs (or *test cases*). If the framework finds a *counterexample* — a test case that causes the SUT to violate the specification — then the framework computes and reports a minimal counterexample. A complete overview is given in Figure 1.

A property consists of three components: (1) *generators*, which compute random test cases, (2) an *oracle*, which decides if a test case is a counterexample, and (3) *shrinkers*, which compute a set of "smaller" test cases from an existing test case. These components are integral to the operation of a PBT framework, which consists of two stages: *testing* and *shrinking*. The testing stage is responsible for identifying a counterexample, and operates using the generators and oracle of the property. The shrinking stage performs counterexample minimization, and operates using the shrinkers and oracle.

The testing stage proceeds by constructing a random test case using the generators, then checking if the test case is a counterexample using the oracle. If the test case is not a counterexample, the testing process restarts with a different test case. Otherwise, the counterexample is passed along to the shrinking stage.

The shrinking stage is essential for the usability of property-based testing. Without it, counterexamples can be arbitrarily large, which complicates debugging. Shrinking operates as follows: (1) starting from the previously discovered counterexample, the shrinkers compute a set of "smaller"

ACM XXXX-XXXX/2025/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

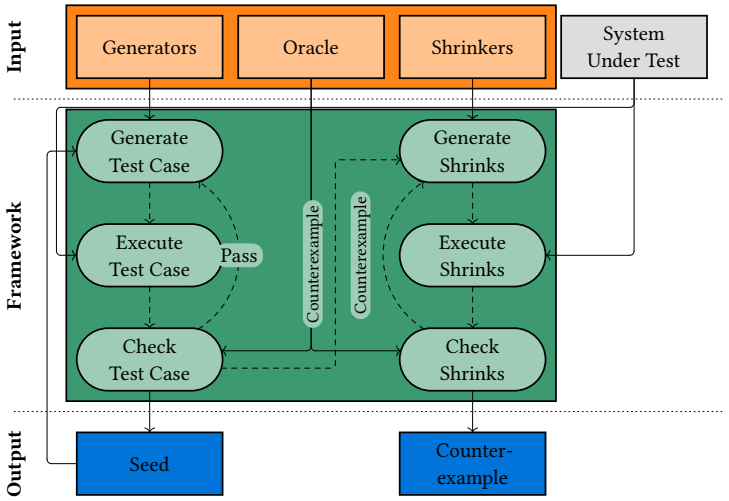


Fig. 1. Property-Based Testing Overview.

```
1  coin :: Gen Bool
2  coin = do x ← prune bool
3            y ← prune bool
4            return (x == y)
```

Fig. 2. Generator Example.

test cases, called *shrinks*, then (2) the oracle determines whether any of the shrinks are counterexamples. If so, the shrinking process restarts using the smaller counterexample. Otherwise, the current counterexample is deemed minimal and reported to the user.

Generators are specified using an embedded domain-specific language (eDSL) provided by the PBT framework. These languages are essentially probabilistic programming languages and hence users view generators as probability distributions. For example, consider the program given in Figure 2, written using the Hedgehog framework [20] for Haskell. Concretely, the `coin` generator computes two random booleans, `x` and `y` (lines 3 and 4), by invoking a subgenerator `prune bool` which performs a fair coin flip. The `coin` generator then computes whether `x` and `y` are equal (line 5). Abstractly, `coin` represents a probability distribution that assigns 0.5 to both `True` and `False`.

Shrinkers are also specified using an eDSL provided by the PBT framework. The QuickCheck [3] framework which popularized PBT employs *manual shrinking*, where shrinkers and generators are specified separately. Shrinkers are surprisingly difficult to write manually, thus subsequent work such as Hedgehog [20] employs *generator-based shrinking*, where shrinkers are derived from annotated generator specifications. Under generator-based shrinking, generators represent distributions of labelled trees, where labels lower in the tree are considered "smaller". While most languages with generator-based shrinking allow users to influence shrinker behaviour [6, 20], others — such as hypothesis [13] — rely solely upon built-in heuristics. This lack of *user-controlled shrinking* saves users from having to think about how to shrink test cases, but inhibits correct and efficient shrinking since the correct notion of "smaller" test case is application-dependent. Furthermore, the heuristics used by these frameworks are unpredictable and can change when the framework is updated [6]. Thus this work focuses on user-controlled generator-based shrinking.

Frameworks	QuickCheck [3]	falsify [6], Hedgehog [20], RapidCheck [7], jqwik [12]	hypothesis [13]
Generator-Based Shrinking	✗	✓	✓
User-Controlled Shrinking	✓	✓	✗
Compositional Reasoning	✓	✗	✓

Table 1. Existing Frameworks and Supported Features.

```
1 context :: Gen Bool → Gen Bool
2 context m = do
3   x ← shrink (\b → if b then [False] else []) (return True)
4   if x then coin else m
```

Fig. 3. Non-Compositionality Example.

Since generators can be arbitrarily complex [10, 16], it is desirable to construct and reason about them in a *compositional* manner. In particular, a generator's distribution should be a pure function of its subgenerator's distributions. For example, we can determine `coin`'s distribution simply by knowing `prune bool`'s distribution. Compositional reasoning holds in frameworks without generator-based shrinking (such as QuickCheck [3]) and frameworks without user-controlled shrinking (such as hypothesis [13]). However, compositional reasoning does not hold in any framework that supports both, such as Hedgehog [20], falsify [6], RapidCheck [7], or jqwik [12]. We summarize existing frameworks and their support for generator-based shrinking, user-controlled shrinking, and compositional reasoning in Table 1.

In this paper, we address the problem of designing a language featuring user-controlled generator-based shrinking and compositional reasoning. We propose a new eDSL, halcheck, where users must manually annotate random choices with unique labels. In exchange for this modest overhead, halcheck generators can be constructed and reasoned about compositionally.

Motivating Example. To illustrate how compositional reasoning fails in a language with user-controlled generator-based shrinking, consider the function `context` given in Figure 3. The `context` function takes in a generator (`m`) and returns another generator. The returned generator first generates a boolean `x` (lines 3 to 5). This boolean is always `True` during the testing stage, but can change to `False` during the shrinking stage. If `x` is `True`, then `coin` is invoked, and otherwise `m` is invoked.

Compositionality implies that `context coin` and `context (prune bool)` should represent the same distribution (since `coin` and `prune bool` represent the same distribution.) This is not the case in practice: `context coin` produces the trees given in Figure 4a, while `context (prune bool)` produces the trees given in both Figures 4a and 4b. This difference indicates that `context` can somehow differentiate between `context coin` and `prune bool`, despite them representing identical distributions.

Contributions. The contributions of this work are as follows:

(a) Expected Trees (b) Additional Trees

Fig. 4. Trees produced by `context` (see example in Figure 3).

- We define a new language, halcheck, which features an additional *labelling* operator. We show that halcheck supports compositional reasoning modulo some minor constraints on the surrounding context. We also provide a complete set of rules for determining whether two halcheck generators have identical distributions.
- To justify our additional constraints, we show that unconstrained compositional reasoning is not possible in a language with user-controlled generator-based shrinking.
- To justify the labelling requirement, we provide a translation from generators defined in an existing language to halcheck generators. We prove that the translation is sound and results in only a linear increase in program size.
- We implement halcheck as a simple modification of the Hedgehog library for Haskell. We evaluate the effects of halcheck's design on generator construction and performance using a data set drawn from the literature and industry.

Organization. The structure of this paper is as follows. In Section 2, we introduce background material, which includes fixing basic notation and reviewing the QuickCheck and Hedgehog languages and their (non-)properties. In Section 3, we present our proof that unconstrained compositional reasoning is not possible in a language with user-controlled generator-based shrinking. In Section 4, we present the language halcheck. In Section 5, we describe our implementation and evaluation of halcheck. In Section 6, we describe and compare related work. In Section 7, we conclude and describe future work.

2 Background

2.1 Notation

Sets. The class of all sets is denoted by Set. The set of natural numbers is denoted by $\mathbb{N}$, the set of rational numbers is denoted by $\mathbb{Q}$, and the set of real numbers is denoted by $\mathbb{R}$. The set of booleans is defined as $\mathbb{B} = \{0, 1\}$. The set of rational weights is defined as $\mathbb{W} = \mathbb{Q} \cap \mathbb{W}$. The cartesian product of sets A and B is written as $A \times B$. For any pair (a, b) we define $\mathrm{fst}\,(a, b) = a$ and $\mathrm{snd}\,(a, b) = b$. The set of all subsets of A is denoted by 2^A.

Functions. We use juxtaposition (e.g., $f\ a$) to denote function application. The composition of two functions $f : B \to C$ and $g : A \to B$ is denoted by $f \circ g$. The notation $a \mapsto \ldots$ denotes an unnamed function with parameter a. We also write functions using blanks for arguments (e.g., $(a + -) = (b \mapsto a + b)$). The pre-image of a function $f : A \to B$ is given by $f^{-1} : 2^B \to 2^A$. Given $f : A \to B$, $a \in A$, and $b \in B$, we define $f[a \mapsto b]$ as

$$f[a \mapsto b]\ c = \begin{cases} b & \text{if } a = c \\ f\ c & \text{otherwise.} \end{cases}$$

The type of dependent functions from A to B (where a variable $a \in A$ may appear in B) is denoted by $\prod_{a \in A} B$.

Words. The set of finite words over a set A is written as A^*. A list with elements $a_0, \ldots, a_{n-1}$ is written as $[a_0, \ldots, a_{n-1}]$. The concatenation of two words as and bs is denoted by $as +\!\!+ bs$. The length of a word as is denoted by $|as|$. The ith element of as is written as as_i, and as_A denotes the substring of as that only contains the elements indexed by elements of $A \subseteq \mathbb{N}$ (e.g., $as_{[0..5)}$ denotes the prefix containing the first five elements of as.) The expression $a :: as$ denotes the word $[a, as_0, \ldots, as_{|as|-1}]$. We define $\mathsf{map} : (A \to B) \to A^* \to B^*$ as $\mathsf{map}\, f\, [a_0, \ldots, a_n] = [f\, a_0, \ldots, f\, a_n]$.

2.2 Probability

In this section, we review the basics of probability and measure theory.

Measurable Sets and Functions. A σ-algebra on a set A is a set of sets $\Sigma \subseteq 2^A$ that includes A and is closed under complementation and countable union. Note that 2^A is a σ-algebra, known as the discrete σ-algebra. We assign each set A a canonical σ-algebra Σ_A which, unless otherwise specified, is the discrete σ-algebra. A set $X \subseteq A$ is measurable iff $X \in \Sigma_A$, and a function $f : A \to B$ is measurable iff $f^{-1}\, X$ is measurable for all for all measurable $X \subseteq B$.

Probability Measures and Random Variables. A probability measure on a set A is a function $\mathbb{P}_A : \Sigma_A \to \mathbb{W}$ satisfying $\mathbb{P}_A\, A = 1$ and $\mathbb{P}_A\, (\bigcup \mathbf{A}) = \sum_{X \in \mathbf{A}} \mathbb{P}_A\, X$ for all countable collections of pairwise disjoint measurable sets $\mathbf{A} \subseteq \Sigma_A$. A probability space is any set A equipped with a probability measure $\mathbb{P}_A$. If A is a probability space, then a measurable function $f : A \to B$ is called a random variable. For a random variable $f : A \to B$, we define the following shorthands within the context of a call to $\mathbb{P}_A$:

$$(f \in X) \quad = \quad f^{-1}\, X \qquad \text{where} \qquad X \subseteq A$$
$$(f = a) \quad = \quad f^{-1}\, \{a\} \qquad \text{where} \qquad a \in A$$

In general, we lift arbitrary operations to random variables in a pointwise manner within the context of a call to $\mathbb{P}_A$, e.g.

$$\mathbb{P}_A\, (f + g \in X) \quad = \quad \mathbb{P}_A\, ((\sigma \mapsto f\, \sigma + g\, \sigma) \in X) \qquad \text{where} \qquad f, g : \mathsf{Seed} \to \mathbb{R}.$$

Two sets $X, Y \subseteq A$ are independent (written $X \perp\!\!\!\perp Y$) iff $\mathbb{P}_A\, (X \cap Y) = \mathbb{P}_A\, (X) \cdot \mathbb{P}_A\, (Y)$, and two random variables $f, g : A \to B$ are independent if $f^{-1}\, X \perp\!\!\!\perp g^{-1}\, Y$ for all measurable $X, Y \subseteq B$.

Seeds. Following [5, 21, 23, 24], we fix a probability space Seed (called an *entropy space*) of seeds with a non-discrete canonical σ-algebra. From this point on we are only concerned with the measure $\mathbb{P}_{\mathsf{Seed}}$ therefore omit the subscript. The set Seed comes equipped with a family of measurable sampling functions $\psi_p : \mathsf{Seed} \to \mathbb{B}$ for all $p \in \mathbb{W}$ and two measurable projection functions $\pi_L, \pi_R : \mathsf{Seed} \to \mathsf{Seed}$ such that (1) the projection functions are measure-preserving, i.e. $\mathbb{P}\, (\pi_L^{-1}\, A) = \mathbb{P}\, (\pi_R^{-1}\, A) = \mathbb{P}\, A$ for all $A \in \Sigma_{\mathsf{Seed}}$, (2) the projection functions are independent, i.e. $\pi_L \perp\!\!\!\perp \pi_R$, (3) different samples are independent, i.e. $\psi_p \perp\!\!\!\perp \psi_q$ for all $p \neq q$, and (4) $\mathbb{P}\, (\psi_p = 1) = p$ for all $p \in \mathbb{W}$. We assume every countable set A is equipped with a canonical injection $\#- : A \to \mathbb{N}$. Then for every countable A and $a \in A$ we define $\pi_a : \mathsf{Seed} \to \mathsf{Seed}$ as

$$\pi_a = \pi_L \circ \overbrace{\pi_R \circ \cdots \circ \pi_R}^{\#a \text{ times}}.$$

Note that any pair of projection functions on different values are independent, i.e.

$$\pi_a \perp\!\!\!\perp \pi_b \iff a \neq b.$$

2.3 Metalanguage

In this section we define a simple metalanguage which serves as a general framework for studying languages with generic effects [17–19] such as probabilistic choice and shrinking. Instances of the language are constructed by providing a *signature* (which specifies a set of language-specific operations, see Definition 2.2) and an *interpretation* (which specifies the meaning of language-specific operations, see Definition 2.5.)

Definition 2.1 (Types). The sets of *value types* Type_v and *computation types* Type_c are generated by

$$\tau_v ::= \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nat} \qquad\qquad \tau_c ::= \mathsf{Gen}\ \tau_v.$$

We define the set of all types as $\mathsf{Type} = \mathsf{Type}_v \cup \mathsf{Type}_c$.

The set of value types consists of the singleton type (Unit), the type of booleans (Bool) and the type of natural numbers (Nat). The computation type $\mathsf{Gen}\ \tau$ represents generators which produce elements of the value type τ. Value and computation types are stratified so that computation types may not be nested.

Definition 2.2 (Signature). A signature is a set of function symbols of the form $f : \tau_1 \times \cdots \times \tau_n \Rightarrow \tau$, for $\tau_1, \ldots, \tau_n, \tau \in \mathsf{Type}_v$.

A signature describes a language-specific set of effectful operations. For example, the signature

$$\Sigma_{\mathsf{St}} = \{\mathsf{get} : \ \Rightarrow \mathsf{Nat}, \mathsf{put} : \mathsf{Nat} \Rightarrow \mathsf{Unit}\}$$

describes operations on some mutable Nat-valued state, where (1) get denotes an operation which takes no arguments and returns the current value of the state, and (2) put denotes an operation which overwrites the current value of the state with its argument. A function symbol may only include value types, i.e. we exclude higher-order generators.

Definition 2.3 (Terms). Given a signature Σ, the set of Σ-terms (denoted by $\mathsf{Term}\ \Sigma$) is generated by

$$\begin{aligned} t ::= {} & x \mid \mathsf{return}\ x \mid \mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ t_2 \mid f\ t_1 \ldots t_n \mid () \mid \mathsf{true} \\ & \mid \mathsf{false} \mid \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \mid n \mid t_1 + t_2 \mid t_1 \cdot t_2 \mid t_1 = t_2 \end{aligned}$$

where variables (e.g. x) are sourced from a countably infinite set of names Var.

Definition 2.3 defines constructs common to all languages we study in this work. Such constructs include variables (x), side-effect-free computations (return), sequential composition (let), the singleton value ($()$), natural numbers (n), booleans (true and false), as well as standard operations on these types. For simplicity, we omit general recursion constructs. In addition to the standard constructs, the set of terms also includes function symbol invocations of the form $f\ t_1 \ldots t_n$. For example, the set of terms generated by Σ_{St} includes terms such as 'get' and 'put $(3 + 2)$'. We define shorthands for some common operations:

$$t_1 \wedge t_2 = \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ \mathsf{false} \qquad t_1 \vee t_2 = \mathsf{if}\ t_1\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ t_2 \qquad \neg t = \mathsf{if}\ t\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}$$

Definition 2.4 (Environments and Typing). The set of environments is defined as the set $\mathsf{Env} = \mathsf{Var} \to \mathsf{Type}_v$ of all functions assigning variables to value types. The typing relation

$$- \vdash - : - \subseteq \mathsf{Env} \times \mathsf{Term}\ \Sigma \times \mathsf{Type}$$

is the smallest relation satisfying the following rules:

$$\frac{}{\Gamma \vdash x : \Gamma\, x} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{return}\, t : \mathsf{Gen}\, \tau} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Gen}\, \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash t_2 : \mathsf{Gen}\, \tau_2}{\Gamma \vdash \mathsf{let}\, x \leftarrow t_1\, \mathsf{in}\, t_2 : \mathsf{Gen}\, \tau_2}$$

$$\frac{(f : \tau_1 \times \cdots \times \tau_n \Rightarrow \tau) \in \Sigma \qquad \Gamma \vdash t_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash f\, t_1 \ldots t_n : \mathsf{Gen}\, \tau} \qquad \frac{}{\Gamma \vdash () : \mathsf{Unit}}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Bool} \qquad \Gamma \vdash t_2 : \tau \qquad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathsf{if}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3 : \tau} \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash n : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Nat} \qquad \Gamma \vdash t_2 : \mathsf{Nat}}{\Gamma \vdash t_1 + t_2 : \mathsf{Nat}} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Nat} \qquad \Gamma \vdash t_2 : \mathsf{Nat}}{\Gamma \vdash t_1 \cdot t_2 : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau \qquad \tau \in \mathsf{Type}_v}{\Gamma \vdash t_1 = t_2 : \mathsf{Bool}}$$

If t has no free variables, we write $t : \tau$ in place of $\forall \Gamma, \Gamma \vdash t : \tau$, since the environment has no bearing on whether t is typed.

Not every term is sensible (e.g. consider '() + true'). The typing relation $- \vdash - : -$ describes exactly which terms are sensible within a given environment. Note that the type argument is unique, i.e. $\Gamma \vdash t : \tau_1$ and $\Gamma \vdash t : \tau_2$ implies $\tau_1 = \tau_2$.

Definition 2.5 (Interpretations and Semantics). The semantics of value types and environments are given by the function $[\![-]\!] : \mathsf{Type}_v \cup \mathsf{Env} \to \mathsf{Set}$, defined as follows:

$$[\![\mathsf{Unit}]\!] = \{0\} \qquad [\![\mathsf{Bool}]\!] = \mathbb{B} \qquad [\![\mathsf{Nat}]\!] = \mathbb{N} \qquad [\![\Gamma]\!] = \prod_{x \in \mathsf{Var}} [\![\Gamma\, x]\!]$$

An interpretation of a signature Σ (or Σ-interpretation) is a tuple $F = (\mathsf{Gen}^F, \mathsf{return}^F, \mathsf{let}^F, (f^F)_{f \in \Sigma})$ satisfying

$$\mathsf{Gen}^F : \mathsf{Set} \to \mathsf{Set} \qquad \mathsf{return}^F : A \to \mathsf{Gen}^F\, A$$
$$\mathsf{let}^F : (A \to \mathsf{Gen}^F\, B) \to \mathsf{Gen}^F\, A \to \mathsf{Gen}^F\, B \qquad f^F : [\![\tau_1]\!] \to \cdots \to [\![\tau_n]\!] \to \mathsf{Gen}\, [\![\tau]\!]$$

for each $A, B \in \mathsf{Set}$ and function symbol $(f : \tau_1 \times \cdots \times \tau_n \Rightarrow \tau) \in \Sigma$. The interpretation F gives rise to a type semantics $[\![-]\!]^F : \mathsf{Type} \to \mathsf{Set}$, defined as

$$[\![\mathsf{Gen}\, \tau]\!]^F = \mathsf{Gen}^F\, [\![\tau]\!] \qquad [\![\tau]\!]^F = [\![\tau]\!] \quad (\text{where } \tau \in \mathsf{Type}_v),$$

and a term semantics $[\![- \vdash -]\!]^F$ satisfying $[\![\Gamma \vdash t]\!]^F : [\![\Gamma]\!] \to [\![\tau]\!]$ for all $\Gamma \vdash t : \tau$, defined as follows:

$$[\![\Gamma \vdash x]\!]^F\, \rho = \rho\, x \qquad [\![\Gamma \vdash \mathsf{return}\, t]\!]^F\, \rho = \mathsf{return}^F\, ([\![\Gamma \vdash t]\!]^F\, \rho)$$

$$[\![\Gamma \vdash \mathsf{let}\, x \leftarrow t_1\, \mathsf{in}\, t_2]\!]^F\, \rho = \mathsf{let}^F\, (a \mapsto [\![\Gamma[x \mapsto a] \vdash t_2]\!]^F\, \rho[x \mapsto a])\, ([\![\Gamma \vdash t_1]\!]^F\, \rho)$$

$$[\![\Gamma \vdash f\, t_1 \ldots t_n]\!]^F\, \rho = f^F\, ([\![\Gamma \vdash t_1]\!]^F\, \rho) \ldots ([\![\Gamma \vdash t_n]\!]^F\, \rho) \qquad [\![\Gamma \vdash ()]\!]^F\, \rho = 0 \qquad [\![\Gamma \vdash \mathsf{true}]\!]^F\, \rho = 1$$

$$[\![\Gamma \vdash \mathsf{false}]\!]^F\, \rho = 0 \qquad [\![\Gamma \vdash \mathsf{if}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3]\!]^F\, \rho = \begin{cases} [\![\Gamma \vdash t_2]\!]^F\, \rho & \text{if } [\![\Gamma \vdash t_1]\!]^F\, \rho = 1 \\ [\![\Gamma \vdash t_3]\!]^F\, \rho & \text{otherwise} \end{cases}$$

$$[\![\Gamma \vdash n]\!]^F \rho = n \qquad [\![\Gamma \vdash t_1 + t_2]\!]^F \rho = [\![\Gamma \vdash t_1]\!]^F \rho + [\![\Gamma \vdash t_2]\!]^F \rho$$

$$[\![\Gamma \vdash t_1 \cdot t_2]\!]^F \rho = [\![\Gamma \vdash t_1]\!]^F \rho \cdot [\![\Gamma \vdash t_2]\!]^F \rho \qquad [\![\Gamma \vdash t_1 = t_2]\!]^F \rho = \begin{cases} 1 & \text{if } [\![\Gamma \vdash t_1]\!]^F \rho = [\![\Gamma \vdash t_2]\!]^F \rho \\ 0 & \text{otherwise} \end{cases}$$

If $t : \tau$ has no free variables, then we define $[\![t]\!]^F = [\![\Gamma \vdash t]\!]^F \rho$ for an arbitrarily chosen $\Gamma \in \mathsf{Env}$ and $\rho \in [\![\Gamma]\!]$, as neither Γ nor ρ influence the semantics of t.

An interpretation assigns language-specific semantics to the type of generators (Gen^F), the return operator (return^F), sequential composition (let^F), and each function symbol (f^F). The remaining (non-Gen) types and terms are assigned standard semantics as functions from variable assignments to suitable domains. As an example, we give the standard interpretation St for Σ_{St}, which interprets computations as state transition functions:

$$\mathsf{Gen}^{\mathrm{St}}\, A = \mathbb{N} \to A \times \mathbb{N} \qquad \mathsf{return}^{\mathrm{St}}\, x = n \mapsto (x, n)$$
$$\mathsf{let}^{\mathrm{St}}\, f\, m = n \mapsto f\, a\, n' \text{ where } (a, n') = m\, n \qquad \mathsf{get}^{\mathrm{St}} = n \mapsto (n, n)$$
$$\mathsf{put}^{\mathrm{St}}\, n = n' \mapsto (0, n)$$

Definition 2.6 (Semantic Value Equivalence). Two terms $\Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \tau$ (for $\tau \in \mathsf{Type}_v$) are semantically equivalent (written $\Gamma \vdash t_1 = t_2$) iff $[\![\Gamma \vdash t_1]\!] = [\![\Gamma \vdash t_2]\!]$.

Two value-typed terms are semantically equivalent if they denote the same value, regardless of their syntactic representation. For example, $\Gamma \vdash x + 0 = x$ holds for all environments $\Gamma \in \mathsf{Env}$ satisfying $\Gamma\, x = \mathsf{Nat}$ — even though $x + 0$ and x are not syntactically identical — since $x + 0$ and x denote the same value.

Definition 2.7 (Semantic Computation Equivalence). Let F be a Σ-interpretation. A semantic equivalence relation on F is an equivalence relation $\approx \subseteq (A \to \mathsf{Gen}^F\, B)^2$ which satisfies the following rules:

$$\textsc{Return-Cong}\ \frac{\Gamma \vdash t = t'}{\Gamma \vdash \mathsf{return}\, t \approx \mathsf{return}\, t'} \qquad \textsc{Let-Cong}\ \frac{\Gamma \vdash t_1 \approx t_1' \qquad \Gamma[x \mapsto \tau] \vdash t_2 \approx t_2'}{\Gamma \vdash (\mathsf{let}\, x \leftarrow t_1\, \mathsf{in}\, t_2) \approx (\mathsf{let}\, x \leftarrow t_1'\, \mathsf{in}\, t_2')}$$

$$\textsc{If-Cong}\ \frac{\Gamma \vdash t_1 = t_1' \qquad \Gamma \vdash t_2 \approx t_2' \qquad \Gamma \vdash t_3 \approx t_3'}{\Gamma \vdash (\mathsf{if}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3) \approx (\mathsf{if}\, t_1'\, \mathsf{then}\, t_2'\, \mathsf{else}\, t_3')}$$

Two terms $\Gamma \vdash t_1 : \mathsf{Gen}\, \tau$ and $\Gamma \vdash t_2 : \mathsf{Gen}\, \tau$ are semantically equivalent with respect to $\approx$ (written $\Gamma \vdash t_1 \approx^F t_2$) iff $[\![\Gamma \vdash t_1]\!]^F \approx [\![\Gamma \vdash t_2]\!]^F$. We omit the superscript when F can be unambiguously determined from the equivalence relation $\approx$. We also write $t_1 \approx^F t_2$ in place of $\forall \Gamma.\, \Gamma \vdash t_1 \approx t_2$, where Γ ranges over all environments satisfying $\Gamma \vdash t_1 : \mathsf{Gen}\, \tau$ and $\Gamma \vdash t_2 : \mathsf{Gen}\, \tau$ for some $\tau \in \mathsf{Type}_v$.

A semantic equivalence relation defines which computation-typed terms are considered to have the same behaviour, regardless of their syntactic representation. Rules Return-Cong to If-Cong are called the *congruence rules*. The congruence rules imply that the semantic equivalence two terms holds independent of the context they appear, and hence these rules are essential for compositional reasoning. For example, the equivalence $(\mathsf{let}\, x \leftarrow \mathsf{get}\, \mathsf{in}\, \mathsf{put}\, x) =^{\mathrm{St}} \mathsf{return}\, ()$ holds since, intuitively, assigning some mutable state to its own value is a no-op.

2.4 QuickCheck

In this section, we review the QuickCheck language for generator specification [3]. We frame the QuickCheck language within our metalanguage framework by providing an appropriate signature and interpretation. At a high level, QuickCheck is a probabilistic programming language and generators are specified using probabilistic operators.

Definition 2.8. The QuickCheck signature is defined as

$$\Sigma_Q = \{\mathsf{flip}_p : \Rightarrow \mathsf{Bool} \mid p \in \mathbb{W}\}$$

and the QuickCheck interpretation Q is defined as follows:

$$\mathsf{Gen}^Q\, A = \mathsf{Seed} \to A \qquad \mathsf{return}^Q\, a = \sigma \mapsto a$$

$$\mathsf{let}^Q\, f\, m = \sigma \mapsto f\,(m\,(\pi_L\,\sigma))\,(\pi_R\,\sigma) \qquad \mathsf{flip}_p^Q = \psi_p$$

In addition to the common return and let operators, the QuickCheck language features a biased coin flip operator flip_p. Intuitively, the flip_p operator produces true with probability p and false with probability $1 - p$. Semantically, generators are represented as random variables (i.e. functions from seeds to values), mirroring the actual implementation of QuickCheck. The core operators are implemented as follows:

- flip_p samples the seed it is given to produce a random boolean.
- $\mathsf{return}\ a$ ignores its input and always returns a.
- $\mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ t_2$ assigns x to a value produced by t_1, then returns the value produced by t_2. The input seed is split between t_1 and t_2 using π_L and π_R, which effectively ensures that the results of t_1 and t_2 are uncorrelated.

As a shorthand, we define a binary probabilistic choice operator $- \oplus_p -$ as

$$t_1 \oplus_p t_2 = \mathsf{let}\ b \leftarrow \mathsf{flip}_p\ \mathsf{in}\ \mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2.$$

Intuitively, $t_1 \oplus_p t_2$ behaves like t_1 with probability p and behaves like t_2 with probability $1 - p$.

Example 2.9. To demonstrate the QuickCheck language, we define a family of generators $\mathsf{nat}^i \in \mathsf{Term}\ \Sigma_Q$ recursively as

$$\mathsf{nat}^0 = \mathsf{return}\ 0 \qquad \mathsf{nat}^{i+1} = \mathsf{let}\ n \leftarrow \mathsf{nat}^i\ \mathsf{in}\ \mathsf{let}\ b \leftarrow \mathsf{flip}_{1/2}\ \mathsf{in}\ \mathsf{return}\ (2 \cdot n + \mathsf{if}\ b\ \mathsf{then}\ 1\ \mathsf{else}\ 0)$$

Intuitively, nat^i computes a random integer in the range $[0\,..\,2^i)$ by combining i random bits.

2.5 Hedgehog

In this section, we review the Hedgehog language for generator-based shrinking specification [20]. As with QuickCheck, we provide an appropriate signature and interpretation. At a high level, the Hedgehog language extends the QuickCheck language with an extra operation for specifying shrinking behaviour.

Definition 2.10 (Shrink Tree). The set $\mathsf{Tree}\ A$ of *shrink trees* is the smallest set satisfying

$$\mathsf{Tree}\ A = A \times (\mathsf{Tree}\ A)^*$$

and let^T, the sequential composition operator for trees, is defined recursively as

$$\begin{aligned}&\mathsf{let}^\mathsf{T} : (A \to \mathsf{Tree}\ B) \to \mathsf{Tree}\ A \to \mathsf{Tree}\ B\\ &\mathsf{let}^\mathsf{T}\ f\ (a, us) = (b, vs +\!\!+ \mathsf{map}\ (\mathsf{let}^\mathsf{T}\ f)\ us)\\ &\quad \text{where } (b, vs) = f\ a.\end{aligned}$$

For a tree $u = (a, us) \in \mathsf{Tree}\, A$, the value $a \in A$ is called the *label* of u, and $us \in (\mathsf{Tree}\, A)^*$ denotes the *children* of u. The set of all labels in a tree u is denoted by $\mathsf{labels}\, u$.

A shrink tree represents a value and its shrinks, its shrink's shrinks, etc. The topmost value is produced during the testing stage, while the lower values are produced during the shrinking stage. In practice, shrink trees may be infinite or contain partial values. For simplicity we assume shrink trees are finite and contain no partial values.

Definition 2.11. The Hedgehog signature is defined as

$$\Sigma_{\mathcal{H}} = \Sigma_Q \cup \{\mathsf{shrinkNat} : \mathsf{Nat} \Rightarrow \mathsf{Nat}\}$$

and the Hedgehog interpretation $\mathcal{H}$ is given as follows:

$$\mathsf{Gen}^{\mathcal{H}}\, A = \mathsf{Seed} \to \mathsf{Tree}\, A \qquad \mathsf{return}^{\mathcal{H}}\, a = \sigma \mapsto (a, [])$$
$$\mathsf{let}^{\mathcal{H}}\, f\, m = \sigma \mapsto \mathsf{let}^{\mathsf{T}}\, (a \mapsto f\, a\, (\pi_L\, \sigma))\, (m\, (\pi_R\, \sigma)) \qquad \mathsf{flip}_p^{\mathcal{H}} = \sigma \mapsto (\psi_p\, \sigma, [])$$
$$\mathsf{shrinkNat}^{\mathcal{H}}\, n = \sigma \mapsto (0, [(1, []), \ldots, (n, [])])$$

Syntactically, the Hedgehog language extends the QuickCheck language with an additional shrinkNat operator, which is used to express shrinking behaviour. Semantically, the primary difference between Hedgehog and QuickCheck is that Hedgehog interprets generators as random variables over shrink trees instead of singular values. Hedgehog's core operators are implemented as follows:

- $\mathsf{shrinkNat}\, n$ always returns 0 during testing and specifies the values $[1\,..\,n]$ as shrinks.
- flip_p returns 1 with probability p and returns 0 with probability $1 - p$ (just like QuickCheck), and specifies no shrinks.
- $\mathsf{return}\, a$ represents a generator which returns a with probability 1, and specifies no shrinks.
- $\mathsf{let}\, x \leftarrow t_1\, \mathsf{in}\, t_2$ behaves as it does under the QuickCheck interpretation during testing, but evaluates t_2 for every value in the tree produced by t_1. This gives a tree of trees, which is flattened for the final result. In practice, this process is performed lazily so that the entire tree is not computed and stored in memory at once.

As a shorthand, we define an n-ary shrinking operator $\mathsf{shrink}\, t_0\, t_1 \ldots t_n$ (for $n \in \mathbb{N}$) as

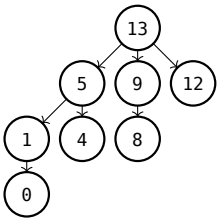
$$\begin{aligned}\mathsf{shrink}\, t_0\, t_1 \ldots t_n = \mathsf{let}\, i \leftarrow \mathsf{shrinkNat}\, n\, \mathsf{in}\, &\mathsf{if}\, i = n\, \mathsf{then}\, t_n\, \mathsf{else}\\ &\ldots\\ &\mathsf{if}\, i = 1\, \mathsf{then}\, t_1\, \mathsf{else}\\ &t_0\end{aligned}$$

Intuitively, $\mathsf{shrink}\, t_0\, t_1 \ldots t_n$ represents a generator that behaves like t_0 during the testing stage, but may behave like any of t_i during the shrinking stage.

Example 2.12. To demonstrate the Hedgehog language, we define a family of generators $\mathsf{snat}^i \in \mathsf{Term}\, \Sigma_{\mathcal{H}}$ recursively in Figure 5a. The term snat^i behaves similarly to nat^i: they both produce a random natural number in the range $[0\,..\,2^i)$. However, snat^i also specifies some shrinking behaviour. Every result produced by sflip_p may shrink to false, which effectively changes one bit snat's output to zero. For example, if snat^4 produces 13, then one possible sequence of shrinks is 13 (1101) → 5 (0101) → 4 (0100). Note that no further shrinks are possible because, in Hedgehog, shrinking does not backtrack: once a later bit is flipped, earlier ones may no longer be flipped. The full shrink tree is given in Figure 5b.

$$
\begin{aligned}
\mathsf{snat}^0 &= \mathsf{return}\ 0 \\
\mathsf{snat}^{i+1} &= \mathsf{let}\ n \leftarrow \mathsf{snat}^i\ \mathsf{in} \\
&\quad\ \mathsf{let}\ b \leftarrow \mathsf{sflip}_{1/2}\ \mathsf{in} \\
&\quad\quad \mathsf{return}\ (2 \cdot n + \mathsf{if}\ b\ \mathsf{then}\ 1\ \mathsf{else}\ 0) \\
\mathsf{sflip}_p &= \mathsf{shrink}\ (\mathsf{return}\ \mathsf{true})\ (\mathsf{return}\ \mathsf{false}) \\
&\quad\ \oplus_p\ \mathsf{return}\ \mathsf{false}.
\end{aligned}
$$

(a) Definition of snat^i.



(b) A shrink tree produced by snat^4.

Fig. 5. Hedgehog example.

2.6 Monads

In this section, we review the notion of monadic interpretations (in the sense of Moggi [15]), which characterizes the expected behaviour of sequential composition. We also review some additional properties which characterize the expected behaviour of probabilistic choice and shrinking.

Definition 2.13 (Monadic Interpretation). An interpretation F is monadic up to an equivalence relation $\approx\ \subseteq (A \to \mathsf{Gen}^F\ B)^2$ iff $\approx$ is a semantic equivalence on F and satisfies the equivalences

$$\textsc{Let}\text{-}\beta \qquad \mathsf{let}\ x \leftarrow \mathsf{return}\ t_1\ \mathsf{in}\ t_2 \approx t_2[t_1/x]$$

$$\textsc{Let}\text{-}\eta \qquad \mathsf{let}\ x \leftarrow t\ \mathsf{in}\ \mathsf{return}\ x \approx t$$

$$\textsc{Let-Assoc} \qquad \mathsf{let}\ x \leftarrow (\mathsf{let}\ y \leftarrow t_1\ \mathsf{in}\ t_2)\ \mathsf{in}\ t_3 \approx \mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ \mathsf{let}\ y \leftarrow t_2\ \mathsf{in}\ t_3$$

where $t_1[t_2/x]$ denotes the substitution of t_2 for x in t_1.

Intuitively, a monadic interpretation ensures that sequential composition behaves as expected, i.e. sequential composition is associative (rule Let-Assoc) and return behaves like a no-op (rules Let-β and Let-η.) It is particularly useful to have an interpretation which is monadic up to standard equality (=): sequential composition satisfies Definition 2.13 in most languages (such as C++), so a monadic interpretation is a necessary (but not sufficient) requirement if the language implementor wishes to reuse the host language's sequential composition constructs.

When reasoning about generator behaviour, the key property of interest is a generator's distribution, i.e. the probability a generator produces a particular value. It is known that the QuickCheck interpretation $\boldsymbol{Q}$ is *not* monadic, although Claessen and Hughes [3] claim that this is "morally" acceptable since rules Let-β to Let-Assoc hold up to some notion of distribution equivalence. We formally investigate this claim in Section 3.

Definition 2.14 (Probabilistic Interpretation). A Σ-interpretation F (for $\Sigma \supseteq \Sigma_{\boldsymbol{Q}}$) is probabilistic up to an equivalence relation $\approx\ \subseteq (A \to \mathsf{Gen}^F\ B)^2$ iff F is monadic up to $\approx$ and satisfies the following rules:

$$\oplus\text{-}1 \qquad t_1 \oplus_1 t_2 \approx t_1$$

$$\oplus\text{-}0 \qquad t_1 \oplus_0 t_2 \approx t_2$$

$$\oplus\text{-Idem} \qquad t \oplus_p t \approx t$$

$$\oplus\text{-Comm} \qquad t_1 \oplus_p t_2 \approx t_2 \oplus_{1-p} t_1$$

⊕-Assoc

$$\frac{0 < p < 1 \qquad 0 < q < 1}{(t_1 \oplus_p t_2) \oplus_q t_3 \approx t_1 \oplus_{p \cdot q} (t_2 \oplus_{(q \cdot (1-p))/(1-p \cdot q)} t_3)}$$

⊕-η

$$(\text{let } x \leftarrow t_1 \text{ in } t_2 \oplus_p t_3) \approx (\text{let } x \leftarrow t_1 \text{ in } t_2) \oplus_p (\text{let } x \leftarrow t_1 \text{ in } t_3)$$

Definition 2.14, adapted from Gibbons and Hinze [8], characterizes the expected behaviour of the probabilistic choice operator $\oplus_p$ (and by extension the operator flip_p). We omit the right distributivity rule from Gibbons and Hinze [8] as it holds for all monadic interpretations in our formulation (Proposition 2.15).

Proposition 2.15. *The following rule holds for all Σ-interpretations F (for $\Sigma \supseteq \Sigma_Q$) that are monadic up to an equivalence relation $\approx \subseteq (A \to \text{Gen}^F\ B)^2$:*

⊕-β

$$(\text{let } x \leftarrow t_1 \oplus_p t_2 \text{ in } t_3) \approx (\text{let } x \leftarrow t_1 \text{ in } t_3) \oplus_p (\text{let } x \leftarrow t_2 \text{ in } t_3)$$

Example 2.16. Probabilistic interpretations allow us to give algebraic proofs of generator properties. For example, nat^1 intuitively generates a 0 or 1 with probability 1/2. We can prove this formally for any interpretation F which is probabilistic up to some equivalence relation $\approx$:

$$\begin{aligned}
\text{nat}^1 &\approx \text{let } n \leftarrow \text{nat}^0 \text{ in} && \text{(by definition)}\\
&\quad\ \text{let } b \leftarrow \text{flip}_{1/2} \text{ in}\\
&\quad\quad \text{return } (2 \cdot n + \text{if } b \text{ then } 1 \text{ else } 0)\\
&\approx \text{let } n \leftarrow \text{nat}^0 \text{ in} && (\text{since } \text{flip}_p \approx \text{return true} \oplus_p \text{return false})\\
&\quad\ \text{let } b \leftarrow \text{return true} \oplus_{1/2} \text{return false in}\\
&\quad\quad \text{return } (2 \cdot n + \text{if } b \text{ then } 1 \text{ else } 0)\\
&\approx \text{let } n \leftarrow \text{nat}^0 \text{ in return } (2 \cdot n + 1) \oplus_{1/2} \text{return } (2 \cdot n) && \text{(by } \oplus\text{-}\beta)\\
&\approx \text{let } n \leftarrow \text{return } 0 \text{ in return } (2 \cdot n + 1) \oplus_{1/2} \text{return } (2 \cdot n) && \text{(by definition)}\\
&\approx \text{return } 1 \oplus_{1/2} \text{return } 0 && \text{(by Let-}\beta\text{)}
\end{aligned}$$

Definition 2.17. A Σ-interpretation F (for $\Sigma \supseteq \Sigma_{\mathcal{H}} \setminus \Sigma_Q$) is a shrinking interpretation up to an equivalence relation $\approx \subseteq (A \to \text{Gen}^F\ B)^2$ iff F is monadic up to $\approx$ and satisfies the following rules:

Shrink-Id	Shrink-Join
$\text{shrink}\ m \approx m$	$\text{shrink}\ (\text{shrink}\ m_0\ m_1 \ldots m_n)\ m'_1 \ldots m'_k \approx \text{shrink}\ m_0\ m_1 \ldots m_n\ m'_1 \ldots m'_k$

Definition 2.17 characterizes the expected behaviour of the shrink operator. As with the probabilistic choice operator $\oplus_p$, the shrink operator distributes over let.

Proposition 2.18. *The following rule holds for all Σ-interpretations F (for $\Sigma \supseteq \Sigma_{\mathcal{H}} \setminus \Sigma_Q$) that are monadic up to a semantic equivalence relation $\approx \subseteq (A \to \text{Gen}^F\ B)^2$:*

Shrink-β

$$\text{let } x \leftarrow \text{shrink}\ m_0 \ldots m_n \text{ in } m' \approx \text{shrink}\ (\text{let } x \leftarrow m_0 \text{ in } m') \ldots (\text{let } x \leftarrow m_n \text{ in } m')$$

Example 2.19. As in Example 2.16, we formally show that snat^1 returns 0 or 1 with probability 1/2. Additionally, our proof shows that, when snat^1 returns 1, it may shrink to 0.

$$\begin{aligned}
\mathsf{snat}^1 &\approx \mathsf{let}\ n \leftarrow \mathsf{snat}^0\ \mathsf{in} && \text{(by definition)}\\
&\quad\ \mathsf{let}\ b \leftarrow \mathsf{sflip}_{1/2}\ \mathsf{in}\\
&\quad\quad \mathsf{return}\ (2 \cdot n + \mathsf{if}\ b\ \mathsf{then}\ 1\ \mathsf{else}\ 0)\\
&\approx \mathsf{let}\ n \leftarrow \mathsf{snat}^0\ \mathsf{in} && \text{(by definition)}\\
&\quad\ \mathsf{let}\ b \leftarrow \mathsf{shrink}\ (\mathsf{return\ true})\ (\mathsf{return\ false}) \oplus_p \mathsf{return\ false}\ \mathsf{in}\\
&\quad\quad \mathsf{return}\ (2 \cdot n + \mathsf{if}\ b\ \mathsf{then}\ 1\ \mathsf{else}\ 0)\\
&\approx \mathsf{let}\ n \leftarrow \mathsf{snat}^0\ \mathsf{in} && \text{(by } \oplus\text{-}\beta \text{ and } \textsc{Shrink}\text{-}\beta)\\
&\quad\ \mathsf{shrink}\ (\mathsf{return}\ (2 \cdot n + 1))\ (\mathsf{return}\ (2 \cdot n)) \oplus_{1/2} \mathsf{return}\ (2 \cdot n)\\
&\approx \mathsf{let}\ n \leftarrow \mathsf{return}\ 0\ \mathsf{in} && \text{(by definition)}\\
&\quad\ \mathsf{shrink}\ (\mathsf{return}\ (2 \cdot n + 1))\ (\mathsf{return}\ (2 \cdot n)) \oplus_{1/2} \mathsf{return}\ (2 \cdot n)\\
&\approx \mathsf{shrink}\ (\mathsf{return}\ 1)\ (\mathsf{return}\ 0) \oplus_{1/2} \mathsf{return}\ 0 && \text{(by } \textsc{Let}\text{-}\beta)
\end{aligned}$$

3 Analysis

In this section, we formally investigate the QuickCheck and Hedgehog languages and their properties. We first define distribution equivalence and show (1) the QuickCheck interpretation is monadic up to distribution equivalence, but (2) the same is not true of Hedgehog's interpretation. Finally, we show that this is not a flaw in Hedgehog, but rather an inherent problem with the combination of probabilistic and shrinking effects.

3.1 Distribution Equivalence

Definition 3.1 (Distribution Equivalence). Two families of random variables $f_1, f_2 : A \to \mathsf{Seed} \to B$ have pointwise equivalent distributions (written $f_1 \doteq f_2$) iff $\mathbb{P}\,(f_1\ a \in X) = \mathbb{P}\,(f_2\ a \in X)$ for all $a \in A$ and $X \subseteq B$.

Two random variables have equivalent distributions if they produce each possible set of values with the same likelihood. Definition 3.1 lifts this idea to families of random variables. Note that Definition 3.1 also applies to Hedgehog's tree-valued random variable generator representation.

Lemma 3.2. *Suppose* $\Gamma \vdash t : \tau$, $\Gamma \vdash t_1 : \mathsf{Gen}\,\tau$, $\Gamma[x \mapsto \tau] \vdash t_2 : \mathsf{Gen}\,\tau'$, $p \in \mathbb{W}$, $X \subseteq [\![\tau]\!]$, *and* $Y \subseteq [\![\tau']\!]$. *The following rules hold:*

$$\mathbb{P}\,([\![\Gamma \vdash \mathsf{return}\ t]\!]^Q\ \rho \in X) = \begin{cases} 1 & \text{if } [\![\Gamma \vdash t]\!]^Q\ \rho \in X \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathbb{P}\,([\![\Gamma \vdash \mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ t_2]\!]^Q\ \rho \in Y) = \sum_{a \in [\![\tau]\!]} \mathbb{P}\,([\![\Gamma \vdash t_1]\!]^Q\ \rho = a) \cdot \mathbb{P}\,([\![\Gamma[x \mapsto \tau] \vdash t_2]\!]^Q\ \rho[x \mapsto a] \in Y)$$

$$\mathbb{P}\,([\![\Gamma \vdash \mathsf{flip}_p]\!]^Q\ \rho = 1) = p$$

Proof. Note that, for any $t \in \mathsf{Term}\,\Sigma_Q$ satisfying $\Gamma \vdash t : \mathsf{Gen}\,\tau$ and $\rho \in [\![\Gamma]\!]$, the random variable $[\![\Gamma \vdash t]\!]^Q\ \rho$ has a countable domain. Hence, it suffices to prove the following three goals:

Goal 1. Suppose $a \in [\![\tau]\!]$. Then

$$\mathbb{P}\,((\sigma \mapsto a) \in X) = \begin{cases} 1 & \text{if } a \in X \\ 0 & \text{otherwise.} \end{cases}$$

Proof. Trivial. ∎

Goal 2. Suppose $A, B \in \mathrm{Set}$ are assigned the discrete σ-algebra, $m : \mathrm{Seed} \to A$ is a random variable with a countable domain, $f : A \to \mathrm{Seed} \to B$ is a family of random variables with countable domains, and $Y \subseteq B$. Then

$$\mathbb{P}\left(\mathrm{let}^{\mathcal{Q}}\, f\, m \in Y\right) = \sum_{a \in [\![\tau]\!]} \mathbb{P}\left(m = a\right) \cdot \mathbb{P}\left(f\, a \in Y\right).$$

Proof. Let A' be the countable subset of A for which $(m \circ \pi_L)^{-1}\, A' \neq \emptyset$. Then

$$\begin{aligned}
\mathbb{P}\left(\mathrm{let}^{\mathcal{Q}}\, f\, m \in Y\right) &= \mathbb{P}\left\{\sigma \mid f\left(m\left(\pi_L\, \sigma\right)\right)\left(\pi_R\, \sigma\right) \in Y\right\} && \text{(by definition)} \\
&= \mathbb{P}\left\{\sigma \mid \exists a \in A.\, m\left(\pi_L\, \sigma\right) = a \wedge f\, a\left(\pi_R\, \sigma\right) \in Y\right\} && \text{(property of } \exists) \\
&= \mathbb{P}\left(\bigcup_{a \in A} \left\{\sigma \mid m\left(\pi_L\, \sigma\right) = a \wedge f\, a\left(\pi_R\, \sigma\right) \in Y\right\}\right) && \text{(by definition)} \\
&= \mathbb{P}\left(\bigcup_{a \in A'} \left\{\sigma \mid m\left(\pi_L\, \sigma\right) = a \wedge f\, a\left(\pi_R\, \sigma\right) \in Y\right\}\right) && \text{(property of } A') \\
&= \sum_{a \in A'} \mathbb{P}\left\{\sigma \mid m\left(\pi_L\, \sigma\right) = a \wedge f\, a\left(\pi_R\, \sigma\right) \in Y\right\} && \text{(property of } \mathbb{P}) \\
&= \sum_{a \in A} \mathbb{P}\left\{\sigma \mid m\left(\pi_L\, \sigma\right) = a \wedge f\, a\left(\pi_R\, \sigma\right) \in Y\right\} && \text{(property of } A') \\
&= \sum_{a \in A} \mathbb{P}\left\{\sigma \mid m\left(\pi_L\, \sigma\right) = a\right\} \cdot \mathbb{P}\left\{\sigma \mid f\, a\left(\pi_R\, \sigma\right) \in Y\right\} && \text{(indep. of } \pi_L \text{ and } \pi_R) \\
&= \sum_{a \in A} \mathbb{P}\left\{\sigma \mid m\, \sigma = a\right\} \cdot \mathbb{P}\left\{\sigma \mid f\, a\, \sigma \in Y\right\} && \text{(measure preservation of } \pi_L \text{ and } \pi_R) \\
&= \sum_{a \in A} \mathbb{P}\left(m = a\right) \cdot \mathbb{P}\left(f\, a \in Y\right). && \text{(by definition)}
\end{aligned}$$

■

Goal 3. We have

$$\mathbb{P}\left(\psi_p = 1\right) = p.$$

Proof. Trivial. ■

□

Lemma 3.2 characterizes the distribution of each of QuickCheck's operators. This lemma allows us to formally prove the claim by Claessen and Hughes [3] that the QuickCheck interpretation is monadic up to distribution equivalence. In fact, we prove a stronger property: we show that the QuickCheck interpretation is probabilistic.

Proposition 3.3. *The QuickCheck interpretation $\mathcal{Q}$ is probabilistic up to distribution equivalence ($\doteq$).*

Proof. Follows by simple calculational proof using Lemma 3.2. □

Can Proposition 3.3 be extended to Hedgehog? This turns out not to be the case: as distribution equivalence is not even a semantic equivalence on $\mathcal{H}$ as the rule

$$\textsc{Let-Cong}\ \frac{\Gamma \vdash t_1 \doteq^{\mathcal{H}} t_1' \qquad \Gamma[x \mapsto \tau] \vdash t_2 \doteq^{\mathcal{H}} t_2'}{\Gamma \vdash \left(\mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ t_2\right) \doteq^{\mathcal{H}} \left(\mathsf{let}\ x \leftarrow t_1'\ \mathsf{in}\ t_2'\right)}.$$

does not hold. For a counterexample, consider the generators $\mathsf{coin}_1, \mathsf{coin}_2 \in \mathsf{Term}\,\Sigma_{\mathcal{H}}$ defined as

$$\mathsf{coin}_1 = \mathsf{return}\ 1 \oplus_{1/2} \mathsf{return}\ 0 \qquad \mathsf{coin}_2 = \mathsf{return}\ 0 \oplus_{1/2} \mathsf{return}\ 1.$$

The coin_2 generator interprets its seed in the exact opposite way from coin_1, i.e. $\mathsf{coin}_1\ \sigma = 1 - \mathsf{coin}_2\ \sigma$ for any $\sigma \in \mathsf{Seed}$. However, we clearly have $\mathsf{coin}_1 \doteq^{\mathcal{H}} \mathsf{coin}_2$ (since both generators return 0 or 1 with probability 0.5) and therefore, for any $x \in \mathsf{Var}$,

$$\mathsf{coin}_1 \doteq^{\mathcal{H}} \mathsf{if}\ x\ \mathsf{then}\ \mathsf{coin}_1\ \mathsf{else}\ \mathsf{coin}_2. \tag{1}$$

Now define

$$\mathsf{step} = \mathsf{shrink}\ (\mathsf{return}\ 1)\ (\mathsf{return}\ 0).$$

The step generator always returns 1 during testing, but returns 0 during shrinking. If rule Let-Cong holds then by equation 1 we have

$$\mathsf{let}\ x \leftarrow \mathsf{step}\ \mathsf{in}\ \mathsf{coin}_1 \doteq^{\mathcal{H}} \mathsf{let}\ x \leftarrow \mathsf{step}\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ \mathsf{coin}_1\ \mathsf{else}\ \mathsf{coin}_2. \tag{2}$$

However, the left program returns the same value during the testing and shrinking stages, as the value x obtained from step is ignored. Conversely, the right program may change its value between stages, since (1) coin_1 is evaluated during testing, (2) coin_2 is evaluated during shrinking, and (3) coin_1 and coin_2 return opposite values. Hence equation 2 is a contradiction.

Proposition 3.4. *Distribution equivalence ($\doteq$) is not a semantic equivalence on the Hedgehog interpretation $\mathcal{H}$.*

3.2 Generator-Based Shrinking Monads

Can we somehow "fix" Hedgehog to make it probabilistic with respect to distribution equivalence? Alternatively, can we use a different equivalence relation? We show that neither of these possibilities work.

Definition 3.5. An interpretation F is a generator-based shrinking interpretation up to an equivalence relation $\approx\ \subseteq (A \to \mathsf{Gen}^F\ B)^2$ iff F is a probabilistic interpretation and a shrinking interpretation up to F.

Definition 3.5 characterizes the minimal set of properties we expect from a language for generator-based shrinking.

Now consider an arbitrary generator-based shrinking interpretation F on $\approx$ and an arbitrary weight $p \in \mathbb{W}$. We define $\mathsf{amb} : \mathsf{Gen\ Bool}$ as

$$\mathsf{amb} = \mathsf{shrink}\ (\mathsf{return}\ 1)\ (\mathsf{return}\ 1) \oplus_p \mathsf{shrink}\ (\mathsf{return}\ 0)\ (\mathsf{return}\ 0).$$

Intuitively, amb denotes a generator which returns 1 or 0 during testing, and returns the exact same value during shrinking. Consider also the generator $\mathsf{amb}' : \mathsf{Gen\ Bool}$ defined as

$$\begin{aligned}\mathsf{amb}' = (&\mathsf{shrink}\ (\mathsf{return}\ 1)\ (\mathsf{return}\ 1) \oplus_p \mathsf{shrink}\ (\mathsf{return}\ 0)\ (\mathsf{return}\ 1)) \oplus_p \\ &(\mathsf{shrink}\ (\mathsf{return}\ 1)\ (\mathsf{return}\ 0) \oplus_p \mathsf{shrink}\ (\mathsf{return}\ 0)\ (\mathsf{return}\ 0)).\end{aligned}$$

Intuitively, amb' behaves similarly to amb but includes the possibility of changing values during shrinking. Clearly these programs should not be equivalent, but the properties of generator-based shrinking interpretations imply that they are.

Theorem 3.6. *The following equivalence holds for any interpretation F which is a generator-based shrinking interpretation up to an equivalence relation $\approx\ \subseteq (A \to \mathsf{Gen}^F\ B)^2$:*

$$\mathsf{amb} \approx^F \mathsf{amb}'$$

Proof. Consider the following lemma:

Lemma 3.6.1. *The following rule holds:*

$$\frac{}{\text{shrink}\ t_0 \dots (t_i \oplus_p t_i') \dots t_n \approx \text{shrink}\ t_0 \dots t_i \dots t_n \oplus_p \text{shrink}\ t_0 \dots t_i' \dots t_n}\ \oplus\text{-Shrink-Distrib}$$

Proof. Define

$$\begin{aligned}\text{select}\ x\ t_0 \dots t_n = {} & \text{if}\ x = n\ \text{then}\ t_n\ \text{else} \\ & \dots \\ & \text{if}\ x = 1\ \text{then}\ t_1\ \text{else} \\ & \quad t_0.\end{aligned}$$

Then

$$\begin{aligned}
& \text{shrink}\ t_0 \dots (t_i \oplus_p t_i') \dots t_n \\
\approx\ & \text{let}\ x \leftarrow \text{shrink}\ 0 \dots n && \text{(by Shrink-}\beta\text{)} \\
& \text{in select}\ x\ t_0 \dots (t_i \oplus_p t_i') \dots t_n \\
\approx\ & \text{let}\ x \leftarrow \text{shrink}\ 0 \dots n && \text{(by }\oplus\text{-Idem)} \\
& \text{in select}\ x\ (t_0 \oplus_p t_0) \dots (t_i \oplus_p t_i') \dots (t_n \oplus_p t_n) \\
\approx\ & \text{let}\ x \leftarrow \text{shrink}\ 0 \dots n && \text{(property of if (select))} \\
& \text{in select}\ x\ t_0 \dots t_i \dots t_n \oplus_p \text{select}\ x\ t_0 \dots t_i' \dots t_n \\
\approx\ & (\text{let}\ x \leftarrow \text{shrink}\ 0 \dots n\ \text{in select}\ x\ t_0 \dots t_i \dots t_n) && \text{(by }\oplus\text{-}\eta\text{)} \\
& \oplus_p (\text{let}\ x \leftarrow \text{shrink}\ 0 \dots n\ \text{in select}\ x\ t_0 \dots t_i' \dots t_n) \\
\approx\ & \text{shrink}\ t_0 \dots t_i \dots t_n \oplus_p \text{shrink}\ t_0 \dots t_i' \dots t_n. && \text{(by }\oplus\text{-}\eta\text{)}
\end{aligned}$$

■

Now consider the term

$$\text{shrink}\ (\text{return}\ 1 \oplus_p \text{return}\ 0)\ (\text{return}\ 1 \oplus_p \text{return}\ 0).$$

We have

$$\begin{aligned}
& \text{shrink}\ (\text{return}\ 1 \oplus_p \text{return}\ 0)\ (\text{return}\ 1 \oplus_p \text{return}\ 0) \\
\approx\ & \text{shrink}\ (\text{return}\ 1)\ (\text{return}\ 1 \oplus_p \text{return}\ 0) && \text{(by }\oplus\text{-Shrink-Distrib)} \\
\approx\ & \oplus_p \text{shrink}\ (\text{return}\ 0)\ (\text{return}\ 1 \oplus_p \text{return}\ 0) \\
\approx\ & (\text{shrink}\ (\text{return}\ 1)\ (\text{return}\ 1) \oplus_p \text{shrink}\ (\text{return}\ 1)\ (\text{return}\ 0)) && \text{(by }\oplus\text{-Shrink-Distrib)} \\
& \oplus_p (\text{shrink}\ (\text{return}\ 0)\ (\text{return}\ 1) \oplus_p \text{shrink}\ (\text{return}\ 0)\ (\text{return}\ 0)) \\
\approx\ & \text{amb}'. && \text{(by definition)}
\end{aligned}$$

We also have

$$\begin{aligned}
& \text{shrink}\ (\text{return}\ 1 \oplus_p \text{return}\ 0)\ (\text{return}\ 1 \oplus_p \text{return}\ 0) \\
\approx\ & \text{let}\ x \leftarrow \text{shrink}\ (\text{return}\ 1)\ (\text{return}\ 0)\ \text{in return}\ 1 \oplus_p \text{return}\ 0 && \text{(by Shrink-}\beta\text{ and Let-}\beta\text{)} \\
\approx\ & (\text{let}\ x \leftarrow \text{shrink}\ (\text{return}\ 1)\ (\text{return}\ 0)\ \text{in return}\ 1) \\
& \oplus_p (\text{let}\ x \leftarrow \text{shrink}\ (\text{return}\ 1)\ (\text{return}\ 0)\ \text{in return}\ 0) && \text{(by }\oplus\text{-}\eta\text{)} \\
\approx\ & \text{shrink}\ (\text{return}\ 1)\ (\text{return}\ 1) \oplus_p \text{shrink}\ (\text{return}\ 0)\ (\text{return}\ 0) && \text{(by Shrink-}\beta\text{ and Let-}\beta\text{)} \\
\approx\ & \text{amb}. && \text{(by definition)}
\end{aligned}$$

Hence amb ≈ amb′. □

4 The halcheck Language

In this section, we introduce the halcheck language for generator-based shrinking. In Section 4.1, we specify halcheck and discuss some basic properties. In Section 4.2, we define our primary reasoning tool — contextual joint distribution equivalence — and show that halcheck's interpretation is probabilistic with respect to this equivalence, modulo some assumptions. In Section 4.3, we introduce a conservative extension of the halcheck language which resolves some issues with compositional generator construction and allows us to define a sound translation from Hedgehog and halcheck.

4.1 Definition and Basic Properties

Definition 4.1 (halcheck). The halcheck signature is defined as

$$\Sigma_{\mathsf{hc}} = \{\mathsf{flip}_{\ell s \sim p} : \Rightarrow \mathsf{Bool} \mid \ell s \in \mathsf{Label}^*, p \in \mathbb{W}\} \cup \{\mathsf{shrinkNat} : \mathsf{Nat} \Rightarrow \mathsf{Nat}\}$$

and the halcheck interpretation hc is given as follows:

$$\begin{aligned}
\mathsf{Gen}^{\mathsf{hc}}\, A &= \mathsf{Seed} \to \mathsf{Tree}\, A & \mathsf{return}^{\mathsf{hc}}\, a &= \sigma \mapsto (a, []) \\
\mathsf{let}^{\mathsf{hc}}\, f\, m &= \sigma \mapsto \mathsf{let}^{\mathsf{T}}\, (a \mapsto f\, a\, \sigma)\, (m\, \sigma) & \mathsf{shrinkNat}^{\mathsf{hc}}\, n &= \sigma \mapsto (0, [(1, []), \ldots, (n, [])]) \\
\mathsf{flip}^{\mathsf{hc}}_{\ell s \sim p} &= \sigma \mapsto (\psi_p\, (\pi_{\ell s}\, \sigma), [])
\end{aligned}$$

Syntactically, the halcheck language is nearly identical to the Hedgehog language but substitutes the coin flip operator flip_p with a *labelled* coin flip operator $\mathsf{flip}_{\ell s \sim p}$. Semantically, halcheck differs from Hedgehog in two important ways:

- $\mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ t_2$ passes the same seed to both t_1 and t_2 instead of splitting it via π_L and π_R. This means that the results produced by t_1 and t_2 may be correllated even if t_2 does not mention x.
- $\mathsf{flip}_{\ell s \sim p}$ samples a different part of the input seed depending on the specified label string ℓs.

As with QuickCheck, we define a labelled probabilistic choice operator $- \oplus_{\ell s \sim p} -$ as

$$t_1 \oplus_{\ell s \sim p} t_2 = \mathsf{let}\ b \leftarrow \mathsf{flip}_{\ell s \sim p}\ \mathsf{in}\ \mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2.$$

Example 4.2. To illustrate the difference between halcheck and Hedgehog, consider the Hedgehog generator $\mathsf{compare}_1$ and its naive halcheck translation $\mathsf{compare}_2$:

$$\begin{aligned}
\mathsf{compare}_1 = {} & \mathsf{let}\ x \leftarrow \mathsf{flip}_{1/2}\ \mathsf{in} \\
& \mathsf{let}\ y \leftarrow \mathsf{flip}_{1/2}\ \mathsf{in} \\
& \mathsf{return}\ (x = y)
\end{aligned}
\qquad\qquad
\begin{aligned}
\mathsf{compare}_2 = {} & \mathsf{let}\ x \leftarrow \mathsf{flip}_{[]\sim 1/2}\ \mathsf{in} \\
& \mathsf{let}\ y \leftarrow \mathsf{flip}_{[]\sim 1/2}\ \mathsf{in} \\
& \mathsf{return}\ (x = y)
\end{aligned}$$

The generator $\mathsf{compare}_1$ may evaluate to true or false. For $\mathsf{compare}_2$, since $\mathsf{let}^{\mathsf{hc}}$ passes the same seed to both of its arguments, the two instances of flip will always produce the same choice, i.e. either they both return true or they both return false. Hence $\mathsf{compare}_2$ may only evaluate to true. A truly equivalent port of $\mathsf{compare}_1$ is written as follows:

$$\begin{aligned}
\mathsf{compare}_3 = {} & \mathsf{let}\ x \leftarrow \mathsf{flip}_{[\underline{x}]\sim 1/2}\ \mathsf{in} \\
& \mathsf{let}\ y \leftarrow \mathsf{flip}_{[\underline{y}]\sim 1/2}\ \mathsf{in} \\
& \mathsf{return}\ (x = y)
\end{aligned}$$

While $\mathsf{let}^{\mathsf{hc}}$ passes the same seed to both instances of flip, the different labels cause different parts of the seed to be used. Thus $\mathsf{compare}_3$ can evaluate to true or false.

Theorem 4.3. *The halcheck interpretation* hc *is a shrinking interpretation up to standard equivalence and also satisfies the following rules:*

⊕-Idem
$t \oplus_{\ell s \sim p} t =^{\mathsf{hc}} t$

⊕-Consistent
$(\mathsf{let}\ x \leftarrow t_1 \oplus_{\ell s \sim p} t_2\ \mathsf{in}\ t_3 \oplus_{\ell s \sim p} t_4) =^{\mathsf{hc}} (\mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ t_3) \oplus_{\ell s \sim p} (\mathsf{let}\ x \leftarrow t_2\ \mathsf{in}\ t_4)$

Proof. Follows trivially by expanding definitions. □

Unlike QuickCheck and Hedgehog, the halcheck interpretation is monadic. This makes halcheck a better candidate for an eDSL implementation which reuses its host language's sequential composition operator. Additionally, halcheck satisfies some of the rules required by probabilistic interpretations (see Definition 2.14), with the exceptions being rules ⊕-1, ⊕-0, ⊕-Comm and ⊕-Assoc. Uniquely, halcheck satisfies the rule ⊕-Consistent, which formalizes the intuition that two instances of $\oplus_{\ell s \sim p}$ always make the same choice. Note that rule ⊕-Consistent implies rule ⊕-η and rule ⊕-β.

4.2 Contextual Joint Distribution Equivalence

Definition 4.4 (Contextual Joint Distribution Equivalence). Two families of random variables $f_1, f_2 : A \to \mathsf{Gen}^{\mathsf{hc}}\ B$ have jointwise equivalent distributions in a finite *context* $L \subseteq \mathsf{Label}^* \times \mathbb{W}$ (written $f_1 \doteq_L f_2$) iff

$$\mathbb{P}\left(X \cap \bigcap_{a \in Z} f_1\ a \in Y\ a\right) = \mathbb{P}\left(X \cap \bigcap_{a \in Z} f_2\ a \in Y\ a\right)$$

for all measurable $X \subseteq \mathsf{Seed}$, $Y : A \to 2^B$, and finite $Z \subseteq A$ where

$$X \perp\!\!\!\perp \bigcap_{(\ell s, p) \in L} (\psi_p \circ \pi_{\ell s})^{-1}\ (W\ (\ell s, p))$$

for all functions $W : \mathsf{Label}^* \times \mathbb{W} \to 2^{\mathbb{B}}$.

Intuitively, $\Gamma \vdash t_1 \doteq_L t_2$ means that t_1 and t_2 produce the same distribution of values in all contexts that *do not* depend on any label in L.

Example 4.5. Consider the terms

$$\mathsf{coin}_1 = \mathsf{return}\ 0 \oplus_{[\underline{x}] \sim 1/2} \mathsf{return}\ 1 \qquad \text{and} \qquad \mathsf{coin}_2 = \mathsf{return}\ 1 \oplus_{[\underline{x}] \sim 1/2} \mathsf{return}\ 0.$$

These two terms have identical distributions but it is not sound to assume one can be substituted for the other in all contexts. For example, the terms

$$\begin{aligned}\mathsf{compare}_{11} = \mathsf{let}\ x \leftarrow \mathsf{coin}_1\ \mathsf{in}\\ \mathsf{let}\ y \leftarrow \mathsf{coin}_1\ \mathsf{in}\\ \mathsf{return}\ (x = y)\end{aligned} \qquad \text{and} \qquad \begin{aligned}\mathsf{compare}_{12} = \mathsf{let}\ x \leftarrow \mathsf{coin}_1\ \mathsf{in}\\ \mathsf{let}\ y \leftarrow \mathsf{coin}_2\ \mathsf{in}\\ \mathsf{return}\ (x = y)\end{aligned}$$

are not equal since $\mathsf{compare}_{11}$ always produces true and $\mathsf{compare}_{12}$ always produces false. However, if no string with prefix $[\underline{x}]$ is mentioned anywhere else in the surrounding program then it is safe to replace coin_1 with coin_2. Hence $\mathsf{coin}_1 \doteq_L \mathsf{coin}_2$ iff $[\underline{x}] \in L$.

Definition 4.6. The support of a term is given by $\mathsf{supp} : \mathsf{Term}\ \Sigma_{\mathsf{hc}} \to 2^{(\mathsf{Label}^* \times \mathbb{W})}$, defined as

$$\mathsf{supp}\ (\mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ t_2) = \mathsf{supp}\ t_1 \cup \mathsf{supp}\ t_2 \qquad \mathsf{supp}\ \mathsf{flip}_{\ell s \sim p} = \{(\ell s, p)\}$$
$$\mathsf{supp}\ (\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) = \mathsf{supp}\ t_2 \cup \mathsf{supp}\ t_3$$

where $\mathsf{supp}\ t = \emptyset$ in all other cases.

The support of a term t is the set of all pairs of label strings and probabilities that appear in t. This includes pairs that appear in instances of $\oplus$, e.g.

$$\begin{aligned}\mathsf{supp}\,(\mathsf{return}\ 0 \oplus_{[\underline{x}]\sim 1/2} \mathsf{return}\ 1) &= \mathsf{supp}\,(\mathsf{let}\ x \leftarrow \mathsf{flip}_{[\underline{x}]\sim 1/2}\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ \mathsf{return}\ 0\ \mathsf{else}\ \mathsf{return}\ 1)\\ &= \{([\underline{x}], 1/2)\}.\end{aligned}$$

Intuitively, two terms with disjoint supports represent independent random variables, and hence their results will be uncorrelated when composed in sequence.

Theorem 4.7. *The following rules hold:*

$$\textsc{Refl}_1{}^* \ \frac{\Gamma \vdash t_1 =^{\mathsf{hc}} t_2}{\Gamma \vdash t_1 \mathrel{\dot{=}}_{\emptyset} t_2} \qquad \textsc{Refl}_2{}^* \ \frac{\Gamma \vdash t_1 \mathrel{\dot{=}}_{L} t_2}{\Gamma \vdash t_1 \doteq t_2} \qquad \alpha\text{-}\textsc{Rename}^* \ \frac{\{(\ell s', p')\} \notin \mathsf{supp}\ t}{t \mathrel{\dot{=}}_{\{(\ell s, p), (\ell s', p')\}} t[\mathsf{flip}_{\ell s \sim p}/\mathsf{flip}_{\ell s' \sim p'}]}$$

$$\textsc{Mono}^* \ \frac{L \subseteq L' \qquad \Gamma \vdash t_1 \mathrel{\dot{=}}_{L} t_2}{\Gamma \vdash t_1 \mathrel{\dot{=}}_{L'} t_2} \qquad \textsc{Return-Cong}^* \ \frac{\Gamma \vdash t = t'}{\Gamma \vdash \mathsf{return}\ t \mathrel{\dot{=}}_{\emptyset} \mathsf{return}\ t'}$$

$$\textsc{Let-Cong}^* \ \frac{\begin{array}{c}\Gamma \vdash t_1 \mathrel{\dot{=}}_{L_1} t_1' \qquad \mathsf{supp}\ t_1 \cap L_2 = \emptyset \qquad \mathsf{supp}\ t_1' \cap L_2 = \emptyset \\ \Gamma[x \mapsto \tau] \vdash t_2 \mathrel{\dot{=}}_{L_2} t_2' \qquad \mathsf{supp}\ t_2 \cap L_1 = \emptyset \qquad \mathsf{supp}\ t_2' \cap L_1 = \emptyset\end{array}}{\Gamma \vdash (\mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ t_2) \mathrel{\dot{=}}_{L_1 \cup L_2} (\mathsf{let}\ x \leftarrow t_1'\ \mathsf{in}\ t_2')}(\Gamma \vdash t_1 : \tau)$$

$$\textsc{If-Cong}^* \ \frac{\begin{array}{c}\Gamma \vdash t_0 = t_0' \qquad \Gamma \vdash t_1 \mathrel{\dot{=}}_{L_1} t_1' \qquad \Gamma \vdash t_2 \mathrel{\dot{=}}_{L_2} t_2' \\ \mathsf{supp}\ t_1 \cap L_2 = \emptyset \qquad \mathsf{supp}\ t_1' \cap L_2 = \emptyset \qquad \mathsf{supp}\ t_2 \cap L_1 = \emptyset \qquad \mathsf{supp}\ t_2' \cap L_1 = \emptyset\end{array}}{\Gamma \vdash (\mathsf{if}\ t_0\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \mathrel{\dot{=}}_{L_1 \cup L_2} (\mathsf{if}\ t_0'\ \mathsf{then}\ t_1'\ \mathsf{else}\ t_2')}$$

$$\oplus\textsc{-Cong}^* \ \frac{\Gamma \vdash t_1 \mathrel{\dot{=}}_{L} t_1' \qquad \Gamma \vdash t_2 \mathrel{\dot{=}}_{L} t_2' \qquad (\ell s, p) \notin L}{\Gamma \vdash t_1 \oplus_{\ell s \sim p} t_2 \mathrel{\dot{=}}_{L} t_1' \oplus_{\ell s \sim p} t_2'} \qquad \oplus\textsc{-1}^* \ t_1 \oplus_{\ell s \sim 1} t_2 \mathrel{\dot{=}}_{\{(\ell s, 1)\}} t_1 \qquad \oplus\textsc{-0}^* \ t_1 \oplus_{\ell s \sim 0} t_2 \mathrel{\dot{=}}_{\{(\ell s, 0)\}} t_2$$

$$\oplus\textsc{-Comm}^* \ \frac{(\ell s, p), (\ell s, 1-p) \notin \mathsf{supp}\ t_1 \cup \mathsf{supp}\ t_2}{t_1 \oplus_{\ell s \sim p} t_2 \mathrel{\dot{=}}_{\{(\ell s, p), (\ell s, 1-p)\}} t_2 \oplus_{\ell s \sim 1-p} t_1}$$

$$\oplus\textsc{-Assoc}^* \ \frac{\begin{array}{c}0 < p < 1 \qquad 0 < q < 1 \qquad (\ell s, p) \neq (\ell s', q) \\ (\ell s, p), (\ell s, p \cdot q), (\ell s', q), (\ell s', q \cdot (1-p)/(1-p \cdot q)) \notin \mathsf{supp}\ t_1 \cup \mathsf{supp}\ t_2 \cup \mathsf{supp}\ t_3 = \emptyset\end{array}}{(t_1 \oplus_{\ell s \sim p} t_2) \oplus_{\ell s' \sim q} t_3 \mathrel{\dot{=}}_{\{(\ell s, p), (\ell s, p \cdot q), (\ell s', q), (\ell s', q \cdot (1-p)/(1-p \cdot q))\}} t_1 \oplus_{\ell s \sim p \cdot q} (t_2 \oplus_{\ell s' \sim q \cdot (1-p)/(1-p \cdot q)} t_3)}$$

Proof. We prove each rule individually.

Case $\textsc{Refl}_1{}^*$. Trivial.

Case $\textsc{Refl}_2{}^*$. It suffices to show that $f \mathrel{\dot{=}} g$ implies $f \doteq g$ for all $f, g : A \to \mathsf{Seed} \to \mathsf{Tree}\ B$, where $A, B \in \mathsf{Set}$ have discrete σ-algebras. To show $f \doteq g$, we must show

$$\mathbb{P}\,(f\ a \in U) = \mathbb{P}\,(g\ a \in U)$$

for all $a \in A$ and $U \subseteq B$. By Definition 4.4,

$$\mathbb{P}\left(X \cap \bigcap_{a \in Z} f\ a \in Y\ a\right) = \mathbb{P}\left(X \cap \bigcap_{a \in Z} g\ a \in Y\ a\right)$$

for all measurable $X \subseteq \text{Seed}$, $Y : A \to 2^B$, and finite $Z \subseteq A$ where

$$X \perp\!\!\!\perp \bigcap_{(\ell s, p) \in L} (\psi_p \circ \pi_{\ell s})^{-1}\ (W\ (\ell s, p))$$

for all functions $W : \text{Label}^* \times \mathbb{W} \to 2^{\mathbb{B}}$. Then we get

$$\mathbb{P}\ (f\ a \in U) = \mathbb{P}\ (g\ a \in U)$$

by instantiating $X = \text{Seed}$, $Z = \{a\}$, and $Y = y \mapsto U$.

Case α-Rename.*

Case Mono.*

Case Return-Cong.*

Case Let-Cong.*

Case If-Cong.*

Case ⊕-Cong.*

Case ⊕-1.*

Case ⊕-0.*

Case ⊕-Comm.*

Case ⊕-Assoc.* □

Theorem 4.7 enumerates the most important properties of contextual joint distribution equivalence. We describe each rule in detail:

- Rule Refl$_1$* allows any of the equivalences from Theorem 4.3 to be lifted to contextual joint distribution equivalence.
- Rule Refl$_2$* states that contextual joint distribution equivalence is sound with respect to distribution equivalence.
- Rule Mono* states that it is sound to enlarge the context of an equivalence. One use of this rule is to apply the transitivity rule to equivalences with different contexts, i.e.

$$t_1 \mathrel{\hat{=}}_L t_2 \ \wedge\ t_2 \mathrel{\hat{=}}_{L'} t_3 \implies t_1 \mathrel{\hat{=}}_{L \cup L'} t_3.$$

- Rules Return-Cong*, Let-Cong* and ⊕-Cong* make contextual joint distribution equivalence *almost* a semantic equivalence. The rules are almost identical to rules Return-Cong to If-Cong but include some side conditions which prevent equivalences from being applied in invalid contexts such as those seen in Example 4.5.
- Rules ⊕-1* to ⊕-Assoc* are almost identical to Rules ⊕-1, ⊕-0, ⊕-Comm and ⊕-Assoc of probabilistic interpretations. The primary difference is that rules ⊕-Comm* and ⊕-Assoc* can only be applied when the labels used for the choice operator are unique within the context they appear. For example, this disallows inferences of the form

$$(t_1 \oplus_{[\underline{x}] \sim 1/2} t_2) \oplus_{[\underline{x}] \sim 1/2} t_3 \mathrel{\hat{=}}_L t_3 \oplus_{[\underline{x}] \sim 1/2} (t_1 \oplus_{[\underline{x}] \sim 1/2} t_2)$$

since $\underline{x}$ is mentioned twice.

- Rule α-Rename* states that it is sound to replace a label with another, as long as *all* instances of the label are replaced. For example, the equation

$$(t_1 \oplus_{[\underline{x}]\sim 1/2} t_2) \oplus_{[\underline{x}]\sim 1/2} t_3 \doteq_{\{[\underline{x}],[\underline{y}]\}} (t_1 \oplus_{[\underline{y}]\sim 1/2} t_2) \oplus_{[\underline{y}]\sim 1/2} t_3$$

  is allowed, but

$$(t_1 \oplus_{[\underline{x}]\sim 1/2} t_2) \oplus_{[\underline{x}]\sim 1/2} t_3 \doteq_{\{[\underline{x}],[\underline{y}]\}} (t_1 \oplus_{[\underline{x}]\sim 1/2} t_2) \oplus_{[\underline{y}]\sim 1/2} t_3$$

  is not.

Theorem 4.8. *Suppose $t_1 \doteq t_2$ for some $\Gamma \vdash t_1 : \mathsf{Gen}\, A$ and $\Gamma \vdash t_2 : \mathsf{Gen}\, A$. Then $t_1 \doteq t_2$ is provable solely using the rules given in Theorems 4.3 and 4.7.*

Proof.

TODO: …

□

Theorem 4.8 essentially states that the rules given in Theorems 4.3 and 4.7 are complete: no other rules are needed to prove that two generators have equivalent distributions.

4.3 Labelling and Translation from Hedgehog

Definition 4.9. The labelling operator $\mathsf{label}_\ell : \mathsf{Term}\, \Sigma_{\mathsf{hc}} \to \mathsf{Term}\, \Sigma_{\mathsf{hc}}$ (for $\ell \in \mathsf{Label}$) is defined by

$$\begin{aligned} \mathsf{label}_\ell\ (\mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ t_2) &= \mathsf{let}\ x \leftarrow \mathsf{label}_\ell\ t_1\ \mathsf{in}\ \mathsf{label}_\ell\ t_2 \\ \mathsf{label}_\ell\ (\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) &= \mathsf{if}\ t_1\ \mathsf{then}\ \mathsf{label}_\ell\ t_2\ \mathsf{else}\ \mathsf{label}_\ell\ t_3 \\ \mathsf{label}_\ell\ (\mathsf{flip}_{\ell s\sim p}) &= \mathsf{flip}_{\ell::\ell s\sim p} \end{aligned}$$

where $\mathsf{label}_\ell\ t = t$ in all other cases.

The expression $\mathsf{label}_\ell\ t$ behaves exactly like t, except every instance of the coin flip operator $\mathsf{flip}_{\ell s\sim p}$ is replaced with $\mathsf{flip}_{\ell::\ell s\sim p}$. The main purpose of this operator is to enable compositional construction of generators.

Example 4.10. Suppose we have a closed term $t : \mathsf{Gen}\ \mathsf{Nat}$ and suppose we want to construct a generator $\mathsf{sum} : \mathsf{Gen}\ \mathsf{Nat}$ that produces the sum of two random values chosen by t. In other words, we want sum to have the distribution

$$\mathbb{P}\left([\![\mathsf{sum}]\!]^{\mathsf{hc}} \in X\right) = \sum_{x,y\in\mathbb{N}} \mathbb{P}\left([\![t]\!]^{\mathsf{hc}} = x\right) \cdot \mathbb{P}\left([\![t]\!]^{\mathsf{hc}} = y\right) \cdot \begin{cases} 1 & \text{if } x+y \in X \\ 0 & \text{otherwise.} \end{cases}$$

Naively, we might write

$$\begin{aligned} \mathsf{sum} = \mathsf{let}\ x \leftarrow t\ \mathsf{in} \\ \mathsf{let}\ y \leftarrow t\ \mathsf{in} \\ \mathsf{return}\ (x+y). \end{aligned}$$

However this definition of sum does not have the right distribution: x and y are always assigned to identical values since any occurrence of flip within the first t will also occur in the second t *with the same label*. To write sum correctly we require another version of t with different labels, which we can achieve with the label operator:

$$\begin{aligned} \mathsf{sum} = \mathsf{let}\ x \leftarrow \mathsf{label}_{\underline{x}}\ t\ \mathsf{in} \\ \mathsf{let}\ y \leftarrow \mathsf{label}_{\underline{y}}\ t\ \mathsf{in} \\ \mathsf{return}\ (x+y) \end{aligned}$$

This example suggests a general pattern for generator construction: in order to prevent unwanted correlation between generated values, all sub-generator calls should be annotated with unique labels via the label operator.

Proposition 4.11. *For any* $t \in \mathsf{Term}\,\Sigma_{\mathsf{hc}}$,

$$\mathsf{supp}\,(\mathsf{label}_\ell\, t) = \{(\ell :: \ell s, p) \mid (\ell s, p) \in \mathsf{supp}\, t\}$$

Proof. Follows trivially by induction over t. □

Proposition 4.11 provides formal justification for the correctness of sum given in Example 4.10: the two terms $\mathsf{label}_{\underline{x}}\, t$ and $\mathsf{label}_{\underline{y}}\, t$ have disjoint supports (since no word can simultaneously begin with both $\underline{x}$ and $\underline{y}$) hence $\mathsf{label}_{\underline{x}}\, t$ and $\mathsf{label}_{\underline{y}}\, t$ produce uncorrelated results.

The label operator allows us to demonstrate that halcheck loses no expressivity relative to Hedgehog. In particular, we provide a translation from Hedgehog to halcheck.

Definition 4.12. The translation $\langle - \rangle : \mathsf{Term}_{\mathcal{H}} \to \mathsf{Term}_{\mathsf{hc}}$ from Hedgehog to halcheck terms is given by

$$\begin{aligned}
\langle \mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ t_2 \rangle &= \mathsf{let}\ x \leftarrow \mathsf{label}_{\underline{l}}\, \langle t_1 \rangle\ \mathsf{in}\ \mathsf{label}_{\underline{r}}\, \langle t_2 \rangle \\
\langle \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rangle &= \mathsf{if}\ t_1\ \mathsf{then}\ \langle t_2 \rangle\ \mathsf{else}\ \langle t_3 \rangle \\
\langle \mathsf{flip}_p \rangle &= \mathsf{flip}_{[\,]\sim p}
\end{aligned}$$

where $\langle t \rangle = t$ in all other cases.

The translation $\langle t \rangle$ of a Hedgehog term t is obtained by replacing all occurrences of $\mathsf{let}\ x \leftarrow t_1\ \mathsf{in}\ t_2$ with $\mathsf{let}\ x \leftarrow \mathsf{label}_{\underline{l}}\, \langle t_1 \rangle\ \mathsf{in}\ \mathsf{label}_{\underline{r}}\, \langle t_2 \rangle$, and all occurrences of flip_p with $\mathsf{flip}_{[\,]\sim p}$. To understand why this works, consider the definition of $\mathsf{let}^{\mathcal{H}}$:

$$\mathsf{let}^{\mathcal{H}}\, f\, m = \sigma \mapsto \mathsf{let}^{\mathsf{T}}\, (a \mapsto f\, a\, (\pi_L\, \sigma))\, (m\, (\pi_R\, \sigma))$$

The only difference between the Hedgehog version ($\mathsf{let}^{\mathcal{H}}$) and the halcheck version ($\mathsf{let}^{\mathsf{hc}}$) is that the Hedgehog version splits the input seed (σ) before forwarding it to its arguments f and m. This ensures that the results of f and m are uncorrelated, which is exactly what the label operator is for. Note that this translation does not preserve the exact semantics of the Hedgehog generator (i.e. $[\![\langle t \rangle]\!]^{\mathsf{hc}}$ and $[\![t]\!]^{\mathcal{H}}$ denote different random variables) but their distributions are guaranteed to be the same (Theorem 4.13).

Theorem 4.13. *If* $\Gamma \vdash t : \tau$ *(where* $t \in \mathsf{Term}\,\Sigma_{\mathcal{H}}$*) then* $[\![\Gamma \vdash \langle t \rangle]\!]^{\mathsf{hc}} \doteq [\![\Gamma \vdash t]\!]^{\mathcal{H}}$.

Proof.

TODO: …

□

In general, our translation produces programs at most 3 times larger than the original Hedgehog program (we count the size of $\mathsf{label}_\ell\, t$ as one plus the size of t). In Section 5, we demonstrate that halcheck programs are not much larger than their Hedgehog counterparts in practice.

4.4 Summary

In this section, we specified halcheck by providing a signature and interpretation (Definition 4.1). We defined our primary reasoning tool — contextual joint distribution equivalence (Definition 4.4) – and showed that halcheck's interpretation is probabilistic with respect to this equivalence, modulo some assumptions (Theorem 4.7). We introduced a conservative extension of the halcheck language called the labelling operator (Definition 4.9) and showed how the labelling operator solves

Example	Sizes		Hedgehog G		Hedgehog S		falsify G		falsify S		QuickCheck G	
List	$\frac{31}{37}$	19%	$\frac{159.9\,\text{ms}}{96.6\,\text{ms}}$	66%	$\frac{1.1\,\text{s}}{700.1\,\text{ms}}$	56%	$\frac{84.3\,\text{ms}}{53.1\,\text{ms}}$	59%	$\frac{1.1\,\text{s}}{921.3\,\text{ms}}$	21%	$\frac{5.1\,\text{ms}}{6.1\,\text{ms}}$	−16%
Sorted List	$\frac{48}{58}$	21%	$\frac{1.3\,\text{s}}{782\,\text{ms}}$	64%	$\frac{4.1\,\text{s}}{2.6\,\text{s}}$	58%	$\frac{115\,\text{ms}}{95.2\,\text{ms}}$	21%	$\frac{1.1\,\text{s}}{935\,\text{ms}}$	16%	$\frac{9.7\,\text{ms}}{13.5\,\text{ms}}$	−28%
Distinct List	$\frac{49}{59}$	20%	$\frac{2\,\text{s}}{1.2\,\text{s}}$	63%	$\frac{15.7\,\text{s}}{10.8\,\text{s}}$	46%	$\frac{603.2\,\text{ms}}{536.3\,\text{ms}}$	12%	$\frac{6.3\,\text{s}}{5.9\,\text{s}}$	7%	$\frac{74.3\,\text{ms}}{92.7\,\text{ms}}$	−20%
Binary Tree	$\frac{33}{41}$	24%	$\frac{185.6\,\text{ms}}{126.7\,\text{ms}}$	46%	$\frac{584.9\,\text{ms}}{375.5\,\text{ms}}$	56%	$\frac{93.3\,\text{ms}}{65.7\,\text{ms}}$	42%	$\frac{432\,\text{ms}}{289\,\text{ms}}$	49%	$\frac{7.2\,\text{ms}}{9.9\,\text{ms}}$	−27%
Binary Search Tree	$\frac{58}{70}$	21%	$\frac{3\,\text{s}}{2.2\,\text{s}}$	38%	$\frac{12.3\,\text{s}}{8.3\,\text{s}}$	49%	$\frac{433.9\,\text{ms}}{358.9\,\text{ms}}$	21%	$\frac{3\,\text{s}}{2.9\,\text{s}}$	3%	$\frac{47.8\,\text{ms}}{64.3\,\text{ms}}$	−26%
Red-Black Tree	$\frac{151}{163}$	8%	$\frac{602\,\text{ms}}{433\,\text{ms}}$	39%	$\frac{5\,\text{s}}{3.6\,\text{s}}$	39%	$\frac{158\,\text{ms}}{116\,\text{ms}}$	36%	$\frac{2.7\,\text{s}}{2.3\,\text{s}}$	17%	$\frac{15.5\,\text{ms}}{19.8\,\text{ms}}$	−22%
Graph	$\frac{56}{62}$	11%	$\frac{139.6\,\text{ms}}{128.6\,\text{ms}}$	9%	$\frac{245.2\,\text{ms}}{239.9\,\text{ms}}$	2%	$\frac{86.3\,\text{ms}}{74.4\,\text{ms}}$	16%	$\frac{1.4\,\text{s}}{1.3\,\text{s}}$	11%	$\frac{9.9\,\text{ms}}{11\,\text{ms}}$	−10%
Connected Graph	$\frac{96}{102}$	6%	$\frac{141.4\,\text{ms}}{128.1\,\text{ms}}$	10%	$\frac{288.6\,\text{ms}}{256.3\,\text{ms}}$	13%	$\frac{87.6\,\text{ms}}{75.4\,\text{ms}}$	16%	$\frac{1.1\,\text{s}}{1\,\text{s}}$	8%	$\frac{2.9\,\text{ms}}{3.4\,\text{ms}}$	−15%
Lambda Term [16]	$\frac{535}{557}$	4%	$\frac{439\,\text{ms}}{320\,\text{ms}}$	37%	$\frac{21.2\,\text{ms}}{20.8\,\text{ms}}$	2%	$\frac{202.9\,\text{ms}}{176.1\,\text{ms}}$	15%	$\frac{399.3\,\text{ms}}{355.5\,\text{ms}}$	12%	$\frac{76.4\,\text{ms}}{78.7\,\text{ms}}$	−3%
Instruction Sequence [10]												
Naive			$\frac{2.8\,\text{s}}{2\,\text{s}}$	35%	$\frac{2.3\,\text{s}}{2.3\,\text{s}}$	0%	$\frac{1.1\,\text{s}}{740.2\,\text{ms}}$	53%	$\frac{157.2\,\text{s}}{151.3\,\text{s}}$	4%	$\frac{267.8\,\text{ms}}{325.2\,\text{ms}}$	−18%
Weighted	$\frac{2488}{2682}$	8%	$\frac{3\,\text{s}}{2.2\,\text{s}}$	32%	$\frac{2.4\,\text{s}}{2.3\,\text{s}}$	5%	$\frac{1.2\,\text{s}}{796.4\,\text{ms}}$	55%	$\frac{185.6\,\text{s}}{176.6\,\text{s}}$	5%	$\frac{289\,\text{ms}}{347.3\,\text{ms}}$	−17%
Execution			$\frac{3.3\,\text{s}}{2.7\,\text{s}}$	21%	$\frac{2.5\,\text{s}}{2.2\,\text{s}}$	11%	$\frac{1.5\,\text{s}}{1.1\,\text{s}}$	36%	$\frac{175.7\,\text{s}}{168.4\,\text{s}}$	4%	$\frac{562\,\text{ms}}{628\,\text{ms}}$	−11%
Variational Execution			$\frac{4.7\,\text{s}}{4\,\text{s}}$	18%	$\frac{2.1\,\text{s}}{2.1\,\text{s}}$	0%	$\frac{1.9\,\text{s}}{1.4\,\text{s}}$	31%	$\frac{263.2\,\text{s}}{220\,\text{s}}$	20%	$\frac{782\,\text{ms}}{856\,\text{ms}}$	−9%

Table 2. Example sizes and interpretation performance. Program sizes and run times are reported as $\left(\frac{\text{Vanilla}}{\text{Modified}}\ \text{Difference\%}\right)$. G - Generation Time S - Shrink Time

some compositionality issues. Finally, we constructed a translation from Hedgehog to halcheck (Definition 4.12) and proved that the translation is sound (Theorem 4.13).

5 Empirical Evaluation

In Section 4, we described a complete, composable, set of rules for reasoning about halcheck generator distributions. Hence, we now evaluate the effects of halcheck's design with respect to other practical considerations such as generator size and performance. Compared to existing solutions for generator-based shrinking [6, 20], halcheck requires that users manually label random choices. Thus, the goal of our evaluation is to answer the following research questions:

RQ1: To what extent can we express existing generators?
RQ2: What is the performance impact of manual labelling?

5.1 Setup

The halcheck interpretation (Definition 4.1) is essentially the same as the Hedgehog interpretation (Definition 2.11) extended with labelled choice. Hence, we implemented a version of halcheck by modifying the existing Hedgehog library (v1.5 [20]) for Haskell. Our modifications are extremely minimal: modified Hedgehog consists of 66 inserted lines and 80 deleted lines across 4 out of 27 of vanilla Hedgehog's files. Changes consist solely of (1) the introduction of the label operator and (2) wrapping of all sub-generator calls in existing combinators with the label operator. There are no changes to the core datatypes. We also updated Hedgehog's test suite to conform with the changes to Hedgehog. These updates consist of 25 inserted lines and 33 deleted lines across 6 out of 9 files.

We use a set of example generators drawn from Lampropoulos et al. [11] and from industry. We summarize our set of examples in Table 2. Each example is written using halcheck's operations (i.e., let, return, label, flip, and shrinkNat), and we interpret each operation using the corresponding operations from each (vanilla and modified) library. Under the vanilla interpretation, labelling is ignored (i.e., $\text{label}_{\ell s}\, t$ behaves like t), and our examples are carefully constructed to produce the same distribution of results under each interpretation.

Note that labelling primarily affects the semantics of random generation and not shrinking. Other libraries, such as QuickCheck [3] and falsify [6], implement random generation (but not

shrinking) in a manner similar to Hedgehog (e.g., compare Definition 2.8 and Definition 2.11). Thus, we also implemented two additional versions of halcheck by modifying the QuickCheck (v2.15) and falsify (v0.2) libraries for Haskell in a similar manner to our Hedgehog modifications. For the vanilla and modified QuickCheck interpretations, shrinkNat always returns 0 (i.e., no shrinking). Studying these other implementations allows us to evaluate our research questions independently of the particular shrinking semantics we gave in Section 4.

The source code for our implementation, examples, and experiments is available at https://github.com/halcheck/halcheck-hs.

5.2 Results

RQ1: *To what extent can we express existing generators?* By Theorem 4.13, every Hedgehog generator can be mechanically translated into a halcheck generator with the same distribution. This naive translation incurs a 200% increase in program size in the worst case, but in practice it is sometimes possible to construct simpler programs. For example, the $\mathsf{compare}_1$ generator given in Example 4.2 is translated as

$$\begin{aligned}\langle \mathsf{compare}_1 \rangle = \mathsf{let}\ x \leftarrow \mathsf{label}_{\underline{1}}\ \mathsf{flip}_{[]\sim 1/2}\ \mathsf{in}\\ \mathsf{label}_{\underline{r}}\ (\mathsf{let}\ y \leftarrow \mathsf{label}_{\underline{1}}\ \mathsf{flip}_{[]\sim 1/2}\ \mathsf{in}\\ \mathsf{label}_{\underline{r}}\ (\mathsf{return}\ (x = y)))\end{aligned}$$

but can be more succinctly expressed as

$$\begin{aligned}&\mathsf{let}\ x \leftarrow \mathsf{label}_{\underline{1}}\ \mathsf{flip}_{[]\sim 1/2}\ \mathsf{in}\\ &\mathsf{let}\ y \leftarrow \mathsf{label}_{\underline{r}}\ \mathsf{flip}_{[]\sim 1/2}\ \mathsf{in}\\ &\quad \mathsf{return}\ (x = y).\end{aligned}$$

Thus, we investigate the practicality of writing halcheck generators by measuring the increase in program size that comes with manual labelling.

The second column of Table 2 lists the size of each example with and without labelling, along with the size increase as a percentage. Sizes correspond to the abstract syntax tree of the program. We report a single pair of numbers for the instruction sequence generators from Hritcu et al. [10] as the various implementations heavily share code.

Our results show that labelling increases program sizes up to 24% in practice, which is much less significant than the 200% upper bound given at the end of Section 4.3.

Summary: We can express all existing generators. Manual labelling incurs a negligible increase in program size.

RQ2: *What is the performance impact of manual labelling?* Generator performance has a direct impact on how quickly bugs are found by property-based testing. Thus we measure and compare the throughput of each interpretation. Specifically, we measure, for each generator and interpretation, 1) the amount of time it takes to generate 1000 test cases, and 2) the amount of time it takes to compute 1000 shrinks. Measurements are made using the Criterion benchmarking library, which repeats each experiment until a time limit is reached (5 minutes in our case), and estimates the running time via linear regression. All measurements are made on an AMD Ryzen Threadripper 2990WX running at 3GHz, with 128 GB of RAM, on Ubuntu 22.04.5 LTS, compiled with GHC 9.4.8.

Table 2 reports the results of our experiments. For each example and library, we report Criterion's estimates for the amount of time it takes to generate and shrink 1000 random values, with and without our modifications. We also report the difference (as a percentage) between the vanilla and modified interpretations of each library. We do not report shrink times for QuickCheck since it does not have generator-based shrinking, and hence shrink times are not affected by our modifications.

For Hedgehog and falsify, manual labelling typically makes generation up to 66% and 59% faster respectively, and makes shrinking up to 58% and 49% faster respectively. Modified Hedgehog and falsify exceed the vanilla interpretations' performance in all benchmarks. On the other hand, modified QuickCheck generally performs worse than the unmodified version, up to 28% slower.

Our results indicate that manual labelling increases generator and shrinker performance for languages featuring generator-based shrinking such as Hedgehog and falsify. However, manual labelling decreases the performance of QuickCheck, which does not employ generator-based shrinking.

Summary: Manual labelling has a positive impact on generator performance when generator-based shrinking is employed, and a negative impact otherwise.

5.3 Threats to Validity

Our examples are written entirely in Haskell, which is a lazy and purely functional language. Our results may not generalize to strict, imperative languages. Our example set is fairly small so we cannot guarantee that our analysis extends to a wide variety of generators and shrinkers in practice. We do not formally investigate the semantics of modified QuickCheck and falsify as we are only concerned with their runtime performance. However, a lack of formal semantics means that we do not know for sure that we have made the "correct" modifications to these libraries, and it is possible that an "incorrect" modification has skewed our measurements.

6 Related Work

Property-Based Testing and Shrinking. Property-based testing was popularized by the QuickCheck [3] framework for Haskell, which features a probabilistic eDSL for specifying generators. In QuickCheck, a shrinker for a type `a` is specified separately as a function `shrinks :: a -> [a]` from values to their immediate shrinks. In contrast, halcheck employs generator-based shrinking, where shrinkers are derived from annotated generator specifications.

Generator-based shrinking approaches were introduced by QuviQ QuickCheck [4] for Erlang. QuviQ QuickCheck implements generator-based shrinking using an approach known as *integrated shrinking*, where generators are represented as tree-valued random variables. Integrated shrinking is used by halcheck as well as many property-based testing frameworks in other languages such as Hedgehog [20] for Haskell and RapidCheck [7] for C++.

The hypothesis [13] library for Python implements a different approach to generator-based shrinking called *internal shrinking*, where generators are represented as random variables drawing from a space of infinite bit sequences, and shrinks are obtained by re-executing generators on modified bit sequences obtained via a set of built-in heuristics. While hypothesis often produces better shrinks than approaches based on integrated shrinking for programs with no user annotations, hypothesis provides no mechanism for user control over shrinking. This is less than ideal when the built-in heuristics are insufficient. The falsify [6] library for Haskell rectifies this downside while retaining the advantages of internal shrinking by allowing users to specify custom shrinking behaviour. Integrating the advantages of internal shrinking into halcheck is left as future work.

The framework of reflective generators [9] are a unique point in the design space of generator specification languages. In addition to supporting hypothesis-style internal shrinking, reflective generators also support running generators "backwards" (i.e. obtaining a seed from a user-provided test case.) In conjunction, these features enable shrinking of user-defined test cases, which is not supported by any other framework save for QuickCheck. However, reflective generators are more limited in what they can express, e.g. rejection sampling cannot be coded using reflective generators.

Supporting "backwards" generation in halcheck without limiting the expressiveness of the language is left as future work.

The formal semantics of generator-based shrinking languages have not been explored in detail prior to this work and hence these languages lack sound, formally-specified, compositional rules for reasoning about combined generator and shrinker behaviour.

Semantics of Probabilistic and Nondeterministic Programming Languages. Shrinking can be viewed as a form of nondeterminism, and there is a large body of work addressing the combination of probabilistic and nondeterministic effects [1, 2, 8, 14, 22]. Various authors [8, 14, 22] observe that allowing probabilistic choice to distribute over nondeterministic choice, combined with the the idempotence of probabilistic choice (rule $\oplus$-Idem), leads to unexpected behaviour. These observations bear resemblance to Theorem 3.6, however our distributivity law goes in the other direction (shrinking distributes over probabilistic choice due to rule $\oplus$-η) and shrinking is not idempotent.

7 Conclusion and Future Work

Existing languages that feature user-controlled generator-based shrinking lack sound compositional rules for reasoning about generator behaviour. In this paper, we showed that this problem is inherent to the combination of generation and shrinking. We then defined a new language, halcheck, which 1) requires users to manually label random choices and 2) features a complete and compositional (modulo some side conditions on labels) set of rules for reasoning about generator behaviour. We also showed that halcheck is able to express all existing Hedgehog programs. We implemented halcheck and measured, using a diverse set of examples, the practical impact of halcheck's labelling mechanism on program size and performance.

Our analysis has some important limitations. First, we do not consider higher-order generators, i.e., generators which produce other generators. Second, labels must be static, i.e., they cannot depend on another generator's behaviour. Lifting these restrictions is left as future work. One non-limitation is that we do not consider programs with general recursion or (pure) higher-order functions. Our results are easily generalized to include such constructs. One of our implementations of halcheck is a modification of falsify. However, this is not a "true" implementation, as falsify's semantics differs significantly with respect to shrinking. In particular, falsify's shrinking is "hierarchical", which essentially means that it allows backtracking (see Example 2.12 for non-backtracking behaviour). One future direction is to formally analyze the combination of hierarchical shrinking (as in falsify) and labelled probabilistic choice (as in halcheck). In particular, we aim to detemine how to formally specify "hierarchical shrinking" and how a language with such a combination of effects can support compositional reasoning.

Data Availability Statement

The source code for our implementation, examples, and experiments is available at https://github.com/halcheck/halcheck-hs.

References

[1] Emanuele Bandini and Roberto Segala. Axiomatizations for probabilistic bisimulation. In *International Colloquium on Automata, Languages, and Programming*, pages 370–381. Springer, 2001.

[2] Yifeng Chen and Jeff W Sanders. Unifying probability with nondeterminism. In *International Symposium on Formal Methods*, pages 467–482. Springer, 2009.

[3] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[4] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in erlang with quickcheck and pulse. *ACM Sigplan Notices*, 44(9):149–160, 2009.

[5] Ryan Culpepper and Andrew Cobb. Contextual equivalence for probabilistic programs with continuous random variables and scoring. In *European Symposium on Programming*, pages 368–392. Springer, 2017.

[6] Edsko de Vries. falsify: Internal shrinking reimagined for haskell. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium*, pages 97–109, 2023.

[7] Emil Eriksson. Rapidcheck. github.com/emil-e/rapidcheck, 2024.

[8] Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. *ACM SIGPLAN Notices*, 46(9):2–14, 2011.

[9] Harrison Goldstein, Samantha Frohlich, Meng Wang, and Benjamin C Pierce. Reflecting on random generation. *Proceedings of the ACM on Programming Languages*, 7(ICFP):322–355, 2023.

[10] Catalin Hritcu, John Hughes, Benjamin C Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. Testing noninterference, quickly. *ACM SIGPLAN Notices*, 48(9):455–468, 2013.

[11] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 114–129, 2017.

[12] Johannes Link. jqwik. github.com/jqwik-team/jqwik, 2024.

[13] David MacIver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 11 2019. ISSN 2475-9066. doi: 10.21105/joss.01891. URL http://dx.doi.org/10.21105/joss.01891.

[14] Michael Mislove, Joël Ouaknine, and James Worrell. Axioms for probability and nondeterminism. *Electronic Notes in Theoretical Computer Science*, 96:7–28, 2004.

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

[16] Michał H Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 91–97, 2011.

[17] Gordon Plotkin and John Power. Semantics for algebraic operations. *Electronic Notes in Theoretical Computer Science*, 45:332–345, 2001.

[18] Gordon Plotkin and John Power. Notions of computation determine monads. In *International Conference on Foundations of Software Science and Computation Structures*, pages 342–356. Springer, 2002.

[19] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied categorical structures*, 11:69–94, 2003.

[20] Jacob Stanley. Hedgehog. hackage.haskell.org/package/hedgehog, 2024.

[21] Marcin Szymczak and Joost-Pieter Katoen. Weakest preexpectation semantics for bayesian inference: Conditioning, continuous distributions and divergence. In *International Summer School on Engineering Trustworthy Software Systems*, pages 44–121. Springer, 2019.

[22] Daniele Varacca and Glynn Winskel. Distributing probability over non-determinism. *Mathematical structures in computer science*, 16(1):87–113, 2006.

[23] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30, 2018.

[24] Yizhou Zhang and Nada Amin. Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proceedings of the ACM on Programming Languages*, 6 (POPL):1–28, 2022.