Compositional Generator Equivalence

ANONYMOUS AUTHOR(S)

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

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

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

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

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

ACM Reference Format:
Anonymous Author(s). 2026. Compositional Generator Equivalence. In *Proceedings of The ACM SIGPLAN International Conference on Functional Programming (ICFP '26).* ACM, New York, NY, USA, 29 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

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

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

ICFP '26, Indianopolis, United States

ACM ISBN XXX-X-XXXX-XXXX-X
https://doi.org/XXXXXXX.XXXXXXX

```
1  coin :: Gen Bool
2  coin = do
3    x ← bool_
4    y ← bool_
5    return (x == y)
6
7  coin' :: Gen Bool
8  coin' = do return ()
9            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.

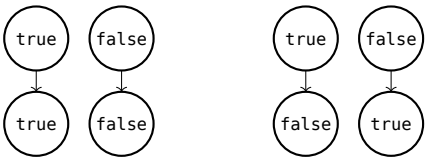
The testing stage proceeds by constructing a random test case using the generators and then checking if the test case is a counterexample using the oracle. If the test case is not a counterexample, the testing process restarts with a different test case; otherwise, the counterexample is passed along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



(a) Expected Trees (b) Additional Trees

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 $\overline{\mathbb{R}}_{\geq 0}$. 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 $[0, 1) \subseteq \overline{\mathbb{R}}_{\geq 0}$.

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 $A_1 \times \ldots \times A_n = \prod_{i=1}^{n} A_i$, and write $(a_1, \ldots, a_n)$ in place of $(a_i)_{i=1}^{n}$. The *nullary product* is denoted by $\mathbf{1}$, 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 $A \to B$. Function application is denoted using juxtaposition $(f\ x)$. The notation $(a \mapsto \ldots)$ 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 $f : B \to C$ and $g : A \to B$ 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 $A_1 + \ldots + A_n = \sum_{i=1}^{n} A_i$. Given a family of functions $f : \prod_{i \in I} (A_i \to B)$, we define the function $[f_i]_{i \in I} : \sum_{i \in I} A_i \to B$ to satisfy $[f_i]_{i \in I}\ (\iota_j\, x) = f_j\, x$ for all $j \in I$. The *nullary coproduct* is denoted by $\mathbf{0}$ $(= \emptyset)$, and is equipped with a (unique) function

$$(\times) : (A \to B) \times (A' \to B') \to A \times A' \to B \times B'$$
$$(f \times g)\,(x, y) = (f\,x, g\,y)$$

$$\mathrm{distrib} : A \times (B + C) \to (A \times B) + (A \times C)$$
$$\mathrm{distrib}\,(x, \iota_1\,y) = \iota_1\,(x, y)$$
$$\mathrm{distrib}\,(x, \iota_2\,y) = \iota_2\,(x, y)$$

$$\mathrm{curry}_{A,B,C} : (A \times B \to C) \to (A \to B \to C)$$
$$\mathrm{curry}_{A,B,C}\,f\,a\,b = f\,(a, b)$$

$$[\ -\ ]_A : \mathbb{B} \to A$$
$$[\mathrm{true}]_A = 1$$
$$[\mathrm{false}]_A = 0$$

Fig. 3. Miscellaneous Functions

$$\mathcal{F}_{\mathbb{B}} = \mathbb{B} \qquad \mathcal{F}_{\mathbb{N}} = \mathbb{N}$$
$$\mathcal{F}_{\overline{\mathbb{R}}_{\geq 0}} = \mathrm{Borel}\,\{\,(r, r') \mid r < r' \in \overline{\mathbb{R}}_{\geq 0}\,\} \qquad \mathcal{F}_{\mathbb{S}} = \mathrm{Borel}\,\{\,(r, r') \mid r < r' \in \mathbb{S}\,\}$$

Fig. 4. Common σ-Algebras.

$$\mathrm{Elem}\,\mathbb{B} = (\overline{\mathbb{R}}_{\geq 0} \to_{\mathbf{M}} \mathbb{B}) \qquad \mathrm{Elem}\,\mathbb{N} = (\overline{\mathbb{R}}_{\geq 0} \to_{\mathbf{M}} \mathbb{N})$$
$$\mathrm{Elem}\,\overline{\mathbb{R}}_{\geq 0} = (\overline{\mathbb{R}}_{\geq 0} \to_{\mathbf{M}} \overline{\mathbb{R}}_{\geq 0}) \qquad \mathrm{Elem}\,\mathbb{S} = (\overline{\mathbb{R}}_{\geq 0} \to_{\mathbf{M}} \mathbb{S})$$
$$\mathrm{Elem} \prod_{i \in I} A_i = \prod_{i \in I} \mathrm{Elem}\,A_i \qquad \mathrm{Elem}\,(A \to_{\mathbf{Q}} B) = \{\mathrm{curry}\,f \mid f : \overline{\mathbb{R}}_{\geq 0} \times A \to_{\mathbf{Q}} B\}$$
$$\mathrm{Elem} \sum_{i \in I} A_i = \left\{ (r \mapsto \iota_{(e\,(i\,r))}\,(\alpha_{(i\,r)}\,r)) \mid i : \overline{\mathbb{R}}_{\geq 0} \to_{\mathbf{Q}} \mathbb{N}, e : \mathbb{N} \to I, \alpha_i \in \mathrm{Elem}\,A_{(e\,i)} \right\}$$

Fig. 5. Common Sets of Random Variables.

$(!_A) : \mathbf{0} \to A$ 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 $\mathrm{true} = \iota_1\,()$ and $\mathrm{false} = \iota_2\,()$.

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 $\mathrm{Borel}\,\mathcal{F}_A \subseteq \mathcal{F}_A$, where $\mathrm{Borel}\,X$ 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 $f : A \to B$ 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 $A \to_{\mathbf{M}} B$.

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

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

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

$$f \circ \mathrm{arr}\,\mathrm{id} = f \qquad (f \circ g) \circ h = f \circ (g \circ h)$$
$$\mathrm{arr}\,(f \circ g) = \mathrm{arr}\,f \circ \mathrm{arr}\,g \qquad \mathrm{first}\,(\mathrm{arr}\,f) = \mathrm{arr}\,(f \times \mathrm{id})$$
$$\mathrm{first}\,(f \circ g) = \mathrm{first}\,f \circ \mathrm{first}\,g \qquad \mathrm{arr}\,(\mathrm{id} \times g) \circ \mathrm{first}\,f = \mathrm{first}\,f \circ \mathrm{arr}\,(\mathrm{id} \times g)$$
$$\mathrm{arr}\,(-_1) \circ \mathrm{first}\,f = f \circ \mathrm{arr}\,(-_1) \qquad \mathrm{arr}\,\mathrm{assoc} \circ \mathrm{first}\,(\mathrm{first}\,f) = \mathrm{first}\,f \circ \mathrm{arr}\,\mathrm{assoc}$$

Fig. 6. Arrow Properties.

Functors, Monads, and Arrows. A *functor* is a mapping $F : \mathbf{Qbs} \to \mathbf{Qbs}$ equipped with a designated function $\mathrm{map}_F : (A \to_{\mathbf{Q}} B) \to_{\mathbf{Q}} (F\,A \to_{\mathbf{Q}} F\,B)$ such that

(1) $\mathrm{map}_F\,\mathrm{id}_A = \mathrm{id}_{F\,A}$ and
(2) $\mathrm{map}_F\,(f \circ g) = \mathrm{map}_F\,f \circ \mathrm{map}_F\,g$ for all $f : B \to_{\mathbf{Q}} C$ and $g : A \to_{\mathbf{Q}} B$.

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

$$\mathrm{unit}_F : A \to_{\mathbf{Q}} F\,A \qquad \mathrm{bind}_F : (A \to_{\mathbf{Q}} F\,B) \to_{\mathbf{Q}} (F\,A \to_{\mathbf{Q}} F\,B).$$

A *monad* is a pre-monad F such that, for all $f : B \to_{\mathbf{Q}} C$ and $g : A \to_{\mathbf{Q}} B$,

$$\mathrm{bind}_F\,\mathrm{unit}_F = \mathrm{id} \qquad \mathrm{bind}_F\,f \circ \mathrm{unit}_F = f \qquad \mathrm{bind}_F\,f \circ \mathrm{bind}_F\,g = \mathrm{bind}_F\,(\mathrm{bind}_F\,f \circ g)$$

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

$$\mathrm{arr}_F : (A \to_{\mathbf{Q}} B) \to_{\mathbf{Q}} F\,(A, B) \qquad \mathrm{first}_F : F\,(A, B) \to_{\mathbf{Q}} F\,(A \times C, B \times C)$$
$$(\circ_F) : F\,(B, C) \times F\,(A, B) \to_{\mathbf{Q}} F\,(A, C) \qquad \mathrm{left}_F : F\,(A, B) \to_{\mathbf{Q}} F\,(A + C, B + C).$$

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

Measures and Distributions. Heunen et al. [2017] construct a *distribution monad* (or *measure monad*) $\mathrm{Dist} : \mathbf{Qbs} \to \mathbf{Qbs}$ equipped with an *integration* operator $(\int - \mathrm{d}-) : (A \to_{\mathbf{M}} \overline{\mathbb{R}}_{\geq 0}) \times \mathrm{Dist}\,A \to_{\mathbf{M}} \overline{\mathbb{R}}_{\geq 0}$ for all $A \in \mathbf{Qbs}$. We write $\int_x y\,\mathrm{d}\mu$ in place of $\int (x \mapsto y)\,\mathrm{d}\mu$ when $\mu \in \mathrm{Dist}\,A$. The integration operator satisfies the properties given in Figure 7 for all $f, f_1, f_2 : A \to_{\mathbf{Q}} \overline{\mathbb{R}}_{\geq 0}$, $x \in A$, $c \in \overline{\mathbb{R}}_{\geq 0}$, $\mu \in \mathrm{Dist}\,A$, $\nu \in \mathrm{Dist}\,B$, $g : B \to_{\mathbf{Q}} A$, and $k_1 \leq k_2 \leq \ldots : A \to_{\mathbf{Q}} \overline{\mathbb{R}}_{\geq 0}$ where k_i are ordered point-wise and converge to a function $k : A \to_{\mathbf{Q}} \overline{\mathbb{R}}_{\geq 0}$. Given $f : A \to_{\mathbf{Q}} B$, $c \in \overline{\mathbb{R}}_{\geq 0}$, and $\mu \in \mathrm{Dist}\,A$, the distributions $\mathrm{map}\,f\,\mu \in \mathrm{Dist}\,B$ (called the *push-forward* of f on μ) and $c \cdot \mu \in \mathrm{Dist}\,A$ satisfy the properties

$$\int g\,\mathrm{d}(\mathrm{map}\,f\,\mu) = \int g \circ f\,\mathrm{d}\mu \qquad \int h\,\mathrm{d}(c \cdot \mu) = c \cdot \int h\,\mathrm{d}\mu$$

for all $g : B \to_{\mathbf{Q}} \overline{\mathbb{R}}_{\geq 0}$ and $h : A \to_{\mathbf{Q}} \overline{\mathbb{R}}_{\geq 0}$.

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

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

for all $f : \mathbb{S} \times \mathbb{S} \to_{\mathbf{Q}} \overline{\mathbb{R}}_{\geq 0}$.

$$\int f \,\mathrm{d}(\mathrm{unit}\, x) = f\, x \qquad \int f \,\mathrm{d}(\mathrm{bind}\, g\, \nu) = \int_x \int f \,\mathrm{d}(g\, x)\,\mathrm{d}\nu$$

$$\int_x c \cdot f\, x \,\mathrm{d}\mu = c \cdot \int f \,\mathrm{d}\mu \qquad \int_x f_1\, x + f_2\, x \,\mathrm{d}\mu = \int f_1 \,\mathrm{d}\mu + \int f_2 \,\mathrm{d}\mu$$

$$\lim_{n\to\infty} \int k_n \,\mathrm{d}\mu = \int k \,\mathrm{d}\mu$$

Fig. 7. Integral Properties.

Monad Operations

$$\begin{array}{ll} \mathrm{unit} : A \to_{\mathbf{Q}} \mathrm{List}\, A & \mathrm{bind} : (A \to_{\mathbf{Q}} \mathrm{List}\, B) \to_{\mathbf{Q}} \mathrm{List}\, A \to_{\mathbf{Q}} \mathrm{List}\, B \\ \mathrm{unit}\, a = a :: \epsilon & \mathrm{bind}\, f\, \epsilon = \epsilon \\ & \mathrm{bind}\, f\, (a :: as) = f\, a +\!\!+ \mathrm{bind}\, f\, as \\ \mathrm{unit} : A \to_{\mathbf{Q}} \mathrm{Tree}\, A & \mathrm{bind} : (A \to_{\mathbf{Q}} \mathrm{Tree}\, B) \to_{\mathbf{Q}} \mathrm{Tree}\, A \to_{\mathbf{Q}} \mathrm{Tree}\, B \\ \mathrm{unit}\, a = \mathrm{node}\, (a, \epsilon) & \mathrm{bind}\, f\, (\mathrm{node}\, (a, as)) = \mathrm{node}\, (b, bs +\!\!+ \mathrm{map}\, (\mathrm{bind}\, f)\, as) \\ & \quad \text{where node}\, (b, bs) = f\, a \end{array}$$

Auxiliary Definitions

$$\begin{array}{l} (+\!\!+) : \mathrm{List}\, A \times \mathrm{List}\, A \to_{\mathbf{Q}} \mathrm{List}\, A \\ \epsilon +\!\!+ bs = bs \\ (a :: as) +\!\!+ bs = a :: (as +\!\!+ bs) \end{array}$$

Fig. 8. Lists and Rose Trees.

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

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

The functions $((x, xs) \mapsto x :: xs)$, $((x, xs) \mapsto \mathrm{node}\, (x, xs))$, and any functions built using structural recursion with quasi-measurable functions, are themselves quasi-measurable. The mappings $\mathrm{List}, \mathrm{Tree} : \mathbf{Qbs} \to \mathbf{Qbs}$ form monads with the operations defined in Figure 8.

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

$$\texttt{Bool} = 1 + 1 \qquad \texttt{true} = \texttt{inl}\,() \qquad \texttt{false} = \texttt{inr}\,()$$

$$\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 = \texttt{match}\ t_1\ \texttt{with}\ \texttt{inl}\,x\texttt{.}\ t_2 \mid \texttt{inr}\,x\texttt{.}\ t_3 \qquad (t_1, t_2, \ldots, t_n) = (t_1, (t_2, \ldots, t_n))$$

$$\texttt{not}\,t_1 = \texttt{if}\ t_1\ \texttt{then false else true} \qquad t_1 = t_2 = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else not}\,t_2$$

$$t_1\ \texttt{and}\ t_2 = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else false}$$

Fig. 9. Derived Constructs.

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.

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

$$\tau ::= \texttt{0} \mid \texttt{1} \mid \tau + \tau \mid \tau * \tau \mid \tau \to \tau \mid \texttt{Gen}\,\tau$$

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

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

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 ($\texttt{0}$), the singleton type (1), sum types ($\tau + \tau$), product types ($\tau * \tau$), function types ($\tau \to \tau$), and generator types ($\texttt{Gen}\,\tau$). Intuitively, the type $\texttt{Gen}\,\tau$ 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 ($\texttt{inl}$, $\texttt{inr}$), eliminations for values of the empty type ($\texttt{absurd}\,t$), case analysis ($\texttt{match}\ t\ \texttt{with}\ \texttt{inl}\,x\texttt{.}\ t \mid \texttt{inr}\,x\texttt{.}\ t$), the singleton value ($\texttt{()}$), tuples ($(t, t)$), anonymous functions ($\texttt{fun}\ x{:}\tau\texttt{.}\ t$), function application ($t\ t$), effect-free computations ($\texttt{return}\,t$), and sequential composition ($\texttt{let}\ x \leftarrow t\ \texttt{in}\ t$). We derive some common constructs in Figure 9. In addition to standard constructs, Hedgehog has the following effectful operations:

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

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

$$\begin{aligned} \texttt{coin}_1 = {} & \texttt{let}\ x \leftarrow \texttt{flip}\,1/2\ \texttt{in} \\ & \texttt{let}\ y \leftarrow \texttt{flip}\,1/2\ \texttt{in} \\ & \texttt{return}\,x = y \end{aligned} \qquad \begin{aligned} \texttt{coin}_2 = {} & \texttt{let}\ x \leftarrow \texttt{flip}\,1/2\ \texttt{in} \\ & \texttt{return}\,x \end{aligned}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \mathtt{inl}\, t : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \mathtt{inr}\, t : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash t : \mathtt{0}}{\Gamma \vdash \mathtt{absurd}\, t : \tau}$$

$$\frac{\begin{array}{c} x_1, x_2 \notin \mathrm{Dom}\,\Gamma \qquad \Gamma \vdash t_0 : \tau_1 + \tau_2 \\ \Gamma, x_1 : \tau_1 \vdash t_1 : \tau_3 \qquad \Gamma, x_2 : \tau_2 \vdash t_2 : \tau_3 \end{array}}{\Gamma \vdash \mathtt{match}\ t_0\ \mathtt{with}\ \mathtt{inl}\, x_1 \,.\, t_1 \mid \mathtt{inr}\, x_2 \,.\, t_2 : \tau_3} \qquad \frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 * \tau_2}{\Gamma \vdash \mathtt{fst}\, t : \tau_1} \qquad \frac{\Gamma \vdash t : \tau_1 * \tau_2}{\Gamma \vdash \mathtt{snd}\, t : \tau_2} \qquad \frac{x \notin \mathrm{Dom}\,\Gamma \qquad \Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \mathtt{fun}\ x : \tau_1 \,.\, t : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathtt{return}\, t : \mathtt{Gen}\, \tau}$$

$$\frac{}{\Gamma \vdash \mathtt{flip}\, p : \mathtt{Gen\,Bool}} \qquad \frac{x \notin \mathrm{Dom}\,\Gamma \qquad \Gamma \vdash t_1 : \mathtt{Gen}\, \tau_1 \qquad \Gamma, x : \tau_1 \vdash t_2 : \mathtt{Gen}\, \tau_2}{\Gamma \vdash \mathtt{let}\ x \leftarrow t_1\ \mathtt{in}\ t_2 : \mathtt{Gen}\, \tau_2}$$

$$\frac{}{\Gamma \vdash \mathtt{shrink} : \tau * \cdots * \tau \to \mathtt{Gen}\, \tau}$$

Fig. 10. Hedgehog's Typing Rules.

Definition 3.3 (Hedgehog Environments and Typing). An *environment* $\Gamma \in \prod_{x \in \mathrm{Dom}\,\Gamma} \mathrm{Type}$ is a family mapping variables from a finite *domain* $\mathrm{Dom}\,\Gamma \subseteq \mathrm{Var}$ to types. The set of environments is denoted by Env, and the singleton environment is written $x : \tau$. The *typing relation* $(- \vdash - : -) \subseteq \mathrm{Env} \times \mathrm{Term} \times \mathrm{Type}$ is the smallest relation satisfying the rules in Figure 10. 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.

3.2 Semantics

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

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

An interpretation induces (1) a *type and environment semantics* $[\![-]\!]^F : \mathrm{Type} \cup \mathrm{Env} \to \mathbf{Qbs}$ and (2) for all $\Gamma \vdash t : \tau$ a *term semantics* $[\![\Gamma \vdash t : \tau]\!]^F : [\![\Gamma]\!]^F \to_{\mathbf{Q}} [\![\tau]\!]^F$ (see Figure 11). When t has no free variables, we write $[\![t : \tau]\!]^F$ in place of $[\![() \vdash t : \tau]\!]^F()$. 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 $[\![\Gamma]\!] = \prod_{x \in \mathrm{Dom}\,\Gamma} \Gamma_x$). 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.

Type and Environment Semantics

$$[\![\mathtt{0}]\!]^F = \mathbf{0} \qquad [\![\mathtt{1}]\!]^F = \mathbf{1} \qquad [\![\tau_1 + \tau_2]\!]^F = [\![\tau_1]\!]^F + [\![\tau_2]\!]^F \qquad [\![\tau_1 * \tau_2]\!]^F = [\![\tau_1]\!]^F \times [\![\tau_2]\!]^F$$

$$[\![\tau_1 \to \tau_2]\!]^F = [\![\tau_1]\!]^F \to_{\mathbf{Q}} [\![\tau_2]\!]^F \qquad [\![\mathsf{Gen}\,\tau]\!]^F = F\,[\![\tau]\!]^F \qquad [\![\Gamma]\!]^F = \prod_{x \in \mathrm{Dom}\,\Gamma} [\![\Gamma_x]\!]^F$$

Term Semantics

$$[\![\Gamma, x : \tau \vdash x : \tau]\!]^F\,\rho = \rho_x \qquad [\![\Gamma \vdash () : \mathtt{1}]\!]^F\,\rho = () \qquad [\![\Gamma \vdash \mathtt{absurd}\,t : \tau]\!]^F\,\rho = !\,([\![\Gamma \vdash t : \mathtt{0}]\!]^F\,\rho)$$

$$([\![\Gamma \vdash \mathtt{fst}\,t : \tau_1]\!]^F\,\rho, [\![\Gamma \vdash \mathtt{snd}\,t : \tau_2]\!]^F\,\rho) = [\![\Gamma \vdash t : \tau_1 * \tau_2]\!]^F\,\rho$$

$$[\![\Gamma \vdash (t_1, t_2) : \tau_1 * \tau_2]\!]^F\,\rho = ([\![\Gamma \vdash t_1 : \tau_1]\!]^F\,\rho, [\![\Gamma \vdash t_2 : \tau_2]\!]^F\,\rho)$$

$$[\![\Gamma \vdash \mathtt{inl}\,t : \tau_1 + \tau_2]\!]^F = \iota_1 \circ [\![\Gamma \vdash t : \tau_1]\!]^F \qquad [\![\Gamma \vdash \mathtt{inr}\,t : \tau_1 + \tau_2]\!]^F = \iota_2 \circ [\![\Gamma \vdash t : \tau_2]\!]^F$$

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

$$[\![\Gamma \vdash \mathtt{fun}\ x : \tau_1\,.\,t : \tau_1 \to \tau_2]\!]^F\,\rho = \rho_x \mapsto [\![\Gamma, x : \tau_1 \vdash t : \tau_2]\!]^F\,(\rho_y)_{y \in \mathrm{Dom}\,\Gamma \cup \{x\}}$$

$$[\![\Gamma \vdash t_1\,t_2 : \tau_2]\!]^F\,\rho = [\![\Gamma \vdash t_1 : \tau_1 \to \tau_2]\!]^F\,\rho\,([\![\Gamma \vdash t_2 : \tau_1]\!]^F\,\rho) \quad \text{where } \Gamma \vdash t_1 : \tau_1 \to \tau_2, \quad \Gamma \vdash t_2 : \tau_1$$

$$[\![\Gamma \vdash \mathtt{return}\,t : \mathsf{Gen}\,\tau]\!]^F\,\rho = \mathrm{unit}\,([\![\Gamma \vdash t : \tau]\!]^F\,\rho)$$

$$[\![\Gamma \vdash \mathtt{let}\ x \leftarrow t_1\ \mathtt{in}\ t_2 : \mathsf{Gen}\,\tau_2]\!]^F\,\rho = \mathrm{bind}_F\,(\rho_x \mapsto [\![\Gamma, x : \tau_1 \vdash t_2 : \mathsf{Gen}\,\tau_2]\!]^F\,(\rho_y)_{y \in \mathrm{Dom}\,\Gamma \cup \{x\}})\,([\![\Gamma \vdash t_1 : \mathsf{Gen}\,\tau_1]\!]^F\,\rho)$$
$$\text{where } \Gamma \vdash t_1 : \mathsf{Gen}\,\tau_1, \quad \Gamma, x : \tau_1 \vdash t_2 : \mathsf{Gen}\,\tau_2$$

$$[\![\Gamma \vdash \mathtt{flip}\,p : \mathsf{Gen}\,\mathsf{Bool}]\!]^F\,\rho = \mathrm{flip}_F\,p \qquad [\![\Gamma \vdash \mathtt{shrink} : \underbrace{\tau * \cdots * \tau}_{n+1 \text{ times}} \to \mathsf{Gen}\,\tau]\!]^F\,\rho = \mathrm{shrink}_F^n$$

Fig. 11. Semantics of Hedgehog.

$$\mathfrak{C}\,A = (\mathbb{S} \to_{\mathbf{Q}} \mathrm{Tree}\,A) \qquad \mathrm{unit}_{\mathfrak{C}}\,a = \sigma \mapsto \mathrm{unit}\,a \qquad \mathrm{bind}_{\mathfrak{C}}\,f\,m = \sigma \mapsto \mathrm{bind}\,(a \mapsto f\,a\,(\pi_r\,\sigma))\,(m\,(\pi_l\,\sigma))$$

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

Fig. 12. Hedgehog's Sampling Interpretation.

Definition 3.5 (Hedgehog Sampling Interpretation). The *sampling interpretation* $\mathfrak{C} : \mathbf{Qbs} \to \mathbf{Qbs}$ of Hedgehog is defined in Figure 12.

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

(1) Generators ($\mathsf{Gen}\,\tau$) 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 `return` t ignores its sample and returns a tree with a single node labelled by the value of t.
(3) Sequential composition `let` $x \leftarrow t_1$ `in` t_2 splits its sample, passing a different half to t_1 and t_2.
(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.

Example 3.6. The semantics of $\texttt{coin}_1, \texttt{coin}_2 : \texttt{Gen Bool}$, given in Example 3.2, are as follows:

$$[\![\texttt{coin}_1]\!]^{\mathfrak{C}}\,\sigma = \text{unit}\,(\pi_l\,\sigma < 1/2 \Leftrightarrow \pi_l\,(\pi_r\,\sigma) < 1/2) \qquad [\![\texttt{coin}_2]\!]^{\mathfrak{C}}\,\sigma = \text{unit}\,(\pi_l\,\sigma < 1/2)$$

While $\texttt{coin}_1$ and $\texttt{coin}_2$ intuitively represent the same distribution of values, they are not equivalent under the sampling semantics: $\mathfrak{C}$ is overly *fine-grained*.

3.3 Program Transformations

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

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

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

Theorem 3.8. *The following equivalences* do not hold*:*

$$\Gamma \vdash \texttt{let}\ x \leftarrow \texttt{return}\ t\ \texttt{in}\ f\ x =^{\mathfrak{C}} f\ t \tag{1}$$

$$\Gamma \vdash \texttt{let}\ x \leftarrow t\ \texttt{in return}\ x =^{\mathfrak{C}} t \tag{2}$$

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

$$\Gamma \vdash \texttt{let}\ x \leftarrow \texttt{flip}\ p\ \texttt{in}\ t =^{\mathfrak{C}} t \tag{4}$$

$$\Gamma \vdash \texttt{let}\ x \leftarrow \texttt{flip}\ p\ \texttt{in let}\ y \leftarrow t\ \texttt{in}\ f\ x\ y =^{\mathfrak{C}} \texttt{let}\ y \leftarrow t\ \texttt{in let}\ x \leftarrow \texttt{flip}\ p\ \texttt{in}\ f\ x\ y \tag{5}$$

$$\Gamma \vdash \texttt{let}\ x \leftarrow \texttt{flip}\ p\ \texttt{in}\ f\ (\texttt{not}\ x) =^{\mathfrak{C}} \texttt{let}\ x \leftarrow \texttt{flip}\ (1-p)\ \texttt{in}\ f\ x \tag{6}$$

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

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

- Equation (1) states that an effect-free computation, when `let`-bound to a variable, can be inlined. This is an optimization: removing a `let` expression reduces the number of sample splits under the sampling semantics. Equation (1) is a form of constant folding [Aho et al. 2006].
- Equation (2) formalizes the intuition that `return` x is a "no-op". Like equation (1), equation (2) is an optimization and reduces the number of sample splits under the sampling semantics.
- Equation (3) states that the order in which multiple sequential compositions are formed does not matter as long as the overall order of computations remains the same. While not

strictly an optimization by itself, equation (3) is necessary to justify further optimizations. For example, if equations (3) and (4) hold then

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

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

- Equation (4) is a standard property of probabilistic choice [Culpepper and Cobb 2017], and states that a `flip` statement whose result is unused can be removed. This is also an optimization that reduces the number of sample splits under the sampling semantics.
- Equation (5) is another standard property of probabilistic choice [Staton 2017], and states that the order in which a random value is generated is irrelevant. While not an optimization by itself, equation (5) is necessary to justify further optimizations. For example, if equations (4) and (5) hold then

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

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

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

Theorem 3.9. *The following equivalences hold:*

$$\Gamma \vdash \texttt{let}\ x \leftarrow \texttt{return}\ t\ \texttt{in}\ \texttt{shrink}\ (f_1\ x)\ldots(f_n\ x) =^{\mathfrak{C}} \texttt{shrink}\ (f_1\ t)\ldots(f_n\ t) \tag{8}$$

$$\Gamma \vdash \texttt{let}\ x \leftarrow \texttt{shrink}\ t_0 \ldots t_n\ \texttt{in}\ \texttt{return}\ x =^{\mathfrak{C}} \texttt{shrink}\ t_0 \ldots t_n \tag{9}$$

$$\Gamma \vdash \texttt{shrink}\ t =^{\mathfrak{C}} \texttt{return}\ t \tag{10}$$

$$\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_2 =^{\mathfrak{C}} t_2 \tag{11}$$

$$\Gamma \vdash \texttt{if}\ \texttt{true}\ \texttt{then}\ t_1\ \texttt{else}\ t_2 =^{\mathfrak{C}} t_1 \tag{12}$$

$$\Gamma \vdash \texttt{if}\ \texttt{false}\ \texttt{then}\ t_1\ \texttt{else}\ t_2 =^{\mathfrak{C}} t_2 \tag{13}$$

$$\Gamma \vdash \texttt{flip}\ 0 =^{\mathfrak{C}} \texttt{return}\ \texttt{false} \tag{14}$$

$$\Gamma \vdash \texttt{flip}\ 1 =^{\mathfrak{C}} \texttt{return}\ \texttt{true} \tag{15}$$

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

We describe each equation as follows:

- Equations (8), (9) and (16) are essentially special cases of Equations (1) to (3) where the computations involved only exhibit shrinking effects.
- Equation (10) states that a `shrink` term with only one argument (i.e., no specified shrinks) is identical to a pure computation (`return`).

- Equations (11) to (13) are common optimizations of `if` expressions. Unlike the other equations we discuss, these equations hold under any interpretation.
- Equations (14) and (15) state that `flip`'s behaviour is deterministic when its parameter is either zero or one.

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

3.4 Distributions

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

Definition 3.10 (Hedgehog Distribution Semantics). The *distribution semantics* of any $t : \mathtt{Gen\ Bool}$ is given by the function

$$\begin{aligned}(\!|-|\!) &: \mathrm{Term} \to \mathrm{Dist}\,\mathbb{B}\\ (\!|\,t\,|\!) &= \mathrm{map}\,[\![\,t\,]\!]^{\mathfrak{C}}\,\lambda.\end{aligned}$$

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

Example 3.11. We compute the distributions of $\mathtt{coin}_1, \mathtt{coin}_2 : \mathtt{Gen\ Bool}$ given in Example 3.2:

$$\begin{aligned}(\!|\mathtt{coin}_1|\!) &= 1/2 \cdot \mathrm{unit}\,(\mathrm{unit}\,\mathrm{true}) + 1/2 \cdot \mathrm{unit}\,(\mathrm{unit}\,\mathrm{false})\\ (\!|\mathtt{coin}_2|\!) &= 1/2 \cdot \mathrm{unit}\,(\mathrm{unit}\,\mathrm{true}) + 1/2 \cdot \mathrm{unit}\,(\mathrm{unit}\,\mathrm{false})\end{aligned}$$

Terms $\mathtt{coin}_1$ and $\mathtt{coin}_2$ both produce single-node trees. Each tree is labelled by either true or false with probability $1/2$.

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

```
discrim : Gen Bool → Gen Bool
discrim = fun m : Gen Bool. let b ← shrink (true, false) in if b then m else coin₁
```

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

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

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

$$(\!|\mathtt{discrim\,coin}_1|\!) = 1/2 \cdot \mathrm{unit}\,(\mathrm{node}\,(\mathrm{true}, \mathrm{unit}\,\mathrm{true} :: \epsilon)) + 1/2 \cdot \mathrm{unit}\,(\mathrm{node}\,(\mathrm{false}, \mathrm{unit}\,\mathrm{false} :: \epsilon))$$

On the other hand, we have

$$\begin{array}{lll} \mathtt{discrim}\,\mathtt{coin}_2 & =^{\mathfrak{C}} & \mathtt{let}\ b \leftarrow \mathtt{shrink}\ (\mathtt{true}, \mathtt{false}) \\ & & \mathtt{in}\ \mathtt{if}\ b\ \mathtt{then}\ \mathtt{coin}_2\ \mathtt{else}\ \mathtt{coin}_1 \end{array} \quad =^{\mathfrak{C}} \quad \begin{array}{l} \mathtt{let}\ b \leftarrow \mathtt{shrink}\ (\mathtt{true}, \mathtt{false}) \\ \mathtt{in}\ \mathtt{if}\ b\ \mathtt{then} \\ \quad \mathtt{let}\ x \leftarrow \mathtt{flip}\ 1/2 \\ \quad \mathtt{in}\ \mathtt{return}\ x \\ \mathtt{else} \\ \quad \mathtt{let}\ y \leftarrow \mathtt{flip}\ 1/2\ \mathtt{in} \\ \quad \mathtt{let}\ z \leftarrow \mathtt{flip}\ 1/2\ \mathtt{in} \\ \quad\ \mathtt{return}\ y = z. \end{array}$$

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

$$\begin{aligned} (\!|\mathtt{discrim}\,\mathtt{coin}_2|\!) = {} & 1/4 \cdot \mathrm{unit}\,(\mathrm{node}\,(\mathrm{true}, \mathrm{unit}\,\mathrm{true} :: \epsilon)) + 1/4 \cdot \mathrm{unit}\,(\mathrm{node}\,(\mathrm{true}, \mathrm{unit}\,\mathrm{false} :: \epsilon)) \\ & + 1/4 \cdot \mathrm{unit}\,(\mathrm{node}\,(\mathrm{false}, \mathrm{unit}\,\mathrm{true} :: \epsilon)) + 1/4 \cdot \mathrm{unit}\,(\mathrm{node}\,(\mathrm{false}, \mathrm{unit}\,\mathrm{false} :: \epsilon)) \end{aligned}$$

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

Proposition 3.12 (Non-Compositionality of Distribution Semantics). *There is no interpretation F such that $[\![t]\!]^F = (\!|t|\!)$ whenever $t : \mathsf{Gen}\,\mathtt{Bool}$.*

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

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

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 $(\!|C[t_1]|\!) = (\!|C[t_2]|\!)$ for all contexts $C : (\Gamma \vdash \tau) \longrightarrow ((\,) \vdash \tau')$, where $\tau' \in \mathrm{Type}$. We write $t_1 \approx t_2 : \tau$ in place of $(\,) \vdash t_1 \approx t_2 : \tau$ 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 $\texttt{coin}_1$ and $\texttt{coin}_2$ are *not* contextually equivalent: they produce different distributions in the context $\texttt{discrim}\,\square$.

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 $\Gamma \vdash t_1 \approx t_2 : \tau \implies \Gamma \vdash t_1 =^F t_2 : \tau$.

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

We proceed by contraposition. In essence, the problems we have observed in Examples 3.2, 3.6 and 3.11 generalize to terms other than $\texttt{coin}_1$ and $\texttt{coin}_2$. Suppose we have two terms $t_1, t_2 : \texttt{Gen}\,\tau$ such that $t_1 \neq^{\mathfrak{C}} t_2$. Then there exists some $\sigma \in \mathbb{S}$ such that $[\![t_1]\!]^{\mathfrak{C}}\,\sigma \neq [\![t_2]\!]^{\mathfrak{C}}\,\sigma$. Define

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

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

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

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

$$[\![\texttt{discrim}'[t_1]]\!]^{\mathfrak{C}}\,\sigma' = \mathrm{node}\,(b, \mathrm{node}\,(b, xs) :: xs) \tag{$\dagger$}$$

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

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

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

$$[\![\texttt{discrim}'[t_2]]\!]^{\mathfrak{C}}\,\sigma' = \mathrm{node}\,(a, \mathrm{node}\,(b, ys) :: xs)$$

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

Theorem 3.15. *The sampling interpretation $\mathfrak{C}$ is sound and complete.*

Proof. *Soundness.* Suppose $\Gamma \vdash t_1 =^{\mathfrak{C}} t_2 : \tau$ and $C : (\Gamma \vdash \tau) \longrightarrow (() \vdash \texttt{Bool})$. Since $\mathfrak{C}$ is (by definition) compositional, we have $\Gamma \vdash C[t_1] =^{\mathfrak{C}} C[t_2] : \tau$. Hence $(\!|C[t_1]|\!) = (\!|C[t_2]|\!)$.

Completeness. Let

$$P\,\tau \equiv \forall \Gamma \vdash t_1, t_2 : \tau, \Gamma \vdash t_1 \approx t_2 \implies \Gamma \vdash t_1 =^{\mathfrak{C}} t_2 : \tau.$$

Given τ, we prove $P\,\tau$ by structural induction on τ.

- *Cases $\tau = \mathtt{0}$ and $\tau = 1$.* These cases are trivial by the fact that all elements of these types are equal.
- *Case $\tau = \tau_1 * \tau_2$.* Suppose $\Gamma \vdash t_1 \approx t_2 : \tau_1 * \tau_2$, where $\Gamma \vdash t_1, t_2 : \tau_1 * \tau_2$, and suppose the inductive hypotheses $P\,\tau_1$ and $P\,\tau_2$. It suffices to show $\Gamma \vdash \mathtt{fst}\,t_1 =^{\mathfrak{C}} \mathtt{fst}\,t_2 : \tau_1$ and $\Gamma \vdash \mathtt{snd}\,t_1 =^{\mathfrak{C}} \mathtt{snd}\,t_2 : \tau_2$, which, by the inductive hypotheses, reduces to showing $\Gamma \vdash \mathtt{fst}\,t_1 \approx \mathtt{fst}\,t_2 : \tau_1$ and $\Gamma \vdash \mathtt{snd}\,t_1 \approx \mathtt{snd}\,t_2 : \tau_2$, which follows from $\Gamma \vdash t_1 \approx t_2 : \tau_1 * \tau_2$.
- *Case $\tau = \tau_1 \to \tau_2$.* Suppose $\Gamma \vdash t_1 \approx t_2 : \tau_1 \to \tau_2$, where $\Gamma \vdash t_1, t_2 : \tau_1 \to \tau_2$, and suppose the inductive hypotheses $P\,\tau_1$ and $P\,\tau_2$. It suffices to show $\Gamma \vdash t_1\,t_3 =^{\mathfrak{C}} t_2\,t_3 : \tau_2$ for all $\Gamma \vdash t_3 : \tau_1$. By the inductive hypothesis, it suffices to show $\Gamma \vdash t_1\,t_3 \approx t_2\,t_3 : \tau_2$, which follows from $\Gamma \vdash t_1 \approx t_2 : \tau_1 \to \tau_2$.
- *Case $\tau = \tau_1 + \tau_2$.* Suppose $\Gamma \vdash t_1 \approx t_2 : \tau_1 + \tau_2$, where $\Gamma \vdash t_1, t_2 : \tau_1 + \tau_2$, and suppose the inductive hypotheses $P\,\tau_1$ and $P\,\tau_2$. We consider, by cases, whether there exists terms $t_3 : \tau_1$ and $t_4 : \tau_2$.
  - *Case t_3 exists and t_4 exists.* It suffices to show

    $\Gamma \vdash \mathtt{match}\ t_1\ \mathtt{with}\ \mathtt{inl}\,x\,.\,x \mid \mathtt{inr}\,x\,.\,t_3 =^{\mathfrak{C}} \mathtt{match}\ t_2\ \mathtt{with}\ \mathtt{inl}\,x\,.\,x \mid \mathtt{inr}\,x\,.\,t_3 : \tau_1$

    $\Gamma \vdash \mathtt{match}\ t_1\ \mathtt{with}\ \mathtt{inl}\,x\,.\,t_4 \mid \mathtt{inr}\,x\,.\,x =^{\mathfrak{C}} \mathtt{match}\ t_2\ \mathtt{with}\ \mathtt{inl}\,x\,.\,t_4 \mid \mathtt{inr}\,x\,.\,x : \tau_2,$

    which, by the inductive hypotheses, reduce to

    $\Gamma \vdash \mathtt{match}\ t_1\ \mathtt{with}\ \mathtt{inl}\,x\,.\,x \mid \mathtt{inr}\,x\,.\,t_3 \approx \mathtt{match}\ t_2\ \mathtt{with}\ \mathtt{inl}\,x\,.\,x \mid \mathtt{inr}\,x\,.\,t_3 : \tau_1$

    $\Gamma \vdash \mathtt{match}\ t_1\ \mathtt{with}\ \mathtt{inl}\,x\,.\,t_4 \mid \mathtt{inr}\,x\,.\,x \approx \mathtt{match}\ t_2\ \mathtt{with}\ \mathtt{inl}\,x\,.\,t_4 \mid \mathtt{inr}\,x\,.\,x : \tau_2,$

    which both follow from $\Gamma \vdash t_1 \approx t_2 : \tau_1 + \tau_2$.
  - *Case t_3 exists and t_4 does not exist.* If t_4 does not exist, then τ_2 is isomorphic to $\mathtt{0}$, and hence there is an alternate term $x : \tau_2 \vdash t_4' : \tau_1$ involving absurd. It suffices to show

    $\Gamma \vdash \mathtt{match}\ t_1\ \mathtt{with}\ \mathtt{inl}\,x\,.\,x \mid \mathtt{inr}\,x\,.\,t_3 =^{\mathfrak{C}} \mathtt{match}\ t_2\ \mathtt{with}\ \mathtt{inl}\,x\,.\,x \mid \mathtt{inr}\,x\,.\,t_3 : \tau_1$

    $\Gamma \vdash \mathtt{match}\ t_1\ \mathtt{with}\ \mathtt{inl}\,x\,.\,t_4' \mid \mathtt{inr}\,x\,.\,x =^{\mathfrak{C}} \mathtt{match}\ t_2\ \mathtt{with}\ \mathtt{inl}\,x\,.\,t_4' \mid \mathtt{inr}\,x\,.\,x : \tau_2,$

    which, by the inductive hypotheses, reduce to

    $\Gamma \vdash \mathtt{match}\ t_1\ \mathtt{with}\ \mathtt{inl}\,x\,.\,x \mid \mathtt{inr}\,x\,.\,t_3 \approx \mathtt{match}\ t_2\ \mathtt{with}\ \mathtt{inl}\,x\,.\,x \mid \mathtt{inr}\,x\,.\,t_3 : \tau_1$

    $\Gamma \vdash \mathtt{match}\ t_1\ \mathtt{with}\ \mathtt{inl}\,x\,.\,t_4' \mid \mathtt{inr}\,x\,.\,x \approx \mathtt{match}\ t_2\ \mathtt{with}\ \mathtt{inl}\,x\,.\,t_4' \mid \mathtt{inr}\,x\,.\,x : \tau_2,$

    which both follow from $\Gamma \vdash t_1 \approx t_2 : \tau_1 + \tau_2$.
  - *Case t_3 does not and t_4 exists.* Symmetric to the previous case.
  - *Case t_3 does not and t_4 does not exist.* In this case, τ_1 and τ_2 are isomorphic to $\mathtt{0}$, and hence $\tau_1 + \tau_2$ is isomorphic to $\mathtt{0}$. Thus $\Gamma \vdash t_1 =^{\mathfrak{C}} t_2 : \mathtt{0}$ trivially.
- *Case $\tau = \mathtt{Gen}\,\tau'$.* We proceed by contraposition. Suppose $\Gamma \vdash t_1 \neq^{\mathfrak{C}} t_2$. Then there exists some $\rho \in [\![\Gamma]\!]$ and $\sigma \in \mathbb{S}$ such that $[\![t_1]\!]^{\mathfrak{C}}\,\rho\,\sigma \neq [\![t_2]\!]^{\mathfrak{C}}\,\rho\,\sigma$. We proceed using $\mathtt{discrim}'$ as described above. Thus
  (1) for all ρ and σ, $[\![\mathtt{discrim}'[t_2]]\!]^{\mathfrak{C}}\,\rho_1\,\sigma_1 = (a, \mathrm{node}\,(a, xs), xs)$ for some a and xs, and
  (2) $[\![\mathtt{discrim}'[t_2]]\!]^{\mathfrak{C}}\,\rho_2\,\sigma_2 = (a, \mathrm{node}\,(b, ys), xs)$ for some ρ, σ, $a \neq b$ and $ys \neq xs$.
  Since our terms represent *terminating* programs, the set of choices of σ_2 must be infinite and of non-zero measure. Thus $(\!|\mathtt{discrim}'[t_1]|\!) = (\!|\mathtt{discrim}'[t_2]|\!)$, which completes the proof.

□

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

$$\Gamma \vdash t_1 =^{F} t_2 \iff \Gamma \vdash t_1 \approx t_2.$$

3.6 Summary

We have formalized Hedgehog's syntax as well as its sampling and distribution semantics with an eye towards proving program optimizations sound. The sampling semantics is compositional but unusably fine-grained: many common program optimizations are unsound under the sampling semantics (Theorem 3.8). On the other hand, the distribution semantics is coarse-grained but non-compositional. By Corollary 3.16, *every* (sound and complete) compositional semantics must identify the exact same set of terms as the sampling semantics. Thus, none of the transformations discussed in Theorem 3.8 are permitted under *any* sound interpretation of Hedgehog, which renders Hedgehog a poor target for optimization. We stress that Theorem 3.15 is a consequence of combining probabilistic and shrinking effects: if either `flip` or `shrink` are removed from Hedgehog, then 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 4.14).

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):

$$\tau ::= \ldots \mid \cancel{\mathsf{Gen}\,\tau} \mid \tau \rightsquigarrow \tau$$

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 $\mathtt{fun}\ x : \tau\ .\ t$).

Hedgehog$^{\rightarrow}$ replaces the generator type $\mathsf{Gen}\,\tau$ 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 $\tau_1 \rightarrow \mathsf{Gen}\,\tau_2$. At the term level, Hedgehog$^{\rightarrow}$ reuses the existing function application ($t_1\ t_2$) and anonymous function constructs ($\mathtt{fun}\ x : \tau\ .\ t$) 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

$$\frac{\Gamma; x : \tau_1 \vdash^\bullet t : \tau_2}{\Gamma \vdash^\bullet \mathtt{fun}\ x : \tau_1 .\ t : \tau_1 \rightsquigarrow \tau_2} \qquad \frac{}{\Gamma \vdash^\bullet \mathtt{flip}\, p : 1 \rightsquigarrow \mathtt{Bool}} \qquad \frac{}{\Gamma \vdash^\bullet \mathtt{shrink} : \tau \times \cdots \times \tau \rightsquigarrow \tau}$$

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

$$\frac{\Gamma, \Delta \vdash^\bullet x : \tau}{\Gamma; \Delta \vdash^\bullet \mathtt{return}\, x : \tau} \qquad \frac{x \notin \mathrm{Dom}\, \Delta \qquad \Gamma; \Delta \vdash^\bullet t_1 : \tau_1 \qquad \Gamma; \Delta, x : \tau_1 \vdash^\bullet t_2 : \tau_2}{\Gamma; \Delta \vdash^\bullet \mathtt{let}\ x \leftarrow t_1\ \mathtt{in}\ t_2 : \tau_2}$$

$$\frac{\Gamma \vdash^\bullet t_1 : \tau_1 \rightsquigarrow \tau_2 \qquad \Gamma; \Delta \vdash^\bullet t_2 : \tau_1}{\Gamma; \Delta \vdash^\bullet t_1\, t_2 : \tau_2} \qquad \frac{\begin{array}{c} x_1, x_2 \notin \mathrm{Dom}\, \Delta \qquad \Gamma, \Delta \vdash^\bullet t_0 : \tau_1 + \tau_2 \\ \Gamma; \Delta, x_1 : \tau_1 \vdash^\bullet t_1 : \tau_3 \qquad \Gamma; \Delta, x_2 : \tau_2 \vdash^\bullet t_2 : \tau_3 \end{array}}{\Gamma; \Delta \vdash^\bullet \mathtt{match}\ t_0\ \mathtt{with}\ \mathtt{inl}\, x_1 .\ t_1 \mid \mathtt{inr}\, x_2 .\ t_2 : \tau_3}.$$

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

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

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 $\mathtt{coin}_1$ and $\mathtt{coin}_2$ (Example 3.2) are still typed in Hedgehog$^{\rightarrow}$, but they are command terms instead of normal terms, i.e.,

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

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

```
coin₁• : 1 ⇝ Bool
coin₁• = fun u : 1. let x ← flip 1/2 in
                    let y ← flip 1/2 in
                      return x = y
```

```
coin₂• : 1 ⇝ Bool
coin₂• = fun u : 1. let x ← flip 1/2 in
                      return x.
```

We similarly wrap the function `discrim` as

```
discrim• : (1 ⇝ Bool) → (1 ⇝ Bool)
discrim• = fun f : 1 ⇝ Bool. fun u : 1. let b ← shrink (true, false) in
                                          if b then f () else coin₁• ().
```

4.2 Semantics

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

$$\mathrm{flip}_F : [0,1] \to_{\mathbf{Q}} F\,(\mathbf{1}, \mathbb{B}) \qquad\qquad \mathrm{shrink}_F^n \in F\,(\textstyle\prod_{i=0}^n A, A).$$

$$[\![\Gamma;\Delta \vdash^\bullet \mathtt{return}\, x : \tau]\!]^F\, \rho = \mathrm{arr}\,(\rho' \mapsto [\![\Gamma,\Delta \vdash^\bullet x : \tau]\!]^F\,(\rho,\rho'))$$

$$\begin{aligned}&[\![\Gamma;\Delta \vdash^\bullet \mathtt{let}\, x \leftarrow t_1\, \mathtt{in}\, t_2 : \tau_2]\!]^F\, \rho\\ &\quad = [\![\Gamma;\Delta,x:\tau_1 \vdash^\bullet t_2 : \tau_2]\!]^F\, \rho\\ &\qquad \circ \mathrm{arr}\,((\rho',\rho'_x) \mapsto (\rho'_y)_{y\in \mathrm{Dom}\,\Delta\cup\{x\}})\\ &\qquad \circ \mathrm{first}\,([\![\Gamma;\Delta \vdash^\bullet t_1 : \tau_1]\!]^F\, \rho)\\ &\qquad \circ \mathrm{arr}\,(\rho' \mapsto (\rho',\rho'))\\ &\quad \text{where } \Gamma;\Delta \vdash^\bullet t_1 : \tau_1, \quad \Gamma;\Delta,x:\tau_1 \vdash^\bullet t_2 : \tau_2\end{aligned}$$

$$[\![\Gamma;\Delta \vdash^\bullet t_1\, t_2 : \tau_2]\!]^F\, \rho = [\![\Gamma \vdash^\bullet t_1 : \tau_1 \rightsquigarrow \tau_2]\!]^F\, \rho \circ [\![\Gamma;\Delta \vdash^\bullet t_2 : \tau_1]\!]^F\, \rho$$
$$\text{where } \Gamma \vdash^\bullet t_1 : \tau_1 \rightsquigarrow \tau_2, \quad \Gamma;\Delta \vdash^\bullet t_2 : \tau_1$$

$$\begin{aligned}&[\![\Gamma;\Delta \vdash^\bullet \mathtt{match}\, t_0\, \mathtt{with}\, \mathtt{inl}\, x_1 .\, t_1 \mid \mathtt{inr}\, x_2 .\, t_2 : \tau_3]\!]^F\, \rho\\ &\quad = \mathrm{arr}\,((a,b) \mapsto (b,a))\\ &\qquad \circ \mathrm{left}\,([\![\Gamma;\Delta,x_2:\tau_2 \vdash^\bullet t_2 : \tau_3]\!]^F\, \rho)\\ &\qquad \circ \mathrm{arr}\,((a,b) \mapsto (b,a))\\ &\qquad \circ \mathrm{left}\,([\![\Gamma;\Delta,x_1:\tau_1 \vdash^\bullet t_1 : \tau_3]\!]^F\, \rho)\\ &\qquad \circ \mathrm{arr}\,(\rho' \mapsto \mathrm{distrib}\,(\rho',[\![\Gamma,\Delta \vdash^\bullet t_0 : \tau_1+\tau_2]\!]^F\,(\rho,\rho')))\\ &\quad \text{where } \Gamma,\Delta \vdash^\bullet t_0 : \tau_1+\tau_2, \quad \Gamma;\Delta,x_1:\tau_1 \vdash^\bullet t_1 : \tau_3, \quad \Gamma;\Delta,x_2:\tau_2 \vdash^\bullet t_2 : \tau_3\end{aligned}$$

Fig. 13. Hedgehog$^{\rightarrow}$ Command Semantics.

An interpretation induces a *type and environment semantics* $[\![-]\!]^F : \mathrm{Type}^\bullet \cup \mathrm{Env}^\bullet \to \mathbf{Qbs}$, defined analogously to Definition 3.4, but with

$$[\![\tau_1 \rightsquigarrow \tau_2]\!]^F = F\,([\![\tau_1]\!]^F, [\![\tau_2]\!]^F).$$

An interpretation also induces, for all $\Gamma \vdash^\bullet t : \tau$, a *term semantics* $[\![\Gamma \vdash^\bullet t : \tau]\!]^F : [\![\Gamma]\!]^F \to_{\mathbf{Q}} [\![\tau]\!]^F$ defined analogously to Definition 3.4 with

$$[\![\Gamma \vdash^\bullet \mathtt{fun}\, x : \tau_1 .\, t : \tau_1 \rightsquigarrow \tau_2]\!]^F = [\![\Gamma;x:\tau_1 \vdash^\bullet t : \tau_2]\!]^F \qquad [\![\Gamma \vdash^\bullet \mathtt{flip}\, p : 1 \rightsquigarrow \mathtt{Bool}]\!]^F\, \rho = \mathrm{flip}_F\, p$$

$$[\![\Gamma \vdash^\bullet \mathtt{shrink} : \underbrace{\tau * \cdots * \tau}_{n+1 \text{ times}} \rightsquigarrow \tau]\!]^F\, \rho = \mathrm{shrink}_F^n$$

where the function $[\![\Gamma;\Delta \vdash^\bullet t : \tau]\!]^F : [\![\Gamma]\!]^F \to_{\mathbf{Q}} F\,([\![\Delta]\!]^F, [\![\tau]\!]^F)$ is defined for all $\Gamma;\Delta \vdash^\bullet t : \tau$ in Figure 13.

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* $\mathfrak{C}^\bullet : \mathbf{Qbs} \times \mathbf{Qbs} \to \mathbf{Qbs}$ of Hedgehog$^{\rightarrow}$ is defined in Figure 14.

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 $f \circ_{\mathfrak{C}^\bullet} g$ splits its sample between arguments f and g.

$$\mathfrak{C}^\bullet\ (A, B) = (\mathbb{S} \times A \to_{\mathfrak{Q}} \mathrm{Tree}\ B) \qquad \mathrm{arr}_{\mathfrak{C}^\bullet}\ f\ (\sigma, a) = \mathrm{unit}\ (f\ a)$$

$$(f \circ_{\mathfrak{C}^\bullet} g)\ (\sigma, a) = \mathrm{bind}\ (b \mapsto f\ (\pi_r\ \sigma, b))\ (g\ (\pi_l\ \sigma, a)) \qquad \mathrm{first}_{\mathfrak{C}^\bullet}\ f\ (\sigma, (a, c)) = \mathrm{map}\ (-, c)\ (f\ (\sigma, a))$$

$$\mathrm{left}_{\mathfrak{C}^\bullet}\ f\ (\sigma, \iota_1\ a) = \mathrm{map}\ \iota_1\ (f\ (\sigma, a)) \qquad \mathrm{left}_{\mathfrak{C}^\bullet}\ f\ (\sigma, \iota_2\ b) = \mathrm{unit}\ (\iota_2\ b)$$

$$\mathrm{flip}_{\mathfrak{C}^\bullet}\ p\ (\sigma, ()) = \mathrm{unit}\ (\sigma < p) \qquad \mathrm{shrink}^n_{\mathfrak{C}^\bullet}\ (\sigma, (a_0, a_1, \ldots, a_n)) = \mathrm{node}\ (a_0, \mathrm{unit}\ a_1 :: \ldots :: \mathrm{unit}\ a_n :: \epsilon)$$

Fig. 14. Hedgehog$^\rightarrow$'s Sampling Interpretation.

Remark 4.6. Only redefining the semantics of `match` is not sufficient to rule out pathological contexts such as `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,

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

This can be expressed equivalently — without any case analysis — as

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

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

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

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

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

$$[\![\texttt{coin}_1^\bullet]\!]^{\mathfrak{C}^\bullet}\ (\sigma, u) = \mathrm{unit}\ (\sigma_1 < 1/2 \Leftrightarrow \sigma_2 < 1/2) \qquad [\![\texttt{coin}_2^\bullet]\!]^{\mathfrak{C}^\bullet}\ (\sigma, u) = \mathrm{unit}\ (\sigma_1 < 1/2)$$

where $\sigma_1, \sigma_2 \in \mathbb{S}$ obtained from σ by some sequence of invocations of π_l and π_r. The exact sequences are unimportant, but they are much longer than in Example 3.6 because the semantics of `let` (Figure 13) involves a large number of compositions ($\circ_{\mathfrak{C}^\bullet}$). Hughes [2004, §4.2] describes how these redundant compositions can be eliminated.

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

$$[\![\texttt{discrim}^\bullet\ \texttt{coin}_1^\bullet]\!]^{\mathfrak{C}^\bullet}\ \sigma = \mathrm{node}\ (\sigma_1 < 1/2 \Leftrightarrow \sigma_2 < 1/2, \mathrm{unit}\ (\sigma_3 < 1/2 \Leftrightarrow \sigma_4 < 1/2) :: \epsilon)$$

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. Unlike Hedgehog, Hedgehog$^\rightarrow$ ensures that each branch in $\texttt{discrim}^\bullet$ receives an independent sample, so $\texttt{discrim}^\bullet\ \texttt{coin}_1^\bullet$ does not return trees where all nodes are identically labelled. This example also demonstrates that the command term $\texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t'$ is generally *not* identical to t', i.e., Equation (11) does not hold in Hedgehog$^\rightarrow$.

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

Definition 4.8 (Hedgehog$^\rightarrow$ Semantic Equivalence). Two Hedgehog$^\rightarrow$ terms $\Gamma \vdash^\bullet t_1, t_2 : \tau$ are *semantically equivalent* with respect to an interpretation F (written $\Gamma \vdash^\bullet t_1 =^F t_2 : \tau$) if $[\![\Gamma \vdash^\bullet t_1 : \tau]\!]^F = [\![\Gamma \vdash^\bullet t_2 : \tau]\!]^F$. 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 can be inferred from the surrounding context.

$$\mathfrak{D}\,(A, B) = \mathrm{Dist}\,(A \to_{\mathbf{Q}} \mathrm{Tree}\,B) \qquad \mathrm{arr}_{\mathfrak{D}}\,\mu = \mathrm{unit}\,(\mathrm{unit} \circ \mu)$$

$$\mu_1 \circ_{\mathfrak{D}} \mu_2 = \mathrm{bind}\,(a_2 \mapsto \mathrm{map}\,(a_1 \mapsto \mathrm{bind}\,a_1 \circ a_2)\,\mu_1)\,\mu_2$$

$$\mathrm{first}_{\mathfrak{D}}\,\mu = \mathrm{map}\,(\mu' \mapsto (a, c) \mapsto \mathrm{map}\,(-, c)\,(\mu' a))\,\mu \qquad \mathrm{left}_{\mathfrak{D}}\,\mu = \mathrm{map}\,(\mu' \mapsto [\mathrm{map}\,\iota_1 \circ \mu', \iota_2])\,\mu$$

$$\mathrm{flip}_{\mathfrak{D}}\,p = \mathrm{map}\,((\sigma, ()) \mapsto \sigma < p)\,\lambda$$

$$\mathrm{shrink}^n_{\mathfrak{D}} = \mathrm{unit}\,((a_0, a_1, \ldots, a_n) \mapsto \mathrm{node}\,(a_0, \mathrm{unit}\,a_1 :: \ldots :: \mathrm{unit}\,a_n :: \epsilon))$$

Fig. 15. Hedgehog$^{\to}$'s Distribution Interpretation.

4.3 Distributions

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

$$(\!|-|\!)^\bullet : \mathrm{Term}^\bullet \to \mathrm{Dist}\,(\mathbf{1} \to_{\mathbf{Q}} \mathbb{B})$$
$$(\!|\,t\,|\!)^\bullet = \mathrm{map}\,(\mathrm{curry}\,[\![\,t\,]\!]^{\mathfrak{C}^\bullet})\,\lambda.$$

Definition 4.9 is the Hedgehog$^{\to}$ analogue of Definition 3.10.

Example 4.10. We compute the distributions of $\mathtt{coin}^\bullet_1, \mathtt{coin}^\bullet_2 : 1 \rightsquigarrow \mathtt{Bool}$ given in Example 4.3:

$$(\!|\mathtt{coin}^\bullet_1|\!)^\bullet = 1/2 \cdot \mathrm{unit}\,(() \mapsto \mathrm{unit}\,\mathrm{true}) + 1/2 \cdot \mathrm{unit}\,(() \mapsto \mathrm{unit}\,\mathrm{false})$$
$$(\!|\mathtt{coin}^\bullet_2|\!)^\bullet = 1/2 \cdot \mathrm{unit}\,(() \mapsto \mathrm{unit}\,\mathrm{true}) + 1/2 \cdot \mathrm{unit}\,(() \mapsto \mathrm{unit}\,\mathrm{false})$$

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

Definition 4.11 (Hedgehog$^{\to}$ Distribution Interpretation). The *distribution interpretation* $\mathfrak{D} : \mathbf{Qbs} \to \mathbf{Qbs}$ is defined in Figure 15.

Proposition 4.12. *The interpretation $\mathfrak{D}$ is an arrow.*

Proof. Follows from the fact that Dist is a monad and $(- \to_{\mathbf{Q}} \mathrm{Tree}\,-)$ is an arrow. □

Lemma 4.13. *The following equations hold:*

$$\mathrm{map}\,(\mathrm{curry}\,(\mathrm{arr}_{\mathfrak{C}^\bullet}\,f))\,\lambda = \mathrm{arr}_{\mathfrak{D}}\,f \tag{17}$$
$$\mathrm{map}\,(\mathrm{curry}\,(\mathrm{first}_{\mathfrak{C}^\bullet}\,f))\,\lambda = \mathrm{first}_{\mathfrak{D}}\,(\mathrm{map}\,(\mathrm{curry}\,f\,\lambda)) \tag{18}$$
$$\mathrm{map}\,(\mathrm{curry}\,(\mathrm{left}_{\mathfrak{C}^\bullet}\,f))\,\lambda = \mathrm{left}_{\mathfrak{D}}\,f \tag{19}$$
$$\mathrm{map}\,(\mathrm{curry}\,(f \circ_{\mathfrak{C}^\bullet} g))\,\lambda = \mathrm{map}\,(\mathrm{curry}\,f)\,\lambda \circ_{\mathfrak{D}} \mathrm{map}\,(\mathrm{curry}\,g)\,\lambda \tag{20}$$
$$\mathrm{map}\,(\mathrm{curry}\,(\mathrm{flip}_{\mathfrak{C}^\bullet}\,p))\,\lambda = \mathrm{flip}_{\mathfrak{D}}\,p \tag{21}$$
$$\mathrm{map}\,(\mathrm{curry}\,\mathrm{shrink}^n_{\mathfrak{C}^\bullet})\,\lambda = \mathrm{shrink}^n_{\mathfrak{D}} \tag{22}$$

Proof.

(17)

$$\begin{aligned}\mathrm{map}\,(\mathrm{curry}\,(\mathrm{arr}_{\mathfrak{C}^\bullet}\,f))\,\lambda &= \mathrm{map}\,(\sigma \mapsto a \mapsto \mathrm{arr}_{\mathfrak{C}^\bullet}\,f\,(\sigma, a))\,\lambda \\ &= \mathrm{map}\,(\sigma \mapsto a \mapsto \mathrm{unit}\,(f\,a))\,\lambda \\ &= \mathrm{unit}\,(a \mapsto \mathrm{unit}\,(f\,a)) \\ &= \mathrm{arr}_{\mathfrak{D}}\,f\end{aligned}$$

(18)

$$\begin{aligned}\mathrm{map}\,(\mathrm{curry}\,(\mathrm{first}_{\mathfrak{C}^\bullet}\,f))\,\lambda &= \mathrm{map}\,(\sigma \mapsto (a,c) \mapsto \mathrm{first}_{\mathfrak{C}^\bullet}\,f\,(\sigma,(a,c)))\,\lambda\\ &= \mathrm{map}\,(\sigma \mapsto (a,c) \mapsto \mathrm{map}\,(-,c)\,(f\,(\sigma,a)))\,\lambda\\ &= \mathrm{map}\,(\sigma \mapsto (a,c) \mapsto \mathrm{map}\,(-,c)\,(\mathrm{curry}\,f\,\sigma\,a))\,\lambda\\ &= \mathrm{map}\,(f' \mapsto (a,c) \mapsto \mathrm{map}\,(-,c)\,(f'a))\,(\mathrm{map}\,(\mathrm{curry}\,f)\,\lambda)\\ &= \mathrm{first}_{\mathfrak{D}}\,(\mathrm{map}\,(\mathrm{curry}\,f)\,\lambda)\end{aligned}$$

(19)

$$\begin{aligned}\mathrm{map}\,(\mathrm{curry}\,(\mathrm{left}_{\mathfrak{C}^\bullet}\,f))\,\lambda &= \mathrm{map}\,(\sigma \mapsto [\mathrm{map}\,\iota_1 \circ \mathrm{curry}\,f\,\sigma, \mathrm{unit} \circ \iota_2])\,\lambda\\ &= \mathrm{map}\,(f' \mapsto [\mathrm{map}\,\iota_1 \circ f', \iota_2])\,(\mathrm{map}\,(\mathrm{curry}\,f)\,\lambda)\\ &= \mathrm{left}_{\mathfrak{D}}\,(\mathrm{map}\,(\mathrm{curry}\,f)\,\lambda)\end{aligned}$$

(20)

$$\begin{aligned}&\mathrm{map}\,(\mathrm{curry}\,(f \circ_{\mathfrak{C}^\bullet} g))\,\lambda\\ &= \mathrm{map}\,(\sigma \mapsto a \mapsto \mathrm{bind}\,(b \mapsto f\,(\pi_r\,\sigma, b))\,(g\,(\pi_l\,\sigma, a)))\,\lambda\\ &= \mathrm{bind}\,(\sigma \mapsto \mathrm{unit}\,(a \mapsto \mathrm{bind}\,(b \mapsto f\,(\pi_r\,\sigma, b))\,(g\,(\pi_l\,\sigma, a))))\,\lambda\\ &= \mathrm{bind}\,(\sigma_1 \mapsto \mathrm{bind}\,(\sigma_2 \mapsto \mathrm{unit}\,(a \mapsto \mathrm{bind}\,(b \mapsto f\,(\sigma_2, b))\,(g\,(\sigma_1, a))))\,\lambda)\,\lambda\\ &= \mathrm{bind}\,(\sigma_1 \mapsto \mathrm{map}\,(\sigma_2 \mapsto a \mapsto \mathrm{bind}\,(b \mapsto f\,(\sigma_2, b))\,(g\,(\sigma_1, a)))\,\lambda)\,\lambda\\ &= \mathrm{bind}\,(\sigma_1 \mapsto \mathrm{map}\,(\sigma_2 \mapsto \mathrm{bind}\,(\mathrm{curry}\,f\,\sigma_2) \circ \mathrm{curry}\,g\,\sigma_1)\,\lambda)\,\lambda\\ &= \mathrm{bind}\,(a_2 \mapsto \mathrm{map}\,(\sigma \mapsto \mathrm{bind}\,(\mathrm{curry}\,f\,\sigma) \circ a_2)\,\lambda)\,(\mathrm{map}\,(\mathrm{curry}\,g)\,\lambda)\\ &= \mathrm{bind}\,(a_2 \mapsto \mathrm{map}\,(a_1 \mapsto \mathrm{bind}\,a_1 \circ a_2)\,(\mathrm{map}\,(\mathrm{curry}\,f)\,\lambda))\,(\mathrm{map}\,(\mathrm{curry}\,g)\,\lambda)\\ &= \mathrm{map}\,(\mathrm{curry}\,f)\,\lambda \circ_{\mathfrak{D}} \mathrm{map}\,(\mathrm{curry}\,g)\,\lambda\end{aligned}$$

□

Theorem 4.14. *Suppose* $() \vdash^\bullet t : 1 \rightsquigarrow \mathtt{Bool}$. *Then* $(\!|t|\!)^\bullet = [\![t]\!]^{\mathfrak{D}}$.

Proof. Follows from Lemma 4.13. □

Theorem 4.14 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 4.14, t is a command term and therefore represents an effectful computation.

Theorem 4.15. *Equations* (1) *to* (16) *(Theorems 3.8 and 3.9) all hold for Hedgehog$^\rightarrow$ terms under* $\mathfrak{D}$, except *for equation* (11).

Theorem 4.15 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 4.16. *The following equivalence holds:*

$$\mathtt{let}\ x \leftarrow \mathtt{flip}\,p\ \mathtt{in}\ \mathtt{if}\ x\ \mathtt{then}\ t\ \mathtt{else}\ t =^{\mathfrak{D}} t \tag{11′}$$

Intuitively, Proposition 4.16 demonstrates that equation (11) holds whenever the condition term does not shrink.

Types

$$\langle 1 \rangle_{\text{ty}} = 1 \qquad \langle \mathtt{0} \rangle_{\text{ty}} = \mathtt{0} \qquad \langle \tau_1 + \tau_2 \rangle_{\text{ty}} = \langle \tau_1 \rangle_{\text{ty}} + \langle \tau_2 \rangle_{\text{ty}} \qquad \langle \tau_1 * \tau_2 \rangle_{\text{ty}} = \langle \tau_1 \rangle_{\text{ty}} * \langle \tau_2 \rangle_{\text{ty}}$$

$$\langle \tau_1 \to \tau_2 \rangle_{\text{ty}} = \langle \tau_1 \rangle_{\text{ty}} \to \langle \tau_2 \rangle_{\text{ty}} \qquad \langle \tau_1 \rightsquigarrow \tau_2 \rangle_{\text{ty}} = \langle \tau_1 \rangle_{\text{ty}} \to \mathtt{Gen}\, \langle \tau_2 \rangle_{\text{ty}}$$

Environments

$$\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle_{\text{env}} = x_1 : \langle \tau_1 \rangle_{\text{ty}}, \ldots, x_n : \langle \tau_n \rangle_{\text{ty}}$$

Terms

$$\langle x \rangle_{\text{tm}} = x \qquad \langle \mathtt{absurd}\, t \rangle_{\text{tm}} = \mathtt{absurd}\, \langle t \rangle_{\text{tm}} \qquad \langle \mathtt{()} \rangle_{\text{tm}} = \mathtt{()} \qquad \langle \mathtt{inl}\, t \rangle_{\text{tm}} = \mathtt{inl}\, \langle t \rangle_{\text{tm}}$$

$$\langle \mathtt{inr}\, t \rangle_{\text{tm}} = \mathtt{inr}\, \langle t \rangle_{\text{tm}}$$

$$\begin{aligned} &\langle \mathtt{match}\ t_1\ \mathtt{with}\ \mathtt{inl}\, x_1 \mathtt{.}\ t_2 \mid \mathtt{inr}\, x_2 \mathtt{.}\ t_3 \rangle_{\text{tm}} \\ &\quad = \mathtt{match}\ \langle t_1 \rangle_{\text{tm}}\ \mathtt{with} \\ &\qquad \mathtt{inl}\, x_1 \mathtt{.}\ \mathtt{let}\ y \leftarrow \langle t_2 \rangle_{\text{tm}}\ \mathtt{in}\ \mathtt{return}\, y \\ &\qquad \mid \mathtt{inr}\, x_2 \mathtt{.}\ \mathtt{let}\ y \leftarrow \mathtt{return}\ \mathtt{()}\ \mathtt{in}\ \langle t_3 \rangle_{\text{tm}} \end{aligned}$$

$$\langle (t_1, t_2) \rangle_{\text{tm}} = (\langle t_1 \rangle_{\text{tm}}, \langle t_2 \rangle_{\text{tm}}) \qquad \langle \mathtt{fst}\, t \rangle_{\text{tm}} = \mathtt{fst}\, \langle t \rangle_{\text{tm}} \qquad \langle \mathtt{snd}\, t \rangle_{\text{tm}} = \mathtt{snd}\, \langle t \rangle_{\text{tm}}$$

$$\langle \mathtt{fun}\ x : \tau \mathtt{.}\ t \rangle_{\text{tm}} = \mathtt{fun}\ x : \langle \tau \rangle_{\text{ty}} \mathtt{.}\ \langle t \rangle_{\text{tm}} \qquad \langle t_1\, t_2 \rangle_{\text{tm}} = \langle t_1 \rangle_{\text{tm}}\, \langle t_2 \rangle_{\text{tm}} \qquad \langle \mathtt{return}\, t \rangle_{\text{tm}} = \mathtt{return}\, \langle t \rangle_{\text{tm}}$$

$$\langle \mathtt{let}\ x \leftarrow t_1\ \mathtt{in}\ t_2 \rangle_{\text{tm}} = \mathtt{let}\ x \leftarrow \langle t_1 \rangle_{\text{tm}}\ \mathtt{in}\ \langle t_2 \rangle_{\text{tm}} \qquad \langle \mathtt{flip}\, p \rangle_{\text{tm}} = \mathtt{flip}\, p \qquad \langle \mathtt{shrink} \rangle_{\text{tm}} = \mathtt{shrink}$$

Fig. 16. Translation from Hedgehog$^{\rightarrow}$ to Hedgehog.

4.4 From Hedgehog$^{\rightarrow}$ to Hedgehog

Definition 4.17. The *translation* from Hedgehog$^{\rightarrow}$ types, environments, and terms to Hedgehog types, environments, and terms is given by the functions $\langle - \rangle_{\text{ty}} : \text{Type}^{\bullet} \to \text{Type}$, $\langle - \rangle_{\text{env}} : \text{Env}^{\bullet} \to \text{Env}$, and $\langle - \rangle_{\text{tm}} : \text{Term}^{\bullet} \to \text{Term}$ defined in Figure 16.

Definition 4.17 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 $\tau_1 \to \mathtt{Gen}\, \tau_2$, and (2) adding additional `return` computations to `match` terms.

Proposition 4.18. *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 4.18 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 4.19. *Suppose* $(); () \vdash^{\bullet} t_1 =^{\mathfrak{D}} t_2 : \mathtt{Bool}$. *Then* $(\!|\langle t_1 \rangle_{\text{tm}}|\!) = (\!|\langle t_2 \rangle_{\text{tm}}|\!)$.

4.5 Summary

We have introduced Hedgehog$^{\rightarrow}$, a restricted variant of Hedgehog with a compositional distribution semantics (Theorem 4.14). 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) 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

```
coin1 :: Gen () Bool
coin1 = proc () → do
  x ← bool_ ⤙ ()
  y ← bool_ ⤙ ()
  returnA ⤙ x == y

coin2 :: Gen () Bool
coin2 = proc () → do
  x ← bool_ ⤙ ()
  returnA ⤙ x

discrim :: Gen () Bool
        → Gen () Bool
discrim m = proc () → do
  x ← shrinkTo ⤙ [True, False]
  if x
    then m ⤙ ()
    else coin1 ⤙ ()
```

Fig. 17. Example 4.3 translated to Haskell.

```
1 prune :: Gen a b → Gen a b
2 prune g = proc x → do
3   y ← freeze g ⤙ x
4   returnA ⤙ treeLabel y
```

Fig. 18. The `prune` Function.

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 4.15). 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 17. 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 "⤙".

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

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

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 18. Intuitively, `prune` behaves exactly like `freeze` but discards everything except the root value in the resulting tree (line 4).

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.

We perform our experiments on Hedgehog's `Hedgehog.Gen` module. The `Hedgehog.Gen` module contains a large collection of common generators and generator combinators. We provide analogues for each of these generators and combinators in `hedgehog-arrow`. We also perform experiments on several modules from Hedgehog's example repository[1]. We rewrote each of these modules to use `hedgehog-arrow`. Our full module set is given in Table 1.

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

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 19. 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 `n`th 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.

[1] Commit 4e9045fafa9b50efdc68326cd9f643c0d8383a18

```haskell
choice :: MonadGen m ⇒ [m a] → m a
choice [] = error "Hedgehog.Gen.choice: used with empty list"
choice xs = do
  n ← integral (Range.constant 0 (length xs - 1))
  xs !! n
```

Fig. 19. The implementation of `choice` in Hedgehog.

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

6 Related Work

Property-Based Testing and Shrinking. Property-based testing was popularized by the QuickCheck [Claessen and Hughes 2000] framework for Haskell, which features a probabilistic eDSL for specifying generators. In QuickCheck, a shrinker for a type a is specified separately as a function `shrinks :: a -> [a]` from values to their immediate shrinks. In contrast, Hedgehog→ employs *generator-based shrinking*, where shrinkers are derived from annotated generator specifications.

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

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

Reflective generators [Goldstein et al. 2023] extend normal generators with *partial monadic profunctor* operations. Goldstein et al. [2023] show how this extension lends itself to a number of features, including the ability to run a generator "backwards" to obtain a random seed from a user-provided test case. The reflective generator framework does not mandate a particular interpretation, 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.

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

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

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

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

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, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings* (Uppsala, Sweden). Springer-Verlag, Berlin, Heidelberg, 368–392. doi:10.1007/978-3-662-54434-1_14

Ugo Dal Lago, Giulio Guerrieri, and Willem Heijltjes. 2020. Decomposing Probabilistic Lambda-Calculi. In *Foundations of Software Science and Computation Structures: 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings* (Dublin, Ireland). Springer-Verlag, Berlin, Heidelberg, 136–156. doi:10.1007/978-3-030-45231-5_8

Edsko de Vries. 2023. falsify: Internal Shrinking Reimagined for Haskell. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium, Haskell 2023, Seattle, WA, USA, September 8-9, 2023*, Trevor L. McDonell and Niki Vazou (Eds.). ACM, New York, NY, 97–109. doi:10.1145/3609026.3609733

Emil Eriksson. 2024. RapidCheck. github.com/emil-e/rapidcheck.

Claudia Faggian and Simona Ronchi della Rocca. 2019. Lambda Calculus and Probabilistic Computation. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS).* 1–13. doi:10.1109/LICS.2019.8785699

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. 2012. The HERMIT in the machine: a plugin for the interactive transformation of GHC core language programs. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12).* Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2364506.2364508

Jeremy Gibbons and Ralf Hinze. 2011. Just do it: simple monadic equational reasoning. *SIGPLAN Not.* 46, 9 (Sept. 2011), 2–14. doi:10.1145/2034574.2034777

Harrison Goldstein, Samantha Frohlich, Meng Wang, and Benjamin C Pierce. 2023. Reflecting on Random Generation. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 322–355.

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science* (Reykjavík, Iceland) *(LICS '17).* IEEE Press, Article 77, 12 pages.

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

John Hughes. 2004. Programming with arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming* (Tartu, Estonia) *(AFP'04).* Springer-Verlag, Berlin, Heidelberg, 73–129. doi:10.1007/11546382_2

Sam Lindley, Philip Wadler, and Jeremy Yallop. 2010. The arrow calculus. *J. Funct. Program.* 20, 1 (2010), 51–69. doi:10.1017/S095679680999027X

David Maciver and Zac Hatfield-Dodds. 2019. Hypothesis: A new approach to property-based testing. *J. Open Source Softw.* 4, 43 (2019), 1891. doi:10.21105/JOSS.01891

Conor Mcbride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (Jan. 2008), 1–13. doi:10.1017/S0956796807006326

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

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. doi:10.1016/0890-5401(91)90052-4

Ross Paterson. 2001. A new notation for arrows. *SIGPLAN Not.* 36, 10 (Oct. 2001), 229–240. doi:10.1145/507669.507664

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

Jaro Reinders. 2024. Higher Order Patterns for Rewrite Rules. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium* (Milan, Italy) *(Haskell 2024)*. Association for Computing Machinery, New York, NY, USA, 14–26. doi:10.1145/3677999.3678275

Jacob Stanley. 2024. Hedgehog. hackage.haskell.org/package/hedgehog.

Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings* (Uppsala, Sweden). Springer-Verlag, Berlin, Heidelberg, 855–879. doi:10.1007/978-3-662-54434-1_32

Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming. *Proc. ACM Program. Lang.* 3, POPL, Article 36 (Jan. 2019), 29 pages. doi:10.1145/3290349

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