

1 Compositional Generator Equivalence

2 ANONYMOUS AUTHOR(S)

3 Property-based testing (PBT) is a powerful technique for software verification that relies on random input
4 generators and “shrinking” processes to find and minimize counterexamples to executable specifications
5 called properties. While optimizing these generators is crucial for testing efficiency, formally justifying
6 such optimizations is currently difficult because existing languages lack a compositional semantics that is
7 coarse-grained enough for high-level reasoning.

8 In this paper, we first provide a formal account of the syntax and semantics of Hedgehog, a popular PBT
9 framework. We demonstrate that Hedgehog’s distribution semantics — which models how users typically
10 reason about generators — is non-compositional. Furthermore, we prove that any sound and complete
11 compositional semantics for Hedgehog must necessarily be equivalent to its sampling semantics, which is too
12 fine-grained to justify common program optimizations.

13 To resolve this dilemma, we introduce Hedgehog \rightarrow , a restricted version of the language based on the
14 arrow calculus, and prove that Hedgehog \rightarrow possesses a compositional distribution semantics. We evaluate
15 Hedgehog \rightarrow through a Haskell implementation and show that it remains expressive enough to capture
16 generators of practical interest, while providing the formal foundation needed for compositional generator
17 equivalence proofs.

18 CCS Concepts: • **Theory of computation → Probabilistic computation; Denotational semantics;**
19 **Program reasoning.**

20 Additional Key Words and Phrases: Property-Based Testing, Denotational Semantics, Probabilistic Programming,
21 Program Equivalence

22 ACM Reference Format:

23 Anonymous Author(s). 2026. Compositional Generator Equivalence. In *Proceedings of The ACM SIGPLAN
24 International Conference on Functional Programming (ICFP ’26)*. ACM, New York, NY, USA, 26 pages. <https://doi.org/XXXXXX.XXXXXXX>

25 1 Introduction

26 *Property-based testing* (PBT) is a powerful tool for identifying software bugs. Users provide an
27 executable specification — or *property* — to a *PBT framework*, which automatically checks the
28 specification against a *system under test* (SUT) over a wide range of randomly generated inputs (or
29 *test cases*). If the framework finds a *counterexample* — a test case that causes the SUT to violate the
30 specification — then the framework computes and reports a minimal counterexample.

31 A property consists of three components: (1) *generators*, which compute random test cases, (2) an
32 *oracle*, which decides if a test case is a counterexample, and (3) *shrinkers*, which compute a set of “smaller”
33 test cases from an existing test case. A property’s components are integral to the
34 operation of a PBT framework, which consists of two stages: testing and shrinking. The *testing stage*
35 is responsible for identifying a counterexample, and operates using the generators and the
36 oracle of the property. The *shrinking stage* performs counterexample generation and minimization,
37 and operates using the shrinkers and the oracle.

38 Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee
39 provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the
40 full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored.
41 Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires
42 prior specific permission and/or a fee. Request permissions from permissions@acm.org.

43 *ICFP ’26, TODO*

44 © 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

45 ACM ISBN TODO

46 <https://doi.org/XXXXXX.XXXXXXX>

```

50   coin :: Gen Bool
51   coin = do
52     x ← bool_
53     y ← bool_
54     return (x == y)
55   coin' :: Gen Bool
56   coin' = do return ()
57   coin
      coin
10  discrim :: Gen Bool → Gen Bool
11  discrim m = do
12    shrink
13    (\b → if b
14      then [False]
15      else [])
16    (return True)
17
18  if x then coin else m

```

Fig. 1. Example generators and distinguishing context.

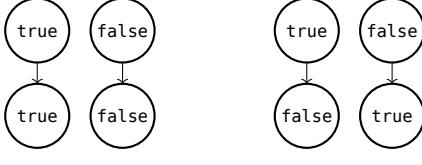
The testing stage proceeds by constructing a random test case using the generators and then checking if the test case is a counterexample using the oracle. If the test case is not a counterexample, the testing process restarts with a different test case; otherwise, the counterexample is passed along to the shrinking stage. In lieu of a counterexample, the testing stage can be terminated (1) after a user-specified number of test cases have been generated, (2) after a user-specified timeout, or (3) manually.

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, it computes a set of “smaller” test cases, called *shrinks*, and 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. Like the testing stage, the shrinking stage can be terminated (1) after a user-specified number of shrinks have been generated, (2) after a user-specified timeout, or (3) manually.

Generators are specified using an embedded domain-specific language (eDSL) provided by the PBT framework. These eDSLs are essentially probabilistic programming languages, and hence users view generators as probability distributions. For example, consider the program given in Figure 1, written using the Hedgehog framework [Stanley 2024] for Haskell. Concretely, the `coin` generator computes two random booleans, `x` and `y` (lines 3 and 4), by invoking a sub-generator `bool_` which performs a fair coin flip. The `coin` generator then checks 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 [Claessen and Hughes 2000] 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 [Stanley 2024] 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 [de Vries 2023; Stanley 2024], others — such as Hypothesis [Maciver and Hatfield-Dodds 2019] — 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 [de Vries 2023]. Thus, this work focuses on user-controlled generator-based shrinking.

Efficient generators and shrinkers are essential for the effectiveness of property-based testing. When timeouts are used, slow generators reduce the likelihood that the testing stage discovers a counterexample, and slow shrinkers reduce the likelihood that the shrinking stage completely minimizes counterexamples. Thus it is important that generators and shrinkers are optimized, either by hand or (preferably) automatically.

Fig. 2. Trees produced by `discrim` (see Figure 1).

By *optimization*, we mean replacing an existing (slow) program with a (faster) *semantically equivalent* one. Thus, in order to perform optimizations, we must formally justify their validity via a semantic equivalence proof. Semantic equivalence in languages with generator-based shrinking is surprisingly subtle. For example, consider the generator `coin'` (Figure 1, line 7). Intuitively, the statement `return ()` is a “no-op”, and so it can ostensibly be removed. However, removing the statement is unsound, as `coin` and `coin'` produce observably different behaviour in combination with the function `discrim` (line 10): `discrim coin` produces the trees given in Figure 2a, while `discrim coin'` produces the trees given in both Figure 2a and Figure 2b. We explain the reason for this in Example 3.11.

Unfortunately, equivalence proofs are currently infeasible as there is currently no way to formally and effectively reason about the behaviour of generators in languages with user-controlled generator-based shrinking. Existing possibilities include:

(1) *Utilize existing probabilistic calculi.* Since generator specification languages are effectively probabilistic programming languages, we can use existing probabilistic calculi to reason about generator behaviour. However, existing probabilistic calculi [Culpepper and Cobb 2017; Dal Lago et al. 2020; Faggian and Rocca 2019] do not model shrinking behaviour.

(2) *Adapt existing calculi for probabilistic non-determinism.* Shrinking can be viewed as a form of non-determinism, which suggests that we can adapt existing calculi which model combinations of probabilistic and non-deterministic effects for our purposes. However, languages which feature generator-based shrinking violate common equational properties – the *monad laws* [Moggi 1991] – which are present in these calculi and also intuitively expected by users. For example, the monad laws imply that the generators `coin` and `coin'` (Figure 1) are equivalent, which is not the case.

(3) *Implementation-level reasoning.* We can reason about languages directly in terms of their implementation (or *sampling semantics*). Existing languages such as Hedgehog [Stanley 2024] represent a generator as a *random variable*, i.e., a deterministic function on some *sample space*. However, sampling semantics is too *fine-grained*, not preserved even by simple changes. For example, `coin` and `coin'` (Figure 1) are not semantically equivalent under Hedgehog’s sampling semantics because they interpret their samples differently. In contrast, an appropriate *coarse-grained* semantics should allow `coin` and `coin'` to be identified, at least in some contexts.

(4) *Distribution-level reasoning.* The effectiveness of property-based testing depends on the distribution of a generator, not the specifics of how samples are turned into test cases. Since Hedgehog has a sampling semantics in terms of random variables, one can derive an appropriate *distribution semantics*. However, this distribution semantics is not *compositional*: the distribution of a generator is not uniquely determined by the distributions of its components. For example, we cannot determine the distribution of `discrim m` (Figure 1) simply by knowing the distribution of `m`: `coin` and `coin'` have the same distribution, but `discrim coin` and `discrim coin'` do not. A compositional semantics is important for reasoning about the behaviour of larger programs.

Contributions. In this paper, we address the problem of formally reasoning about generator behaviour in languages with user-controlled generator-based shrinking, for the purpose of justifying

optimizations. We focus on Hedgehog [Stanley 2024], although our results are easily transferred to other languages. We show Hedgehog's **only** compositional semantics is its sampling semantics; Hedgehog generators can (at best) be interpreted as deterministic programs with no opportunity to optimize or reason about their probabilistic behaviour. In light of this result, we define a restricted version of Hedgehog based on the arrow calculus [Lindley et al. 2010] called Hedgehog \rightarrow , and prove that its distribution semantics is compositional and validates many common optimizations. We evaluate the expressiveness of Hedgehog \rightarrow relative to Hedgehog in practice and demonstrate Hedgehog \rightarrow 's utility for equivalence proofs on a set of examples. Specifically, (1) we give a formal account of the Hedgehog language's syntax and its sampling semantics, where generators represent random variables on an appropriate sample space, (2) we also define Hedgehog's distribution semantics, where generators represent probability distributions, and show that it is not compositional, (3) Hedgehog's distribution semantics induces an *observational equivalence* relation, and we show that the only compositional semantics that is sound and complete with respect to observational equivalence is the sampling semantics, (4) we define Hedgehog \rightarrow , a version of Hedgehog based on the arrow calculus [Lindley et al. 2010], and show that the distribution semantics for this version of Hedgehog is compositional, and (5) we evaluate the expressiveness of Hedgehog \rightarrow , its effect on program size, and its utility for program optimization.

Organization. In Section 2, we review background material. In Section 3, we present our formalization and analysis of the Hedgehog language. In Section 4, we present Hedgehog \rightarrow . In Section 5, we evaluate the practical utility of Hedgehog \rightarrow . In Section 6, we describe and compare related work. In Section 7, we conclude and discuss future work.

2 Background

In this section, we fix some basic notation (Section 2.1) and then review the requisite probability theory (Section 2.2). We work within the framework of quasi-Borel spaces [Heunen et al. 2017], which allows reasoning about probabilistic behaviour of programs with higher-order functions.

2.1 Notation

Sets. The set of *natural numbers* is denoted by \mathbb{N} and the set of *extended non-negative real numbers* is denoted by $\bar{\mathbb{R}}_{\geq 0}$. The *union* of two sets A and B is denoted by $A \cup B$, the *intersection* is denoted by $A \cap B$, the *empty set* is denoted by \emptyset , and the *powerset* of A is denoted by $\mathcal{P}A$. The *sample space* \mathbb{S} is defined as the interval $[0, 1) \subseteq \bar{\mathbb{R}}_{\geq 0}$.

Products. The *(cartesian) product* of a family of sets $(A_i)_{i \in I}$ is defined as $\prod_{i \in I} A_i = \{ (a_i)_{i \in I} \mid \forall i \in I, a_i \in A_i \}$. We define the *n-ary product* as $A_1 \times \dots \times A_n = \prod_{i=1}^n A_i$, and write (a_1, \dots, a_n) in place of $(a_i)_{i=1}^n$. The *nullary product* is denoted by 1 , and its unique element is denoted by $()$. The *joining* of two families $a = (a_i)_{i \in I}$ and $b = (b_j)_{j \in J}$ such that $I \cap J = \emptyset$ is denoted by a, b .

Functions. The set of *functions* from set A to set B is denoted by $A \rightarrow B$. Function application is denoted using juxtaposition ($f x$). The notation $(a \mapsto \dots)$ denotes an *(anonymous) function* with parameter a . Functions can be denoted using *blanks*, e.g., $(1 + -) = (n \mapsto 1 + n)$. The *identity function* on A is denoted by id_A and the *composition* of two functions $f : B \rightarrow C$ and $g : A \rightarrow B$ is denoted by $f \circ g$. We omit the subscripts on id when it can be inferred from the context.

Coproducts. The *coproduct* (or *disjoint sum*) of a family of sets $(A_i)_{i \in I}$ is defined as $\sum_{i \in I} A_i = \{ \iota_i x \mid x \in A_i \}$. We define the *n-ary coproduct* as $A_1 + \dots + A_n = \sum_{i=1}^n A_i$. Given a family of functions $f : \prod_{i \in I} (A_i \rightarrow B)$, we define the function $[f_i]_{i \in I} : \sum_{i \in I} A_i \rightarrow B$ to satisfy $[f_i]_{i \in I} (\iota_j x) = f_j x$ for all $j \in I$. The *nullary coproduct* is denoted by $0 (= \emptyset)$, and is equipped with a (unique) function

197 198 $(\times) : (A \rightarrow B) \times (A' \rightarrow B') \rightarrow A \times A' \rightarrow B \times B'$ 199 $(f \times g)(x, y) = (fx, gy)$	distrib : $A \times (B + C) \rightarrow (A \times B) + (A \times C)$ distrib $(x, \iota_1 y) = \iota_1(x, y)$ distrib $(x, \iota_2 y) = \iota_2(x, y)$
201 202 $\text{curry}_{A,B,C} : (A \times B \rightarrow C) \rightarrow (A \rightarrow B \rightarrow C)$ 203 $\text{curry}_{A,B,C} f a b = f(a, b)$	$[-]_A : \mathbb{B} \rightarrow A$ $[\text{true}]_A = 1$ $[\text{false}]_A = 0$

Fig. 3. Miscellaneous Functions

$$\begin{array}{ll} \mathcal{F}_{\mathbb{B}} = \mathbb{B} & \mathcal{F}_{\mathbb{N}} = \mathbb{N} \\ \mathcal{F}_{\overline{\mathbb{R}}_{\geq 0}} = \text{Borel } \{(r, r') \mid r < r' \in \overline{\mathbb{R}}_{\geq 0}\} & \mathcal{F}_{\mathbb{S}} = \text{Borel } \{(r, r') \mid r < r' \in \mathbb{S}\} \end{array}$$

Fig. 4. Common σ -Algebras.

$$\begin{array}{ll} \text{Elem } \mathbb{B} = (\overline{\mathbb{R}}_{\geq 0} \rightarrow_{\mathbf{M}} \mathbb{B}) & \text{Elem } \mathbb{N} = (\overline{\mathbb{R}}_{\geq 0} \rightarrow_{\mathbf{M}} \mathbb{N}) \\ \text{Elem } \overline{\mathbb{R}}_{\geq 0} = (\overline{\mathbb{R}}_{\geq 0} \rightarrow_{\mathbf{M}} \overline{\mathbb{R}}_{\geq 0}) & \text{Elem } \mathbb{S} = (\overline{\mathbb{R}}_{\geq 0} \rightarrow_{\mathbf{M}} \mathbb{S}) \\ \text{Elem } \prod_{i \in I} A_i = \prod_{i \in I} \text{Elem } A_i & \text{Elem } (A \rightarrow_{\mathbf{Q}} B) = \{\text{curry } f \mid f : \overline{\mathbb{R}}_{\geq 0} \times A \rightarrow_{\mathbf{Q}} B\} \\ \text{Elem } \sum_{i \in I} A_i = \left\{ (r \mapsto \iota_{(e(ir))}(\alpha_{(ir)} r)) \mid i : \overline{\mathbb{R}}_{\geq 0} \rightarrow_{\mathbf{Q}} \mathbb{N}, e : \mathbb{N} \rightarrow I, \alpha_i \in \text{Elem } A_{(ei)} \right\} \end{array}$$

Fig. 5. Common Sets of Random Variables.

(!A) : $\mathbf{0} \rightarrow A$ for all sets A. We omit the subscript on (!) when it can be inferred from the context. The set of booleans is defined as $\mathbb{B} = \mathbf{1} + \mathbf{1}$, where true = $\iota_1()$ and false = $\iota_2()$.

Miscellaneous. We implicitly treat proposition as booleans. We define common miscellaneous functions in Figure 3. We omit subscripts on these functions when they can be inferred from the context.

2.2 Probability Theory

Measure Theory. A set A is a *measurable space* if it is equipped with a designated set $\mathcal{F}_A \subseteq \mathcal{P}A$ (called a σ -algebra) such that Borel $\mathcal{F}_A \subseteq \mathcal{F}_A$, where Borel X is the closure of X under complements and countable unions. We define several (but not all) instances of \mathcal{F} in Figure 4. The class of measurable spaces is denoted by **Meas**. A function $f : A \rightarrow B$ for $A, B \in \mathbf{Meas}$ is *measurable* if $f^{-1}X \in \mathcal{F}_A$ for all $X \in \mathcal{F}_B$. The set of measurable functions from A to B is denoted by $A \rightarrow_{\mathbf{M}} B$.

Quasi Borel Spaces. A set A is a *quasi-Borel space* if there is a designated set $\text{Elem } A \subseteq \mathcal{P}(\overline{\mathbb{R}}_{\geq 0} \rightarrow A)$ of random variables such that

- (1) $\text{Elem } A$ contains all constant functions,
- (2) $\text{Elem } A$ is closed under precomposition with all measurable $f : \overline{\mathbb{R}}_{\geq 0} \rightarrow \overline{\mathbb{R}}_{\geq 0}$, and
- (3) $\text{Elem } A$ is closed under countable case analysis, i.e., $(r \mapsto f(ir)r) \in \text{Elem } A$ for all $f : \mathbb{N} \rightarrow \text{Elem } A$ and $i : \overline{\mathbb{R}}_{\geq 0} \rightarrow_{\mathbf{M}} \mathbb{N}$.

We define several (but not all) instances of Elem in Figure 5. The class of quasi-Borel spaces is denoted by **Qbs**. A function $f : A \rightarrow B$ on $A, B \in \mathbf{Qbs}$ is *quasi-measurable* if $\alpha \in \text{Elem } A$ implies $(f \circ \alpha) \in \text{Elem } B$. The set of quasi-measurable functions from A to B is denoted by $A \rightarrow_{\mathbf{Q}} B$. A function $f : A \times \overline{\mathbb{R}}_{\geq 0} \rightarrow_{\mathbf{Q}} B$ is called a *random function*.

$$\begin{array}{ll}
246 \quad f \circ \text{arr} \text{id} = f & (f \circ g) \circ h = f \circ (g \circ h) \\
247 \quad \text{arr}(f \circ g) = \text{arr} f \circ \text{arr} g & \text{first}(\text{arr} f) = \text{arr}(f \times \text{id}) \\
248 \quad \text{first}(f \circ g) = \text{first} f \circ \text{first} g & \text{arr}(\text{id} \times g) \circ \text{first} f = \text{first} f \circ \text{arr}(\text{id} \times g) \\
249 \quad \text{arr}(-_1) \circ \text{first} f = f \circ \text{arr}(-_1) & \text{arr} \text{assoc} \circ \text{first}(\text{first} f) = \text{first} f \circ \text{arr} \text{assoc} \\
250 \\
251
\end{array}$$

Fig. 6. Arrow Properties.

254 *Functors, Monads, and Arrows.* A *functor* is a mapping $F : \mathbf{Qbs} \rightarrow \mathbf{Qbs}$ equipped with a designated
255 function $\text{map}_F : (A \rightarrow_Q B) \rightarrow_Q (FA \rightarrow_Q FB)$ such that

- 256 (1) $\text{map}_F \text{id}_A = \text{id}_{FA}$ and
257 (2) $\text{map}_F(f \circ g) = \text{map}_F f \circ \text{map}_F g$ for all $f : B \rightarrow_Q C$ and $g : A \rightarrow_Q B$.

258 A (strong) *pre-monad* is a mapping $F : \mathbf{Qbs} \rightarrow \mathbf{Qbs}$ equipped with designated operations

$$261 \quad \text{unit}_F : A \rightarrow_Q FA \qquad \text{bind}_F : (A \rightarrow_Q FB) \rightarrow_Q (FA \rightarrow_Q FB).$$

262 A *monad* is a pre-monad F such that, for all $f : B \rightarrow_Q C$ and $g : A \rightarrow_Q B$,

$$264 \quad \text{bind}_F \text{unit}_F = \text{id} \qquad \text{bind}_F f \circ \text{unit}_F = f \qquad \text{bind}_F f \circ \text{bind}_F g = \text{bind}_F(\text{bind}_F f \circ g)$$

266 Every monad F is also a functor by defining $\text{map}_F f = \text{bind}_F(\text{unit}_F \circ f)$ for functions $f : A \rightarrow_Q B$.
267 A *pre-arrow* is a mapping $F : \mathbf{Qbs} \times \mathbf{Qbs} \rightarrow \mathbf{Qbs}$ equipped with designated operations

$$269 \quad \text{arr}_F : (A \rightarrow_Q B) \rightarrow_Q F(A, B) \qquad \text{first}_F : F(A, B) \rightarrow_Q F(A \times C, B \times C) \\
270 \quad (\circ_F) : F(B, C) \times F(A, B) \rightarrow_Q F(A, C) \qquad \text{left}_F : F(A, B) \rightarrow_Q F(A + C, B + C).$$

272 An *arrow* is a pre-arrow F which satisfies the properties in Figure 6. We omit the subscripts on
273 map, unit, bind, arr, \circ , first, and left when they can be inferred from the context.

274 *Measures and Distributions.* Heunen et al. [2017] construct a *distribution monad* (or *measure*
275 *monad*) $\text{Dist} : \mathbf{Qbs} \rightarrow \mathbf{Qbs}$ equipped with an *integration operator* $(\int - d-) : (A \rightarrow_M \bar{\mathbb{R}}_{\geq 0}) \times$
276 $\text{Dist} A \rightarrow_M \bar{\mathbb{R}}_{\geq 0}$ for all $A \in \mathbf{Qbs}$. We write $\int_x y d\mu$ in place of $\int(x \mapsto y) d\mu$ when $\mu \in \text{Dist} A$. The
277 integration operator satisfies standard properties. Given $f : A \rightarrow_Q B$, $c \in \bar{\mathbb{R}}_{\geq 0}$, and $\mu \in \text{Dist} A$, the
278 distributions map $f \mu \in \text{Dist} B$ (called the *push-forward* of f on μ) and $c \cdot \mu \in \text{Dist} A$ satisfy the
279 properties

$$282 \quad \int g d(\text{map} f \mu) = \int g \circ f d\mu \qquad \int h d(c \cdot \mu) = c \cdot \int h d\mu$$

285 for all $g : B \rightarrow_Q \bar{\mathbb{R}}_{\geq 0}$ and $h : A \rightarrow_Q \bar{\mathbb{R}}_{\geq 0}$.

286 The *uniform distribution* on samples $\lambda \in \text{Dist} \mathbb{S}$ is the (unique) distribution satisfying $\int_\sigma [\sigma_1 \leq$
287 $\sigma \leq \sigma_2] d\lambda = \sigma_2 - \sigma_1$ for all $\sigma_1 \leq \sigma_2 \in \mathbb{S}$. There exists two functions $\pi_l, \pi_r : \mathbb{S} \rightarrow \mathbb{S}$ such that

$$289 \quad \int_\sigma f(\pi_l \sigma, \pi_r \sigma) d\lambda = \int_{\sigma_1}^{\sigma_2} \int_{\sigma_1}^{\sigma_2} f(\sigma_1, \sigma_2) d\lambda d\lambda$$

293 for all $f : \mathbb{S} \times \mathbb{S} \rightarrow_Q \bar{\mathbb{R}}_{\geq 0}$.

```

295
296     unit : A →Q List A
297     unit a = a :: ε
298
299     unit : A →Q Tree A
300     unit a = node (a, ε)
301
302
303
304     bind : (A →Q List B) →Q List A →Q List B
305     bind f ε = ε
306     bind f (a :: as) = f a ++ bind f as
307
308     bind : (A →Q Tree B) →Q Tree A →Q Tree B
309     bind f (node (a, as)) = node (b, bs ++ map (bind f) as)
310     where node (b, bs) = f a

```

Monad Operations

```

304     (++) : List A × List A →Q List A
305     ε ++ bs = bs
306     (a :: as) ++ bs = a :: (as ++ bs)
307
308
309
310     Fig. 7. Lists and Rose Trees.

```

Lists and Trees. The quasi-Borel spaces of *lists* $\text{List } A$ and (*rose*) *trees* $\text{Tree } A$ over $A \in \mathbf{Qbs}$ are the smallest sets satisfying

$$\text{List } A = \{\epsilon\} \cup \{x :: xs \mid x \in A, xs \in \text{List } A\} \quad \text{Tree } A = \{\text{node}(x, xs) \mid x \in A, xs \in \text{List}(\text{Tree } A)\}.$$

The functions $((x, xs) \mapsto x :: xs)$, $((x, xs) \mapsto \text{node}(x, xs))$, and any functions built using structural recursion with quasi-measurable functions, are themselves quasi-measurable. The mappings $\text{List}, \text{Tree} : \mathbf{Qbs} \rightarrow \mathbf{Qbs}$ form monads with the operations defined in Figure 7.

3 Hedgehog

In this section, we provide a formal account of the Hedgehog [Stanley 2024] language. Our formalization is given in the style of the monadic metalanguage by Moggi [1991]. At a high level, Hedgehog is a probabilistic programming language, where generators are specified using probabilistic operators. In addition to probabilistic behaviour, Hedgehog contains operators for specifying shrinking behaviour.

Generator optimizations require formal justification relative to an appropriate semantics. We formalize two candidate semantics for Hedgehog: (1) a sampling semantics (Definition 3.5), where generators represent random variables, which closely mirrors Hedgehog’s actual implementation, and (2) a distribution semantics (Definition 3.10), where generators represent distributions, which more closely model how users actually think about generators. Both of these semantics have significant issues which make them unsuitable for equivalence proofs. The sampling semantics is overly *fine-grained*, i.e., it distinguishes many terms which users expect to be identical (Example 3.6) and invalidates many common program transformations (Theorem 3.8). The distribution semantics is not compositional, i.e., the distribution of a term is not uniquely determined by the distributions of its immediate sub-terms (Proposition 3.12), which complicates reasoning about larger programs. Unfortunately, this state of affairs cannot be fixed by simply finding the “right semantics”: we derive soundness and completeness conditions from the distribution semantics (Definition 3.14) and show that *every* (sound and complete) compositional semantics is equivalent to the sampling semantics (Corollary 3.16). This result implies that several common program transformation are unsound, which justifies our need for a restricted version of Hedgehog.

3.1 Syntax

Syntactic elements are written in `teletype` font.

```

344           Bool = 1 + 1           true = inl ()           false = inr ()
345
346 if t1 then t2 else t3 = match t1 with inl x. t2 | inr x. t3      (t1, t2, ..., tn) = (t1, (t2, ..., tn))
347
348 not t1 = if t1 then false else true          t1 = t2 = if t1 then t2 else not t2
349
350           t1 and t2 = if t1 then t2 else false

```

Fig. 8. Derived Constructs

Definition 3.1 (Hedgehog Syntax). The set of Hedgehog *types* Type is generated by

$$\tau ::= 0 \mid 1 \mid \tau + \tau \mid \tau * \tau \mid \tau \rightarrow \tau \mid \text{Gen } \tau$$

and the set of Hedgehog *terms* Term is generated by

$$t ::= x \mid \text{absurd } t \mid () \mid \text{inl } t \mid \text{inr } t \mid \text{match } t \text{ with inl } x. t \mid \text{inr } x. t \mid (t, t) \\ \mid \text{fst } t \mid \text{snd } t \mid \text{fun } x : \tau. t \mid t t \mid \text{return } t \mid \text{let } x \leftarrow t \text{ in } t \mid \text{flipp } t \mid \text{shrink } t$$

where *variables* (e.g., x) are sourced from a countably infinite set of names Var , and probabilities (e.g., p) are elements of the real interval $[0, 1]$. We identify terms up to α -equivalence.

The set of types consists of the empty type (\emptyset), the singleton type (1), sum types ($\tau + \tau$), product types ($\tau * \tau$), function types ($\tau \rightarrow \tau$), and generator types ($\text{Gen } \tau$). Intuitively, the type $\text{Gen } \tau$ is the type of generators which produce elements of the type τ . The set of terms contains many standard constructs including variables (x), left and right injections (inl , inr), eliminations for values of the empty type ($\text{absurd } t$), case analysis ($\text{match } t \text{ with } \text{inl } x. t \mid \text{inr } x. t$), the singleton value (()), tuples ((t, t)), anonymous functions ($\text{fun } x : \tau. t$), function application ($t t$), effect-free computations ($\text{return } t$), and sequential composition ($\text{let } x \leftarrow t \text{ in } t$). We derive some common constructs in Figure 8. In addition to standard constructs, Hedgehog has the following effectful operations:

- (1) The biased *coin flip operation* $\text{flip } p$ generates true with probability p and false with probability $1 - p$.
 - (2) The *shrinking operation* shrink may be applied to an n -tuple (t_0, t_1, \dots, t_n) . The operation $\text{shrink} (t_0, t_1, \dots, t_n)$ behaves like t_0 during the testing stage, but attempts each of t_1, \dots, t_n in order during the shrinking stage.

Example 3.2. We recast our examples from Section 1 in our chosen syntax:

```

coin1 = let x ← flip1/2 in
           let y ← flip1/2 in
               return x = y

coin2 = let x ← flip1/2 in
           return x

```

Definition 3.3 (Hedgehog Environments and Typing). An environment $\Gamma \in \prod_{x \in \text{Dom } \Gamma} \text{Type}$ is a family mapping variables from a finite domain $\text{Dom } \Gamma \subseteq \text{Var}$ to types. The set of environments is denoted by Env , and the singleton environment is written $x : \tau$. The *typing relation* $(- \vdash - : -) \subseteq \text{Env} \times \text{Term} \times \text{Type}$ is the smallest relation satisfying the rules in Figure 9. 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 ‘`fst ()`’). The typing relation $\Gamma \vdash t : \tau$ describes exactly which terms are sensible within a given environment.

$$\begin{array}{c}
\frac{}{\Gamma, x : \tau \vdash x : \tau} \\
\frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \text{inl } t : \tau_1 + \tau_2} \\
\frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \text{inr } t : \tau_1 + \tau_2} \\
\frac{\Gamma \vdash t : \theta}{\Gamma \vdash \text{absurd } t : \tau}
\end{array}
\quad
\begin{array}{c}
\frac{x_1, x_2 \notin \text{Dom } \Gamma}{\Gamma, x_1 : \tau_1 \vdash t_1 : \tau_3} \quad \frac{\Gamma \vdash t_0 : \tau_1 + \tau_2}{\Gamma, x_2 : \tau_2 \vdash t_2 : \tau_3} \\
\frac{\Gamma \vdash \text{match } t_0 \text{ with inl } x_1 . t_1 \mid \text{inr } x_2 . t_2 : \tau_3}{\Gamma \vdash () : 1} \\
\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 * \tau_2}
\end{array}
\quad
\begin{array}{c}
\frac{\Gamma \vdash t : \tau_1 * \tau_2}{\Gamma \vdash \text{fst } t : \tau_1} \quad \frac{\Gamma \vdash t : \tau_1 * \tau_2}{\Gamma \vdash \text{snd } t : \tau_2} \\
\frac{x \notin \text{Dom } \Gamma}{\Gamma \vdash \text{fun } x : \tau_1 . t : \tau_1 \rightarrow \tau_2} \\
\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \text{return } t : \text{Gen } \tau}
\end{array}
\quad
\begin{array}{c}
\frac{x \notin \text{Dom } \Gamma}{\Gamma \vdash \text{flip } p : \text{Gen Bool}} \\
\frac{\Gamma \vdash t_1 : \text{Gen } \tau_1 \quad \Gamma, x : \tau_1 \vdash t_2 : \text{Gen } \tau_2}{\Gamma \vdash \text{let } x \leftarrow t_1 \text{ in } t_2 : \text{Gen } \tau_2} \\
\frac{}{\Gamma \vdash \text{shrink} : \tau * \dots * \tau \rightarrow \text{Gen } \tau}
\end{array}$$

Fig. 9. Hedgehog's Typing Rules.

3.2 Semantics

Definition 3.4 (Hedgehog Interpretation and Semantics). A Hedgehog *interpretation* is a pre-monad $F : \mathbf{Qbs} \rightarrow \mathbf{Qbs}$ equipped with additional operations

$$\text{flip}_F : [0, 1] \rightarrow_{\mathbf{Q}} F \mathbb{B} \qquad \text{shrink}_F^n : \prod_{i=0}^n A \rightarrow_{\mathbf{Q}} FA.$$

An interpretation induces (1) a *type and environment semantics* $\llbracket - \rrbracket^F : \text{Type} \cup \text{Env} \rightarrow \mathbf{Qbs}$ and (2) for all $\Gamma \vdash t : \tau$ a *term semantics* $\llbracket \Gamma \vdash t : \tau \rrbracket^F : \llbracket \Gamma \rrbracket^F \rightarrow_{\mathbf{Q}} \llbracket \tau \rrbracket^F$ (see Figure 10). When t has no free variables, we write $\llbracket t : \tau \rrbracket^F$ in place of $\llbracket () \vdash t : \tau \rrbracket^F$. We also omit the type ascription $(: \tau)$ when it can be inferred from the surrounding context.

The semantics of most types and terms are standard, where types denote suitable quasi-Borel spaces and terms denote quasi-measurable functions over *variable assignments* (i.e., elements of $\llbracket \Gamma \rrbracket = \prod_{x \in \text{Dom } \Gamma} \Gamma_x$). An interpretation specifies the semantics of generator types and terms (Gen, return, let, flip, and shrink). We only require that an interpretation F is a pre-monad, and not a monad: QuickCheck generators are known not to form a monad [Claessen and Hughes 2000], and Hedgehog's sampling interpretation (Definition 3.5) inherits this property. An important feature of Definition 3.4 is that it is *compositional*, i.e., the semantics of a term is uniquely determined by the semantics of its immediate sub-terms.

Definition 3.5 (Hedgehog Sampling Interpretation). The *sampling interpretation* $\mathfrak{C} : \mathbf{Qbs} \rightarrow \mathbf{Qbs}$ of Hedgehog is defined in Figure 11.

Definition 3.5 closely mirrors the actual implementation of Hedgehog. We describe each component of the sampling interpretation as follows:

- (1) Generators ($\text{Gen } \tau$) are represented as deterministic functions from the sample space \mathbb{S} to rose trees (i.e., generators are random variables). Intuitively, the probabilistic behaviour of a generator arises by feeding it a uniformly distributed random sample, and the resulting tree (called a *shrink tree*) describes a value and all the ways it can be shrunk.
- (2) The “pure” computation $\text{return } t$ ignores its sample and returns a tree with a single node labelled by the value of t .

Type and Environment Semantics

$$\llbracket \emptyset \rrbracket^F = \mathbf{0} \quad \llbracket 1 \rrbracket^F = \mathbf{1} \quad \llbracket \tau_1 + \tau_2 \rrbracket^F = \llbracket \tau_1 \rrbracket^F + \llbracket \tau_2 \rrbracket^F \quad \llbracket \tau_1 * \tau_2 \rrbracket^F = \llbracket \tau_1 \rrbracket^F \times \llbracket \tau_2 \rrbracket^F$$

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket^F = \llbracket \tau_1 \rrbracket^F \rightarrow_Q \llbracket \tau_2 \rrbracket^F \quad \llbracket \text{Gen } \tau \rrbracket^F = F \llbracket \tau \rrbracket^F \quad \llbracket \Gamma \rrbracket^F = \prod_{x \in \text{Dom } \Gamma} \llbracket \Gamma_x \rrbracket^F$$

Term Semantics

$$\llbracket \Gamma, x : \tau \vdash x : \tau \rrbracket^F \rho = \rho_x \quad \llbracket \Gamma \vdash () : 1 \rrbracket^F \rho = () \quad \llbracket \Gamma \vdash \text{absurd } t : \tau \rrbracket^F \rho = !(\llbracket \Gamma \vdash t : \emptyset \rrbracket^F \rho)$$

$$(\llbracket \Gamma \vdash \text{fst } t : \tau_1 \rrbracket^F \rho, \llbracket \Gamma \vdash \text{snd } t : \tau_2 \rrbracket^F \rho) = \llbracket \Gamma \vdash t : \tau_1 * \tau_2 \rrbracket^F \rho$$

$$\llbracket \Gamma \vdash (t_1, t_2) : \tau_1 * \tau_2 \rrbracket^F \rho = (\llbracket \Gamma \vdash t_1 : \tau_1 \rrbracket^F \rho, \llbracket \Gamma \vdash t_2 : \tau_2 \rrbracket^F \rho)$$

$$\llbracket \Gamma \vdash \text{inl } t : \tau_1 + \tau_2 \rrbracket^F = \iota_1 \circ \llbracket \Gamma \vdash t : \tau_1 \rrbracket^F \quad \llbracket \Gamma \vdash \text{inr } t : \tau_1 + \tau_2 \rrbracket^F = \iota_2 \circ \llbracket \Gamma \vdash t : \tau_2 \rrbracket^F$$

$$\begin{aligned} & \llbracket \Gamma \vdash \text{match } t_0 \text{ with inl } x_1 . t_1 \mid \text{inr } x_2 . t_2 : \tau_3 \rrbracket^F \rho \\ &= \begin{cases} \llbracket \Gamma, x_1 : \tau_1 \vdash t_1 : \tau_3 \rrbracket^F (\rho_x)_{x \in \text{Dom } \Gamma \cup \{x_1\}} & \text{if } \llbracket \Gamma \vdash t_0 : \tau_1 + \tau_2 \rrbracket^F \rho = \iota_1 \rho_{x_1} \\ \llbracket \Gamma, x_2 : \tau_2 \vdash t_2 : \tau_3 \rrbracket^F (\rho_x)_{x \in \text{Dom } \Gamma \cup \{x_2\}} & \text{if } \llbracket \Gamma \vdash t_0 : \tau_1 + \tau_2 \rrbracket^F \rho = \iota_2 \rho_{x_2} \end{cases} \\ & \text{where } \Gamma \vdash t_0 : \tau_1 + \tau_2, \quad \Gamma, x_1 : \tau_1 \vdash t_1 : \tau_3, \quad \Gamma, x_2 : \tau_2 \vdash t_2 : \tau_3 \end{aligned}$$

$$\llbracket \Gamma \vdash \text{fun } x : \tau_1 . t : \tau_1 \rightarrow \tau_2 \rrbracket^F \rho = \rho_x \mapsto \llbracket \Gamma, x : \tau_1 \vdash t : \tau_2 \rrbracket^F (\rho_y)_{y \in \text{Dom } \Gamma \cup \{x\}}$$

$$\begin{aligned} & \llbracket \Gamma \vdash t_1 t_2 : \tau_2 \rrbracket^F \rho = \llbracket \Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \rrbracket^F \rho (\llbracket \Gamma \vdash t_2 : \tau_1 \rrbracket^F \rho) \\ & \text{where } \Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2, \quad \Gamma \vdash t_2 : \tau_1 \end{aligned}$$

$$\llbracket \Gamma \vdash \text{return } t : \text{Gen } \tau \rrbracket^F \rho = \text{unit } (\llbracket \Gamma \vdash t : \tau \rrbracket^F \rho)$$

$$\begin{aligned} & \llbracket \Gamma \vdash \text{let } x \leftarrow t_1 \text{ in } t_2 : \text{Gen } \tau_2 \rrbracket^F \rho \\ &= \text{bind}_F (\rho_x \mapsto \llbracket \Gamma, x : \tau_1 \vdash t_2 : \text{Gen } \tau_2 \rrbracket^F (\rho_y)_{y \in \text{Dom } \Gamma \cup \{x\}}) (\llbracket \Gamma \vdash t_1 : \text{Gen } \tau_1 \rrbracket^F \rho) \\ & \text{where } \Gamma \vdash t_1 : \text{Gen } \tau_1, \quad \Gamma, x : \tau_1 \vdash t_2 : \text{Gen } \tau_2 \end{aligned}$$

$$\llbracket \Gamma \vdash \text{flip } p : \text{GenBool} \rrbracket^F \rho = \text{flip}_F p \quad \llbracket \Gamma \vdash \text{shrink} : \underbrace{\tau * \dots * \tau}_{n+1 \text{ times}} \rightarrow \text{Gen } \tau \rrbracket^F \rho = \text{shrink}_F^n$$

Fig. 10. Semantics of Hedgehog.

$$\mathfrak{C} A = (\mathbb{S} \rightarrow_Q \text{Tree } A) \quad \text{unit}_{\mathfrak{C}} a = \sigma \mapsto \text{unit } a \quad \text{bind}_{\mathfrak{C}} f m = \sigma \mapsto \text{bind } (a \mapsto f a (\pi_r \sigma)) (m (\pi_l \sigma))$$

$$\text{flip}_{\mathfrak{C}} p = \sigma \mapsto \text{unit } (\sigma < p) \quad \text{shrink}_{\mathfrak{C}}^n (a_0, a_1, \dots, a_n) = \sigma \mapsto \text{node } (a_0, \text{unit } a_1 :: \dots :: \text{unit } a_n :: \epsilon)$$

Fig. 11. Hedgehog's Sampling Interpretation.

- (3) Sequential composition `let x ← t1 in t2` splits its sample, passing a different half to t₁ and t₂.
- (4) The probabilistic choice operation `flip p` returns `true` if its sample lies in the range [0, p).
- (5) The shrink operation `shrink` ignores its sample and returns a shrink tree comprising of its arguments.

491 Example 3.6. The semantics of $\text{coin}_1, \text{coin}_2 : \text{GenBool}$, given in Example 3.2, are as follows:

$$492 \quad \llbracket \text{coin}_1 \rrbracket^{\mathfrak{C}} \sigma = \text{unit}(\pi_l \sigma < 1/2 \Leftrightarrow \pi_l(\pi_r \sigma) < 1/2) \quad \llbracket \text{coin}_2 \rrbracket^{\mathfrak{C}} \sigma = \text{unit}(\pi_l \sigma < 1/2)$$

494 While coin_1 and coin_2 intuitively represent the same distribution of values, they are not equivalent
495 under the sampling semantics: \mathfrak{C} is overly *fine-grained*.

496 3.3 Program Transformations

498 We often talk about term equivalence under an interpretation (e.g., does $\llbracket \text{coin}_1 \rrbracket^F = \llbracket \text{coin}_2 \rrbracket^F$ for
499 some other interpretation F ?), so we define a shorthand in Definition 3.7.

500 *Definition 3.7 (Hedgehog Semantic Equivalence).* Two Hedgehog terms $\Gamma \vdash t_1, t_2 : \tau$ are *semantically equivalent*
501 under an interpretation F (written $\Gamma \vdash t_1 =^F t_2 : \tau$) if $\llbracket \Gamma \vdash t_1 : \tau \rrbracket^F = \llbracket \Gamma \vdash t_2 : \tau \rrbracket^F$.
502 We write $t_1 =^F t_2 : \tau$ for $(\cdot) \vdash t_1 =^F t_2 : \tau$ and omit the type ascription ($: \tau$) when it can be inferred
503 from the surrounding context.

505 To further illustrate how the sampling interpretation is unsuitable for semantic equivalence
506 proofs, we show that several common program transformations are not permitted under \mathfrak{C} .

507 THEOREM 3.8. *The following equivalences do not hold:*

$$509 \quad \Gamma \vdash \text{let } x \leftarrow \text{return } t \text{ in } f x =^{\mathfrak{C}} f t \quad (1)$$

$$510 \quad \Gamma \vdash \text{let } x \leftarrow t \text{ in return } x =^{\mathfrak{C}} t \quad (2)$$

$$512 \quad \Gamma \vdash \text{let } y \leftarrow (\text{let } x \leftarrow t \text{ in } f x) \text{ in } g y =^{\mathfrak{C}} \text{let } x \leftarrow t \text{ in let } y \leftarrow f x \text{ in } g y \quad (3)$$

$$513 \quad \Gamma \vdash \text{let } x \leftarrow \text{flipp in } t =^{\mathfrak{C}} t \quad (4)$$

$$514 \quad \Gamma \vdash \text{let } x \leftarrow \text{flipp in let } y \leftarrow t \text{ in } f x y =^{\mathfrak{C}} \text{let } y \leftarrow t \text{ in let } x \leftarrow \text{flipp in } f x y \quad (5)$$

$$516 \quad \Gamma \vdash \text{let } x \leftarrow \text{flipp in } f(\text{not } x) =^{\mathfrak{C}} \text{let } x \leftarrow \text{flip}(1 - p) \text{ in } f x \quad (6)$$

$$517 \quad \Gamma \vdash \text{let } x \leftarrow \text{flipp in let } y \leftarrow \text{flip } q \text{ in } f(x \text{ and } y) =^{\mathfrak{C}} \text{let } x \leftarrow \text{flip}(p \cdot q) \text{ in } f x \quad (7)$$

519 Intuitively, none of these transformations are sound under the sampling interpretation because,
520 similar to Example 3.6, they all change how samples are interpreted. We discuss each equation in
521 detail:

- 522 • Equation (1) states that an effect-free computation, when let-bound to a variable, can be
523 inlined. This is an optimization: removing a let expression reduces the number of sample
524 splits under the sampling semantics. Equation (1) is a form of constant folding [Aho et al.
525 2006].
- 526 • Equation (2) formalizes the intuition that `return x` is a “no-op”. Like equation (1), equation (2)
527 is an optimization and reduces the number of sample splits under the sampling semantics.
- 528 • Equation (3) states that the order in which multiple sequential compositions are formed
529 does not matter as long as the overall order of computations remains the same. While not
530 strictly an optimization by itself, equation (3) is necessary to justify further optimizations.
531 For example, if equations (3) and (4) hold then

$$532 \quad \begin{aligned} \text{let } x \leftarrow (\text{let } y \leftarrow t_1 \text{ in } \text{flip}) \text{ in } t_2 &= \text{let } y \leftarrow t_1 \text{ in let } x \leftarrow \text{flip in } t_2 \\ &= \text{let } y \leftarrow t_1 \text{ in } t_2. \end{aligned}$$

535 Equations (1) to (3) are essentially the monad laws, and hence \mathfrak{C} is not a monad.

- 536 • Equation (4) is a standard property of probabilistic choice [Culpepper and Cobb 2017],
537 and states that a flip statement whose result is unused can be removed. This is also an
538 optimization that reduces the number of sample splits under the sampling semantics.

- 540 • **Equation (5)** is another standard property of probabilistic choice [Staton 2017], and states that
 541 the order in which a random value is generated is irrelevant. While not an optimization by
 542 itself, **equation (5)** is necessary to justify further optimizations. For example, if **equations (4)**
 543 and **(5)** hold then

$$\begin{pmatrix} \text{let } x \leftarrow \text{flipp in} \\ \text{let } y \leftarrow \text{shrink... in} \\ \text{let } z \leftarrow \text{flip } q \text{ in} \\ f(x \text{ and } z) \end{pmatrix} = \begin{pmatrix} \text{let } x \leftarrow \text{flipp in} \\ \text{let } z \leftarrow \text{flip } q \text{ in} \\ \text{let } y \leftarrow \text{shrink... in} \\ f(x \text{ and } z) \end{pmatrix} = \begin{pmatrix} \text{let } x \leftarrow \text{flip}(p \cdot q) \text{ in} \\ \text{let } y \leftarrow \text{shrink... in} \\ f x \end{pmatrix}.$$

- 544 • **Equation (6)** states that flipping a biased coin with parameter p is the same as flipping a
 545 biased coin with parameter $1 - p$ and interchanging the results. Since p is a constant in our
 546 setting, **equation (6)** is an optimization because it eliminates a negation (*not*).
 547 • **Equation (7)** provides a condition for merging two calls to `flip`. Since `flip`'s parameter is
 548 a constant in our setting, **equation (7)** is an optimization because it reduces the number of
 549 sample splits under the sampling semantics.

550 In contrast to **Theorem 3.8**, the following transformations *are* permitted under \mathfrak{C} .

551 **THEOREM 3.9.** *The following equivalences hold:*

$$\Gamma \vdash \text{let } x \leftarrow \text{return } t \text{ in shrink } (f_1 x) \dots (f_n x) =^{\mathfrak{C}} \text{shrink } (f_1 t) \dots (f_n t) \quad (8)$$

$$\Gamma \vdash \text{let } x \leftarrow \text{shrink } t_0 \dots t_n \text{ in return } x =^{\mathfrak{C}} \text{shrink } t_0 \dots t_n \quad (9)$$

$$\Gamma \vdash \text{shrink } t =^{\mathfrak{C}} \text{return } t \quad (10)$$

$$\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_2 =^{\mathfrak{C}} t_2 \quad (11)$$

$$\Gamma \vdash \text{if true then } t_1 \text{ else } t_2 =^{\mathfrak{C}} t_1 \quad (12)$$

$$\Gamma \vdash \text{if false then } t_1 \text{ else } t_2 =^{\mathfrak{C}} t_2 \quad (13)$$

$$\Gamma \vdash \text{flip } 0 =^{\mathfrak{C}} \text{return false} \quad (14)$$

$$\Gamma \vdash \text{flip } 1 =^{\mathfrak{C}} \text{return true} \quad (15)$$

$$\Gamma \vdash \begin{pmatrix} \text{let } y \leftarrow (\text{let } x \leftarrow \text{shrink } t_0 \dots t_n \\ \text{in shrink } (f_1 x) \dots (f_m x)) \end{pmatrix} =^{\mathfrak{C}} \begin{pmatrix} \text{let } x \leftarrow \text{shrink } t_0 \dots t_n \text{ in} \\ \text{let } y \leftarrow \text{shrink } (f_1 x) \dots (f_m x) \text{ in} \\ \text{shrink } (g_0 y) \dots (g_k y) \end{pmatrix} \quad (16)$$

575 We describe each equation as follows:

- 576 • **Equations (8), (9) and (16)** are essentially special cases of **Equations (1) to (3)** where the
 577 computations involved only exhibit shrinking effects.
 578 • **Equation (10)** states that a `shrink` term with only one argument (i.e., no specified shrinks)
 579 is identical to a pure computation (`return`).
 580 • **Equations (11) to (13)** are common optimizations of `if` expressions. Unlike the other equa-
 581 tions we discuss, these equations hold under any interpretation.
 582 • **Equations (14) and (15)** state that `flip`'s behaviour is deterministic when its parameter is
 583 either zero or one.

584 From **Theorems 3.8** and **3.9** we infer that the sampling interpretation only permits transformations
 585 of *non-probabilistic terms*. Since generators are primarily probabilistic programs, this renders the
 586 sampling interpretation unsuitable for program optimization.

589 **3.4 Distributions**

590 We have presented in Definitions 3.4 and 3.5 a sampling semantics which closely mirrors the actual
 591 implementation of Hedgehog. However, the specifics of how samples are turned into values are irre-
 592 relevant from the user's perspective. Users are more concerned with a generator's distribution, which
 593 determines how well the input space of the SUT is covered. To this end, the sampling semantics is
 594 not suitable for proving the soundness because it is too fine-grained (e.g., see Example 3.6).

595 *Definition 3.10 (Hedgehog Distribution Semantics).* The *distribution semantics* of any $t : \text{GenBool}$
 596 is given by the function

$$\begin{aligned} 598 \quad \langle - \rangle &: \text{Term} \rightarrow \text{Dist } \mathbb{B} \\ 599 \quad \langle t \rangle &= \text{map } \llbracket t \rrbracket^{\mathfrak{C}} \lambda. \end{aligned}$$

600 Since generators represent random variables, we obtain a distribution via the push-forward
 601 operation.

602 *Example 3.11.* We compute the distributions of $\text{coin}_1, \text{coin}_2 : \text{GenBool}$ given in Example 3.2:

$$\begin{aligned} 604 \quad \langle \text{coin}_1 \rangle &= 1/2 \cdot \text{unit}(\text{unit true}) + 1/2 \cdot \text{unit}(\text{unit false}) \\ 605 \quad \langle \text{coin}_2 \rangle &= 1/2 \cdot \text{unit}(\text{unit true}) + 1/2 \cdot \text{unit}(\text{unit false}) \end{aligned}$$

607 Terms coin_1 and coin_2 both produce single-node trees. Each tree is labelled by either true or false
 608 with probability $1/2$.

609 Building on Example 3.11, we show that Hedgehog's distribution semantics is non-compositional,
 610 i.e., it cannot be represented as an interpretation. Define

612 $\text{discrim} : \text{GenBool} \rightarrow \text{GenBool}$

613 $\text{discrim} = \text{fun } m : \text{GenBool}. \text{ let } b \leftarrow \text{shrink}(\text{true}, \text{false}) \text{ in if } b \text{ then } m \text{ else } \text{coin}_1$

614 and consider the sampling semantics of $\text{discrim} \text{coin}_1$ and $\text{discrim} \text{coin}_2$. On one hand, we have

$$\begin{aligned} 616 \quad \text{discrim} \text{coin}_1 &=^{\mathfrak{C}} \text{let } b \leftarrow \text{shrink}(\text{true}, \text{false}) \text{ in if } b \text{ then } \text{coin}_1 \text{ else } \text{coin}_1 \\ 617 \quad &=^{\mathfrak{C}} \text{let } b \leftarrow \text{shrink}(\text{true}, \text{false}) \text{ in } \text{coin}_1. \end{aligned}$$

618 During shrinking, the sample fed to coin_1 does not change, so any tree produced by $\text{discrim} \text{coin}_1$
 619 will have identical node labels. Hence,

$$621 \quad \langle \text{discrim} \text{coin}_1 \rangle = 1/2 \cdot \text{unit}(\text{node}(\text{true}, \text{unit true} :: \epsilon)) + 1/2 \cdot \text{unit}(\text{node}(\text{false}, \text{unit false} :: \epsilon))$$

622 On the other hand, we have

$$\begin{aligned} 624 \quad \text{discrim} \text{coin}_2 &=^{\mathfrak{C}} \text{let } b \leftarrow \text{shrink}(\text{true}, \text{false}) &=^{\mathfrak{C}} \text{let } b \leftarrow \text{shrink}(\text{true}, \text{false}) \\ 625 \quad &\text{in if } b \text{ then } \text{coin}_2 \text{ else } \text{coin}_1 &\text{in if } b \text{ then} \\ 626 \quad && \text{let } x \leftarrow \text{flip } 1/2 \\ 627 \quad && \text{in return } x \\ 628 \quad && \text{else} \\ 629 \quad && \text{let } y \leftarrow \text{flip } 1/2 \text{ in} \\ 630 \quad && \text{let } z \leftarrow \text{flip } 1/2 \text{ in} \\ 631 \quad && \text{return } y = z. \end{aligned}$$

632 During shrinking, the values of x and y will be identical (as they obtain their values from the same
 633 part of the sample), but there is only a 50% chance that $y = z$. Hence,

$$\begin{aligned} 634 \quad \langle \text{discrim} \text{coin}_2 \rangle &= 1/4 \cdot \text{unit}(\text{node}(\text{true}, \text{unit true} :: \epsilon)) + 1/4 \cdot \text{unit}(\text{node}(\text{true}, \text{unit false} :: \epsilon)) \\ 635 \quad &+ 1/4 \cdot \text{unit}(\text{node}(\text{false}, \text{unit true} :: \epsilon)) + 1/4 \cdot \text{unit}(\text{node}(\text{false}, \text{unit false} :: \epsilon)) \end{aligned}$$

If distribution semantics can be represented as an interpretation, then $(\text{discrim coin}_1) = (\text{discrim coin}_2)$ implies $(\text{discrim coin}_1) = (\text{discrim coin}_2)$. Hence, no such interpretation exists.

PROPOSITION 3.12 (NON-COMPOSITIONALITY OF DISTRIBUTION SEMANTICS). *There is no interpretation F such that $\llbracket t \rrbracket^F = \llbracket t \rrbracket$ whenever $t : \text{GenBool}$.*

3.5 Contextual Equivalence

The sampling interpretation given in [Definition 3.5](#) induces a compositional semantics, but does not accurately model user-level reasoning and is too fine-grained to be useful for generator equivalence proofs ([Example 3.6](#)). Conversely, the distribution semantics given in [Definition 3.10](#) has neither of these drawbacks but is non-compositional. Can we obtain a coarse-grained interpretation that more closely models user-level reasoning? To answer this question, we first define what it means for an interpretation to be *correct*.

In the denotational approach to semantics, two standard tasks are to show that a particular denotational semantics is sound and complete with respect to an existing operational semantics [[Plotkin 1977](#)]. A denotational semantics is sound if any two denotationally equal programs are contextually equivalent, i.e., they produce the same result in any context under the operational semantics. Soundness rules out “bad” interpretations (such as the one where all terms are equal) and ensures that all conclusions made using the denotational semantics are valid in the operational semantics. Conversely, a denotational semantics is complete if any two contextually equivalent programs are denotationally equal. Completeness rules out excessively fine interpretations (such as one where terms are equal iff they are syntactically equal) and ensures that all conclusions that can be made using the operational semantics can also be made using the denotational semantics (i.e., there are no “blind spots” in the reasoning power of the denotational semantics). In the context of Hedgehog, we consider distribution semantics as an operational semantics, and define contextual equivalence accordingly. Then, a correct (i.e., sound and complete) interpretation is one that coincides with contextual equivalence.

Definition 3.13 (Contexts and Contextual Equivalence). The set of contexts Ctx is generated by

$$\begin{aligned} C ::= & \square \mid \text{absurd } C \mid \text{inl } C \mid \text{inr } C \mid \text{match } C \text{ with } \text{inl } x. t \mid \text{inr } x. t \\ & \mid \text{match } t \text{ with } \text{inl } x. C \mid \text{inr } x. t \mid \text{match } t \text{ with } \text{inl } x. t \mid \text{inr } x. C \\ & \mid (C, t) \mid (t, C) \mid \text{fst } C \mid \text{snd } C \mid \text{fun } x : \tau. C \mid C t \mid t C \mid \text{return } C \\ & \mid \text{let } x \leftarrow C \text{ in } t \mid \text{let } x \leftarrow t \text{ in } C. \end{aligned}$$

The *substitution* $C[t]$ of a term t into a context C is obtained by substituting the (unique) instance of \square in C for t . We write $C : (\Gamma \vdash \tau) \longrightarrow (\Gamma' \vdash \tau')$ when $\Gamma' \vdash C[t] : \tau'$ for all terms $\Gamma \vdash t : \tau$. Two terms $\Gamma \vdash t_1, t_2 : \tau$ are *contextually equivalent* (written $\Gamma \vdash t_1 \approx t_2 : \tau$) if $(\llbracket C[t_1] \rrbracket) = (\llbracket C[t_2] \rrbracket)$ for all contexts $C : (\Gamma \vdash \tau) \longrightarrow ((\square) \vdash \tau')$, where $\tau' \in \text{Type}$. We write $t_1 \approx t_2 : \tau$ in place of $(\square) \vdash t_1 \approx t_2 : \tau$ and omit the type ascription $(: \tau)$ when it can be inferred from the surrounding context.

Intuitively, [Definition 3.13](#) states that two terms are contextually equivalent if one can be substituted for the other in any context without changing the overall distribution of the term. For example, we refer to [Example 3.11](#), from which we conclude coin_1 and coin_2 are *not* contextually equivalent: they produce different distributions in the context $\text{discrim } \square$.

We now define our correctness criteria for interpretations.

Definition 3.14. An interpretation F is (1) *sound* if $\Gamma \vdash t_1 =^F t_2 : \tau \implies \Gamma \vdash t_1 \approx t_2 : \tau$, and (2) *complete* if $\Gamma \vdash t_1 \approx t_2 : \tau \implies \Gamma \vdash t_1 =^F t_2 : \tau$.

687 The sampling interpretation \mathfrak{C} is trivially sound (generators which represent the same random
 688 variables have the same distribution). Surprisingly, \mathfrak{C} is also complete! We sketch our completeness
 689 proof for terminating generator terms with no free variables.

690 We proceed by contraposition. In essence, the problems we have observed in Examples 3.2, 3.6
 691 and 3.11 generalize to terms other than coin_1 and coin_2 . Suppose we have two terms $t_1, t_2 : \text{Gen } \tau$
 692 such that $t_1 \neq^{\mathfrak{C}} t_2$. Then there exists some $\sigma \in \mathbb{S}$ such that $\llbracket t_1 \rrbracket^{\mathfrak{C}} \sigma \neq \llbracket t_2 \rrbracket^{\mathfrak{C}} \sigma$. Define

$$\begin{aligned} 693 \text{discrim}' &: ((\text{()}) \vdash \text{Gen } \tau) \longrightarrow ((\text{()}) \vdash \text{Gen } \tau) \\ 694 \text{discrim}' &= \text{let } b \leftarrow \text{shrink}(\text{true}, \text{false}) \text{ in if } b \text{ then } \square \text{ else } t_1. \\ 695 \end{aligned}$$

696 First, we consider the case of $\text{discrim}'[t_1]$:

$$\begin{aligned} 698 \text{discrim}'[t_1] &=^{\mathfrak{C}} \text{let } b \leftarrow \text{shrink}(\text{true}, \text{false}) \text{ in if } b \text{ then } t_1 \text{ else } t_1 \\ 699 &=^{\mathfrak{C}} \text{let } b \leftarrow \text{shrink}(\text{true}, \text{false}) \text{ in } t_1 \\ 700 \end{aligned}$$

701 The first shrink produced by $\text{discrim}'[t_1]$ is always the same element returned during the testing
 702 phase, as b first shrinks from true to false. More concretely, for any $\sigma' \in \mathbb{S}$,

$$704 \llbracket \text{discrim}'[t_1] \rrbracket^{\mathfrak{C}} \sigma' = \text{node}(b, \text{node}(b, \text{xs} :: \text{xs})) \quad (\dagger)$$

705 where $\text{node}(b, \text{xs}) = \llbracket t_1 \rrbracket^{\mathfrak{C}} (\pi_r \sigma')$. Now consider the case of $\text{discrim}'[t_2]$:

$$707 \text{discrim}'[t_2] =^{\mathfrak{C}} \text{let } b \leftarrow \text{shrink}(\text{true}, \text{false}) \text{ in if } b \text{ then } t_2 \text{ else } t_1.$$

708 Without loss of generalization, assume $\sigma = \pi_r \sigma'$ for some $\sigma' \in \mathbb{S}$. Then

$$711 \llbracket \text{discrim}'[t_2] \rrbracket^{\mathfrak{C}} \sigma' = \text{node}(a, \text{node}(b, \text{ys} :: \text{xs}))$$

712 where $(a, \text{xs}) = \llbracket t_2 \rrbracket^{\mathfrak{C}} \sigma$ and $(b, \text{ys}) = \llbracket t_1 \rrbracket^{\mathfrak{C}} \sigma$. Since $\llbracket t_1 \rrbracket^{\mathfrak{C}} \sigma \neq \llbracket t_2 \rrbracket^{\mathfrak{C}} \sigma$, either $a \neq b$ or $\text{xs} \neq \text{ys}$. In either case, the result of $\llbracket \text{discrim}'[t_1] \rrbracket^{\mathfrak{C}} \sigma''$ is *not* of the form described in (†). Thus
 713 $\text{node}(a, \text{node}(b, \text{ys} :: \text{xs}))$ occurs with positive probability under $\llbracket \text{discrim}'[t_2] \rrbracket$ but occurs with
 714 zero probability under $\llbracket \text{discrim}'[t_1] \rrbracket$.

717 THEOREM 3.15. *The sampling interpretation \mathfrak{C} is sound and complete.*

719 COROLLARY 3.16. *Let F be a sound and complete interpretation. Then two terms $\Gamma \vdash t_1, t_2 : \tau$ are
 720 semantically equivalent under F iff they are semantically equivalent under \mathfrak{C} , i.e.,*

$$722 \Gamma \vdash t_1 =^F t_2 \iff \Gamma \vdash t_1 =^{\mathfrak{C}} t_2.$$

723 3.6 Summary

725 We have formalized Hedgehog's syntax as well as its sampling and distribution semantics with
 726 an eye towards proving program optimizations sound. The sampling semantics is compositional
 727 but unusably fine-grained: many common program optimizations are unsound under the sampling
 728 semantics (Theorem 3.8). On the other hand, the distribution semantics is coarse-grained but
 729 non-compositional. By Corollary 3.16, every (sound and complete) compositional semantics must
 730 identify the exact same set of terms as the sampling semantics. Thus, none of the transformations
 731 discussed in Theorem 3.8 are permitted under *any* sound interpretation of Hedgehog, which renders
 732 Hedgehog a poor target for optimization. We stress that Theorem 3.15 is a consequence of combining
 733 probabilistic and shrinking effects: if either `flip` or `shrink` are removed from Hedgehog, then
 734 Theorem 3.15 no longer holds.

4 Hedgehog \rightarrow

In Section 3, we demonstrated that Hedgehog is unsuited for performing generator optimizations. In particular, any two programs with identical distributions can be distinguished within a pathological context, *discrim'*, unless they have identical sampling semantics. To overcome this flaw, we propose to instead work with *Hedgehog \rightarrow* , a restricted version of Hedgehog based on the arrow calculus [Lindley et al. 2010] which rules out such pathological contexts.

Intuitively, *discrim'* exploits hidden statistical dependence between generators in different branches. This dependence is a direct result of both sides of a *match* term being evaluated with the same sample. Thus, Hedgehog \rightarrow rules out this behaviour by ensuring that each branch of a case analysis (*match*) receives different samples. This is achieved by (1) redefining the semantics of case analysis and (2) adding additional type system restrictions.

In this section, we define the Hedgehog \rightarrow language. This section is structurally similar to Section 3. We first define Hedgehog \rightarrow 's syntax and type system as a modification of Hedgehog's (Definition 4.1, Definition 4.2). We then define a sampling semantics for Hedgehog \rightarrow (Definition 4.5) and derive from it a distribution semantics (Definition 4.9). Unlike Hedgehog, Hedgehog \rightarrow 's distribution semantics is compositional (we show it in Theorem 4.12).

4.1 Syntax

Definition 4.1 (Hedgehog \rightarrow Types and Terms). The set of Hedgehog \rightarrow types (denoted by Type $^\bullet$) is defined by the following modifications to Type (Definition 3.1):

$$\tau ::= \dots \mid \text{Gen } \tau \mid \tau \rightsquigarrow \tau$$

The set of Hedgehog \rightarrow terms (denoted by Term $^\bullet$) is defined identically to Term (Definition 3.1), but types are drawn from Type $^\bullet$ instead of Type (e.g., in *fun* $x : \tau . t$).

Hedgehog \rightarrow replaces the generator type Gen τ with a generator function type $\tau_1 \rightsquigarrow \tau_2$. Intuitively, an element of $\tau_1 \rightsquigarrow \tau_2$ is a function that (1) takes in an element of type τ_1 , (2) performs some probabilistic and shrinking effects, and then (3) returns a value of type τ_2 . In other words, $\tau_1 \rightsquigarrow \tau_2$ is analogous to $\tau_1 \rightarrow \text{Gen } \tau_2$. At the term level, Hedgehog \rightarrow reuses the existing function application ($t_1 t_2$) and anonymous function constructs (*fun* $x : \tau . t$) for generator functions.

The most significant changes are in the typing rules (Definition 4.2).

Definition 4.2 (Hedgehog \rightarrow Typing). The set of environments Env $^\bullet$ and the typing relation $(- \vdash^\bullet - : -) \subseteq \text{Env}^\bullet \times \text{Term}^\bullet \times \text{Type}^\bullet$ are defined analogously to Definition 3.3, but we remove the rules for *let*, *return*, *flip*, and *shrink*, and add

$$\frac{\Gamma; x : \tau_1 \vdash^\bullet t : \tau_2}{\Gamma \vdash^\bullet \text{fun } x : \tau_1 . t : \tau_1 \rightsquigarrow \tau_2} \quad \frac{}{\Gamma \vdash^\bullet \text{flip } p : 1 \rightsquigarrow \text{Bool}} \quad \frac{}{\Gamma \vdash^\bullet \text{shrink} : \tau \times \dots \times \tau \rightsquigarrow \tau}$$

where the command typing relation $(- ; - \vdash^\bullet - : -) \subseteq \text{Env}^\bullet \times \text{Env}^\bullet \times \text{Term}^\bullet \times \text{Type}^\bullet$ is mutually defined to be the smallest relation satisfying

$$\frac{\Gamma, \Delta \vdash^\bullet x : \tau}{\Gamma; \Delta \vdash^\bullet \text{return } x : \tau} \quad \frac{x \notin \text{Dom } \Delta \quad \Gamma; \Delta \vdash^\bullet t_1 : \tau_1 \quad \Gamma; \Delta, x : \tau_1 \vdash^\bullet t_2 : \tau_2}{\Gamma; \Delta \vdash^\bullet \text{let } x \leftarrow t_1 \text{ in } t_2 : \tau_2} \quad \frac{\begin{array}{c} x_1, x_2 \notin \text{Dom } \Delta \quad \Gamma, \Delta \vdash^\bullet t_0 : \tau_1 + \tau_2 \\ \Gamma; \Delta, x_1 : \tau_1 \vdash^\bullet t_1 : \tau_3 \quad \Gamma; \Delta, x_2 : \tau_2 \vdash^\bullet t_2 : \tau_3 \end{array}}{\Gamma; \Delta \vdash^\bullet \text{match } t_0 \text{ with inl } x_1 . t_1 \mid \text{inr } x_2 . t_2 : \tau_3}.$$

Hedgehog \rightarrow 's typing rules divide terms into two classes: normal terms (satisfying $- \vdash^\bullet - : -$) and *command terms* (satisfying $- ; - \vdash^\bullet - : -$). Command terms only occur inside an anonymous function with a generator function type. Command terms include effect-free computations (*return*), sequential composition (*let*), effectful function application ($t_1 t_2$), and case analyses which result in commands (*match*). The typing rules for command terms have two contexts: one for normal values (Γ) and one for values produced by the result of another command or given as an input to the function (Δ). In an effectful function application $t_1 t_2$, only the argument term t_2 can refer to any variable in Δ . This rules out command terms such as

$\text{let } f \leftarrow t_1 \text{ in let } x \leftarrow t_2 \text{ in } f x,$

where the effects performed by the last term ($f x$) are determined by the (arbitrarily complex) behaviour of t_1 , i.e. the control flow of a command term is *fixed*.

Example 4.3. The terms coin_1 and coin_2 (Example 3.2) are still typed in Hedgehog \rightarrow , but they are command terms instead of normal terms, i.e.,

$$\Gamma; \Delta \vdash^\bullet \text{coin}_1 : \text{Bool} \quad \Gamma; \Delta \vdash^\bullet \text{coin}_2 : \text{Bool}.$$

To form normal terms, we place each term inside an anonymous function:

$$\begin{array}{ll} \text{coin}_1^\bullet : 1 \rightsquigarrow \text{Bool} & \text{coin}_2^\bullet : 1 \rightsquigarrow \text{Bool} \\ \text{coin}_1^\bullet = \text{fun } u : 1. \text{ let } x \leftarrow \text{flip } 1/2 \text{ in} & \text{coin}_2^\bullet = \text{fun } u : 1. \text{ let } x \leftarrow \text{flip } 1/2 \text{ in} \\ \quad \text{let } y \leftarrow \text{flip } 1/2 \text{ in} & \quad \text{return } x. \\ \quad \text{return } x = y & \end{array}$$

We similarly wrap the function *discrim* as

$$\begin{aligned} \text{discrim}^\bullet : (1 \rightsquigarrow \text{Bool}) \rightarrow (1 \rightsquigarrow \text{Bool}) \\ \text{discrim}^\bullet = \text{fun } f : 1 \rightsquigarrow \text{Bool}. \text{ fun } u : 1. \text{ let } b \leftarrow \text{shrink}(\text{true}, \text{false}) \text{ in} \\ \quad \text{if } b \text{ then } f() \text{ else } \text{coin}_1^\bullet(). \end{aligned}$$

4.2 Semantics

Definition 4.4 (Hedgehog \rightarrow Interpretation and Semantics). A Hedgehog \rightarrow interpretation $F : \mathbf{Qbs} \times \mathbf{Qbs} \rightarrow \mathbf{Qbs}$ is a pre-arrow equipped with additional operations

$$\text{flip}_F : [0, 1] \rightarrow_Q F(1, \mathbb{B}) \quad \text{shrink}_F^n \in F(\prod_{i=0}^n A, A).$$

An interpretation induces a *type and environment semantics* $\llbracket - \rrbracket^F : \text{Type}^\bullet \cup \text{Env}^\bullet \rightarrow \mathbf{Qbs}$, defined analogously to Definition 3.4, but with

$$\llbracket \tau_1 \rightsquigarrow \tau_2 \rrbracket^F = F(\llbracket \tau_1 \rrbracket^F, \llbracket \tau_2 \rrbracket^F).$$

An interpretation also induces, for all $\Gamma \vdash^\bullet t : \tau$, a *term semantics* $\llbracket \Gamma \vdash^\bullet t : \tau \rrbracket^F : \llbracket \Gamma \rrbracket^F \rightarrow_Q \llbracket \tau \rrbracket^F$ defined analogously to Definition 3.4 with

$$\llbracket \Gamma \vdash^\bullet \text{fun } x : \tau_1. t : \tau_1 \rightsquigarrow \tau_2 \rrbracket^F = \llbracket \Gamma; x : \tau_1 \vdash^\bullet t : \tau_2 \rrbracket^F \quad \llbracket \Gamma \vdash^\bullet \text{flip } p : 1 \rightsquigarrow \text{Bool} \rrbracket^F \rho = \text{flip}_F p$$

$$\llbracket \Gamma \vdash^\bullet \text{shrink} : \underbrace{\tau * \dots * \tau}_{n+1 \text{ times}} \rightsquigarrow \tau \rrbracket^F \rho = \text{shrink}_F^n$$

where the function $\llbracket \Gamma; \Delta \vdash^\bullet t : \tau \rrbracket^F : \llbracket \Gamma \rrbracket^F \rightarrow_Q F(\llbracket \Delta \rrbracket^F, \llbracket \tau \rrbracket^F)$ is defined for all $\Gamma; \Delta \vdash^\bullet t : \tau$ in Figure 12.

```

834    $\llbracket \Gamma; \Delta \vdash^* \text{return } x : \tau \rrbracket^F \rho = \text{arr}(\rho' \mapsto \llbracket \Gamma, \Delta \vdash^* x : \tau \rrbracket^F (\rho, \rho'))$ 
835
836    $\llbracket \Gamma; \Delta \vdash^* \text{let } x \leftarrow t_1 \text{ in } t_2 : \tau_2 \rrbracket^F \rho$ 
837    $= \llbracket \Gamma; \Delta, x : \tau_1 \vdash^* t_2 : \tau_2 \rrbracket^F \rho$ 
838    $\circ \text{arr}((\rho', \rho'_x) \mapsto (\rho'_y)_{y \in \text{Dom } \Delta \cup \{x\}})$ 
839    $\circ \text{first}(\llbracket \Gamma; \Delta \vdash^* t_1 : \tau_1 \rrbracket^F \rho)$ 
840    $\circ \text{arr}(\rho' \mapsto (\rho', \rho'))$ 
841   where  $\Gamma; \Delta \vdash^* t_1 : \tau_1, \quad \Gamma; \Delta, x : \tau_1 \vdash^* t_2 : \tau_2$ 
842
843    $\llbracket \Gamma; \Delta \vdash^* t_1 t_2 : \tau_2 \rrbracket^F \rho = \llbracket \Gamma \vdash^* t_1 : \tau_1 \rightsquigarrow \tau_2 \rrbracket^F \rho \circ \llbracket \Gamma; \Delta \vdash^* t_2 : \tau_1 \rrbracket^F \rho$ 
844   where  $\Gamma \vdash^* t_1 : \tau_1 \rightsquigarrow \tau_2, \quad \Gamma; \Delta \vdash^* t_2 : \tau_1$ 
845
846    $\llbracket \Gamma; \Delta \vdash^* \text{match } t_0 \text{ with } \text{inl } x_1 . t_1 \mid \text{inr } x_2 . t_2 : \tau_3 \rrbracket^F \rho$ 
847    $= \text{arr}((a, b) \mapsto (b, a))$ 
848    $\circ \text{left}(\llbracket \Gamma; \Delta, x_2 : \tau_2 \vdash^* t_2 : \tau_3 \rrbracket^F \rho)$ 
849    $\circ \text{arr}((a, b) \mapsto (b, a))$ 
850    $\circ \text{left}(\llbracket \Gamma; \Delta, x_1 : \tau_1 \vdash^* t_1 : \tau_3 \rrbracket^F \rho)$ 
851    $\circ \text{arr}(\rho' \mapsto \text{distrib}(\rho', \llbracket \Gamma, \Delta \vdash^* t_0 : \tau_1 + \tau_2 \rrbracket^F (\rho, \rho')))$ 
852   where  $\Gamma, \Delta \vdash^* t_0 : \tau_1 + \tau_2, \quad \Gamma; \Delta, x_1 : \tau_1 \vdash^* t_1 : \tau_3, \quad \Gamma; \Delta, x_2 : \tau_2 \vdash^* t_2 : \tau_3$ 
853
854   Fig. 12. Hedgehog $^\rightarrow$  Command Semantics.
855

```

$\mathfrak{C}^*(A, B) = (\mathbb{S} \times A \rightarrow_Q \text{Tree } B)$ $(f \circ \mathfrak{C}^* g)(\sigma, a) = \text{bind}(b \mapsto f(\pi_r \sigma, b))(g(\pi_l \sigma, a))$ $\text{left}_{\mathfrak{C}^*} f(\sigma, \iota_1 a) = \text{map } \iota_1(f(\sigma, a))$ $\text{flip}_{\mathfrak{C}^*} p(\sigma, ()) = \text{unit } (\sigma < p)$	$\text{arr}_{\mathfrak{C}^*} f(\sigma, a) = \text{unit}(f a)$ $\text{first}_{\mathfrak{C}^*} f(\sigma, (a, c)) = \text{map } (-, c)(f(\sigma, a))$ $\text{left}_{\mathfrak{C}^*} f(\sigma, \iota_2 b) = \text{unit } (\iota_2 b)$ $\text{shrink}_{\mathfrak{C}^*}^n(\sigma, (a_0, a_1, \dots, a_n)) = \text{node}(a_0, \text{unit } a_1 :: \dots :: \text{unit } a_n :: \epsilon)$
---	---

Fig. 13. Hedgehog $^\rightarrow$'s Sampling Interpretation.

The semantics for terms are adapted from Lindley et al. [2010]. The semantics of Hedgehog $^\rightarrow$ terms are defined analogously to Hedgehog, where an interpretation provides implementations of command terms. One significant difference is that the semantics of match (specifically within a command term) is now also defined by the interpretation. As with Definition 3.4, we only require an interpretation to be a pre-arrow and not an arrow, since our sampling interpretation is not an arrow.

Definition 4.5 (Hedgehog $^\rightarrow$ Sampling Interpretation). The sampling interpretation $\mathfrak{C}^* : \mathbf{Qbs} \times \mathbf{Qbs} \rightarrow \mathbf{Qbs}$ of Hedgehog $^\rightarrow$ is defined in Figure 13.

Hedgehog's sampling semantics defines generators as random variables, and Hedgehog $^\rightarrow$ analogously defines generator functions as random functions. The remaining definitions also mirror Definition 3.4, e.g., where composition $f \circ \mathfrak{C}^* g$ splits its sample between arguments f and g .

Remark 4.6. Only redefining the semantics of match is not sufficient to rule out pathological contexts such as $\text{discrim}'$. Hedgehog $^\rightarrow$'s additional type system restrictions are necessary due to the presence of higher-order computation. For example, recall from Proposition 3.12,

$\text{discrimcoin}_2 =^{\mathfrak{C}} \text{let } b \leftarrow \text{shrink}(\text{true}, \text{false}) \text{ in if } b \text{ then } \text{coin}_1 \text{ else } \text{coin}_2.$

883 This can be expressed equivalently – without any case analysis – as

884 $\text{let } m \leftarrow \text{shrink}(\text{coin}_1, \text{coin}_2) \text{ in } m.$

885 The Hedgehog $^\rightarrow$ version of this term is

887 $\text{let } m \leftarrow \text{shrink}(\text{coin}_1^\bullet, \text{coin}_2^\bullet) \text{ in } m()$

888 which does not satisfy the typing relation because let-bound variables (i.e., m), cannot appear on
889 the left hand side of a function application (as in $m()$).

890 Example 4.7. The sampling semantics of $\text{coin}_1^\bullet, \text{coin}_2^\bullet : 1 \rightsquigarrow \text{Bool}$ (Example 4.3) are nearly
891 identical to their counterparts in Example 3.6:

892 $\llbracket \text{coin}_1^\bullet \rrbracket^{\mathfrak{C}^\bullet}(\sigma, u) = \text{unit}(\sigma_1 < 1/2 \Leftrightarrow \sigma_2 < 1/2) \quad \llbracket \text{coin}_2^\bullet \rrbracket^{\mathfrak{C}^\bullet}(\sigma, u) = \text{unit}(\sigma_1 < 1/2)$

893 where $\sigma_1, \sigma_2 \in \mathbb{S}$ obtained from σ by some sequence of invocations of π_l and π_r . The exact
894 sequences are unimportant, but they are much longer than in Example 3.6 because the semantics
895 of let (Figure 12) involves a large number of compositions ($\circ_{\mathfrak{C}^\bullet}$). Hughes [2004, §4.2] describes
896 how these redundant compositions can be eliminated.

897 On the other hand, the semantics of $\text{discrim}^\bullet \text{ coin}_1^\bullet$ is

898 $\llbracket \text{discrim}^\bullet \text{ coin}_1^\bullet \rrbracket^{\mathfrak{C}^\bullet} \sigma = \text{node}(\sigma_1 < 1/2 \Leftrightarrow \sigma_2 < 1/2, \text{unit}(\sigma_3 < 1/2 \Leftrightarrow \sigma_4 < 1/2) :: \epsilon)$

899 where $\sigma_1, \sigma_2, \sigma_3, \sigma_4 \in \mathbb{S}$ are also obtained from σ by some sequence of invocations of π_l and π_r .
900 Unlike Hedgehog, Hedgehog $^\rightarrow$ ensures that each branch in discrim^\bullet receives an independent
901 sample, so $\text{discrim}^\bullet \text{ coin}_1^\bullet$ does not return trees where all nodes are identically labelled. This
902 example also demonstrates that the command term $\text{if } t \text{ then } t' \text{ else } t'$ is generally *not* identical
903 to t' , i.e., Equation (11) does not hold in Hedgehog $^\rightarrow$.

904 Like Hedgehog, Hedgehog $^\rightarrow$ interpretations induce a semantic equivalence relation.

905 Definition 4.8 (Hedgehog $^\rightarrow$ Semantic Equivalence). Two Hedgehog $^\rightarrow$ terms $\Gamma \vdash^\bullet t_1, t_2 : \tau$ are *semantically equivalent*
906 with respect to an interpretation F (written $\Gamma \vdash^\bullet t_1 =^F t_2 : \tau$) if $\llbracket \Gamma \vdash^\bullet t_1 : \tau \rrbracket^F =$
907 $\llbracket \Gamma \vdash^\bullet t_2 : \tau \rrbracket^F$. We write $t_1 =^F t_2 : \tau$ for $() \vdash^\bullet t_1 =^F t_2 : \tau$ and omit the type ascription $(: \tau)$ when it
908 can be inferred from the surrounding context.

909 4.3 Distributions

910 Definition 4.9 (Hedgehog $^\rightarrow$ Distribution Semantics). The *distribution semantics* of term $t : 1 \rightsquigarrow \text{Bool}$
911 is given by the function

912
$$\begin{aligned} \langle - \rangle^\bullet : \text{Term}^\bullet &\rightarrow \text{Dist}(1 \rightarrow_Q \mathbb{B}) \\ \langle t \rangle^\bullet &= \text{map}(\text{curry} \llbracket t \rrbracket^{\mathfrak{C}^\bullet}) \lambda. \end{aligned}$$

913 Definition 4.9 is the Hedgehog $^\rightarrow$ analogue of Definition 3.10.

914 Example 4.10. We compute the distributions of $\text{coin}_1^\bullet, \text{coin}_2^\bullet : 1 \rightsquigarrow \text{Bool}$ given in Example 4.3:

915 $\langle \text{coin}_1^\bullet \rangle^\bullet = 1/2 \cdot \text{unit}((\text{id}) \mapsto \text{unit true}) + 1/2 \cdot \text{unit}((\text{id}) \mapsto \text{unit false})$

916 $\langle \text{coin}_2^\bullet \rangle^\bullet = 1/2 \cdot \text{unit}((\text{id}) \mapsto \text{unit true}) + 1/2 \cdot \text{unit}((\text{id}) \mapsto \text{unit false})$

917 These distributions are essentially identical to the ones given in Example 3.11.

918 Definition 4.11 (Hedgehog $^\rightarrow$ Distribution Interpretation). The *distribution interpretation* $\mathfrak{D} : \mathbf{Qbs} \rightarrow \mathbf{Qbs}$ is defined in Figure 14.

919 THEOREM 4.12. Suppose $() \vdash^\bullet t : 1 \rightsquigarrow \text{Bool}$. Then $\langle t \rangle^\bullet = \llbracket t \rrbracket^{\mathfrak{D}}$.

$$\begin{aligned}
932 \quad \mathfrak{D}(A, B) &= \text{Dist}(A \rightarrow_{\mathbb{Q}} \text{Tree } B) & \text{arr}_{\mathfrak{D}} \mu &= \text{unit}(\text{unit} \circ \mu) \\
933 \\
934 \quad \mu_1 \circ_{\mathfrak{D}} \mu_2 &= \text{bind}(a_2 \mapsto \text{map}(a_1 \mapsto \text{bind } a_1 \circ a_2) \mu_1) \mu_2 & \text{first}_{\mathfrak{D}} \mu &= \text{map}(\mu' \mapsto (a, c) \mapsto (\mu' a, c)) \mu \\
935 \\
936 \quad \text{left}_{\mathfrak{D}} \mu &= \text{map}(\mu' \mapsto [\iota_1 \circ \mu', \iota_2]) \mu & \text{flip}_{\mathfrak{D}} p &= \text{map}((\sigma, ()) \mapsto \sigma < p) \lambda \\
937 \\
938 \quad \text{shrink}_{\mathfrak{D}}^n &= \text{unit}((a_0, a_1, \dots, a_n) \mapsto \text{node}(a_0, \text{unit } a_1 :: \dots :: \text{unit } a_n :: \epsilon)) \\
939 \\
940 \end{aligned}$$

Fig. 14. Hedgehog \rightarrow 's Distribution Interpretation.

Theorem 4.12 states that, unlike Hedgehog's distribution semantics, Hedgehog \rightarrow 's distribution semantics *are* compositional, as witnessed by the interpretation \mathfrak{D} . Under \mathfrak{D} , generator functions are interpreted as distributions of functions into shrink trees. Note that, in Theorem 4.12, t is a command term and therefore represents an effectful MC: ?? computation.

THEOREM 4.13. *Equations (1) to (16) (Theorems 3.8 and 3.9) all hold for Hedgehog \rightarrow terms under \mathfrak{D} , except for equation (11).*

Theorem 4.13 demonstrates that Hedgehog \rightarrow is considerably more suited for program optimization than Hedgehog. The only transformation which is not valid (as demonstrated in Example 4.7) is equation (11). However, we can recover a special case.

PROPOSITION 4.14. *The following equivalence holds:*

$$\text{let } x \leftarrow \text{flip } p \text{ in if } x \text{ then } t \text{ else } t =^{\mathfrak{D}} t \quad (11')$$

Intuitively, Proposition 4.14 demonstrates that equation (11) holds whenever the condition term does not shrink.

4.4 From Hedgehog \rightarrow to Hedgehog

Definition 4.15. The *translation* from Hedgehog \rightarrow types, environments, and terms to Hedgehog types, environments, and terms is given by the functions $\langle - \rangle_{\text{ty}} : \text{Type}^{\bullet} \rightarrow \text{Type}$, $\langle - \rangle_{\text{env}} : \text{Env}^{\bullet} \rightarrow \text{Env}$, and $\langle - \rangle_{\text{tm}} : \text{Term}^{\bullet} \rightarrow \text{Term}$ defined in Figure 15.

Definition 4.15 defines our translation from Hedgehog \rightarrow terms to Hedgehog terms. The only changes performed by $\langle - \rangle_{\text{tm}}$ are (1) replacing every instance of $\tau_1 \rightsquigarrow \tau_2$ with $\tau_1 \rightarrow \text{Gen } \tau_2$, and (2) adding additional return computations to match terms.

PROPOSITION 4.16. *The following statements hold:*

- (1) Suppose $\Gamma \vdash^{\bullet} t : \tau$. Then $\langle \Gamma \rangle_{\text{env}} \vdash \langle t \rangle_{\text{tm}} : \langle \tau \rangle_{\text{ty}}$.
- (2) Suppose $\Gamma; \Delta \vdash^{\bullet} t : \tau$. Then $\langle \Gamma \rangle_{\text{env}}, \langle \Delta \rangle_{\text{env}} \vdash \langle t \rangle_{\text{tm}} : \langle \tau \rangle_{\text{ty}}$.

Proposition 4.16 shows that our translation preserves typing. As shown in Example 4.7, it is not the case that our translation preserves sampling semantics. However, distribution semantics is preserved.

LEMMA 4.17. *Suppose $() ; () \vdash^{\bullet} t_1 =^{\mathfrak{D}} t_2 : \text{Bool}$. Then $\llbracket \langle t_1 \rangle_{\text{tm}} \rrbracket = \llbracket \langle t_2 \rangle_{\text{tm}} \rrbracket$.*

4.5 Summary

We have introduced Hedgehog \rightarrow , a restricted variant of Hedgehog with a compositional distribution semantics (Theorem 4.12). Hedgehog \rightarrow achieves this by forcing different branches in a case analysis (match) to produce statistically independent values. This cannot be a simple semantic change due to the presence of higher-order computation, so Hedgehog \rightarrow also places restrictions on (effectful)

Types							
981	$\langle 1 \rangle_{\text{ty}} = 1$	$\langle \emptyset \rangle_{\text{ty}} = \emptyset$	$\langle \tau_1 + \tau_2 \rangle_{\text{ty}} = \langle \tau_1 \rangle_{\text{ty}} + \langle \tau_2 \rangle_{\text{ty}}$				
982			$\langle \tau_1 * \tau_2 \rangle_{\text{ty}} = \langle \tau_1 \rangle_{\text{ty}} * \langle \tau_2 \rangle_{\text{ty}}$				
983							
984	$\langle \tau_1 \rightarrow \tau_2 \rangle_{\text{ty}} = \langle \tau_1 \rangle_{\text{ty}} \rightarrow \langle \tau_2 \rangle_{\text{ty}}$	$\langle \tau_1 \rightsquigarrow \tau_2 \rangle_{\text{ty}} = \langle \tau_1 \rangle_{\text{ty}} \rightarrow \text{Gen} \langle \tau_2 \rangle_{\text{ty}}$					
Environments							
985							
986	$\langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle_{\text{env}} = x_1 : \langle \tau_1 \rangle_{\text{ty}}, \dots, x_n : \langle \tau_n \rangle_{\text{ty}}$						
987							
Terms							
988							
989	$\langle x \rangle_{\text{tm}} = x$	$\langle \text{absurd } t \rangle_{\text{tm}} = \text{absurd} \langle t \rangle_{\text{tm}}$	$\langle () \rangle_{\text{tm}} = ()$	$\langle \text{inl } t \rangle_{\text{tm}} = \text{inl} \langle t \rangle_{\text{tm}}$			
990							
991							
992	$\langle \text{inr } t \rangle_{\text{tm}} = \text{inr} \langle t \rangle_{\text{tm}}$	$\langle \text{match } t_1 \text{ with inl } x_1 . t_2 \mid \text{inr } x_2 . t_3 \rangle_{\text{tm}}$ $= \text{match} \langle t_1 \rangle_{\text{tm}} \text{ with}$ $\quad \text{inl } x_1 . \text{let } y \leftarrow \langle t_2 \rangle_{\text{tm}} \text{ in return } y$ $\quad \mid \text{inr } x_2 . \text{let } y \leftarrow \text{return} () \text{ in } \langle t_3 \rangle_{\text{tm}}$					
993							
994							
995							
996	$\langle (t_1, t_2) \rangle_{\text{tm}} = \langle (t_1)_{\text{tm}}, (t_2)_{\text{tm}} \rangle$	$\langle \text{fst } t \rangle_{\text{tm}} = \text{fst} \langle t \rangle_{\text{tm}}$	$\langle \text{snd } t \rangle_{\text{tm}} = \text{snd} \langle t \rangle_{\text{tm}}$				
997	$\langle \text{fun } x : \tau . t \rangle_{\text{tm}} = \text{fun } x : \langle \tau \rangle_{\text{ty}} . \langle t \rangle_{\text{tm}}$	$\langle t_1 t_2 \rangle_{\text{tm}} = \langle t_1 \rangle_{\text{tm}} \langle t_2 \rangle_{\text{tm}}$	$\langle \text{return } t \rangle_{\text{tm}} = \text{return} \langle t \rangle_{\text{tm}}$				
998							
999	$\langle \text{let } x \leftarrow t_1 \text{ in } t_2 \rangle_{\text{tm}} = \text{let } x \leftarrow \langle t_1 \rangle_{\text{tm}} \text{ in } \langle t_2 \rangle_{\text{tm}}$	$\langle \text{flip } p \rangle_{\text{tm}} = \text{flip } p$	$\langle \text{shrink } \rangle_{\text{tm}} = \text{shrink}$				
1000							
1001							
1002							
1003							
1004							
1005							
1006							
1007							
1008							
1009							
1010							
1011							
1012							
1013							
1014							
1015							
1016							
1017							
1018							
1019							
1020							
1021							
1022							
1023							
1024							
1025							
1026							
1027							
1028							
1029							

Fig. 15. Translation from Hedgehog \rightarrow to Hedgehog.

function application in the style of the arrow calculus [Lindley et al. 2010]. The resulting language admits nearly all the program transformations discussed in Theorems 3.8 and 3.9, except for one which holds in a restricted context. We provide a simple translation from Hedgehog \rightarrow back to Hedgehog, which indicates that any Hedgehog program which corresponds to some Hedgehog \rightarrow program is equally amenable to optimization.

5 Evaluation

In Section 4, we described a modification of the Hedgehog language, Hedgehog \rightarrow , which has a compositional distribution semantics which supports a much wider variety of program transformations than Hedgehog (Theorem 4.13). We now evaluate the effects of Hedgehog \rightarrow 's design with respect to other practical considerations. Given that the Hedgehog \rightarrow language obtains a compositional semantics by placing syntactic restrictions on the form of Hedgehog programs, the goal of our evaluation is to answer the following research questions:

RQ1: To what extent can a Hedgehog \rightarrow -based language express existing generators?

RQ2: What is the impact of Hedgehog \rightarrow 's design on program size?

5.1 Setup

We have implemented Hedgehog \rightarrow as a companion library to Hedgehog. The library, hedgehog-arrow, includes a type Gen a b of effectful functions from a to b. The Glasgow Haskell Compiler (GHC) includes syntactic extensions for arrows [Paterson 2001], which allows programs to be written in a syntax similar to what is presented in Section 4. For example, we translate Example 4.3 to Haskell in Figure 16. Anonymous generator functions are constructed using the proc keyword, sequential composition is denoted using do notation, and generator function application is explicitly denoted using the operator “ \rightarrow ”.

In addition to the two basic operators flip and shrink presented in Sections 3 and 4, hedgehog-arrow also includes the following primitives:

```

1030   1 coin1 :: Gen () Bool
1031   2 coin1 = proc () -> do
1032     3   x <- bool_ -> ()
1033     4   y <- bool_ -> ()
1034     5   returnA -> x == y
1035   6
1036   7 coin2 :: Gen () Bool
1037   8 coin2 = proc () -> do
1038     9   x <- bool_ -> ()
1039    10  returnA -> x

```

Fig. 16. Example 4.3 translated to Haskell.

```

1040   1 prune :: Gen a b -> Gen a b
1041   2 prune g = proc x -> do
1042     3   y <- freeze g -> x
1043     4   returnA -> treeLabel y

```

Fig. 17. The prune Function.

Module	# Expressible	Module Size
Hedgehog.Gen	64/68/70	N/A
Test.Example.Basic	5/18/18	417/393 (6.1%)
Test.Example.Confidence	1/1/1	35/37 (-5.4%)
Test.Example.Coverage	7/7/7	251/261 (-3.8%)
Test.Example.Exception	3/3/3	82/85 (-3.5%)
Test.Example.QuasiShow	2/2/2	63/60 (5%)
Test.Example.Resource	10/10/10	398/401 (-0.7%)
Test.Example.RoundTrip	0/3/3	225/216 (4.2%)
Test.Example.STLC	0/16/16	425/402 (5.7%)

Table 1. Numbers of expressible generators per module and module sizes. Generator counts are reported as number of (not expressible/semi-expressible/fully-expressible) generators. Module sizes are reported as (Hedgehog \rightarrow /Hedgehog) module AST nodes.

- 1058
- 1059 (1) Operations `getSize :: Gen a Size` and `resize :: Gen a b -> Gen (Size, a) b` for
 - 1060 controlling the size of generated values.
 - 1061 (2) An operation `freeze :: Gen a b -> Gen a (Tree b)` which runs its argument, capturing
 - 1062 all of its shrinking behaviour and saving it to a tree.

1063 The first feature comes from QuickCheck and solves certain termination issues [Claessen and Hughes 2000]. The second feature is used to modify or eliminate existing shrinking behaviour. For example, the function `prune`, which eliminates all shrinking behaviour from its argument, is implemented using `freeze` in Figure 17. Intuitively, `prune` behaves exactly like `freeze` but discards everything except the root value in the resulting tree (line 4).

1064 We perform our experiments on Hedgehog’s Hedgehog.Gen module. The Hedgehog.Gen module

1065 contains a large collection of common generators and generator combinators. We provide analogues

1066 for each of these generators and combinators in hedgehog-arrow. We also perform experiments

1067 on several modules from Hedgehog’s example repository¹. We rewrote each of these modules to

1068 use hedgehog-arrow. Our full module set is given in Table 1.

1069 The source code for our implementation, examples, and experiments is available at <https://github.com/hedgehog-arr/hedgehog-arr>.

1070
1071
1072
1073
1074
1075
1076
1077
1078

¹Commit 4e9045fafaf9b50efdc68326cd9f643c0d8383a18

```

1079   1 choice :: MonadGen m => [m a] -> m a
1080   2 choice [] = error "Hedgehog.Gen.choice: used with empty list"
1081   3 choice xs = do
1082     4   n <- integral (Range.constant 0 (length xs - 1))
1083     5   xs !! n

```

Fig. 18. The implementation of choice in Hedgehog.

5.2 Results

RQ1: To what extent can a Hedgehog \rightarrow -based language express existing generators? Hedgehog \rightarrow is intuitively a restricted version of Hedgehog, hence we investigate the impact this restriction has on expressivity in practice.

Table 1 lists, per module, how many generators and combinators within those modules

- (1) cannot be expressed in Hedgehog \rightarrow ,
- (2) are *semi-expressible*, i.e., can be expressed using non-expressible generators, and
- (3) are expressible in Hedgehog \rightarrow .

We observed only two combinators in the first category: choice and frequency. To illustrate why these combinators are not expressible, we present a simplified implementation of choice in Figure 18. Intuitively, choice xs selects a random generator x from xs and executes it. During shrinking, choice xs may behave like any x' in xs which occurs earlier than x. Operationally, choice xs works by selecting a random integer n (line 4) and then executing the nth element of xs (line 5). The reason choice cannot be implemented is because it always passes the same sample to its selection xs !! n, which introduces a statistical dependency, which is explicitly disallowed by Hedgehog \rightarrow (see Remark 4.6). The frequency combinator exhibits the same issue. We provide alternative implementations of choice and frequency in hedgehog-arrow which guarantee statistical independence during shrinking. Note that combinators are not generators themselves, but functions used to construct other generators: to our knowledge, no generator deliberately depends on the original behaviour of choice and frequency.

Summary: hedgehog-arrow expresses almost all generators in our evaluation corpus, including those used in practical examples.

RQ2: What is the impact of Hedgehog \rightarrow 's design on program size? The restrictions Hedgehog \rightarrow places on command terms may require users to employ verbose workarounds. We determine whether this is the case by measuring the impact of Hedgehog \rightarrow 's design on program size.

Table 1 lists the size (in AST nodes) of both the Hedgehog and hedgehog-arrow version of each module. We do not compare the Hedgehog.Gen module because the Hedgehog.Gen primarily consists of re-exported internal code.

Our results show that using hedgehog-arrow has a negligible impact on program size.

Summary: hedgehog-arrow programs are not significantly larger than their Hedgehog counterparts.

5.3 Threats to Validity

Our example set is entirely written in Haskell, which is a lazy and purely functional language. Our results may not generalize to strict, imperative languages. Our example set, while fairly large, does not include any particularly large examples (the largest being 122 lines), so our conclusions may not hold for larger programs.

1128 6 Related Work

1129 *Property-Based Testing and Shrinking.* Property-based testing was popularized by the QuickCheck
 1130 [Claessen and Hughes 2000] framework for Haskell, which features a probabilistic eDSL for spec-
 1131 ifying generators. In QuickCheck, a shrinker for a type a is specified separately as a function
 1132 $\text{shrinks} :: a \rightarrow [a]$ from values to their immediate shrinks. In contrast, Hedgehog $^\rightarrow$ employs
 1133 *generator-based shrinking*, where shrinkers are derived from annotated generator specifications.
 1134

1135 Generator-based shrinking approaches were introduced by QuviQ QuickCheck [Claessen et al.
 1136 2009] for Erlang. QuviQ QuickCheck implements generator-based shrinking using *integrated*
 1137 *shrinking*, where generators are represented as tree-valued random variables. Integrated shrinking
 1138 is used by Hedgehog [Stanley 2024] (and hence Hedgehog $^\rightarrow$) and many property-based testing
 frameworks in other languages such as RapidCheck [Eriksson 2024] for C++.
 1139

1140 The Hypothesis [Maciver and Hatfield-Dodds 2019] library for Python implements a different
 1141 approach to generator-based shrinking called *internal shrinking*, where generators are represented
 1142 as random variables drawing from a space of infinite bit sequences, and shrinks are obtained
 1143 by re-executing generators on modified bit sequences obtained via a set of built-in heuristics.
 1144 While hypothesis often produces better shrinks than approaches based on integrated shrinking
 1145 for programs with no user annotations, hypothesis provides no mechanism for user control over
 1146 shrinking. This is less than ideal when the built-in heuristics are insufficient. The falsify library for
 1147 Haskell [de Vries 2023] rectifies this downside while retaining the advantages of internal shrinking
 1148 by allowing users to specify custom shrinking behaviour. De Vries [2023] identify the primary
 1149 advantage of internal shrinking as *hierarchical shrinking* behaviour, where shrinking is allowed
 1150 to backtrack. The Hedgehog language actually exhibits hierarchical shrinking for generators
 1151 implemented using Applicative functor operations [Mcbride and Paterson 2008]. Applicative
 1152 functors and arrows are closely related [Mcbride and Paterson 2008]. Integrating hierarchical
 shrinking into Hedgehog $^\rightarrow$ is left as future work.
 1153

1154 Reflective generators [Goldstein et al. 2023] extend normal generators with *partial monadic*
 1155 *profunctor* operations. Goldstein et al. [2023] show how this extension lends itself to a number of
 1156 features, including the ability to run a generator “backwards” to obtain a random seed from a user-
 1157 provided test case. The reflective generator framework does not mandate a particular interpretation,
 1158 but is incompatible with Hedgehog $^\rightarrow$ ’s semantics since Hedgehog $^\rightarrow$ ’s explicitly disallows the
 general (monadic) form of sequential composition that reflective generators support.
 1159

1160 The formal semantics of generator-based shrinking languages have not been explored in detail
 1161 prior to this work and hence these languages lack sound, formally-specified, compositional rules
 1162 for reasoning about combined generator and shrinker behaviour.
 1163

1164 *Semantics of Probabilistic Programming Languages.* Quasi-Borel Spaces [Heunen et al. 2017]
 1165 form a semantic model of higher-order probabilistic programs, which we use to justify our various
 1166 semantics. Quasi-Borel Pre-Domains [Vákár et al. 2019] combine quasi-Borel spaces with ω -complete
 1167 partial orders [Abramsky and Jung 1995] to allow modelling probabilistic programs with recursion.
 1168 All of our results generalize to this setting except for (possibly) **Theorem 3.15**. The proof of
 1169 **Theorem 3.15** relies on the fact that, whenever $\llbracket t \rrbracket^{\mathbb{C}} \sigma = x$, then $\llbracket t \rrbracket^{\mathbb{C}} \sigma' = x$ for all σ' in some set of
 1170 positive measure. This holds for all *terminating* programs, but not necessarily for non-terminating
 1171 programs. Investigating this case is left for future work. Note that a failure to generalize does not
 1172 reduce the impact of **Theorem 3.15** since it still applies to all terminating programs.
 1173

1174 *Semantics of Probabilistic and Nondeterministic Programming Languages.* Shrinking can be viewed
 1175 as a form of nondeterminism, and there is a large body of work addressing the combination of
 1176 probabilistic and nondeterministic effects [Bandini and Segala 2001; Chen and Sanders 2009; Gibbons
 1177

and Hinze 2011; Mislove et al. 2004; Varacca and Winskel 2006]. Various authors [Gibbons and Hinze 2011; Mislove et al. 2004; Varacca and Winskel 2006] observe that allowing probabilistic choice to distribute over nondeterministic choice, combined with the the idempotence of probabilistic choice, leads to unexpected behaviour. **Theorem 3.15** provides a similar type of observation, specific to Hedgehog. Placing this result in a more general setting is left to future work.

Arrows. Arrows are a generalization of monads [Moggi 1991] introduced to functional programming by Hughes [2000]. Arrows model languages with more restricted control flow than monads allow, which is crucial for our solution. Arrows alone do not model languages with case analysis: the ArrowChoice class [Hughes 2000] in Haskell extends arrows with this ability, and corresponds to the left function required by Hedgehog \rightarrow interpretations. Hedgehog \rightarrow is built on the *arrow calculus* [Lindley et al. 2010], which provides a more familiar presentation of arrows and their equational properties. Paterson [2001] describes a similar notation for Haskell, on which our implementation is based.

7 Conclusion and Future Work

Existing languages that feature user-controlled generator-based shrinking lack sound, compositional semantics for reasoning about generator equivalence. In this paper, we showed that this problem is inherent to the design of existing languages, using Hedgehog as a canonical example. We defined Hedgehog \rightarrow , a restricted version of Hedgehog, and showed that it has an appropriate compositional semantics. We also showed that Hedgehog \rightarrow is expressive enough in practice by translating a large set of existing programs into Hedgehog \rightarrow , and showed that Hedgehog \rightarrow programs are not significantly more tedious to write than Hedgehog programs.

We have shown that many common optimizations are valid within Hedgehog \rightarrow , but these optimizations remain to be implemented. This can be done for Haskell programs either with rewrite rules [Reinders 2024] or a compiler plugin [Farmer et al. 2012]. As mentioned in Section 6, integrating falsify-style [de Vries 2023] hierarchical shrinking into Hedgehog \rightarrow is also left as future work.

Data Availability Statement

The source code for our implementation, examples, and experiments is available at <https://github.com/hedgehog-arr/hedgehog-arr>. The proofs for our theorems can be found in the extended version at <https://github.com/hedgehog-arr/hedgehog-arr/blob/master/extended.pdf>.

References

- Samson Abramsky and Achim Jung. 1995. *Domain theory*. Oxford University Press, Inc., USA, 1–168.
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA.
- Emanuele Bandini and Roberto Segala. 2001. Axiomatizations for Probabilistic Bisimulation. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming, (ICALP '01)*. Springer-Verlag, Berlin, Heidelberg, 370–381.
- Yifeng Chen and J. W. Sanders. 2009. Unifying Probability with Nondeterminism. In *FM 2009: Formal Methods*, Ana Cavalcanti and Dennis R. Dams (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 467–482.
- Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18–21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, New York, NY, 268–279. doi:[10.1145/351240.351266](https://doi.org/10.1145/351240.351266)
- Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. 2009. Finding race conditions in Erlang with QuickCheck and PULSE. *SIGPLAN Not.* 44, 9 (Aug. 2009), 149–160. doi:[10.1145/1631687.1596574](https://doi.org/10.1145/1631687.1596574)
- Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017*,

- 1226 *Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29,*
 1227 *Proceedings* (Uppsala, Sweden). Springer-Verlag, Berlin, Heidelberg, 368–392. doi:[10.1007/978-3-662-54434-1_14](https://doi.org/10.1007/978-3-662-54434-1_14)
- 1228 Ugo Dal Lago, Giulio Guerrieri, and Willem Heijltjes. 2020. Decomposing Probabilistic Lambda-Calculi. In *Foundations of*
 1229 *Software Science and Computation Structures: 23rd International Conference, FOSSACS 2020, Held as Part of the European*
 1230 *Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings* (Dublin,
 1231 Ireland). Springer-Verlag, Berlin, Heidelberg, 136–156. doi:[10.1007/978-3-030-45231-5_8](https://doi.org/10.1007/978-3-030-45231-5_8)
- 1232 Edsko de Vries. 2023. falsify: Internal Shrinking Reimagined for Haskell. In *Proceedings of the 16th ACM SIGPLAN International*
 1233 *Haskell Symposium, Haskell 2023, Seattle, WA, USA, September 8–9, 2023*, Trevor L. McDonell and Niki Vazou (Eds.). ACM,
 1234 New York, NY, 97–109. doi:[10.1145/3609026.3609733](https://doi.org/10.1145/3609026.3609733)
- 1235 Emil Eriksson. 2024. RapidCheck. github.com/emil-e/rapidcheck.
- 1236 Claudia Faggian and Simona Ronchi della Rocca. 2019. Lambda Calculus and Probabilistic Computation. In *2019 34th Annual*
 1237 *ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. doi:[10.1109/LICS.2019.8785699](https://doi.org/10.1109/LICS.2019.8785699)
- 1238 Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. 2012. The HERMIT in the machine: a plugin for the interactive
 1239 transformation of GHC core language programs. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark)
 1240 (*Haskell '12*). Association for Computing Machinery, New York, NY, USA, 1–12. doi:[10.1145/2364506.2364508](https://doi.org/10.1145/2364506.2364508)
- 1241 Jeremy Gibbons and Ralf Hinze. 2011. Just do it: simple monadic equational reasoning. *SIGPLAN Not.* 46, 9 (Sept. 2011),
 1242 2–14. doi:[10.1145/2034574.2034777](https://doi.org/10.1145/2034574.2034777)
- 1243 Harrison Goldstein, Samantha Frohlich, Meng Wang, and Benjamin C Pierce. 2023. Reflecting on Random Generation.
 1244 *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 322–355.
- 1245 Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability
 1246 theory. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science* (Reykjavík, Iceland) (*LICS*
 1247 '17). IEEE Press, Article 77, 12 pages.
- 1248 John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.* 37, 1–3 (May 2000), 67–111. doi:[10.1016/S0167-6423\(99\)00023-4](https://doi.org/10.1016/S0167-6423(99)00023-4)
- 1249 John Hughes. 2004. Programming with arrows. In *Proceedings of the 5th International Conference on Advanced Functional*
 1250 *Programming* (Tartu, Estonia) (*AFP'04*). Springer-Verlag, Berlin, Heidelberg, 73–129. doi:[10.1007/11546382_2](https://doi.org/10.1007/11546382_2)
- 1251 Sam Lindley, Philip Wadler, and Jeremy Yallop. 2010. The arrow calculus. *J. Funct. Program.* 20, 1 (2010), 51–69. doi:[10.1017/S095679680999027X](https://doi.org/10.1017/S095679680999027X)
- 1252 David Maciver and Zac Hatfield-Dodds. 2019. Hypothesis: A new approach to property-based testing. *J. Open Source Softw.*
 1253 4, 43 (2019), 1891. doi:[10.21105/JOSS.01891](https://doi.org/10.21105/JOSS.01891)
- 1254 Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (Jan. 2008), 1–13.
 1255 doi:[10.1017/S0956796807006326](https://doi.org/10.1017/S0956796807006326)
- 1256 Michael Mislove, Joël Ouaknine, and James Worrell. 2004. Axioms for probability and nondeterminism. *Electronic Notes in*
 1257 *Theoretical Computer Science* 96 (2004), 7–28.
- 1258 Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. doi:[10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4)
- 1259 Ross Paterson. 2001. A new notation for arrows. *SIGPLAN Not.* 36, 10 (Oct. 2001), 229–240. doi:[10.1145/507669.507664](https://doi.org/10.1145/507669.507664)
- 1260 Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theor. Comput. Sci.* 5, 3 (1977), 223–255. doi:[10.1016/0304-3975\(77\)90044-5](https://doi.org/10.1016/0304-3975(77)90044-5)
- 1261 Jaro Reinders. 2024. Higher Order Patterns for Rewrite Rules. In *Proceedings of the 17th ACM SIGPLAN International*
 1262 *Haskell Symposium* (Milan, Italy) (*Haskell 2024*). Association for Computing Machinery, New York, NY, USA, 14–26.
 1263 doi:[10.1145/3677999.3678275](https://doi.org/10.1145/3677999.3678275)
- 1264 Jacob Stanley. 2024. Hedgehog. hackage.haskell.org/package/hedgehog.
- 1265 Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *Programming Languages and Systems: 26th*
 1266 *European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice*
 1267 *of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings* (Uppsala, Sweden). Springer-Verlag, Berlin,
 1268 Heidelberg, 855–879. doi:[10.1007/978-3-662-54434-1_32](https://doi.org/10.1007/978-3-662-54434-1_32)
- 1269 Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming. *Proc. ACM*
 1270 *Program. Lang.* 3, POPL, Article 36 (Jan. 2019), 29 pages. doi:[10.1145/3290349](https://doi.org/10.1145/3290349)
- 1271 Daniele Varacca and Glynn Winskel. 2006. Distributing probability over non-determinism. *Mathematical structures in*
 1272 *computer science* 16, 1 (2006), 87–113.
- 1273
- 1274