Abstract

Property-based random testing in the style of QuickCheck demands efficient generators for well-distributed random data satisfying complex logical predicates, but writing these generators can be difficult and error prone. We propose a better alternative: a domain-specific language in which generators are expressed by decorating predicates with lightweight annotations to control both the distribution of generated values as well as the amount of constraint solving that happens before each variable is instantiated. This language, called Luck, makes generators easier to write, read, and maintain.

We give Luck a formal semantics and prove several fundamental properties, including the soundness and completeness of random generation with respect to a standard predicate semantics. We evaluate Luck on common examples from the property-based testing literature and on two significant case studies; we show that it can be used in complex domains with comparable bug-finding effectiveness and a significant reduction in testing code size, compared to handwritten generators.

1. Introduction

Since being popularized by QuickCheck [19], property-based random testing has become a standard technique for improving software quality in a wide variety of programming languages [2, 38, 45, 54] and for streamlining interaction with proof assistants [6, 15, 22, 53, 57].

To use a property-based random testing tool, one writes properties in the form of executable predicates. For example, a natural property of a school database system: every student in a list registeredStudents should be taking at least one course,

\[
\text{member } x \, \lbrack \rbrack = \text{False} \\
\text{member } x \, (h:t) = (x == h) \lor \text{member } x \, t
\]

If there are many possible student ids (e.g., because they are represented as machine integers), then a randomly generated id is very unlikely to belong to registeredStudents, so almost all test cases will be discarded.

To enable effective testing in such cases, the QuickCheck user can provide a property-based generator for inputs satisfying p—here, a generator that always returns student ids drawn from the members of registeredStudents. Indeed, QuickCheck provides a library of combinators for defining such generators. These combinators also allow fine control over the distribution of generated values—a crucial feature in practice [19, 32, 56].

Property-based generators generators work well for small to medium-sized examples, but writing them can become challenging as p becomes more complex—sometimes becoming a research contribution in its own right! For example, papers have been written about generation techniques for well-typed lambda-terms [23, 56, 60, 64] and for indistinguishable machine states for finding bugs in information-flow monitors [36, 37]. Moreover, if we use QuickCheck to test an invariant property (e.g., type preservation), then the same predicate will appear in both the precondition and the conclusion of the property, requiring that we write both a predicate p and a generator whose outputs all satisfy p. These two artifacts must then be kept in sync, which can become both a maintenance issue and a rich source of testing bugs. These difficulties are not hypothetical: Hrițcu et al.’s machine-state generator [36] is over 1500 lines of tricky Haskell, while Pałka et al.’s generator for well-typed lambda-terms [56] is over 1600 even trickier ones. To enable effective property-based random testing of complex software artifacts, we need a better way of writing predicates and corresponding generators.

A natural idea is to derive an efficient generator for a given predicate directly from p itself. Indeed, two variants of this idea, with complementary strengths and weaknesses, have been explored in the literature—one based on local choices and backtracking, the other on general constraint solving. Our language, Luck, synergistically combines these two approaches.

The first approach can be thought of as a kind of incremental generate-and-test: rather than generating completely random valuations and then testing them against p, we instead walk over the structure of p and instantiate each unknown variable x at the first point where we meet a constraint involving x. In the member example above, on each recursive call, we make a random choice between the branches of the ||. If we choose the left, we instantiate x to the head of the list; otherwise we leave x unknown and continue with the recursive call to member on the tail. This has the effect of traversing the list of registered students and picking one of its elements. This process resembles narrowing from functional logic programming [1, 34, 45, 53]. It is attractively lightweight, admits natural control over distributions (as we will see in the next section), and has been used successfully [16, 24, 26, 59], even in challenging domains such as generating well-typed programs to test compilers [17, 24].
However, choosing a value for each unknown when we encounter the first constraint on it risks making choices that do not satisfy later constraints, forcing us to backtrack and make a different choice when the problem is discovered. For example, consider the `notMember` predicate:

\[
\text{notMember } x \sqsubseteq \text{True} \\
\text{notMember } x (h : t) = (x /= h) \land \text{notMember } x \; t
\]

Suppose we wish to generate values for \(x\) such that `notMember x ys` for some predetermined list `ys`. When we first encounter the constraint \(x /= h\), we generate a value for \(x\) that is not equal to the known value \(h\). We then proceed to the recursive call of `notMember`, where we check that the chosen \(x\) does not appear in the rest of the list. Since the values in the rest of the list are not taken into account when choosing \(x\), this may force us to backtrack if our choice of \(x\) was unlucky. If the space of possible values for \(x\) is not much bigger than \(ys\)—say, just \(2x\)—we will backtrack 50% of the time. Worse yet, if `notMember` is used to define another predicate—e.g., `distinct`, which tests whether each element of an input list is different from all the others—and we similarly generate lists satisfying `distinct`, then `notMember`’s 50% chance of backtracking will be compounded on each recursive call of `distinct`, leading to unacceptably low rates of successful generation.

The second existing approach uses a constraint solver to generate a diverse set of valuations satisfying a predicate. This approach has been widely investigated, both for generating inputs directly from predicates [12,31,43,62] and for symbolic-execution-based testing [10,27,63,66], which additionally uses the system under test to guide generation of inputs that exercise different control-flow paths. For `notMember`, gathering a set of disequality constraints on \(x\) before choosing its value avoids any backtracking.

However, pure constraint-solving approaches do not give us everything we need. They do not provide effective control over the distribution of generated valuations. At best, they might guarantee a uniform (or near uniform) distribution [14], but this is typically not the distribution we want in practice (see [52]). Moreover, the overhead of maintaining and solving constraints can make these approaches significantly less efficient than the more lightweight, local approach of needed narrowing when the latter does not lead to backtracking, as for instance in `member`.

The complementary strengths and weaknesses of local instantiation and global constraint solving suggest a hybrid approach, where limited constraint propagation, under explicit user control, is used to refine the domains (sets of possible values) of unknowns before instantiation. Exploring this hybrid approach is the goal of this paper. Our main contributions are:

- We propose a new domain-specific language, Luck, for writing generators via lightweight annotations on predicates, combining the strengths of the local-instantiation and constraint-solving approaches to generation. Section [22] illustrates Luck’s novel features using binary search trees as an example.

- To place Luck’s design on a firm formal foundation, we define a core calculus and establish key properties, including the soundness and completeness of its probabilistic generator semantics with respect to a straightforward interpretation of expressions as predicates [53].

- We provide a prototype interpreter [54] including a simple implementation of the constraint-solving primitives used by the generator semantics. We do not use an off-the-shelf constraint solver because we want to experiment with a per-variable uniform sampling approach [52], which is not supported by modern solvers. In addition, using such a solver would require translating Luck expressions—datatypes, pattern matching, etc.—into a form that it can handle. We leave this for future work.

- We evaluate Luck’s expressiveness on a collection of common examples from the random testing literature [53]. Two significant case studies show that Luck can be used (1) to find bugs in a widely used compiler (GHC) by randomly generating well-typed lambda terms and (2) to help design information-flow abstract machines by generating “low-indistinguishable” machine states. Compared to hand-written generators, these experiments demonstrate comparable bug-finding effectiveness (measured in test cases generated per counterexample found) and a significant reduction in the size of testing code. These interpreted Luck generators run an order of magnitude slower than compiled QuickCheck versions (8 to 36 times per test), but many opportunities for optimization remain.

[50] and [57] discuss related work and future directions. The auxiliary materials with this submission contain: (1) a Coq formalization of the narrowing semantics of Luck and machine-checked proofs of its properties [53], and (2) an extended version of the paper with full definitions and paper proofs for the whole semantics; (3) (non-anonymous) a prototype Luck interpreter and a battery of example programs, including all the ones we used for evaluation [55].

2. Luck by Example

Fig. 1 shows a recursive Haskell predicate `bst` that checks whether a given tree with labels strictly between `low` and `high` satisfies the standard binary-search tree (BST) invariant [52]. It is followed by a QuickCheck generator `genTree`, which generates BSTs with a given maximum depth, controlled by the `size` parameter. This generator first checks whether `low + 1 >= high`, in which case it returns the only valid BST satisfying this constraint—the Empty one. Otherwise, it uses QuickCheck’s `frequency` combinator, which takes a list of pairs of positive integer weights and associated generators and randomly selects one of the generators using the probabilities specified by the weights. In this example, \(\frac{\text{size}}{\text{size} + 1}\) of the time it creates an Empty tree, while \(\frac{\text{size} + 1}{\text{size} + 1}\) of the time it returns a Node. The `bst` generator is specified using monadic syntax: first it generates an integer \(x\) that is strictly between `low` and `high`, and then the left and right subtrees \(l\) and \(r\) by calling `genTree` recursively; finally it returns `Node x l r`.

The generator for BSTs allows us to efficiently test conditional properties of the form “if \(\text{bst}\; t\) then \(\langle\text{some other property of } t\rangle\),” but it raises some new issues of its own. First, even for this simple example, getting the generator right is a bit tricky (for instance because of potential off-by-one errors in generating \(x\)), and, even then, it is not immediately obvious that the set of trees generated by the generator is exactly the set accepted by the predicate. Worse, we now need to maintain two similar but distinct artifacts and keep them in sync. As predicates and generators become more complex, these issues can become quite problematic (e.g., [56]).

Enter Luck. The bottom of Fig. 1 shows a Luck program that represents both a BST predicate and a generator for random BSTs. Modulo variations in concrete syntax, the Luck code follows the Haskell `bst` predicate quite closely. The significant differences are: (1) the sample-after expression `!x`, which controls when node labels are generated, and (2) the `size` parameter, which is used, as in the QuickCheck generator, to annotate the branches of the case with relative weights. Together, these enable us to give the program both a natural interpretation as a predicate (by simply ignor-

---

1 Constraint solvers can, of course, be used to directly search for counterexamples to a property of interest by software model checking [12,41,4,9] etc.]. We are interested here in the rather different task of quickly generating a large number of diverse inputs, so that we can thoroughly test systems like compilers whose state spaces are too large to be exhaustively explored.
Binary tree datatype (in both Haskell and Luck):

data Tree a = Empty | Node a (Tree a) (Tree a)

Test predicate for BSTs (in Haskell):

bst :: Int -> Int -> Tree Int -> Bool
bst low high size
case tree of
    Empty -> True
    Node x l r ->
    low < x && x < high
    && bst low x l && bst x high r

QuickCheck generator for BSTs (in Haskell):

genTree :: Int -> Int -> Int -> Gen (Tree Int)
genTree size low high
    | low + 1 > high = return Empty
    | otherwise =
        frequency [(1, return Empty),
            (size, do
                x <- choose (low + 1, high - 1)
                l <- genTree (size `div` 2) low x
                r <- genTree (size `div` 2) x high
                return (Node x l r))]

Luck generator (and predicate) for BSTs:

sig bst :: Int -> Int -> Int -> Gen (Tree Int)
sig bst size low high
    | size == 0 then tree == Empty
    | otherwise case tree of
        | 1 % Empty -> True
        | size % Node x l r ->
        | (low < x && x < high) !x
        | && bst (size / 2) low x
        | && bst (size / 2) x high

Figure 1. Binary Search Tree tester and two generators

Alternatively, we can evaluate the top-level query

bst 10 0 42 t = True

by instantiating unknowns uniformly from the domain associated with each unknown. While this behavior is sometimes useful, effective property-based random testing often requires fine user control over the distribution of generated test cases. QuickCheck provides a library of basic combinators with which to build up complex generators; one of the most commonly used is frequency, which we saw in genTree (Fig. 1). Similarly, Luck allows weight annotations on the branches of a case expression, called narrow expressions, which have a frequency-like effect. In the Luck version of bst, for example, the unknown tree is either instantiated to an Empty tree at the time or partially instantiated to a Node (with fresh unknowns for x and the left and right subtrees) at the time. Narrow expressions give the user control over the probabilities of local choices. These do not necessarily correspond to a specific posterior probability, but the QuickCheck community has established techniques for guiding the user in tuning local weights to obtain good testing. For example, the user can wrap properties inside a collect combinator; during testing, QuickCheck will gather information on x, grouping equal values to provide an estimate of the posterior distribution that is being sampled. The collect combinator is an extremely effective tool for helping adjust frequency weights and dramatically increasing bug-finding rates (see, e.g., [5]). The Luck implementation provides a similar primitive.

Note that, while locally instantiating unknowns uniformly from their domain is a useful default, generating globally uniform distributions of test cases is usually not what we want, as they often lead to inefficient testing in practice. A simple example comes from the information flow control experiments of Hritcu et al. [56]. There are two “security levels”, called labels, Low and High, and pairs of
integers and labels are considered “indistinguishable” to a Low observer if the labels are equal and, if the labels are Low, so are the integers. In Haskell:

\[
\begin{align*}
\text{indist } (v_1, \text{High}) (v_2, \text{High}) &= \text{True} \\
\text{indist } (v_1, \text{Low}) (v_2, \text{Low}) &= v_1 == v_2 \\
\text{indist } _- _- &= \text{False}
\end{align*}
\]

A uniform distribution of such indistinguishable pairs is strongly skewed towards pairs with High labels: if we use 32-bit integers, for every Low indistinguishable pair there are $2^{32}$ High ones!

However, if a Luck user really wants to draw from a uniform distribution over a set of structured values satisfying some complex predicate, this effect can be achieved by emulating Boltzmann samplers \[20\]. This technique fits naturally in Luck, providing an efficient way of drawing samples from combinatorial structures of an approximate size \(n\)—in time linear in \(n\)—where any two objects with the same size have an equal probability of being generated. Details can be found in the extended version.

### 3. Semantics of Core Luck

We next present a core calculus for Luck—a minimal subset into which the examples in the previous section can in principle be desugared (though our implementation does not do this). The core language omits primitive booleans and integers and replaces general datatypes with binary sums, products, and iso-recursive types.

We begin in \[3.1\] with the syntax and standard predicate semantics of the core. (We call it the “predicate” semantics because, in our examples, the result of evaluating a top-level expression will typically be a boolean, though this expectation is not baked into the formalism.) We then build up to the full generator semantics \[3.2\]. Then we define a probabilistic narrowing semantics \[3.3\], abstracting over the primitives required to implement our semantics. Then we define a probabilistic narrowing semantics, which enhances the local-instantiation approach to random generation with QuickCheck-style distribution control \[3.4\]. Finally, we introduce a matching semantics, building on the narrowing semantics, that unifies constraint solving and narrowing into a single evaluator \[3.5\]. In the long version, we also show how integers and booleans are encoded and how the semantics is applied to the binary search tree example. The key properties of the generator semantics (both narrowing and matching versions) are soundness and completeness with respect to the predicate semantics \[3.5\]. Informally, whenever we use a Luck program to generate a valuation that satisfies some predicate, the valuation will satisfy the boolean predicate semantics (soundness), and it will generate every possible satisfying valuation with non-zero probability (completeness).

#### 3.1 Syntax, Typing, and Predicate Semantics

The syntax of Core Luck is given in Fig. 2. Except for the last line in the definitions of values and expressions, it is a standard simply typed call-by-value lambda calculus with sums, products, and iso-recursive types. We include recursive lambdas for convenience, although in principle they could be encoded using recursive types.

Values in Luck include unit, pairs of values, sum constructors \((L \text{ and } R)\) applied to values (and annotated with types, to eliminate ambiguity), first class recursive functions \((\text{rec})\), fold-annotated values (indicating where an iso-recursive type should be “folded”), and unknowns drawn from an infinite set. The standard expression forms include variables, unit, functions, function applications, pairs with a single-branch pattern-matching construct for deconstructing them, value tagging \((L \text{ and } R)\), pattern matching on tagged values, and fold/unfold. Luck’s nonstandard additions are unknowns \((u)\), narrow \((e \leftarrow (e_1, e_2))\), sample \((\& e)\) and after \((e_1; e_2)\) expressions.

After is a sequencing operator, evaluating both \(e_1\) and \(e_2\) in sequence. However, unlike the standard sequencing operator \(e_1; e_2\), the result of \(e_1; e_2\) is the result of \(e_1\), rather than that of \(e_2\); the expression \(e_2\) is evaluated for its side-effects. For example, the sample-after expression \(e \& \text{int}\) of the previous section is desugared to a combination of sample and after: \(x : \text{int}\). Before this snippet can be evaluated, \(x\) has to be substituted with a value, for instance with an unknown \(u\). The expression \(e\) is the first to be evaluated, refining the domain of \(u\) (amongst other unknowns). Finally, the sample expression \(u\&\) is evaluated for its side effect, instantiating \(u\) to a uniformly generated value from its domain. A reasonable way to implement \(e_1; e_2\) using standard lambda abstractions would be \((\lambda x. (\lambda _x e_2)) e_1\). However, there is a slight difference in the semantics of such an encoding compared to our intended semantics, which will be discussed later in this section \[3.5\].

Weight annotations like the ones in the \texttt{bst} example, can be similarly desugared using narrow expressions, assuming a standard encoding of binary search trees \((\text{Tree} = \mu X.1 + \text{int} \times X \times X)\) and naturals, using syntactic sugar for constant naturals:

\[
\text{case } (\text{unfold}_{\text{tree}} \text{tree} \leftarrow (1, \text{size})) \text{ of } (L x \rightarrow \ldots)(R y \rightarrow \ldots)
\]

Most of the typing rules are standard; they can be found in \[3.2\]. An unknown \(u\) has non-functional type \(T\) if \(U(u) = T\). The four non-standard rules appear in Fig. 3. An unknown \(u\) has non-functional type \(T\) if \(U(u) = T\). If \(e_1\) and \(e_2\) are well-typed, then \(e_1; e_2\) shares the type of \(e_1\). A narrow expression \(e \leftarrow (e_1, e_2)\) is well-typed if \(e\) has sum type \(T_1 + T_2\) and \(e_1, e_2\) are natural numbers. A sample expression \(\& e\) has the (non-functional) type \(T\) when \(e\) has type \(T\). We define the predicate semantics for Core Luck, written \(\models v\), as a big-step operational semantics. We assume that \(e\) is closed with respect to ordinary variables and free of unknowns. The rules for the standard constructs are unsurprising (see the extended version). The only non-standard rules are for narrow, sample and after expressions, which are essentially ignored (Fig. 4). Using the predicate semantics we can implement the naive generate-and-test method for generating valuations satisfying a predicate: generate arbitrary valuations and filter them.
A constraint set \( \kappa \) denotes a set of valuations (written \([\kappa]\)), representing the solutions to the constraints. Constraint sets also carry type information about existing unknowns: \( U(\kappa) \) is a mapping from \( \kappa \)'s unknowns to types. A constraint set \( \kappa \) is well-typed (\( \vdash \kappa \)) if for every valuation \( \sigma \) in the denotation of \( \kappa \) and every unknown \( u \) bound in \( \sigma \), the type map \( U(\kappa) \) contains \( u \) and \( [\emptyset]; U(\kappa) \vdash \sigma(u) : U(\kappa)(u) \).

Many of the semantic rules need to introduce fresh unknowns. The \textit{fresh} function takes as inputs a constraint set \( \kappa \) and a sequence of types of length \( k \); it draws the next \( k \) unknowns (in some deterministic order) from the infinite set \( U \) and extends \( U(\kappa) \) with the respective bindings. Note that the types given to \textit{fresh} cannot contain any arrows (its argument type is \( U^+ \)).

The main way constraints are introduced during evaluation is unification. Given a constraint set \( \kappa \) and two values, each potentially containing unknowns, \textit{unify} updates \( \kappa \) to preserve only those valuations in which the values match.

\( SAT \) is a total predicate on constraint sets that holds if the denotation of the input \( \kappa \) contains at least one valuation.

The value-extraction function \( \kappa[u] \) returns an optional (non-unknown) value: if in the denotation of \( \kappa \), all valuations map \( u \) to the same value \( v \), then that value is returned (written \([v] \)); otherwise we return nothing (written \([\emptyset]\)). Finally, the \textit{sample} operation is used to implement sample expressions (\( \{e\} \)): given a constraint set \( \kappa \) and an unknown \( u \in U(\kappa) \), it returns a list of constraint sets representing all possible concrete choices for \( u \), in all of which \( u \) is completely determined—that is, \( \forall \sigma \in \{\kappa\} \), \( \exists u. \, \kappa[u] = [v] \). To allow for reasonable implementations of this interface, we maintain an invariant that the input unknown to \textit{sample} will always have a finite denotation; thus, the resulting list is also finite.

### 3.3 Narrowing Semantics

As a step toward a semantics for Luck that incorporates both constraint solving and local instantiation, we first define a simpler narrowing semantics. This semantics may be of some interest in its own right, in that it extends traditional “needed narrowing” with explicit probabilistic instantiation points, but our aim here is to use it as a subroutine of the matching semantics in \( \S3.4 \). The narrowing evaluation judgment takes as inputs an expression \( e \) and a constraint set \( \kappa \). As in the predicate semantics, evaluating \( e \) returns a value \( v \), but now it depends on a constraint set \( \kappa \), and also returns a new constraint set \( \kappa' \). The latter is intuitively a refinement of \( \kappa \)—i.e., evaluation will only remove valuations.

\[ e \equiv [v] \kappa \kappa' \equiv v \]

We annotate the semantics with a representation of the sequence of random choices made during evaluation, in the form of a \textit{trace} \( t \). A trace is a sequence of \textit{Choices}: integer pairs \((m, n)\) with \( 0 \leq m < n \), where \( n \) denotes the number of possibilities and \( m \) is the index of the one actually taken. We write \( e \) for the empty trace and \( t' \) for the concatenation of two traces. We also annotate the judgment with the probability \( q \) of making the choices represented in the trace. Recording traces is useful after the fact in calculating the total probability of some given outcome of evaluation (which may be reached by many different derivations). Traces play no role in determining how evaluation proceeds.

We maintain the invariant that the input constraint set \( \kappa \) is well typed and the input expression \( e \) is well typed with respect to an empty variable context and the unknown context \( U(\kappa) \). Another invariant is that every constraint set \( \kappa \) that appears as input to a judgment is satisfiable, and the restriction of its denotation to the unknowns in \( e \) is finite. These invariants are established at the top-level (as discussed in \( \S3.4 \)). The finiteness invariant ensures the output of \textit{sample} will always be a finite collection (and therefore the probabilities involved will be positive rational numbers). Moreover, they guarantee termination of constraint solving, as we will see in the next subsection (3.4). Finally, we assume that the type of every expression has been determined by an initial type-checking phase. We write \( e \downarrow t \) to show that \( e \) has type \( T \). This is used in the semantic rules to provide type information for fresh unknowns.

The narrowing semantics is given in Fig. 5 for the standard constructs (omitting \textit{fold/unfold} and \textit{N-R} and \textit{N-Case-R} rules analogous to the \textit{N-L} and \textit{N-Case-L} rules shown) and in Fig. 6 for the instantiation expressions. Fig. 7 and Fig. 8 give some auxiliary definitions. Most of the rules are intuitive. A common pattern is sequencing two narrowing judgments \( e_1 \equiv [v_1] \kappa_1 \kappa_2 \equiv [v] \) and \( e_2 \equiv [v_2] \kappa_2 \kappa_3 \equiv [v] \). The constraint-set result of the first narrowing judgment is given as input to the second, while traces and probabilities are accumulated by concatenation \((t_1 \cdot t_2)\) and multiplication \((q_1 \cdot q_2)\). We now explain the rules in detail.

Rule \textit{N-Base} is the base case of the evaluation relation, treating values that are not handled by other rules by returning them as-is. There is no choice to be made, so the probability of the result is \( 1 \) and the trace is empty.

Rule \textit{N-Pair}: To evaluate \((e_1, e_2)\) given a constraint set \( \kappa \), we sequence the derivations for \( e_1 \) and \( e_2 \).
\[
\begin{align*}
N-Base & \quad v = () \lor v = (rec (f : T_1 \to T_2) x = e') \lor v \in \mathcal{U} \\
& \quad v = \kappa \triangledown_1 \kappa = v
\end{align*}
\]

\[
\begin{align*}
N-Pair & \quad e_1 = \kappa \triangledown_1 \kappa_1 \vdash v_1, e_2 = \kappa_1 \triangledown_2 \kappa_2 \vdash v_2 \\
& \quad (e_1, e_2) = \kappa \triangledown_{q_1+q_2} \kappa_2 \vdash (v_1, v_2)
\end{align*}
\]

\[
\begin{align*}
N-Case-P & \quad e' [v_1/x, v_2/y] = \kappa \triangledown_{q_1+q_2} \kappa' \vdash v
\end{align*}
\]

\[
\begin{align*}
N-Case-U & \quad e_1 [v_1/x, v_2/y] = \kappa \triangledown_{q_1+q_2} \kappa' \vdash v
\end{align*}
\]

\[
\begin{align*}
N-App & \quad e_1 [v_1/x, v_2/y] = \kappa \triangledown_{q_1+q_2} \kappa' \vdash v
\end{align*}
\]

\[
\begin{align*}
N-After & \quad e_1 = \kappa \triangledown_1 \kappa_1 \vdash v_1, e_2 = \kappa_1 \triangledown_2 \kappa_2 \vdash v_2
\end{align*}
\]

\[
\begin{align*}
N-Bang & \quad l \in e \triangledown_{q_1+q_2} \kappa' \vdash v
\end{align*}
\]

\[
\begin{align*}
N-Narrow & \quad v = () \lor v = (rec (f : T_1 \to T_2) x = e') \lor v \in \mathcal{U} \\
& \quad v = \kappa \triangledown_1 \kappa = v
\end{align*}
\]

\[
\begin{align*}
N-Pair & \quad e_1 = \kappa \triangledown_1 \kappa_1 \vdash v_1, e_2 = \kappa_1 \triangledown_2 \kappa_2 \vdash v_2 \\
& \quad (e_1, e_2) = \kappa \triangledown_{q_1+q_2} \kappa_2 \vdash (v_1, v_2)
\end{align*}
\]

\[
\begin{align*}
N-Case-P & \quad e' [v_1/x, v_2/y] = \kappa \triangledown_{q_1+q_2} \kappa' \vdash v
\end{align*}
\]

\[
\begin{align*}
N-Case-U & \quad e_1 [v_1/x, v_2/y] = \kappa \triangledown_{q_1+q_2} \kappa' \vdash v
\end{align*}
\]

\[
\begin{align*}
N-App & \quad e_1 [v_1/x, v_2/y] = \kappa \triangledown_{q_1+q_2} \kappa' \vdash v
\end{align*}
\]

\[
\begin{align*}
N-After & \quad e_1 = \kappa \triangledown_1 \kappa_1 \vdash v_1, e_2 = \kappa_1 \triangledown_2 \kappa_2 \vdash v_2
\end{align*}
\]

\[
\begin{align*}
N-Bang & \quad l \in e \triangledown_{q_1+q_2} \kappa' \vdash v
\end{align*}
\]

\[
\begin{align*}
N-Narrow & \quad v = () \lor v = (rec (f : T_1 \to T_2) x = e') \lor v \in \mathcal{U} \\
& \quad v = \kappa \triangledown_1 \kappa = v
\end{align*}
\]

\[
\begin{align*}
N-Pair & \quad e_1 = \kappa \triangledown_1 \kappa_1 \vdash v_1, e_2 = \kappa_1 \triangledown_2 \kappa_2 \vdash v_2 \\
& \quad (e_1, e_2) = \kappa \triangledown_{q_1+q_2} \kappa_2 \vdash (v_1, v_2)
\end{align*}
\]

\[
\begin{align*}
N-Case-P & \quad e' [v_1/x, v_2/y] = \kappa \triangledown_{q_1+q_2} \kappa' \vdash v
\end{align*}
\]

\[
\begin{align*}
N-Case-U & \quad e_1 [v_1/x, v_2/y] = \kappa \triangledown_{q_1+q_2} \kappa' \vdash v
\end{align*}
\]

\[
\begin{align*}
N-App & \quad e_1 [v_1/x, v_2/y] = \kappa \triangledown_{q_1+q_2} \kappa' \vdash v
\end{align*}
\]
Both outcomes are equiprobable (because of the 1 arguments to choose), so the probability is one half in each case. This binary uniform choice is recorded in the trace $T_{k}$ as either (0, 2) or (1, 2). Finally, we evaluate the expression corresponding to the chosen index, with the corresponding unknown substituted for the variable.

Checking for satisfiability enforces the invariant that constraint sets are satisfiable, which in turn ensures that $\kappa_{1}$ and $\kappa_{2}$ cannot both be unsatisfiable at the same time.

Rule N-App: To evaluate an application $(e_{0}, e_{1})$, we first evaluate $e_{0}$ to $(\mathit{rec} (f : T_{1} \rightarrow T_{2}) \ x = e_{2})$ (since unknowns only range over arrow-free types $\overline{T}$, the result cannot be unknown) and its argument $e_{1}$ to a value $v_{1}$. We then perform the appropriate substitutions, evaluate $e_{2}$ of $(\mathit{rec} (f : T_{1} \rightarrow T_{2}) \ x = e_{2})/f, v_{1}/x$, and combine the various probabilities and traces appropriately.

Rule N-After is similar to N-Pair; however, we only keep the result of the first narrowing to implement the reverse form of sequencing described in the introduction of this section.

Rule N-Bang: To evaluate $e$ to a value $v$, then use the auxiliary relation $\mathit{sample} V$ [Fig. 7] to completely instantiate $v$, walking down the structure of $e$. When unknowns are encountered, $\mathit{sample}$ is produced to a list of constraint sets $S$; with probability $\frac{1}{|S|}$ (where $|S|$ is the size of the list) we can select the $m$th constraint set in $S$, for each $0 \leq m < |S|$.

The narrow rule N-Narrow is similar to N-Case-U. The main difference is the “weight” arguments $e_{1}$ and $e_{2}$. These are evaluated to values $v_{1}$ and $v_{2}$, and $\mathit{sample} V$ is called to ensure that they are fully instantiated in all subsequent constraint sets, in particular in $\kappa_{1}$. The relation $\mathit{nat}_{\kappa_{1}} (v_{1}) = n_{1}$ walks down the structure of the value $v_{1}$ (like $\mathit{sample} V$) and calculates the unique integer $n_{1}$ corresponding to $v_{1}$. Specifically, when the input value is an unknown, $\mathit{nat}_{\kappa_{1}} (u) = n$ holds if $\kappa[u] = v'_{n}$ and $\mathit{nat}_{\kappa_{0}} (v_{n}) = n$ (the notation $\mathit{nat}_{\kappa_{i}} (v)$ is defined in Fig. 3). The rest of the rule is the same as N-Case-U, except that the computed weights $n_{1}$ and $n_{2}$ are given as arguments to choose in order to shape the distribution accordingly.

Using the narrowing semantics, we can implement a more efficient method for generating valuations than the naive generate and-test described in Section §3.3. Instead of generating arbitrary valuations we only lazily instantiate a subset of unknowns as we encounter them. This method has the additional advantage that, if a generated valuation yields an unwanted result, the implementation can backtrack to the point of the latest choice, which can drastically increase performance [17].

### 3.4 Matching Semantics

The narrowing semantics gives a straightforward way of generating random values from terms containing unknowns. Typically, though, we don’t want to obtain any old random valuation; in particular, for an expression of type Boo1, we may want only valuations for which the predicate semantics yields True. Unfortunately, this is not the case for the narrowing semantics.

To see why this is a problem, consider three unknowns, $u_{1}, u_{2}$, and $u_{3}$, and a constraint set $\kappa$ where each unknown has type Boo1—i.e., $1 + 1 + 1$—and the domain each unknown is associated with contains both True and False—$L_{1+1} ()$ and $R_{1+1} ()$. Suppose we want to generate valuations for these three unknowns such that the conjunction $u_{1} \& u_{2} \& u_{3}$ holds, where $e_{1} \& e_{2} \& e_{3}$ is shorthand for case $e_{1}$ of $(L \ x \rightarrow e_{2}) (R \ y \rightarrow \mathit{False})$. If we attempt to narrow the expression $u_{1} \& u_{2} \& u_{3}$, we first apply the N-Case-U rule with $e = u_{1}$. That means that $u_{1}$ will be unified with either $L_{1}$ or $R_{1}$ and a fresh unknown with equal probability, yielding a False value for the entire conjunction 50% of the time. If we chose to unify $u_{3}$ with an $L$, then we apply the N-Case-U rule again, returning either True or $u_{3}$ (since unknowns are values and returned by narrowing) with equal probability. Therefore, we will have generated a desired valuation only 25% of the time, and we would need to backtrack.

The problem is that the narrowing semantics is agnostic to the desired result. In this section we present a matching semantics that takes as an additional input a pattern (a value not containing lambdas but possibly containing unknowns). The matching semantics propagates this pattern backwards to guide the generation process. By allowing our semantics to look ahead, we can potentially avoid case branches that lead to non-matching results.

The big-step matching judgment has the form

\[
\begin{align*}
\mathit{M-Base} & \quad v = () \lor v \in \mathcal{U} \quad \kappa' = \mathit{unify} \ k \upsilon \ v \ p \\
& \quad p = \mathit{v} = \kappa \uplus v \ U \\
& \quad \text{if} \ SAT (\kappa') \text{then} \{ \kappa' \} \text{ else } \emptyset
\end{align*}
\]

where $p$ can mention the unknowns in $U (\kappa)$ and where the metavariable $\kappa'$ stands for an optional constraint set ($\emptyset$ or $\kappa$) returned by matching. Returning an option allows us to calculate the probability of backtracking by summing the q's of all failing derivations. (The combined probability of failures and successes may be less than 1 because some reduction paths may diverge.)

We keep the invariants from §3.3: the input constraint set $\kappa$ is well-typed and so is the input expression $v$ (with respect to an empty variable context and $U (\kappa)$); moreover $\kappa$ is satisfiable, and the restriction of its denotation to the unknowns in $v$ is finite. To these invariants we add that the input pattern $p$ is well-typed in $U (\kappa)$ and that the common type of $e$ and $p$ does not contain any arrows ($e$ can still contain functions and applications internally; these are handled by calling the narrowing semantics).

The rules except for case are similar to the narrowing semantics. The rules [Fig. 9] show several of the rest appear in the extended version. Rule M-Base: To generate valuations for a unit value or an unknown, we unify $v$ and the target pattern $p$ under the input constraint set $\kappa$. Unlike in N-Base, there is no case for functions, since the expression being evaluated must have a non-function type.

Rules M-Pair, M-Pair-Fail: To evaluate $(e_{1}, e_{2})$, where $e_{1}$ and $e_{2}$ have types $T_{1}$ and $T_{2}$, we first generate fresh unknowns $u_{1}$ and $u_{2}$. We unify the pair $(u_{1}, u_{2})$ with the target pattern $p$, obtaining a new constraint set $\kappa'$. We then proceed as in N-Pair, evaluating $e_{1}$ against pattern $u_{1}$ then $e_{2}$ against $u_{2}$, threading...
We model the behavior of constraint solving: instead of actually using the patterns to guide the evaluation of the scrutinee as well; $e_2$ is not evaluated, and the final trace and probability are $t_1$ and $q_1$.

Rules M-App, M-After: To evaluate an application $(e_0 \ e_1)$, we use the narrowing semantics to reduce $e_0$ to $(\text{rec } f \ x = e_2)$ and $e_1$ to a value $v_1$, then evaluate $e_2((\text{rec } f \ x = e_2)/f,v_2/x)$ against the original $p$. In this rule we cannot use a pattern during the evaluation of $e_1$: we do not have any candidates! This is the main reason for introducing the sequencing operator $e_1 : e_2$. In M-After, we evaluate $e_1$ against $p$ and then evaluate $e_2$ using narrowing, just for its side effects. If we used lambdas to encode sequencing, $e_1$ would be narrowed instead, which is not what we want.

The interesting rules are the ones for case when the type of the scrutinee does not contain functions. For these rules, we can actually use the patterns to guide the evaluation of the scrutinee as well. We model the behavior of constraint solving: instead of choosing which branch to follow with some probability (50% in N-Case-U), we evaluate both branches, just like a constraint solver would exhaustively search the entire domain.

Before looking at the rules in detail, we need to extend the constraint set interface with two new functions:

```latex
\text{rename} :: \ U^* \rightarrow \mathcal{C} \rightarrow \mathcal{C}
\text{union} :: \ \mathcal{C} \rightarrow \mathcal{C} \rightarrow \mathcal{C}
```

The rename operation freshens the provided constraint set by replacing all the unknowns in the provided sequence with freshly generated ones. The union of two constraint sets intuitively denotes the union of their corresponding denotations. However, to permit simple yet reasonably efficient implementations like the one in [8], this operation is permitted to overapproximate the actual union. To ensure that the overapproximation does not go too far, we pass it an additional argument that acts as an upper bound. That is, \( \text{union} \ \kappa_1 \ \kappa_2 \) yields a constraint set that contains both \( \kappa_1 \) and \( \kappa_2 \) and is contained in \( \kappa \).

Two of the rules appear in Fig. 10. (A third is symmetric to M-Case-2; a fourth handles failures.) We independently evaluate \( e \) against both an \( L \) pattern and an \( R \) pattern. If both of them yield failure, then the whole evaluation yields failure (elided). If exactly one succeeds, we evaluate just the corresponding branch (M-Case-2 or the other elided rule). If both succeed (M-Case-1), we evaluate both branches and combine the results with \( \text{union} \). The initial constraint set is again provided, to bound any over-approximation. We use \( \text{rename} \) to avoid conflicts, since we may generate the same fresh unknowns while independently computing \( \kappa_{1u} \) and \( \kappa_{2u} \).

When desired, the user can ensure that only one branch will be executed by using a narrow expression before the case is reached. Since \( e \) will then begin with a concrete constructor, only one of the evaluations of \( e \) against the patterns \( L \) and \( R \) will succeed, and only the corresponding branch will be executed.

The M-Case-1 rule is the second place where the need for finiteness of the restriction of \( \kappa \) to the input expression \( e \) arises. In order for the semantics to terminate in the presence of (terminating) recursive calls, it is necessary that the domain be finite. Consider a simple recursive function that holds for every natural number:

\[
\text{rec} \ (f : \text{nat} \rightarrow \text{bool}) \ u = \ \\
\text{case unfold} \ u \text{ of} \ (\text{L } x \rightarrow \text{True}) \ (R \ y \rightarrow (f \ y))
\]

Even though \( f \) terminates in the predicate semantics for every input \( u \), if we allow a constraint set to map \( u \) to the infinite domain of all natural numbers, the matching semantics will not terminate. In practice, using the case rule for evaluation with large finite domains can also be prohibitively costly. For this reason, our implementation introduces primitive integers; constraints on them are implemented efficiently and proved to implement the same behavior as recursive functions over Peano representations in the core calculus.

3.5 Properties

In this section, we summarize the properties that we have proven about core Luck. More details and proofs can be found in the extended version. Intuitively, when we evaluate an expression \( e \) against a pattern \( p \) in the presence of a constraint set \( \kappa \), we can only remove valuations from the denotation of \( \kappa \) (decreasingness); every valuation that matches \( p \) will be found in the denotation of the resulting constraint set of some derivation (completeness); and any derivation in the generator semantics corresponds to an execution in the predicate semantics (soundness).

Since we have two flavors of generator semantics, narrowing and matching, we also present these properties in two steps. First, we present the properties for the narrowing semantics. (Their proofs have been verified using Coq.) Then, we present the properties for the matching semantics; these proofs are similar to the narrowing ones (details can be found in the extended version; the only real difference is the case rule).

We write \( \text{dom}(\kappa) \) for the domain of \( \kappa \) (the common domain of its valuations) and \( \sigma_{\kappa_1} \) for the restriction of a valuation \( \sigma \) to a set of unknowns \( x \) or to the domain of another valuation \( x \). Two constraint sets are said to be ordered, written \( \kappa_1 \leq \kappa_2 \) if \( \text{dom}(\kappa_1) \subseteq \text{dom}(\kappa_2) \) and, for all valuations \( \sigma \in [\kappa_1] \), the restriction of \( \sigma \) to the domain of \( \kappa_2 \) is in its denotation \( [\sigma]_{\text{dom}(\kappa_2)} \subseteq [\kappa_2] \).

Properties of the Narrowing Semantics

Decreasingness informally states that we never add new valuations to our constraint sets; our semantics can only refine the denotation of the input \( \kappa \).

**Theorem 3.5.1 (Decreasingness)**

\[
e = \kappa \Downarrow \kappa' \Downarrow v \Rightarrow \kappa' \leq \kappa
\]
Completeness and soundness can be visualized as follows:

\[
e_p \downarrow v_p \quad \Rightarrow \quad \exists \sigma' \in \sigma_p[q.]\ \sigma'|_\sigma = \sigma \land \sigma' \in [e]\]

Given the top and left sides of the diagram, completeness guarantees that we can fill in the bottom and right. That is, given a predicate derivation \(e_p \downarrow v_p\) and a “factoring” of \(e_p\) into an expression \(e\) and a constraint set \(\kappa\) such that for some valuation \(\sigma \in [e]\) substituting \(\sigma\) in \(e\) yields \(e_p\), and under the assumption that everything is well typed, there is always a nonzero probability of obtaining some factoring of \(v_p\) as the result of a narrowing judgment.

**Theorem 3.5.2 (Completeness).**

\[
e_p \downarrow v_p \quad \Rightarrow \quad \exists \sigma' \in \sigma_p[q.]\ \sigma'|_\sigma = \sigma \land \sigma' \in [e]\]

\[
\sigma(e) = e_p \quad \sigma \in [e] \quad \frac{\sigma'(v) = v_p}{\sigma'(v) = v_p} \quad e \models e_p \downarrow v_p \quad \Rightarrow \quad \exists \sigma' \in \sigma_p[q.]\ \sigma'|_\sigma = \sigma \land \sigma' \in [e]\]

Soundness guarantees the opposite direction: any narrowing derivation \(e \equiv \kappa \downarrow \kappa' \models v\) directly corresponds to a derivation in the predicate semantics, with the additional assumption that all the unknowns in \(e\) are included in the domain of the input constraint set \(\kappa\) (which can be replaced by a stronger assumption that \(e\) is well typed in \(\kappa\)).

**Theorem 3.5.3 (Soundness).**

\[
e \models e \equiv \kappa \downarrow \kappa' \models v \quad \sigma' \models e \quad \sigma' \in [\kappa'] \quad \forall u. \ u \in e \Rightarrow u \in \text{dom} (\kappa) \quad \Rightarrow \quad \exists \sigma_p \models e_p. \ [\sigma']_\sigma = \sigma \land \sigma' \in [e] \quad \sigma(e) = e_p \quad e_p \downarrow v_p\]

**Properties of the Matching Semantics**

The decreasingness property for the matching semantics is very similar to the narrowing semantics: if the matching semantics yields \(\{\kappa\}\), then \(\kappa'\) is smaller than the input constraint set.

**Theorem 3.5.4 (Decreasingness).**

\[
p \models e \equiv \kappa \downarrow \kappa' \models \kappa \quad \Rightarrow \quad \kappa' \ll \kappa\]

Because the matching semantics may explore both branches of a case, it can fall into a loop when the predicate semantics would not (by exploring a non-terminating branch that the predicate semantics does not take). For this reason we need to strengthen the premises of completeness to require that all valuations in the input constraint set result in a terminating execution.

**Theorem 3.5.5 (Completeness).**

\[
\sigma \models e \equiv \kappa \downarrow \kappa' \models \kappa \quad \frac{\sigma(e) = e_p \quad \sigma(p) = v_p}{\exists \kappa' \sigma' \models \text{True}.} \quad \frac{\forall \sigma' \in [\kappa]. \ [\sigma']_\sigma = \sigma \land \sigma' \in [\kappa'] \quad \sigma(e) = e_p \downarrow v_p}{p \models e \equiv \kappa \downarrow \kappa' \models \kappa}\]

An “opposite” strengthening of the premise is required in soundness: we only guarantee that the valuations we obtain in the result are satisfying if the input constraint set \(\kappa\) denotes a single valuation \(\sigma\), with respect to the unknowns contained in the input expression \(e\). The matching semantics can generate constraint sets containing valuations that do not satisfy the predicate, since the \textit{union} operation can be imprecise (discussed in §3.3.4 and in the extended version). However, the matching semantics \textit{can} be used to simulate the predicate semantics for an expression, all of whose unknowns are assigned unique values by \(\kappa\). We return to this point in §4.

**Theorem 3.5.6 (Soundness).**

\[
p \equiv e \equiv \kappa \downarrow \kappa' \models \kappa \quad \text{size}([\kappa], \kappa) = 1 \quad \sigma'(p) = v_p \land \sigma' \in [\kappa'] \quad \forall u. \ u \in e \downarrow u \in \text{dom} (\kappa) \quad \Rightarrow \quad \exists \sigma \models e_p. \ [\sigma']_\sigma = \sigma \land \sigma(e) = e_p \quad e_p \downarrow v_p\]

### 4. Implementation

We next describe the Luck top level, and we give an implementation of constraint sets as \textit{orthogonal maps}, instantiating the abstract specification presented in §3.

**At the Top Level**

The inputs provided to the Luck interpreter consist of an expression \(e\) of type \texttt{bool} containing zero or more free unknowns \(u\) (but no free variables), and an initial constraint set \(\kappa\) providing types and finite domains for each unknown in \(u\), such that their occurrences in \(e\) are well-typed (\(\emptyset; U(\kappa) \models e : 1 + 1\)). The interpreter matches \(e\) against True (that is, \(L_{1+1}()\)), to derive a refined constraint set \(\kappa'\):

\[
L_{1+1}() \equiv e \equiv \kappa \downarrow \kappa' \models \kappa'\]

This involves random choices, and there is also the possibility that matching fails (and the semantics generates \(\emptyset\) instead of \(\{\kappa'\}\)). In this case, we simply try again (up to an ad hoc limit). All the valuations \(\sigma \in [\kappa']\) should map the unknowns in \(u\) to the same values; that is, \(\sigma|_u\) must be the same for each \(\sigma\). The interpreter returns this common \(\sigma|_u\) if it exists, and retries otherwise.

There is no guarantee that the generator semantics will yield a \(\kappa'\) mapping every \(u\) to a unique value, or, even if it does, that these values will satisfy the predicate semantics. Consider, for example, the expression \((x < y) : \{!x \& \& y\}\) evaluated in an initial constraint set in which \(x, y \in \{1, \ldots, 10\}\). After matching True against \(x < y\), we know \(x \neq 10\) and \(y \neq 1\), but sampling may very well choose \(x = 9\) and \(y = 2\), which does not satisfy the predicate. The soundness theorem in §3.5 only guarantees that the valuations in the resulting constraint set satisfy the predicate interpretation if the initial \(\kappa\) denotes a single valuation \(\{\sigma\}\). To guarantee that the valuations we return do have this property, we transform expressions before supplying them to the interpreter, replacing an expression \(e\) with free unknowns \(u\) by \(e : \{!u \& \& \sigma\}\) (where \(!u\) means \(!u_1\ ; \ldots ; \; !u_n\) for each \(u_i \in u\)). That is, we match \(e\) against True, hopefully refining the domains of the unknowns, then we instantiate all the unknowns using their refined domains, and finally we check that the instantiation actually makes \(e\) true. If this is not the case, the interpreter will backtrack and try again. In fact, this “repetition after instantiation” trick is one that can be useful in Luck programs in general: if \((x < y) : \{!x \& \& y\}\) were to occur inside a Luck program, it could make sense to replace it by \(x < y\ ; \{!x \& \& x < y\ ; \; !y\}\), thus ensuring that \(y\) is not instantiated until we have refined its domain using the chosen value for \(x\); this reduces the probability of later backtracking at the cost of some repeated evaluation.

Since the Luck interpreter retries at the top-level if generation fails, no internal backtracking is actually \textit{necessary} in the implementation of the generator semantics. However, some local backtracking allows wrong choices to be reversed quickly and leads to an enormous improvement in performance [15]. Our implementation backtracks locally in calls to \textit{choose}: if \textit{choose} has two choices available and the first one fails when matching the instantiated expression against a pattern, then we immediately try the second choice instead. Effectively, this means that if \(e\) is already known to be of the form \(L_x\) then narrow will not choose to instantiate it us-

---

2 Our implementation actually allows the user to skip this transformation, which avoids evaluating \(e\) twice; this makes sense if the user knows that \(e\) can only generate constraint sets with the right property.
ing \( R_{\preceq} \) and vice versa. This may require matching against \( \varepsilon \) twice, and our implementation shares work between these two matches as far as possible. (It also seems useful to give the user explicit control over where backtracking occurs, but we leave this for future work.)

**Orthogonal Maps** The constraint set specification can be implemented in a variety of ways. On one extreme, we could represent constraint sets explicitly as sets of valuations, but this would cause big efficiency problems, since even unifying two unknowns would require traversing the whole set filtering out all valuations in which the unknowns are different. On the other extreme, we could represent a constraint set as an arbitrary logical formula over unknowns. While this is a compact representation, it does not really support the per-variable sampling we require. For our interpreter we choose a middle way, using a data structure we call orthogonal maps to represent sets of valuations; this allows for a very simple implementation at the cost of *soundly overapproximating* sets. An orthogonal map is a map from unknowns to ranges, which have the following syntax:

\[
\text{r} ::= () \mid \text{u} \mid (\text{r}, \text{r}) \mid \text{fold } \text{r} \mid L\text{r} \mid R\text{r} \mid \{L\text{r}, R\text{r}\}
\]

Ranges represent sets of (non-functional) values: units, unknowns, pairs of ranges, and \( L \) and \( R \) applied to ranges. We also include the option for a range to be a pair of an \( L \) applied to some range and an \( R \) applied to another. For example, the set of all Boolean values can be encoded compactly in a range (elding folds and type information) as \( \{L\}(R()) \). Similarly, the set \( \{0, 2, 3\} \) can be encoded as \( \{L\}(L(R(L(R())))) \), assuming a standard Peano encoding of naturals. And while this compact representation can represent all sets of naturals, not all sets of Luck values can be precisely represented: For instance the set \( \{(0, 1), (1, 0)\} \) cannot be represented using ranges, only approximated to \( \{(L()), R(L())\} \cup \{(L()), L(R())\} \), which represents the larger set \( \{(0, 0), (0, 1), (1, 0), (1, 1)\} \). This corresponds to a form of Cartesian abstraction, in which we lose any relation between the components of a pair, so if one used ranges as an abstract domain for abstract interpretation it would be hard to prove say sortedness of lists. Ranges are a rather imprecise abstract domain for algebraic datatypes, but the efficiency constraints in our constraint solving setting are quite different than in static analysis.

Orthogonal maps have a natural denotation as sets of valuations: just take the Cartesian product of the sets of values in the range of each unknown. This corresponds to a non-relational domain in abstract interpretation in terms—i.e., one that ignores the relationship between unknowns.

We implement constraint sets as pairs of a typing environment and an optional map from unknowns to ranges. A constraint set \( \kappa \) is \( SAT \) if the second element is not \( \emptyset \). The sample primitive indexes into the map and collects all possible values for an unknown. The more interesting operations with this representation are \text{unify} and \text{union}. The \text{unify} operation is implemented by straightforwardly translating the values to ranges and unifying those. For simplicity, unification of two ranges \( r_1 \) and \( r_2 \) in the presence of a constraint set \( \kappa \) returns both a constraint set \( \kappa' \) where \( r_1 \) and \( r_2 \) are unified and the unified range \( r' \). If \( r_1 = r_2 = () \) there is nothing to be done. If both ranges have the same top-level constructor, we recursively unify the inner subranges. If one of the ranges, say \( r_1 \), is an unknown \( u \) we index into \( \kappa \) to find the range \( r_u \) corresponding to \( u \), unify \( r_u \) with \( r_2 \) in \( \kappa \) to obtain a range \( r' \), and then map \( u \) to \( r' \) in the resulting constraint set \( \kappa' \). If both ranges are unknowns \( u_1, u_2 \) we unify their corresponding ranges to obtain \( r' \). We then pick one of the two unknowns, say \( u_1 \), to map to \( r' \) while mapping \( u_2 \) to \( u_1 \). To keep things deterministic we introduce an ordering on unknowns and always map \( u_i \) to \( u_j \) if \( u_i < u_j \). Finally, if one range is the compound range \( \{L\text{r}_1, R\text{r}_1\} \) while the other is \( L\text{r}_2 \), the resulting range is only \( L \) applied to the result of the unification of \( r_1 \) and \( r_2 \). The \text{union} operator is similar; the only difference is that, when \( r_1 = L\text{r}_1 \) and \( r_2 = R\text{r}_2 \), instead of failing (as unification does), we create the compound range \( \{L\text{r}_1, R\text{r}_2\} \).

## 5. Evaluation

To evaluate the expressiveness and efficiency of Luck’s hybrid approach to test case generation, we tested it with a number of small examples and two significant case studies: generating well-typed lambda terms and information-flow control machine states. The Luck code is generally much smaller and cleaner than that of existing handwritten generators, though the Luck interpreter takes longer to generate each example (around 20 \( \times \) to 36 \( \times \) for the more complex generators). Finally, while this is admittedly a subjective impression, we found it significantly easier to get the generators right in Luck.

**Small Examples** The literature on random test generation includes many small examples, including various list predicates such as \text{sorted, member, and distinct} and tree predicates like BSTs and red-black trees. In the extended version we show the implementation of all of these examples in Luck, illustrating how we can write predicates and generators together with minimal effort.

We use red-black trees to compare the efficiency of our Luck interpreter to generators provided by commonly used tools, like QuickCheck (random testing), SmallCheck (exhaustive testing) and Lazy SmallCheck [61]. Lazy SmallCheck leverages laziness to greatly improve upon out-of-the-box QuickCheck and SmallCheck generators in the presence of sparse preconditions, by using partially defined inputs to explore large parts of the search space at once. Using both Luck and Lazy SmallCheck, we attempted to generate 1000 red black trees with a specific black height \( bh \)—meaning that the depth of the tree can be as large as \( 2 \cdot bh + 1 \). Results are shown in Fig. 11. Lazy SmallCheck was able to generate all 227 trees of black height 2 in 17 seconds, fully exploring all trees up to depth 5. When generating trees of black height 3, which required exploring trees up to depth 7, Lazy SmallCheck was unable to generate 1000 red black trees within 5 minutes. At the same time, the Luck implementation lies consistently within an order of magnitude of a very efficient handwritten generator in QuickCheck that generates valid Red-Black trees directly. Using rejection-sampling approaches by generating trees and discarding those that don’t satisfy the red-black tree invariant (like using QuickCheck’s \text{or} SmallCheck’s \text{or} with \text{or} instead of \text{and}) is prohibitively costly: these approaches perform much worse than Lazy SmallCheck.

**Well-Typed Lambda Terms** Using our prototype implementation we reproduced the experiments of Palka et al. [55], who generated well-typed lambda terms in order to discover bugs in GHC’s strict-
ness analyzer. We encoded a model of simply typed lambda calculus with polymorphism in Luck, providing a large typing environment with standard functions from the Haskell Prelude to generate interesting well-typed terms. The generated ASTs were then pretty-printed into Haskell syntax and each one was applied to a partial list of the form: \([1, 2, \text{undefined}]\). Using the same version of GHC (6.12.1), we compiled each application twice: once with optimizations \((-O2)\) and once without and compared the outputs.

A straightforward Luck implementation of the polymorphic lambda calculus was not adequate for finding bugs efficiently. To improve its performance we borrowed tricks from the similar case study of Fetscher et al. [23], seeding the environment with monomorphic versions of possible constants and increasing the frequency of seq, a basic Haskell function that introduces strictness, to increase the chances of exercising the strictness analyzer. Using this, we discovered bugs that are similar (under quick manual inspection) to those found by Palka et al. and Fetscher et al.

The generation speed was slower than that of Palka’s handwritten generator. We were able to generate terms of average size 50 (number of internal nodes), and, grouping terms together in batches of 100, we got a total time of generation, unparsing, compilation and execution of around 30-35 seconds per batch. This is a slowdown of around 20x compared to that of Palka’s. However, our implementation is a total of 82 lines of fairly simple code, while the handwritten development is 1684 lines, accompanied by the phrase “...the code is difficult to understand, so reading it is not recommended” in its distribution page [55].

**Information-Flow Control** For a second large case study, we reimplemented a method for generating information-flow control machine states [36]. Given an abstract stack machine with data and instruction memories, a stack and a program counter, one attaches labels—security levels—to runtime values, propagating them during execution and restricting potential flows of information from high (secret) to low (public) data. The desired security property, termination-insensitive noninterference, states that if we start with two indistinguishable abstract machines \(s1\) and \(s2\) (i.e., all their low-tagged parts are identical) and run each of them to completion, then the resulting states \(s1’\) and \(s2’\) are also indistinguishable.

Hritcu et al. [56] found that efficient testing of this property could be achieved in two ways: either by generating instruction memories that allow for long executions and checking for indistinguishability at each low step (called LLNI, low-lockstep noninterference), or by looking for counter-examples to a stronger invariant (strong enough to prove noninterference), generating two arbitrary indistinguishable states and then running for a single step (SSNI, single step noninterference). In both cases, there is some effort involved in generating indistinguishable machines: for efficiency, one must first generate one abstract machine \(a\) and then vary \(a\), to generate an indistinguishable one \(a’\). In writing such a generator for variations, one must effectively reverse the indistinguishability predicate between states and then keep the two artifacts in sync.

We first investigated their stronger property (SSNI), by encoding the indistinguishability predicate in Luck, and using our prototype to generate small, indistinguishable pairs of states. In 216 lines we were able to describe both the predicate and the generator for indistinguishable machines. The same functionality required \(>1000\) lines of complex Haskell code in the handwritten version.

The handwritten generator is reported to generate an average of 18400 tests per second, while the Luck prototype generates 1400 tests per second, around 13 times slower.

The real promise of Luck, however, became apparent when we turned to LLNI. Hritcu et al. [56] generate long sequences of instructions using generation by execution: starting from a machine state where data memories and stacks are instantiated, they generate the current instruction ensuring it does not cause the machine to crash, then allow the machine to take a step and repeat. While intuitively simple, this extra piece of generator functionality took significant effort to code, debug, and optimize for effectiveness, resulting in more than 100 additional lines of code. The same effect was achieved in Luck by the following 6 intuitive lines, where we just put the previous explanation in code:

```
sig runsLong :: Int -> AS -> Bool
fun runsLong len st =
  if len <= 0 then True
  else case step st of
    | 99 % Just st' -> runsLong (len - 1) st'
    | 1 % Nothing -> True
```

We evaluated the test generator on the same set of buggy information-flow analyses as in Hritcu et al. [56]. We were able to find all of the same bugs, with similar effectiveness (in terms of number of bugs found per 100 tests). However, the Luck generator was 36 times slower (Luck: 100 tests/s, Haskell: 3600 tests/s). We expect to be able to greatly improve this result (and the rest of the results in this section) with a more efficient implementation that compiles Luck programs to QuickCheck generators directly, instead of interpreting them in an unoptimized prototype.

The success of the prototype in giving the user enough flexibility to achieve similar effectiveness with state-of-the-art generators, while significantly reducing the amount of code and effort required, suggests that the approach Luck takes is promising and points towards the need for a real, optimizing implementation.

### 6. Related Work

Luck lies in the intersection of many different topics in programming languages, and the potentially related literature is huge. Here we present just the most closely related work.

**Random Testing** The works that are most closely related to our own are the narrowing based approaches of Gligoric et al. [25], Claessen et al. [17][18] and Fetscher et al. [23]. Gligoric et al. use a “delayed choice” approach, which amounts to needed-narrowing, to generate test cases in Java. Claessen et al. exploit the laziness of Haskell, combining a needed-narrowing-like technique with FEAT [21], a tool for functional enumeration of algebraic types, to efficiently generate near-uniform random inputs satisfying a precondition. While their use of FEAT allows them to get uniformity by default, it is not clear how user control over the resulting distribution could be achieved. Fetscher et al. [23] also use an algorithm that makes local choices with the potential to backtrack in case of failure. Moreover, they add a simple version of constraint solving, handling equality and disequality constraints. This allows them to achieve excellent performance in testing GHC for bugs (as in [59]) using the “trick” of monomorphizing the polymorphic constants of the context. They present two different strategies for making local choices: uniformly at random, or by ordering branches based on their branching factor. While both of these strategies seem reasonable (and somewhat complementary), there is no way of exerting control over the distribution as necessary.

**Enumeration-Based Testing** An interesting related approach appears in the inspiring work of Bulwahn [1]. In the context of Isabelle’s [50] QuickCheck [6], Bulwahn automatically constructs enumerators for a given precondition via a compilation to logic programs using mode inference. This work successfully addresses the issue of generating satisfying valuations for preconditions directly and serves for exhaustive testing of “small” instances, significantly pushing the limit of what is considered “small” compared to previous approaches. Lindblad [45] and Runciman et al. [61] also provide support for exhaustive testing using narrowing-based techniques. Instead of implementing mechanisms that resemble nar-
rowing in standard functional languages, Fischer and Kuchen leverage the built in engine of the functional logic language Curry to enumerate tests satisfying a coverage criterion. In a later, black-box approach for Curry, Christiansen and Fischer additionally use level diagonalization and randomization to bring larger tests earlier in the enumeration order. While exhaustive testing is useful and has its own merits and advantages over random testing in a lot of domains, we turn to random testing because the complexity of our applications—testing noninterference or optimizing compilers—makes enumeration impractical.

**Constraint Solving**

Many researchers have turned to constraint-solving based approaches to generate random inputs satisfying preconditions. In the constraint solving literature around SAT witness generation, the pioneering work of Chakraborty et al. stands out because of its efficiency and its guarantees of approximate uniformity. However, there is no way—and no obvious way to add it—of controlling distributions. In addition, their efficiency relies crucially on the independent support being small relative to the entire space (where the support $X$ of a boolean formula $p$ is the set of variables appearing in $p$ and the independent support is a subset $D$ of $X$ such that no two satisfying assignments for $p$ differ only in $X \setminus D$). While true for typical SAT instances, this is not the case for random testing properties, like, for example, noninterference. In fact, a minimal independent support for indistinguishable machines includes one entire machine state and the high parts of another; thus, the benefit from their heuristics may be minimal. Finally, they require logical formulae as inputs, which would require a rather heavy translation from a high-level language like Haskell.

Such a translation from a higher-level language to the logic of a constraint solver has been attempted a few times to support testing [12][31], the most recent and efficient for Haskell being Target [62]. Target translates preconditions in the form of refinement types, and uses a constraint solver to generate a satisfying valuation for testing. Then it introduces the negation of the generated input to the formula, in order to generate new, different ones. While more efficient than Lazy SmallCheck in a variety of cases, there are still cases where a narrowing-like approach outperforms the tool, further pointing towards the need to combine the two approaches as in Luck. Moreover, the use of an automatic translation and constraint solving does not give any guarantees on the resulting distribution, neither does it allow for user control.

Constraint-solving is also used in symbolic evaluation based techniques, where the goal is to generate diverse inputs that achieve higher coverage [3][8][11][27][28][46][63]. Recently, in the context of Rosette [65], symbolic execution was used to successfully find bugs in the same information-flow control case study.

**Semantics for narrowing-based solvers**

Recently, Fowler and Hutton put needed-narrowing-based solvers on a formal mathematical foundation. They presented an operational semantics of a purely narrowing-based solver, named Reach, proving soundness and completeness. In their concluding remarks, they mention that native representations of primitive datatypes do not fit with the notion of lazy narrowing since they are “large, flat datatypes with strict semantics.” In Luck, we were able to exhibit the same behavior for both the primitive integers and their datatype encodings successfully addressing this issue, while at the same time incorporating constraint solving into our formalization.

**Probabilistic programming**

Semantics for probabilistic programs share many similarities with the semantics of Luck [29][40][48], while the problem of generating satisfying valuations shares similarities with probabilistic sampling [13][44][47][51]. For example, the semantics of PROB in the recent probabilistic programming survey of Gordon et al. [10] takes the form of probability distributions over valuations, while Luck semantics can be viewed as (sub)probability distributions over constraint sets, which induces a distribution over valuations. Moreover, in probabilistic programs, observations serve a similar role to preconditions in random testing, creating problems for simplistic probabilistic samplers that use rejection sampling—i.e., generate and test. Recent advances in this domain, like the work on Microsoft’s R2 Markov Chain Monte Carlo sampler [51], have shown promise in providing more efficient sampling, using pre-imaging transformations in analyzing programs. An important difference is in the type of programs usually targeted by such tools. The difficulty in probabilistic programming arises mostly from dealing with a large number of complex observations, modeled by relatively small programs. For example, Microsoft’s TrueSkill [39] ranking program is a very small program, powered by millions of observations. In contrast, random testing deals with very complex programs (for example a type checker) and a single observation without noise (observe true).

We did a simple experiment with R2, using the following probabilistic program to model the indistinguishability of $\phi$ where we use booleans to model labels:

```plaintext
    double v1 = Uniform.Sample(0, 10);
    double v2 = Uniform.Sample(0, 10);
    bool l1 = Bernoulli.Sample(0.5);
    bool l2 = Bernoulli.Sample(0.5);
    Observer.Observe(l1==l2 && (v1==v2 || l1));
```

Two pairs of doubles and booleans will be indistinguishable if the booleans are equal and, if the booleans are false, so are the doubles. The result was somewhat surprising at first, since all the generated samples have their booleans set to true. However, that is an accurate estimation of the posterior distribution: for every “false” indistinguishable pair there exist $2^{16}$ “true” ones! Of course, one could probably come up with a better prior or use a tool that allows arbitrary conditioning to skew the distribution appropriately. If, however, for such a trivial example the choices are non-obvious, imagine replacing pairs of doubles and booleans with arbitrary lambda terms and indistinguishability by a well-typed relation. Coming up with suitable priors that lead to efficient testing would become an ambitious research problem on its own!

7. Conclusions and Future Work

In this paper we presented Luck, a language for writing generators in the form of lightly annotated predicates. We presented the semantics of Luck, combining local instantiation and constraint solving in a unified framework, exploring their interesting interactions. We also developed a prototype implementation of this semantics and used it to replicate the results of state-of-the-art handwritten random generators for two complex domains. The results showed the significant potential of Luck’s approach, allowing us to replicate the generation presented by the handwritten generators with extremely reduced code and effort. The prototype was slower by an order of magnitude, but there is still a lot of room for improvement.

In the future it will be interesting to explore compilation of Luck into generators in a language like Haskell to improve the performance of our interpreted prototype. Another way to improve performance would be to experiment with more expressive domain representations (e.g., ones allowing a precise implementation of the union operation from [33][54]). We also want to investigate Luck’s equational theory, showing for instance that the encoded conjunction, negation, and disjunction satisfy the usual logical laws. Another potential direction for this work is automatically deriving smart shrinkers. Shrinking, or delta-debugging, is crucial in property-based testing, and can also require a lot of user effort and domain specific knowledge to be efficient [58]. It would be interesting to see if there is a counterpart to narrowing or constraint solving that allows shrinking to stay within the space of the preconditions.
References

[38] J. Hughes. QuickCheck testing for fun and profit. PADL. 2007.


