Beginner's Luck

A Language for Property-Based Generators

Leonidas Lampropoulos[1] Diane Gallois-Wong[2,3] Cătălin Hriţcu[2]
John Hughes[4] Benjamin C. Pierce[1] Li-yao Xia[2,3]

[1]University of Pennsylvania, USA [2]INRIA Paris, France [3]ENS Paris, France [4]Chalmers University, Sweden

Abstract

Property-based random testing *à la* QuickCheck requires building efficient generators for well-distributed random data satisfying complex logical predicates, but writing these generators can be difficult and error prone. We propose a domain-specific language in which generators are conveniently expressed by decorating predicates with lightweight annotations to control both the distribution of generated values and 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, showing 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.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Testing tools (e.g., data generators, coverage testing)

Keywords random testing; property-based testing; narrowing; constraint solving; domain specific language

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, 39, 46, 56] and for streamlining interaction with proof assistants [6, 15, 22, 55, 59].

When using a property-based random testing tool, one writes *properties* in the form of executable predicates. For example, a natural property to test for a list `reverse` function is that, for any list `xs`, reversing `xs` twice yields `xs` again. In QuickCheck notation:

```
prop_reverse xs  =  (reverse (reverse xs) == xs)
```

To test this property, QuickCheck generates random lists until either it finds a counterexample or a certain number of tests succeed.

An appealing feature of QuickCheck is that it offers a library of property combinators resembling standard logical operators. For example, a property of the form `p ==> q`, built using the implication combinator `==>`, will be tested automatically by generating *valuations* (assignments of random values, of appropriate type, to the free variables of `p` and `q`), discarding those valuations that fail to satisfy `p`, and checking whether any of the ones that remain are counterexamples to `q`.

QuickCheck users soon learn that this default generate-and-test approach sometimes does not give satisfactory results. In particular, if the precondition `p` is satisfied by relatively few values of the appropriate type, then most of the random inputs that QuickCheck generates will be discarded, so that `q` will seldom be exercised. Consider, for example, testing a simple property of a school database system: that every student in a list of `registeredStudents` should be taking at least one course,

```
prop_registered studentId =
  member studentId registeredStudents ==>
  countCourses studentId > 0
```

where, as usual:

```
member x [] = False
member x (h:t) = (x == h) || member x t
```

If the space of possible student ids is large (e.g., because they are represented as machine integers), then a randomly generated id is very unlikely to be a member of `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, 33, 37].

Property-based generators generators work well for small to medium-sized examples, but writing them can become challenging as `p` gets more complex—sometimes turning into a research contribution in its own right! For example, papers have been written about generation techniques for well-typed lambda-terms [23, 58, 62, 66] and for "indistinguishable" machine states that can be used for finding bugs in information-flow monitors [37, 38]. Moreover, if we use QuickCheck to test an *invariant* property (e.g., type preservation), then the same condition will appear in both the precondition and the conclusion of the property, requiring that we express this condition both as a boolean predicate `p` and as 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 confusion in the testing process. These difficulties are not hypothetical: Hriţcu *et al.*'s machine-state generator [37] is over 1500

lines of tricky Haskell, while Pałka *et al.*'s generator for well-typed lambda-terms [58] 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 `p` directly from `p` itself. Indeed, two variants of this idea, with complementary strengths and weaknesses, have been explored by others—one based on local choices and backtracking, one 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, 35, 46, 67]. It is attractively lightweight, admits natural control over distributions (as we will see in the next section), and has been used successfully [16, 24, 27, 61], even in challenging domains such as generating well-typed programs to test compilers [17, 23].

However, choosing a value for an 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:

```
notMember x []     =  True
notMember x (h:t)  =  (x /= h) && 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 $2\times$—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 want to generate a list 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.[1] This approach has been widely investigated, both for generating inputs directly from predicates [12, 32, 44, 64] and for symbolic-execution-based testing [3, 9, 28, 65, 68], 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 §2). 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 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 §2 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 (§3).
- We provide a prototype interpreter (§4) 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 (§2) 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 (§5) and on two significant case studies; the latter demonstrate 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 show comparable bug-finding effectiveness (measured in test cases generated per counterexample found) and a significant reduction in the size of testing code. The interpreted Luck generators run an order of magnitude slower than compiled QuickCheck versions (8 to 24 times per test), but many opportunities for optimization remain.

Sections §6 and §7 discuss related work and future directions. This paper is accompanied by several auxiliary materials: (1) a Coq formalization of the narrowing semantics of Luck and machine-checked proofs of its properties (available at `https://github.com/QuickChick/Luck`) (§3.3); (2) the prototype Luck interpreter and a battery of example programs, including all the ones we used for evaluation (also at `https://github.com/QuickChick/Luck`) (§5); (3) an extended version of the paper with full definitions and paper proofs for the whole semantics (`https://arxiv.org/abs/1607.05443`).

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 [54]. 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 re-

[1] Constraint solvers can, of course, be used to *directly* search for counterexamples to a property of interest by software model checking [4, 5, 40, 42, 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 tree =
  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 -> Tree Int -> Bool
fun bst size low high tree =
  if size == 0 then tree == Empty
  else case tree of
       | 1    % Empty -> True
       | size % Node x l r ->
          ((low < x && x < high) !x)
          && bst (size / 2) low x l
          && bst (size / 2) x high r
```

Figure 1. Binary Search Tree tester and two generators

turns 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, $\frac{1}{size+1}$ of the time it creates an `Empty` tree, while $\frac{size}{size+1}$ of the time it returns a `Node`. The `Node` 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 `bst t` then ⟨*some other property of* `t`⟩," 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 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., [37]).

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 ignoring weights and sampling expressions) and an efficient interpretation as a generator of random trees with the same distribution as the QuickCheck version. For example, evaluating the top-level query `bst 10 0 42 u = True`—i.e., "generate values `t` for the unknown `u` such that `bst 10 0 42 t` evaluates to `True`"—will yield random binary search trees of size up to 10 with node labels strictly between 0 and 42, with the same distribution as the QuickCheck generator `genTree 10 0 42`.

An *unknown* in Luck is a special kind of value, similar to logic variables found in logic programming languages and unification variables used by type-inference algorithms. Unknowns are typed, and each is associated with a domain of possible values from its type. Given an expression e mentioning some set U of unknowns, our goal is to generate *valuations* over these unknowns (maps from U to concrete values) by iteratively refining the unknowns' domains, so that, when any of these valuations is substituted into e, the resulting concrete term evaluates to a desired value (e.g., `True`).

Unknowns can be introduced both explicitly, as in the top-level query above (see also §4), and implicitly, as in the generator semantics of `case` expressions. In the `bst` example, when the `Node` branch is chosen, the pattern variables `x`, `l`, and `r` are replaced by fresh unknowns, which are then instantiated by evaluating the constraint `low < x && x < high` and the recursive calls to `bst`.

Varying the placement of unknowns in the top-level `bst` query yields different behaviors. For instance, if we change the query to `bst 10 ul uh u = True`, replacing the `low` and `high` parameters with unknowns `ul` and `uh`, the domains of these unknowns will be refined during tree generation and the result will be a generator for random valuations (`ul` $\mapsto$ `i`, `uh` $\mapsto$ `j`, `u` $\mapsto$ `t`) where `i` and `j` are lower and upper bounds on the node labels in `t`.

Alternatively, we can evaluate the top-level query `bst 10 0 42 t = True`, replacing `u` with a concrete tree `t`. In this case, Luck will return a trivial valuation only if `t` is a binary search tree; otherwise it will report that the query is unsatisfiable. A less useful possibility is that we provide explicit values for `low` and `high` but choose them with `low` $>$ `high`, e.g., `bst 10 6 4 u = True`. Since there are no satisfying valuations for `u` other than `Empty`, Luck will now generate only `Empty` trees.

A *sample-after expression* of the form `e !x` is used to control instantiation of unknowns. Typically, `x` will be an unknown `u`, and evaluating `e !u` will cause `u` to be instantiated to a concrete value (after evaluating `e` to refine the domains of all of the unknowns in `e`). If `x` reduces to a value rather than an unknown, we similarly instantiate any unknowns appearing within this value.

As a concrete example, consider the compound inequality constraint `0 < x && x < 4`. A generator based on pure narrowing (as in [27]), would instantiate `x` when the evaluator meets the first constraint where it appears, namely `0 < x` (assuming left-to-right evaluation order). We can mimic this behavior in Luck by writing `((0 < x) !x) && (x < 4)`. However, picking a value for `x` at this point ignores the constraint `x < 4`, which can lead to backtracking. If, for instance, the domain from which we are choosing values for `x` is 32-bit integers, then the probability that a random choice satisfying `0 < x` will also satisfy `x < 4` is minuscule. It is better in this case to write `(0 < x && x < 4) !x`, instantiating `x` after the entire conjunction has been evaluated and all the constraints on the domain of `x` recorded and thus avoiding backtracking completely. Finally, if we do not include a sample-after expression for `x` here at all, we can further refine its domain with constraints later on, at the cost of dealing with a more abstract representation of it internally in the meantime. Thus, sample-after expressions give Luck users explicit control over the tradeoff between the expense of possible backtracking—when unknowns are instantiated early—and the expense of maintaining constraints on unknowns—so that they can be instantiated late (e.g., so that `x` can be instantiated after the recursive calls to `bst`).

Sample-after expressions choose random values with *uniform* probability from the domain associated with each unknown. While this behavior is sometimes useful, effective property-based random testing often requires fine control over the distribution of generated test cases. Drawing inspiration from the QuickCheck combinator library for building complex generators, and particularly `frequency` (which we saw in `genTree` (Fig. 1)), Luck also allows weight annotations on the branches of a `case` expression which have a `frequency`-like effect. In the Luck version of `bst`, for example, the unknown `tree` is either instantiated to an `Empty` tree $\frac{1}{1+size}$ of the time or partially instantiated to a `Node` (with fresh unknowns for `x` and the left and right subtrees) $\frac{size}{1+size}$ of the time.

Weight annotations 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 x` 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 effective tool for adjusting `frequency` weights and dramatically increasing bug-finding rates (e.g., [37]). The Luck implementation provides a similar primitive.

One further remark on uniform sampling: 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 this often leads to inefficient testing in practice. A simple example comes from the information flow control experiments of Hriţcu *et al.* [37]. 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:

```
indist (v1,High) (v2,High)  =  True
indist (v1,Low ) (v2,Low)   =  v1 == v2
indist _          _         =  False
```

If we use 32-bit integers, then for every `Low` indistinguishable pair there are 2^{32} `High` ones! Thus, choosing a uniform distribution over indistinguishable pairs means that we will essentially never generate pairs with `Low` labels.

However, in other areas where random sampling is used, it is sometimes important to be able to generate globally uniform distributions; if desired, this effect can be achieved in Luck by emulating *Boltzmann samplers* [20]. This technique fits naturally in Luck, providing an efficient way of drawing samples from combinatorial structures of 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 omits primitive booleans and integers and replaces 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 in three steps. First, we give an interface to a *constraint solver* (§3.2), 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 (§3.3). Finally, we introduce a *matching semantics*, building on the narrowing semantics, that unifies constraint solving and narrowing into a single evaluator (§3.4). In the long version, we also show how integers and booleans can be encoded and how the semantics applies 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 (§3.6); 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).

$$
\begin{aligned}
v &::= () \mid (v, v) \mid L_T\ v \mid R_T\ v \\
&\quad \mid \mathit{rec}\ (f : T_1 \to T_2)\ x = e \mid \mathit{fold}_T\ v \\
&\quad \mid u \\
e &::= x \mid () \mid \mathit{rec}\ (f : T_1 \to T_2)\ x = e \mid (e\ e) \\
&\quad \mid (e, e) \mid \mathit{case}\ e\ \mathit{of}\ (x, y) \to e \\
&\quad \mid L_T\ e \mid R_T\ e \mid \mathit{case}\ e\ \mathit{of}\ (L\ x \to e)\ (R\ x \to e) \\
&\quad \mid \mathit{fold}_T\ e \mid \mathit{unfold}_T\ e \\
&\quad \mid u \mid e \leftarrow (e, e) \mid\ !e \mid e\ ;\ e \\
\overline{T} &::= X \mid 1 \mid \overline{T} + \overline{T} \mid \overline{T} \times \overline{T} \mid \mu X.\ \overline{T} \\
T &::= X \mid 1 \mid T + T \mid T \times T \mid \mu X.\ T \mid T \to T \\
\Gamma &::= \varnothing \mid \Gamma, x : T
\end{aligned}
$$

Figure 2. Core Luck Syntax

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 in examples, although in principle they could be encoded using recursive types.

Values include unit, pairs of values, sum constructors (L and R) applied to values (and annotated with types, to eliminate ambiguity), first class recursive functions (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 and R), pattern matching on tagged values, and fold/unfold. The nonstandard additions are unknowns (u), *instantiation* ($e \leftarrow (e_1, e_2)$), *sample* ($!e$) and *after* ($e_1\ ;\ e_2$) expressions.

The "after" operator, written with a backwards semicolon, evaluates 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; the expression e_2 is evaluated just for its side-effects. For example, the sample-after expression `e !x` of the previous section is desugared to a combination of sample and after: $e\ ;\ !x$. If we evaluate this snippet in a context where x is bound to some unknown u, then the expression e is evaluated first, refining the domain of u (amongst other unknowns); then the sample expression $!u$ is evaluated for its side effect, instantiating u to a uniformly generated value from its domain; and finally the result of e is returned as the result of the whole expression. A reasonable way to implement $e_1\ ;\ e_2$ using standard lambda abstractions would be as $(\lambda\ x.\ (\lambda\_.\ x)\ e_2)\ e_1$. However, there is a slight difference in the semantics of this encoding compared to our intended semantics—we will return to this point in §3.4.

$$\textbf{T-U}\ \frac{U(u) = \overline{T}}{\Gamma; U \vdash u : \overline{T}} \qquad \textbf{T-After}\ \frac{\Gamma; U \vdash e_1 : T_1 \qquad \Gamma; U \vdash e_2 : T_2}{\Gamma; U \vdash e_1 \,;\, e_2 : T_1}$$

$$\textbf{T-Bang}\ \frac{\Gamma; U \vdash e : \overline{T}}{\Gamma; U \vdash !e : \overline{T}} \qquad \textbf{T-Narrow}\ \frac{\Gamma; U \vdash e : \overline{T}_1 + \overline{T}_2 \qquad \Gamma; U \vdash e_l : nat \qquad \Gamma \vdash e_r : nat}{\Gamma; U \vdash e \leftarrow (e_l, e_r) : \overline{T}_1 + \overline{T}_2}$$

$$nat := \mu X.\ 1 + X$$

Figure 3. Typing Rules for Nonstandard Constructs

$$\textbf{P-Narrow}\ \frac{e \Downarrow v \qquad e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad [\![v_1]\!] > 0 \qquad [\![v_2]\!] > 0}{e \leftarrow (e_1, e_2) \Downarrow v} \qquad \textbf{P-Bang}\ \frac{e \Downarrow v}{!e \Downarrow v}$$

$$\textbf{P-After}\ \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \,;\, e_2 \Downarrow v_1}$$

$$\begin{array}{rcl} [\![fold_{nat}\ (L_{1+nat}\ ())]\!] & = & 0 \\ [\![fold_{nat}\ (R_{1+nat}\ v)]\!] & = & 1 + [\![v]\!] \end{array}$$

Figure 4. Predicate Semantics for Nonstandard Constructs

Weight annotations like the ones in the `bst` example can be desugared using *instantiation expressions*. For example, assuming a standard encoding of binary search trees ($Tree = \mu X.\ 1 + int \times X \times X$) and naturals, plus syntactic sugar for constant naturals:

$$case\ (unfold_{Tree}\ tree \leftarrow (1, size))\ of\ (L\ x \rightarrow \ldots)(R\ y \rightarrow \ldots)$$

Most of the typing rules are standard (these can be found in the extended version of the paper.) The four non-standard rules are given in Fig. 3. Unknowns are typed: each will be associated with a domain (set of values) drawn from a type $\overline{T}$ that does not contain arrows. Luck does not support constraint solving over functional domains (which would require something like higher-order unification), and the restriction of unknowns to non-functional types reflects this. To remember the types of unknowns, we extend the typing context to include a component U, a map from unknowns to non-functional types. When the variable typing environment $\Gamma = \varnothing$, we write $U \vdash e : T$ as a shorthand for $\varnothing; U \vdash e : T$. An unknown u has type $\overline{T}$ if $U(u) = \overline{T}$. If e_1 and e_2 are well typed, then $e_1 \,;\, e_2$ shares the type of e_1. An instantiation expression $e \leftarrow (e_l, e_r)$ is well typed if e has sum type $\overline{T}_1 + \overline{T}_2$ and e_l and e_r are natural numbers. A sample expression $!e$ has the (non-functional) type $\overline{T}$ when e has type $\overline{T}$.

The predicate semantics for Core Luck, written $e \Downarrow v$, are defined 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 the ones for narrow, sample and after expressions, which are essentially ignored (Fig. 4). With the predicate semantics we can implement a naive generate-and-test method for generating valuations satisfying some predicate by generating arbitrary well-typed valuations and filtering out those for which the predicate does not evaluate to `True`.

3.2 Constraint Sets

The rest of this section develops an alternative probabilistic generator semantics for Core Luck. This semantics will use *constraint sets* $\kappa \in \mathcal{C}$ to describe the possible values that unknowns can take. For the moment, we leave the implementation of constraint sets open (the one used by our prototype interpreter is described in §4), simply requiring that they support the following operations:

$$\begin{array}{lcl} [\![\cdot]\!] & :: & \mathcal{C} \rightarrow Set\ Valuation \\ U & :: & \mathcal{C} \rightarrow Map\ \mathcal{U}\ \overline{T} \\ fresh & :: & \mathcal{C} \rightarrow \overline{T}^* \rightarrow (\mathcal{C} \times \mathcal{U}^*) \\ unify & :: & \mathcal{C} \rightarrow Val \rightarrow Val \rightarrow \mathcal{C} \\ SAT & :: & \mathcal{C} \rightarrow Bool \\ [\cdot] & :: & \mathcal{C} \rightarrow \mathcal{U} \rightarrow Maybe\ Val \\ sample & :: & \mathcal{C} \rightarrow \mathcal{U} \rightarrow \mathcal{C}^* \end{array}$$

Here we describe these operations informally, deferring technicalities until after we have presented the generator semantics (§3.6).

A constraint set κ denotes a set of valuations ($[\![\kappa]\!]$), representing the solutions to the constraints. Constraint sets also carry type information about existing unknowns: $U(\kappa)$ is a mapping from κ's unknowns to types. A constraint set κ is *well typed* ($\vdash \kappa$) if, for every valuation σ in the denotation of κ and every unknown u bound in σ, the type map $U(\kappa)$ contains u and $\varnothing; U(\kappa) \vdash \sigma(u) : U(\kappa)(u)$.

Many of the semantic rules will need to introduce fresh unknowns. The *fresh* function takes as inputs a constraint set κ and a sequence of (non-functional) types of length k; it draws the next k unknowns (in some deterministic order) from the infinite set $\mathcal{U}$ and extends $U(\kappa)$ with the respective bindings.

The main way constraints are introduced during evaluation is unification. Given a constraint set κ and two values, each potentially containing unknowns, *unify* updates κ to preserve only those valuations in which the values match.

SAT is a total predicate that holds on constraint sets whose denotation contains at least one valuation. The totality requirement implies that our constraints must be decidable.

The value-extraction function $\kappa[u]$ returns an optional (non-unknown) value: if in the denotation of κ, all valuations map u to the same value v, then that value is returned (written $\{v\}$); otherwise nothing (written $\varnothing$).

The *sample* operation is used to implement sample expressions ($!e$): given a constraint set κ 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 $\forall \kappa \in (sample\ \kappa\ u).\ \exists v.\ \kappa[u] = \{v\}$. To allow for reasonable implementations of this interface, we maintain an invariant that the input unknown to *sample* will always have a finite denotation; thus, the resulting list is also finite.

3.3 Narrowing Semantics

As a first step toward a semantics for Core Luck that incorporates both constraint solving and local instantiation, we define a simpler *narrowing* semantics. This semantics is of some interest in its own right, in that it extends traditional "needed narrowing" with explicit probabilistic instantiation points, but its role here is as a subroutine of the matching semantics in §3.4.

The narrowing evaluation judgment takes as inputs an expression e and a constraint set κ. As in the predicate semantics, evaluating e returns a value v, but now it also depends on a constraint set κ and returns a new constraint set κ'. The latter is intuitively a refinement of κ—i.e., evaluation will only remove valuations.

$$e \Rightarrow \kappa \Downarrow_q^t \kappa' \vDash v$$

The semantics is annotated with a representation of the sequence of random choices made during evaluation, in the form of a *trace* t. A trace is a sequence of *choices*: integer pairs (m, n) with $0 \leqslant m < n$, where n denotes the number of possibilities chosen among and m is the index of the one actually taken. We write ϵ for the empty trace and $t \cdot t'$ for the concatenation of two traces.

$$\textbf{N-Base}\ \frac{v = () \ \vee\ v = (rec\ (f : T_1 \to T_2)\ x = e') \ \vee\ v \in \mathcal{U}}{v \rightrightarrows \kappa \Downarrow_1^{\epsilon} \kappa \vDash v}$$

$$\textbf{N-Pair}\ \frac{e_1 \rightrightarrows \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v_1 \qquad e_2 \rightrightarrows \kappa_1 \Downarrow_{q_2}^{t_2} \kappa_2 \vDash v_2}{(e_1, e_2) \rightrightarrows \kappa \Downarrow_{q_1 * q_2}^{t_1 \cdot t_2} \kappa_2 \vDash (v_1, v_2)}$$

$$\textbf{N-CasePair-P}\ \frac{\begin{array}{c} e \rightrightarrows \kappa \Downarrow_q^t \kappa_a \vDash (v_1, v_2) \\ e'[v_1/x,\ v_2/y] \rightrightarrows \kappa_a \Downarrow_{q'}^{t'} \kappa' \vDash v \end{array}}{case\ e\ of\ (x, y) \to e' \rightrightarrows \kappa \Downarrow_{q*q'}^{t \cdot t'} \kappa' \vDash v}$$

$$\textbf{N-CasePair-U}\ \frac{\begin{array}{c} e \rightrightarrows \kappa \Downarrow_q^t \kappa_a \vDash u \\ (\kappa_b, [u_1, u_2]) = fresh\ \kappa_a\ [\overline{T}_1, \overline{T}_2] \\ \kappa_c = unify\ \kappa_b\ (u_1, u_2)\ u \\ e'[u_1/x,\ u_2/y] \rightrightarrows \kappa_c \Downarrow_{q'}^{t'} \kappa' \vDash v \end{array}}{case\ e^{\overline{T}_1 \times \overline{T}_2}\ of\ (x, y) \to e' \rightrightarrows \kappa \Downarrow_{q*q'}^{t \cdot t'} \kappa' \vDash v}$$

$$\textbf{N-L}\ \frac{e \rightrightarrows \kappa \Downarrow_q^t \kappa' \vDash v}{L_{T_1+T_2}\ e \rightrightarrows \kappa \Downarrow_q^t \kappa' \vDash L_{T_1+T_2}\ v}$$

$$\textbf{N-Case-L}\ \frac{\begin{array}{c} e \rightrightarrows \kappa \Downarrow_q^t \kappa_a \vDash L_T\ v_l \\ e_l[v_l/x_l] \rightrightarrows \kappa_a \Downarrow_{q'}^{t'} \kappa' \vDash v \end{array}}{case\ e\ of\ (L\ x_l \to e_l)(R\ x_r \to e_r) \rightrightarrows \kappa \Downarrow_{q*q'}^{t \cdot t'} \kappa' \vDash v}$$

$$\textbf{N-Case-U}\ \frac{\begin{array}{c} e \rightrightarrows \kappa \Downarrow_{q_1}^{t_1} \kappa_a \vDash u \\ (\kappa_0, [u_l, u_r]) = fresh\ \kappa_a\ [\overline{T}_l, \overline{T}_r] \\ \kappa_l = unify\ \kappa_0\ u\ (L_{\overline{T}_l+\overline{T}_r}\ u_l) \qquad \kappa_r = unify\ \kappa_0\ u\ (R_{\overline{T}_l+\overline{T}_r}\ u_r) \\ choose\ 1\ \kappa_l\ 1\ \kappa_r \to_{q_2}^{t_2} i \\ e_i[u_i/x_i] \rightrightarrows \kappa_i \Downarrow_{q_3}^{t_3} \kappa' \vDash v \end{array}}{case\ e^{\overline{T}_l+\overline{T}_r}\ of\ (L\ x_l \to e_l)(R\ x_r \to e_r) \rightrightarrows \kappa \Downarrow_{q_1*q_2*q_3}^{t_1 \cdot t_2 \cdot t_3} \kappa' \vDash v}$$

$$\textbf{N-App}\ \frac{\begin{array}{c} e_0 \rightrightarrows \kappa \Downarrow_{q_0}^{t_0} \kappa_a \vDash (rec\ (f : T_1 \to T_2)\ x = e_2) \\ e_1 \rightrightarrows \kappa_a \Downarrow_{q_1}^{t_1} \kappa_b \vDash v_1 \\ e_2[(rec\ (f : T_1 \to T_2)\ x = e_2)/f,\ v_1/x] \rightrightarrows \kappa_b \Downarrow_{q_2}^{t_2} \kappa' \vDash v \end{array}}{(e_0\ e_1) \rightrightarrows \kappa \Downarrow_{q_0*q_1*q_2}^{t_0 \cdot t_1 \cdot t_2} \kappa' \vDash v}$$

Figure 5. Narrowing Semantics of Standard Core Luck Constructs

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 κ is well typed and that 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 κ 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 (see §4). The finiteness invariant ensures the output of *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 §3.4. Finally, we assume that the type of every expression has been determined by an initial type-checking phase. We write e^T to show that e has type T. This information is used in the semantic rules to provide types for fresh unknowns.

The narrowing semantics is given in Fig. 5 for the standard constructs (omitting *fold/unfold* and **N-R** and **N-Case-R** rules analogous to the **N-L** and **N-Case-L** rules shown) and in Fig. 6 for instantiation expressions; Fig. 8 and Fig. 7 give some auxiliary definitions. Most of the rules are intuitive. A common pattern is sequencing two narrowing judgments $e_1 \rightrightarrows \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v$ and $e_2 \rightrightarrows \kappa_1 \Downarrow_{q_2}^{t_2} \kappa_2 \vDash v$. The constraint-set result of the first narrowing judgment (κ_1) is given as input to the second, while traces and probabilities are accumulated by concatenation ($t_1 \cdot t_2$) and multiplication ($q_1 * q_2$). We now explain the rules in detail.

$$\textbf{N-After}\ \frac{e_1 \rightrightarrows \kappa \Downarrow_{q_1}^{t_1} \kappa_1 \vDash v_1 \qquad e_2 \rightrightarrows \kappa_1 \Downarrow_{q_2}^{t_2} \kappa_2 \vDash v_2}{e_1\ ;\ e_2 \rightrightarrows \kappa \Downarrow_{q_1*q_2}^{t_1 \cdot t_2} \kappa_2 \vDash v_1}$$

$$\textbf{N-Bang}\ \frac{e \rightrightarrows \kappa \Downarrow_q^t \kappa_a \vDash v \qquad sampleV\ \kappa_a\ v \Rightarrow_{q'}^{t'} \kappa'}{!e \rightrightarrows \kappa \Downarrow_{q*q'}^{t \cdot t'} \kappa' \vDash v}$$

$$\textbf{N-Narrow}\ \frac{\begin{array}{c} e \rightrightarrows \kappa \Downarrow_q^t \kappa_a \vDash v \\ e_1 \rightrightarrows \kappa_a \Downarrow_{q_1}^{t_1} \kappa_b \vDash v_1 \qquad e_2 \rightrightarrows \kappa_b \Downarrow_{q_2}^{t_2} \kappa_c \vDash v_2 \\ sampleV\ \kappa_c\ v_1 \Rightarrow_{q_1'}^{t_1'} \kappa_d \qquad sampleV\ \kappa_d\ v_2 \Rightarrow_{q_2'}^{t_2'} \kappa_e \\ nat_{\kappa_e}(v_1) = n_1 \qquad n_1 > 0 \qquad nat_{\kappa_e}(v_2) = n_2 \qquad n_2 > 0 \\ (\kappa_0, [u_1, u_2]) = fresh\ \kappa_e\ [\overline{T}_1, \overline{T}_2] \\ \kappa_l = unify\ \kappa_0\ v\ (L_{\overline{T}_1+\overline{T}_2}\ u_1) \qquad \kappa_r = unify\ \kappa_0\ v\ (R_{\overline{T}_1+\overline{T}_2}\ u_2) \\ choose\ n_1\ \kappa_l\ n_2\ \kappa_r \to_{q'}^{t'} i \end{array}}{e^{\overline{T}_1+\overline{T}_2} \leftarrow (e_1^{nat}, e_2^{nat}) \rightrightarrows \kappa \Downarrow_{q*q_1*q_2*q_1'*q_2'*q'}^{t \cdot t_1 \cdot t_2 \cdot t_1' \cdot t_2' \cdot t'} \kappa_i \vDash v}$$

Figure 6. Narrowing Semantics for Non-Standard Expressions

$$\frac{SAT(\kappa_1) \qquad SAT(\kappa_2)}{choose\ n\ \kappa_1\ m\ \kappa_2 \to_{n/(n+m)}^{[(0,2)]} l} \qquad \frac{\neg SAT(\kappa_1) \qquad SAT(\kappa_2)}{choose\ n\ \kappa_1\ m\ \kappa_2 \to_1^{\epsilon} r}$$

$$\frac{SAT(\kappa_1) \qquad SAT(\kappa_2)}{choose\ n\ \kappa_1\ m\ \kappa_2 \to_{m/(n+m)}^{[(1,2)]} r} \qquad \frac{SAT(\kappa_1) \qquad \neg SAT(\kappa_2)}{choose\ n\ \kappa_1\ m\ \kappa_2 \to_1^{\epsilon} l}$$

Figure 7. Auxiliary relation *choose*

$$\frac{sample\ \kappa\ u = S \qquad S[m] = \kappa'}{sampleV\ \kappa\ u \Rightarrow_{1/|S|}^{[(m,|S|)]} \kappa'}$$

$$\frac{}{sampleV\ \kappa\ () \Rightarrow_1^{\epsilon} \kappa} \qquad \frac{sampleV\ \kappa\ v \Rightarrow_q^t \kappa'}{sampleV\ \kappa\ (fold_T\ v) \Rightarrow_q^t \kappa'}$$

$$\frac{sampleV\ \kappa\ v \Rightarrow_q^t \kappa'}{sampleV\ \kappa\ (L_T\ v) \Rightarrow_q^t \kappa'} \qquad \frac{sampleV\ \kappa\ v \Rightarrow_q^t \kappa'}{sampleV\ \kappa\ (R_T\ v) \Rightarrow_q^t \kappa'}$$

$$\frac{sampleV\ \kappa\ v_1 \Rightarrow_{q_1}^{t_1} \kappa_1 \qquad sampleV\ \kappa_1\ v_2 \Rightarrow_{q_2}^{t_2} \kappa'}{sampleV\ \kappa\ (v_1, v_2) \Rightarrow_{q_1*q_2}^{t_1 \cdot t_2} \kappa'}$$

Figure 8. Auxiliary relation $sampleV$

Rule **N-Base** is the base case of the evaluation relation, handling values that are not handled by other rules by returning them as-is. No choices are made, so the probability of the result is 1 and the trace is empty.

Rule **N-Pair**: To evaluate (e_1, e_2) given a constraint set κ, we sequence the derivations for e_1 and e_2.

Rules **N-CasePair-P**, **N-CasePair-U**: To evaluate the pair elimination expression $case\ e\ of\ (x, y) \to e'$ in a constraint set κ, we first evaluate e in κ. Typing ensures that the resulting value is either

a pair or an unknown. If it is a pair (**N-CasePair-P**), we substitute its components for x and y in e' and continue evaluating. If it is an unknown u of type $\overline{T}_1 \times \overline{T}_2$ (**N-CasePair-U**), we first use $\overline{T}_1$ and $\overline{T}_2$ as types for fresh unknowns u_1, u_2 and remember the constraint that the pair (u_1, u_2) must unify with u. We then proceed as above, this time substituting u_1 and u_2 for x and y.

(The first pair rule might appear unnecessary since, even in the case where the scrutinee evaluates to a pair, we could generate unknowns, unify, and substitute, as in **N-CasePair-U**. However, unknowns in Luck only range over non-functional types $\overline{T}$, so this trick does not work when the type of the e contains arrows.)

The **N-CasePair-U** rule also shows how the finiteness invariant is preserved: when we generate the unknowns u_1 and u_2, their domains are unconstrained, but before we substitute them into an expression used as "input" to a subderivation, we unify them with the result of a narrowing derivation, which already has a finite representation in κ_a.

Rule **N-L**: To evaluate $L_{T_1+T_2}\ e$, we evaluate e and tag the resulting value with $L_{T_1+T_2}$, with the resulting constraint set, trace, and probability unchanged. $R_{T_1+T_2}\ e$ is handled similarly (the rule is elided) .

Rules **N-Case-L**,**N-Case-U**: As in the pair elimination rule, we first evaluate the discriminee e to a value, which must have one of the shapes $L_T\ v_l$, $R_T\ v_r$, or $u \in \mathcal{U}$, thanks to typing. The cases for $L_T\ v_l$ (rule **N-Case-L**) and $R_T\ v_r$ (elided) are similar to **N-CasePair-P**: v_l or v_r can be directly substituted for x_l or x_r in e_l or e_r. The unknown case (**N-Case-U**) is similar to **N-CasePair-U** but a bit more complex. Once again e shares with the unknown u a type $\overline{T}_l + \overline{T}_r$ that does not contain any arrows, so we can generate fresh unknowns u_l, u_r with types $\overline{T}_l$, $\overline{T}_r$. We unify $L_{\overline{T}_l+\overline{T}_r}\ v_l$ with u to get the constraint set κ_l and $R_{\overline{T}_l+\overline{T}_r}\ v_r$ with u to get κ_r. We then use the auxiliary relation $choose$ (Fig. 7), which takes two integers n and m (here equal to 1) as well as two constraint sets (here κ_l and κ_r), to select either l or r. If exactly one of κ_l and κ_r is satisfiable, then $choose$ will return the corresponding index with probability 1 and an empty trace (because no random choice were made). If both are satisfiable, then the resulting index is randomly chosen. Both outcomes are equiprobable (because of the 1 arguments to $choose$), so the probability is one half in each case. This uniform binary choice is recorded in the trace t_2 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. The satisfiability checks enforce the invariant that constraint sets are satisfiable, which in turn ensures that κ_l and κ_r cannot both be unsatisfiable at the same time, since there must exist at least one valuation in κ_0 that maps u to a value (either L or R) which ensures that the corresponding unification will succeed.

Rule **N-App**: To evaluate an application $(e_0\ e_1)$, we first evaluate e_0 to $rec\ (f : T_1 \to T_2)\ x = e_2$ (since unknowns only range over arrow-free types $\overline{T}$, the result cannot be an unknown) and its argument e_1 to a value v_1. We then evaluate the appropriately substituted body, $e_2[(rec\ (f : T_1 \to T_2)\ x = e_2)/f, v_1/x]$, and combine the various probabilities and traces appropriately.

Rule **N-After** is similar to **N-Pair**; however, the value result of the derivation is that of the first narrowing evaluation, implementing the reverse form of sequencing described in the introduction of this section.

Rule **N-Bang**: To evaluate $!e$ we evaluate e to a value v, then use the auxiliary relation $sample V$ (Fig. 8) to completely instantiate v, walking down the structure of v. When unknowns are encountered, $sample$ is used to produce a list of constraint sets S; with probability $\frac{1}{|S|}$ (where $|S|$ is the size of the list) we can select the mth constraint set in S, for each $0 \leqslant m < |S|$.

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 $sample V$ is called to ensure that they are fully instantiated in all subsequent constraint sets, in particular in κ_e. The relation $nat_{\kappa_e}(v_1) = n_1$ walks down the structure of the value v_1 (like $sample V$) and calculates the unique natural number n_1 corresponding to v_1. Specifically, when the input value is an unknown, $nat_\kappa(u) = n$ holds if $\kappa[u] = v'$ and $[\![v]\!] = n$, where the notation $[\![v]\!]$ is defined in Fig. 4. 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 §3.1: 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 improve performance [17].

Unfortunately, using the narrowing semantics in this way can lead to a lot of backtracking. To see why, consider three unknowns, u_1, u_2, and u_3, and a constraint set κ where each unknown has type `Bool` (i.e., $1 + 1$) and the domain associated with each 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 is shorthand for $case\ e_1\ of\ (L\ x \to e_2)(R\ y \to \texttt{False})$. If we attempt to evaluate the expression u_1 && u_2 && u_3 using the narrowing semantics, we first apply the **N-Case-U** rule with $e = u_1$. That means that u_1 will be unified with either L or R (applied to a fresh unknown) with equal probability, leading to a `False` result for the entire expression 50% of the time. If we choose to unify u_1 with an L, then we apply the **N-Case-U** rule again, returning either `False` or u_3 (since unknowns are values—rule **N-Base**) with equal probability. Therefore, we will have generated a desired valuation only 25% of the time; we will need to backtrack 75% of the time.

The problem here is that the narrowing semantics is agnostic to the desired result of the whole computation—we only find out at the very end that we need to backtrack. But we can do better...

3.4 Matching Semantics

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) and propagates this pattern backwards to guide the generation process. By allowing our semantics to look ahead in this way, we can often avoid case branches that lead to non-matching results.

The matching judgment is again a variant of big-step evaluation; it has the form

$$p \Leftarrow e \Rightarrow \kappa \Uparrow_q^t \kappa^?$$

where p can mention the unknowns in $U(\kappa)$ and where the metavariable $\kappa^?$ stands for an *optional* constraint set ($\varnothing$ 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 κ is well typed and so is the input expression e (with respect to an empty variable context and $U(\kappa)$); moreover κ is satisfiable, and the restriction of its denotation to the unknowns in e 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).

$$\textbf{M-Base}\ \frac{v = () \ \vee \ v \in \mathcal{U} \qquad \kappa' = \mathit{unify}\ \kappa\ v\ p}{p \Leftarrow v \rightrightarrows \kappa \Uparrow_1^{\epsilon}\ \mathit{if}\ \mathit{SAT}(\kappa')\ \mathit{then}\ \{\kappa'\}\ \mathit{else}\ \varnothing}$$

$$\textbf{M-Pair}\ \frac{\begin{array}{c}(\kappa', [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ \kappa_0 = \mathit{unify}\ \kappa'\ (u_1, u_2)\ p \\ u_1 \Leftarrow e_1 \rightrightarrows \kappa_0 \Uparrow_{q_1}^{t_1}\ \{\kappa_1\} \qquad u_2 \Leftarrow e_2 \rightrightarrows \kappa_1 \Uparrow_{q_2}^{t_2}\ \kappa_2^?\end{array}}{p \Leftarrow (e_1^{\overline{T}_1}, e_2^{\overline{T}_2}) \rightrightarrows \kappa \Uparrow_{q_1 * q_2}^{t_1 \cdot t_2}\ \kappa_2^?}$$

$$\textbf{M-Pair-Fail}\ \frac{\begin{array}{c}(\kappa', [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ \kappa_0 = \mathit{unify}\ \kappa'\ (u_1, u_2)\ p \\ u_1 \Leftarrow e_1 \rightrightarrows \kappa_0 \Uparrow_{q_1}^{t_1}\ \varnothing\end{array}}{p \Leftarrow (e_1^{\overline{T}_1}, e_2^{\overline{T}_2}) \rightrightarrows \kappa \Uparrow_{q_1}^{t_1}\ \varnothing}$$

$$\textbf{M-App}\ \frac{\begin{array}{c}e_0 \rightrightarrows \kappa \Downarrow_{q_0}^{t_0}\ \kappa_0 \vDash (\mathit{rec}\ f\ x = e_2) \\ e_1 \rightrightarrows \kappa_0 \Downarrow_{q_1}^{t_1}\ \kappa' \vDash v_1 \\ p \Leftarrow e_2[(\mathit{rec}\ f\ x = e_2)/f, v_1/x] \rightrightarrows \kappa' \Uparrow_{q_2}^{t_2}\ \kappa^?\end{array}}{p \Leftarrow (e_0\ e_1) \rightrightarrows \kappa \Uparrow_{q_0 * q_1 * q_2}^{t_0 \cdot t_1 \cdot t_2}\ \kappa^?}$$

$$\textbf{M-After}\ \frac{p \Leftarrow e_1 \rightrightarrows \kappa \Uparrow_{q_1}^{t_1}\ \{\kappa_1\} \qquad e_2 \rightrightarrows \kappa_1 \Downarrow_{q_2}^{t_2}\ \kappa_2 \vDash v}{p \Leftarrow e_1\ ;\ e_2 \rightrightarrows \kappa \Uparrow_{q_1 * q_2}^{t_1 \cdot t_2}\ \{\kappa_2\}}$$

Figure 9. Matching Semantics of Selected Core Luck Constructs

$$\textbf{M-Case-1}\ \frac{\begin{array}{c}(\kappa_0, [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ (L_{\overline{T}_1 + \overline{T}_2}\ u_1) \Leftarrow e \rightrightarrows \kappa_0 \Uparrow_{q_1}^{t_1}\ \{\kappa_1\} \\ (R_{\overline{T}_1 + \overline{T}_2}\ u_2) \Leftarrow e \rightrightarrows \kappa_0 \Uparrow_{q_2}^{t_2}\ \{\kappa_2\} \\ p \Leftarrow e_1[u_1/x_l] \rightrightarrows \kappa_1 \Uparrow_{q_1'}^{t_1'}\ \kappa_a^? \qquad p \Leftarrow e_2[u_2/y_r] \rightrightarrows \kappa_2 \Uparrow_{q_2'}^{t_2'}\ \kappa_b^? \\ \kappa^? = \mathit{combine}\ \kappa_0\ \kappa_a^?\ \kappa_b^?\end{array}}{\begin{array}{c}p \Leftarrow \mathit{case}\ e^{\overline{T}_1 + \overline{T}_2}\ \mathit{of}\ (L\ x_l \to e_1)(R\ y_r \to e_2) \rightrightarrows \kappa \\ \Uparrow_{q_1 * q_2 * q_1' * q_2'}^{t_1 \cdot t_2 \cdot t_1' \cdot t_2'}\ \kappa^?\end{array}}$$

where
$$\begin{array}{l}\mathit{combine}\ \kappa\ \varnothing\ \varnothing = \varnothing \\ \mathit{combine}\ \kappa\ \{\kappa_1\}\ \varnothing = \{\kappa_1\} \\ \mathit{combine}\ \kappa\ \varnothing\ \{\kappa_2\} = \{\kappa_2\} \\ \mathit{combine}\ \kappa\ \{\kappa_1\}\ \{\kappa_2\} = \\ \qquad \{\mathit{union}\ \kappa_1\ (\mathit{rename}\ (U(\kappa_1)\text{-}U(\kappa))\ \kappa_2)\}\end{array}$$

$$\textbf{M-Case-2}\ \frac{\begin{array}{c}(\kappa_0, [u_1, u_2]) = \mathit{fresh}\ \kappa\ [\overline{T}_1, \overline{T}_2] \\ (L_{\overline{T}_1 + \overline{T}_2}\ u_1) \Leftarrow e \rightrightarrows \kappa_0 \Uparrow_{q_1}^{t_1}\ \varnothing \\ (R_{\overline{T}_1 + \overline{T}_2}\ u_2) \Leftarrow e \rightrightarrows \kappa_0 \Uparrow_{q_2}^{t_2}\ \{\kappa_2\} \\ p \Leftarrow e_2[u_2/y] \rightrightarrows \kappa_2 \Uparrow_{q_2'}^{t_2'}\ \kappa_b^?\end{array}}{p \Leftarrow \mathit{case}\ e^{\overline{T}_1 + \overline{T}_2}\ \mathit{of}\ (L\ x \to e_1)(R\ y \to e_2) \rightrightarrows \kappa \Uparrow_{q_1 * q_2 * q_2'}^{t_1 \cdot t_2 \cdot t_2'}\ \kappa_b^?}$$

Figure 10. Matching Semantics for Constraint-Solving *case*

The rules except for *case* are similar to the narrowing semantics. Fig. 9 shows several; 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 κ. Unlike **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 $\overline{T}_1$ and $\overline{T}_2$, 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 κ'. We then proceed as in **N-Pair**, evaluating e_1 against pattern u_1 and e_2 against u_2, threading constraint sets and accumulating traces and probabilities. **M-Pair** handles the case where the evaluation of e_1 succeeds, while **M-Pair-Fail** handles failure: if evaluating e_1 yields $\varnothing$, the whole computation immediately yields $\varnothing$ 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 $\mathit{rec}\ f\ x = e_2$ and e_1 to a value v_1, then evaluate $e_2[(\mathit{rec}\ f\ x = e_2)/f, v_2/x]$ 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 as a primitive $e_1\ ;\ e_2$ instead of encoding it using lambda abstractions. 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 generation that occurs during 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:

$$\begin{array}{lll}\mathit{rename} & :: & \mathcal{U}^* \to \mathcal{C} \to \mathcal{C} \\ \mathit{union} & :: & \mathcal{C} \to \mathcal{C} \to \mathcal{C}\end{array}$$

The *rename* operation freshens a constraint set by replacing all the unknowns in a given sequence with freshly generated ones. The *union* of two constraint sets intuitively denotes the union of their corresponding denotations.

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 branch bodies and combine the results with *union*. We use *rename* to avoid conflicts, since we may generate the same fresh unknowns while independently computing $\kappa_a^?$ and $\kappa_b^?$.

If desired, the user can ensure that only one branch will be executed by using an instantiation 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 κ 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. To see this, consider a simple recursive predicate that holds for every number:

$$\begin{array}{l}\mathit{rec}\ (f : \mathit{nat} \to \mathit{bool})\ u = \\ \quad \mathit{case}\ \mathit{unfold}_{\mathit{nat}}\ u\ \mathit{of}\ (L\ x \to \texttt{True})(R\ y \to (f\ y))\end{array}$$

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. While this finiteness restriction feels a bit unnatural, we have not found it to be a problem in practice—see §4.

3.5 Example

To show how all this works, let's trace the main steps of the matching derivations of two given expressions against the pattern `True` in a given constraint set. We will also extract probability distributions about optional constraint sets from these derivations.

We are going to evaluate $A := (0 < u \;\&\&\; u < 4)\;;\; !u$ and $B := (0 < u\;;\; !u) \;\&\&\; u < 4$ against the pattern `True` in a constraint set κ, in which u is independent from other unknowns and its possible values are $0, ..., 9$. Similar expressions were introduced as examples in §2; the results we obtain here confirm the intuitive explanation given there.

Recall that the conjunction expression e_1 && e_2 is shorthand for $case\ e_1\ of\ (L\ a \to e_2)(R\ b \to \texttt{False})$, and that we are using a standard Peano encoding of naturals: $nat = \mu X.\ 1 + X$. We elide folds for brevity. The inequality $a < b$ can be encoded as $lt\ a\ b$, where:

$$lt = rec\ (f : nat \to nat \to bool)\ x = rec\ (g : nat \to bool)\ y = \\ case\ y\ of \quad (L\ \_ \to \texttt{False}) \\ (R\ y_R \to case\ x\ of \quad (L\ \_ \to \texttt{True}) \\ (R\ x_R \to f\ x_R\ y_R))$$

Many rules introduce fresh unknowns, many of which are irrelevant: they might be directly equivalent to some other unknown, or there might not exist any reference to them. We abusively use the same variable for two constraint sets which differ only in the addition of a few irrelevant variables to one of them.

Evaluation of A We first derive $\texttt{True} \Leftarrow (0 < u) \rightleftharpoons \kappa \Uparrow_1^{\epsilon} \{\kappa_0\}$. Since in the desugaring of $0 < u$ as an application lt is already in rec form and both 0 and u are values, the constraint set after the narrowing calls of **M-App** will stay unchanged. We then evaluate $case\ u\ of\ (L\ \_ \to \texttt{False})(R\ y_R \to ...)$. Since the domain of u contains both zero and non-zero elements, unifying u with $L_{1+nat}\ u_1$ and $R_{1+nat}\ u_2$ (**M-Base**) will produce some non-empty constraint sets. Therefore, rule **M-Case-1** applies. Since the body of the left hand side of the match is `False`, the result of the left derivation in **M-Case-1** is $\varnothing$ and in the resulting constraint set κ_0 the domain of u is $\{1, ..., 9\}$.

Next, we turn to $\texttt{True} \Leftarrow (0 < u \;\&\&\; u < 4) \rightleftharpoons \kappa \Uparrow_1^{\epsilon} \{\kappa_1\}$, where, by a similar argument following the recursion, the domain of u in κ_1 is $\{1, 2, 3\}$. There are 3 possible narrowing-semantics derivations for $!u$: (1) $!u \rightleftharpoons \kappa_1 \Downarrow_{1/3}^{[(0,3)]} \kappa_1^A \vDash u$, (2) $!u \rightleftharpoons \kappa_1 \Downarrow_{1/3}^{[(1,3)]} \kappa_2^A \vDash u$, and (3) $!u \rightleftharpoons \kappa_1 \Downarrow_{1/3}^{[(2,3)]} \kappa_3^A \vDash u$, where the domain of u in κ_i^A is $\{i\}$. (We have switched to narrowing-semantics judgments because of the rule **M-After**.) Therefore all the possible derivations for $A = (0 < u \;\&\&\; u < 4)\;;\; !u$ matching `True` in κ are:

$$\texttt{True} \Leftarrow A \rightleftharpoons \kappa \Uparrow_{1/3}^{[(i-1,3)]} \{\kappa_i^A\} \qquad \text{for } i \in \{1, 2, 3\}$$

From the set of possible derivations, we can extract a probability distribution: for each resulting optional constraint set, we sum the probabilities of each of the traces that lead to this result. Thus the probability distribution associated with $\texttt{True} \Leftarrow A \rightleftharpoons \kappa$ is

$$[\{\kappa_1^A\} \mapsto \frac{1}{3};\quad \{\kappa_2^A\} \mapsto \frac{1}{3};\quad \{\kappa_3^A\} \mapsto \frac{1}{3}].$$

Evaluation of B The evaluation of $0 < u$ is the same as before, after which we narrow $!u$ directly in κ_0 and there are 9 possibilities: $!u \rightleftharpoons \kappa_0 \Downarrow_{1/9}^{[(i-1,9)]} \kappa_i^B \vDash u$ for each $i \in \{1, ..., 9\}$, where the domain of u in κ_i^B is $\{i\}$. Then we evaluate $\texttt{True} \Leftarrow u < 4 \rightleftharpoons \kappa_i^B$: if i is 1, 2 or 3 this yields $\{\kappa_i^B\}$; if $i > 3$ this yields a failure $\varnothing$. Therefore the possible derivations for $B = (0 < u\;;\; !u) \;\&\&\; u < 4$ are:

$$\texttt{True} \Leftarrow B \rightleftharpoons \kappa \Uparrow_{1/9}^{[(i-1,9)]} \{\kappa_i^B\} \qquad \text{for } i \in \{1, 2, 3\}$$

$$\texttt{True} \Leftarrow B \rightleftharpoons \kappa \Uparrow_{1/9}^{[(i-1,9)]} \varnothing \qquad \text{for } i \in \{4, ..., 9\}$$

We can again compute the corresponding probability distribution:

$$[\{\kappa_1^B\} \mapsto \frac{1}{9};\quad \{\kappa_2^B\} \mapsto \frac{1}{9};\quad \{\kappa_3^B\} \mapsto \frac{1}{9};\quad \varnothing \mapsto \frac{2}{3}]$$

Note that if we were just recording the probability of an execution and not its trace, we would not know that there are six distinct executions leading to $\varnothing$ with probability $\frac{1}{9}$, so we would not be able to compute its total probability.

The probability associated with $\varnothing$ (0 for A, $2/3$ for B) is the probability of backtracking. As stressed in §2, A is much better than B in terms of backtracking—i.e., it is more efficient in this case to instantiate u only after all the constraints on its domain have been recorded. For a more formal treatment of backtracking strategies in Luck using Markov Chains, see [26].

3.6 Properties

We close our discussion of Core Luck by summarizing some key properties; more details and proofs can be found in the extended version. Intuitively, we show that, when we evaluate an expression e against a pattern p in the presence of a constraint set κ, we can only remove valuations from the denotation of κ (*decreasingness*), any derivation in the generator semantics corresponds to an execution in the predicate semantics (*soundness*), and every valuation that matches p will be found in the denotation of the resulting constraint set of some derivation (*completeness*).

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; for these, we have only paper proofs, but these proofs are quite similar to the narrowing ones (details are in the extended version; the only real difference is the case rule).

We begin by giving the formal specification of constraint sets. We introduce one extra abstraction, the *domain* of a constraint set κ, written $dom(\kappa)$. This domain corresponds to the unknowns in a constraint set that actually have bindings in $[\![\kappa]\!]$. For example, when we generate a fresh unknown u from κ, u does not appear in the domain of κ; it only appears in the denotation after we use it in a unification. The domain of κ is a subset of the set of keys of $U(\kappa)$.

When we write that for a valuation and constraint set $\sigma \in [\![\kappa]\!]$, it also implies that the unknowns that have bindings in σ are exactly the unknowns that have bindings in $[\![\kappa]\!]$, i.e., in $dom(\kappa)$. We use the overloaded notation $\sigma|_x$ to denote the restriction of σ to x, where x is either a set of unknowns or another valuation.

Specification of fresh

$$(\kappa', u) = fresh\ \kappa\ T \Rightarrow \begin{cases} u \notin U(\kappa) \\ U(\kappa') = U(\kappa) \oplus (u \mapsto T) \\ [\![\kappa']\!] = [\![\kappa]\!] \end{cases}$$

Intuitively, when we generate a fresh unknown u of type T from κ, u is really fresh for κ, meaning $U(\kappa)$ does not have a type binding for it. The resulting constraint set κ' has an extended unknown typing map, where u maps to T and its denotation remains unchanged. That means that $dom(\kappa') = dom(\kappa)$.

Specification of sample

$$\kappa' \in sample\ \kappa\ u \Rightarrow \begin{cases} U(\kappa') = U(\kappa) \\ SAT(\kappa') \\ \exists v.\ [\![\kappa']\!] = \{\, \sigma \mid \sigma \in [\![\kappa]\!],\ \sigma(u) = v \,\} \end{cases}$$

When we sample u in a constraint set κ and obtain a list, for every member constraint set κ', the typing map of κ remains unchanged and all of the valuations that remain in the denotation of κ' are the ones that mapped to some specific value v in κ. Clearly, the domain of κ remains unchanged. We also require a completeness property from *sample*, namely that if we have a valuation $\sigma \in [\![\kappa]\!]$ where $\sigma(u) = v$ for some u, v, then σ is in some member κ' of the result:

$$\left.\begin{array}{l}\sigma(u) = v \\ \sigma \in [\![\kappa]\!]\end{array}\right\} \Rightarrow \exists \kappa'.\ \left\{\begin{array}{l}\sigma \in [\![\kappa']\!] \\ \kappa' \in \mathit{sample}\ \kappa\ u\end{array}\right.$$

Specification of unify

$$U(\mathit{unify}\ \kappa\ v_1\ v_2) = U(\kappa)$$
$$[\![\mathit{unify}\ \kappa\ v_1\ v_2]\!] = \{\, \sigma \in [\![\kappa]\!] \mid \sigma(v_1) = \sigma(v_2) \,\}$$

When we unify in a constraint set κ two (well-typed for κ) values v_1 and v_2, the typing map remains unchanged while the denotation of the result contains just the valuations from κ that when substituted into v_1 and v_2 make them equal. The domain of κ' is the union of the domain of κ and the unknowns in v_1, v_2.

Properties of the Narrowing Semantics The first theorem, *decreasingness* states that we never add new valuations to our constraint sets; our semantics can only refine the denotation of the input κ.

Theorem 3.6.1 (Decreasingness)**.**

$$e \dashv \kappa \Downarrow^t_q \kappa' \vDash v \ \Rightarrow\ \kappa' \leqslant \kappa$$

Soundness and completeness can be visualized as follows:

$$\begin{array}{ccc} e_p & \xrightarrow{\ \Downarrow\ } & v_p \\ {\scriptstyle \sigma \in [\![\kappa]\!]}\uparrow & & \uparrow{\scriptstyle \sigma' \in [\![\kappa']\!]} \\ e \dashv \kappa & \xrightarrow{\ \Downarrow^t_q\ } & v \vDash \kappa' \end{array}$$

Given the bottom and right sides of the diagram, soundness guarantees that we can fill in the top and left. That is, any narrowing derivation $e \dashv \kappa \Downarrow^q_t \kappa' \vDash v$ directly corresponds to some 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 κ (which can be replaced by a stronger assumption that e is well typed in κ).

Theorem 3.6.2 (Soundness)**.**

$$\left.\begin{array}{l}e \dashv \kappa \Downarrow^q_t \kappa' \vDash v \\ \sigma'(v) = v_p\ \wedge\ \sigma' \in [\![\kappa']\!] \\ \forall u.\ u \in e \Rightarrow u \in \mathit{dom}(\kappa)\end{array}\right\} \Rightarrow\ \exists \sigma\ e_p.\ \left\{\begin{array}{l}\sigma'|_\sigma \equiv \sigma \\ \sigma \in [\![\kappa]\!] \\ \sigma(e) = e_p \\ e_p \Downarrow v_p\end{array}\right.$$

Completeness guarantees the opposite direction: given a predicate derivation $e_p \Downarrow v_p$ and a "factoring" of e_p into an expression e and a constraint set κ such that for some valuation $\sigma \in [\![\kappa]\!]$ substituting σ 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 3.6.3 (Completeness)**.**

$$\left.\begin{array}{l}e_p \Downarrow v_p \\ \sigma(e) = e_p \\ \sigma \in [\![\kappa]\!] \wedge \vdash \kappa \\ \varnothing; U(\kappa) \vdash e : T\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\exists v\ \kappa'\ \sigma'\ q\ t. \\ \quad \sigma'|_\sigma \equiv \sigma\ \ \wedge\ \ \sigma' \in [\![\kappa']\!] \\ \quad \sigma'(v) = v_p \\ \quad e \dashv \kappa \Downarrow^t_q \kappa' \vDash v\end{array}\right.$$

Properties of the Matching Semantics Before we proceed to the theorems for the matching semantics, we need a specification for the *union* operation.

Specification of union

$$\left.\begin{array}{l}U(\kappa_1)|_{U(\kappa_1)\cap U(\kappa_2)} = U(\kappa_2)|_{U(\kappa_1)\cap U(\kappa_2)} \\ \mathit{union}\ \kappa_1\ \kappa_2 = \kappa\end{array}\right\}$$
$$\Rightarrow$$
$$\left\{\begin{array}{l}U(\kappa) = U(\kappa_1) \cup U(\kappa_2) \\ [\![\kappa]\!] = [\![\kappa_1]\!] \cup [\![\kappa_2]\!]\end{array}\right.$$

To take the *union* of two constraint sets, their typing maps must obviously agree on any unknowns present in both. The denotation of the *union* of two constraint sets is then just the union of their corresponding denotations.

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

Theorem 3.6.4 (Decreasingness)**.**

$$p \Leftarrow e \dashv \kappa \Uparrow^t_q \{\kappa'\}\ \Rightarrow\ \kappa' \leqslant \kappa$$

Soundness is again similar to the matching semantics.

Theorem 3.6.5 (Soundness)**.**

$$\left.\begin{array}{l}p \Leftarrow e \dashv \kappa \Uparrow^t_q \{\kappa'\} \\ \sigma'(p) = v_p \wedge \sigma' \in [\![\kappa']\!] \\ \forall u.\ (u \in e \vee u \in p) \Rightarrow u \in \mathit{dom}(\kappa)\end{array}\right\} \Rightarrow \exists \sigma\ e_p.\ \left\{\begin{array}{l}\sigma'|_\sigma \equiv \sigma \\ \sigma \in [\![\kappa]\!] \\ \sigma(e) = e_p \\ e_p \Downarrow v_p\end{array}\right.$$

For the completeness theorem, we need to slightly strengthen its premise; since 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). Thus, we require that *all* valuations in the input constraint set result in a terminating execution.

Theorem 3.6.6 (Completeness)**.**

$$\left.\begin{array}{l}e_p \Downarrow v_p\ \wedge\ \sigma \in [\![\kappa]\!] \\ \varnothing; U(\kappa) \vdash e : \overline{T}\ \wedge\ \vdash \kappa \\ \sigma(e) = e_p \wedge \sigma(p) = v_p \\ \forall \sigma' \in [\![\kappa]\!].\ \exists v'.\ \sigma'(e) \Downarrow v'\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\exists \kappa'\ \sigma'\ q\ t. \\ \quad \sigma'|_\sigma \equiv \sigma \\ \quad \sigma' \in [\![\kappa']\!] \\ \quad p \Leftarrow e \dashv \kappa \Uparrow^t_q \{\kappa'\}\end{array}\right.$$

4. Implementation

We next describe the Luck prototype: its top level, its treatment of backtracking, and the implementation of primitive integers 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 *bool* containing zero or more free unknowns $\vec{u}$ (but no free variables), and an initial constraint set κ providing types and finite domains[2] for each unknown in $\vec{u}$, such that their occurrences in e are well typed ($\varnothing; U(\kappa) \vdash e : 1 + 1$).

The interpreter matches e against `True` (that is, $L_{1+1}\ ()$), to derive a refined constraint set κ':

$$L_{1+1}\ () \Leftarrow e \dashv \kappa \Uparrow^t_q \{\kappa'\}$$

This involves random choices, and there is also the possibility that matching fails (and the semantics generates $\varnothing$ instead of $\{\kappa'\}$).

[2] This restriction to finite domains appears to be crucial for our technical development to work, as discussed in the previous section. In practice, we have not yet encountered a situation where it was important to be able to generate examples of *unbounded* size (as opposed to examples up to some large maximum size). We do sometimes want to generate structures containing large numbers, since they can be represented efficiently, but here, too, choosing an enormous finite bound appears to be adequate for the applications we've tried. The implementation allows for representing all possible ranges of a corresponding type up to a given size bound. Such bounds are initialized at the top level, and they are propagated (and reduced a bit) to fresh unknowns created by pattern matching before these unknowns are used as inputs to the interpreter.

In this case, a simple *global* backtracking approach could simply try the whole thing again (up to an ad hoc limit). While not strictly necessary for a correct implementation of the matching semantics, some *local* backtracking allows wrong choices to be reversed quickly and leads to an enormous improvement in performance [18]. Our prototype backtracks locally in calls to $choose$: if *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 $L\_\ \_$, then narrow will not choose to instantiate it using $R\_\ \_$, and vice versa. This may require matching against e 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.)

After the interpreter matches e against `True`, all the resulting valuations $\sigma \in [\![\kappa']\!]$ should map the unknowns in $\vec{u}$ to some values. However, there is no guarantee that the generator semantics will yield a κ' mapping every $\vec{u}$ to a unique values. The Luck top-level then applies the $sample$ constraint set function to each unknown in $\vec{u}$, ensuring that $\sigma|_{\vec{u}}$ is the same for each σ in the final constraint set. The interpreter returns this common $\sigma|_{\vec{u}}$ if it exists, and backtracks otherwise.

Pattern Match Compiler In Section 2, we saw an example using a standard `Tree` datatype and instantiation expressions assigning different weights to each branch. While the desugaring of simple pattern matching to core Luck syntax is straightforward (3.1), nested patterns—as in Fig. 11—complicate things in the presence of probabilities. We expand such expressions to a tree of simple case expressions that match only the outermost constructors of their scrutinees. However, there is generally no unique choice of weights in the expanded predicate: a branch from the source predicate may be duplicated in the result. We guarantee the intuitive property that the *sum* of the probabilities of the clones of a branch is proportional to the weights given by the user, but that still does not determine the individual probabilities that should be assigned to these clones.

The most obvious way to distribute weights is to simply share the weight equally with all duplicated branches. But the probability of a single branch then depends on the total number of expanded branches that come from the same source, which can be hard for users to determine and can vary widely even between sets of patterns that appear similar. Instead, Luck's default weighing strategy works as follows. For any branch B from the source, at any intermediate case expression of the expansion, the subprobability distribution over the immediate subtrees that contain at least one branch derived from B is uniform. This makes modifications of the source patterns in nested positions affect the distribution more locally.

In Figure 11, the `False` branch should have probability $\frac{1}{3}$. It is expanded into four branches, corresponding to subpatterns `Var _`, `Lam _ _`, `App (Var _) _`, `App (App _ _) _`. The latter two are grouped under the pattern `App _ _`, while the former two are in their own groups. These three groups receive equal shares of the total probability of the original branch, that is $\frac{1}{9}$ each. The two branches for `App (Var _) _` and `App (App _ _) _` split that further into twice $\frac{1}{18}$. On the other hand, `True` remains a single branch with probability $\frac{2}{3}$. The weights on the left of every pattern are calculated to reflect this distribution.

Constraint Set Implementation Our desugaring of source-level pattern matching to core case expressions whose discriminee e is first narrowed means that rule **M-Case-1** is not executed for datatypes; only one of the evaluations of e against the L and R patterns will succeed and only one branch will be executed. This means that our constraint-set representation for datatypes doesn't need to implement $union$. We leverage this to provide a simple and efficient implementation of the unification constraints. For our prototype, the constraint solving behavior of $case$ is only exploited in our treatment of primitive integers, which we detail at the end of this section.

```
data T = Var Int | Lam Int T | App T T

sig isRedex :: T → Bool          -- Original
fun isRedex t =
  case t of
  | 2 % App (Lam _ _) _ → True -- 2/3
  | 1 % _ → False               -- 1/3

sig isRedex :: T → Bool          -- Expansion
fun isRedex t =
  case t of
  | 1 % Var _ → False           -- 1/9
  | 1 % Lam _ _ → False         -- 1/9
  | 7 % App t1 _ →
      case t1 of
      | 1 % Var _ → False       -- 1/18
      | 12 % Lam _ _ → True     -- 2/3
      | 1 % App _ _ → False     -- 1/18
```

Figure 11. Expanding case expression with a nested pattern and a wildcard. Comments show the probability of each alternative.

The constraint set interface could be implemented in a variety of different ways. The simplest would be to explicitly represent constraint sets as sets of valuations, but this would lead to 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 directly support the per-variable sampling that we require.

For our prototype we choose a middle way, using a simple data structure we call *orthogonal maps* to represent sets of valuations. An orthogonal map is a map from unknowns to *ranges*, which have the following syntax:

$$r ::= () \mid u \mid (r, r) \mid \mathit{fold}\ r \mid L\ r \mid R\ r \mid \{L\ r, R\ r\}$$

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 (eliding folds and type information) as $\{L(), R()\}$. Similarly, the set $\{0, 2, 3\}$ can be encoded as $\{L(), R(R\{L(), R()\})\}$, assuming a standard Peano encoding of naturals.

However, while this compact representation can represent all sets of naturals, not all sets of Luck non-functional values can be precisely represented. For instance the set $\{(0, 1), (1, 0)\}$ cannot be represented using ranges, only approximated to $(\{L(), R(L())\}$, $\{L(), R(L())\})$, 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 [41, 43, 51].

We implement constraint sets as pairs of a typing environment and an optional map from unknowns to ranges. The typing environment of a constraint set ($U(\cdot)$ operation), is just the first projection of the tuple. A constraint set κ is SAT if the second element is not $\varnothing$. The $sample$ primitive indexes into the map and collects all possible values for an unknown.

The only interesting operation with this representation is *unify*. It 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 κ returns both a constraint set κ' 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 κ to find the range r_u corresponding to u, unify r_u with r_2 in κ to obtain a range r', and then map u to r' in the resulting constraint set κ'. 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 $\{L\ r_{1l}, R\ r_{1r}\}$ while the other is $L\ r_2$, the resulting range is only L applied to the result of the unification of r_{1l} and r_2.

It is easy to see that if we start with a set of valuations that is representable as an orthogonal map, non-*union* operations will result in constraint sets whose denotation is still representable, which allows us to get away with this simple implementation of datatypes. The **M-Case-1** rule is used to model our treatement of integers. We introduce primitive integers in our prototype accompanied by standard integer equality and inequality constraints. In Section 3.5 we saw how a recursive less-than function can be encoded using Peano-style integers and *case* expressions that do *not* contain instantiation expressions in the discriminee. All integer constraints can be desugared into such recursive functions with the exact same behavior—modulo efficiency.

To implement integer constraints, we extend the codomain of the mapping in the constraint set implementation described above to include a compact representation of sets of intervals of primitive integers as well as a set of the unknown's associated constraints. Every time the domain of an unknown u is refined, we use an incremental variant of the AC-3 *arc consistency* algorithm [47] to efficiently refine the domains of all the unknowns linked to u, first iterating through the constrains associated with u and then only through the constraints of other "affected" unknowns.

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× to 24× for the more complex generators. (Also, 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—list predicates such as `sorted`, `member`, and `distinct`, tree predicates like BSTs (§2) and red-black trees, and so on. In the extended version we show the implementation of many such 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 [63]. Lazy SmallCheck leverages Haskell's 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

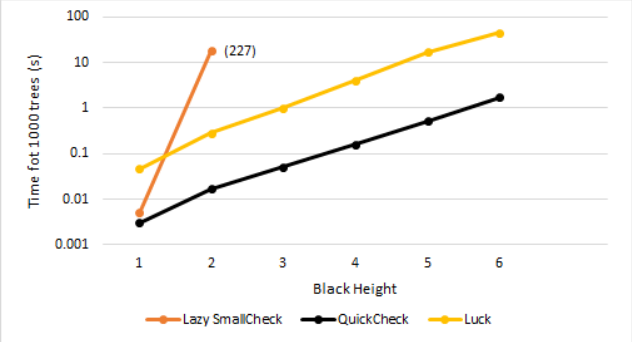

Figure 12. Red-Black Tree Experiment

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. 12. 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 QuickCheck generator 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 (e.g., QuickCheck or SmallCheck's `==>`) is prohibitively costly: these approaches perform much worse than Lazy SmallCheck.

Well-Typed Lambda Terms Using our prototype implementation we reproduced the experiments of Pałka *et al.* [58], who generated well-typed lambda terms in order to discover bugs in GHC's strictness analyzer. We also use this case study to indirectly compare to two narrowing-based tools that are arguably closer to Luck and that use the same case study to evaluate their work: Claessen *et al.* [17, 18] and Fetscher *et al.* [23].

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,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 a type system for 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 seem similar (under quick manual inspection) to those found by Pałka *et al.* and Fetscher *et al.*.

Luck's generation speed was slower than that of Pałka's handwritten generator. We were able to generate terms of average size 50 (internal nodes), and, grouping terms together in batches of 100, we got a total time of generation, unparsing, compilation and execution of around 35 seconds per batch. This is a slowdown of 20x compared to that of Pałka's. However, our implementation is a total of 82 lines of fairly simple code, while the handwritten development is 1684 lines, with the warning "...the code is difficult to understand, so reading it is not recommended" in its distribution page [57].

The derived generators of Claessen *et al.* [17] achieved a 7x slowdown compared to the handwritten generator, while the Redex generators [23] also report a 7x slowdown in generation time for their best generator. However, by seeding the environment with monomorphised versions of the most common constants present in the counterexamples, they were able to achieve a time per counterexample on par with the handwritten generator.

Information-Flow Control For a second large case study, we reimplemented a method for generating information-flow-control machine states [37]. 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.

Hriţcu *et al.* [37] 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 `s` and then *vary* `s`, to generate an indistinguishable one `s'`. 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 the stronger property (SSNI), by encoding the indistinguishability predicate in Luck and using our prototype to generate small, indistinguishable pairs of states. In 216 lines of code 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 1450 tests per second, around 12.5 times slower.

The real promise of Luck, however, became apparent when we turned to LLNI. Hriţcu *et al.* [37] 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 our generator on the same set of buggy information-flow analyses as in Hriţcu *et al.* [37]. We were able to find all of the same bugs, with similar effectiveness (number of bugs found per 100 tests). However, the Luck generator was 24 times slower (Luck: 150 tests/s, Haskell: 3600 tests/s). We expect to be able to 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 a minimally tuned 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 closest related work.

Random Testing The works that are most closely related to our own are the narrowing based approaches of Gligoric *et al.* [27], 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 narrowing-like technique with FEAT [21], a tool for functional enumeration of algebraic types, to efficiently generate near-uniform random inputs satisfying some 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 [58]) using the "trick" of monomorphizing the polymorphic constants of the context as discussed in the previous section. 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 [7]. In the context of Isabelle's [52] 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 [46] and Runciman *et al.* [63] also provide support for exhaustive testing using narrowing-based techniques. Instead of implementing mechanisms that resemble narrowing in standard functional languages, Fischer and Kuchen [24] leverage the built-in engine of the functional logic programming language Curry [34] to enumerate tests satisfying a coverage criterion. In a later, black-box approach for Curry, Christiansen and Fischer [16] 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.* [14] 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 $X \backslash D$). 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, 32], the most recent and efficient for Haskell being Target [64]. 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, 28, 29, 48, 65]. Recently, in the context of Rosette [68], 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 [25] put needed-narrowing based solvers on a firmer 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 [30, 31, 50], while the problem of generating satisfying valuations shares similarities with probabilistic sampling [13, 45, 49, 53]. For example, the semantics of PROB in the recent probabilistic programming survey of Gordon *et al.* [31] 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 [53], 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 [36] ranking program is a very small program, powered by millions of observations. In contrast, random testing deals with very complex programs (e.g., 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 §2, where we use booleans to model labels:

```
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^{64} "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-typedness 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

We have 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 and exploring their interactions. We described a prototype implementation of this semantics, which we used to replicate the results of state-of-the-art handwritten random generators for two complex domains. The results showed the potential of Luck's approach, allowing us to replicate the generation presented by the handwritten generators with reduced code and effort. The prototype was slower by an order of magnitude, but there is still significant 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 other domain representations. We also want to investigate Luck's equational theory, showing, for instance, that the encoded conjunction, negation, and disjunction satisfy the usual logical laws. Finally, the backtracking strategies in our implementation can be abstractly modeled on top of our notion of choice-recording trace; Gallois-Wong [26] shows promising preliminary results using Markov chains for this.

Another potential direction for future work is automatically deriving smart shrinkers. Shrinking, or delta-debugging, is crucial in property-based testing, and it can also require significant user effort and domain specific knowledge to be efficient [60]. It would be interesting to see if there is a counterpart to narrowing or constraint solving that allows shrinking to preserve desired properties.

Acknowledgments

We are grateful to Maxime Dénès, Nate Foster, Thomas Jensen, Gowtham Kaki, George Karachalias, Michał Pałka, Zoe Paraskevopoulou, Christine Rizkallah, Antal Spector-Zabusky, Perdita Stevens, and Andrew Tolmach for useful comments. This work was supported by NSF awards #1421243, *Random Testing for Language Design* and #1521523, *Expeditions in Computing: The Science of Deep Specification*.

References

[1] S. Antoy. A needed narrowing strategy. *JACM*. 2000.

[2] T. Arts, L. M. Castro, and J. Hughes. Testing Erlang data types with QuviQ QuickCheck. In *7th ACM SIGPLAN Workshop on Erlang*. 2008.

[3] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with Veritesting. *ICSE*, 2014.

[4] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[5] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. *ITP*. 2010.

[6] L. Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. *CPP*. 2012.

[7] L. Bulwahn. Smart testing of functional programs in Isabelle. *LPAR*. 2012.

[8] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. *ASE*, 2008.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI*. 2008.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. *CCS*. 2006.

[11] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. *ICSE*. 2011.

[12] M. Carlier, C. Dubois, and A. Gotlieb. Constraint reasoning in FocalTest. *ICSOFT*. 2010.

[13] A. T. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. *AISTATS*, 2013.

[14] S. Chakraborty, K. S. Meel, and M. Y. Vardi. Balancing scalability and uniformity in SAT witness generator. *DAC*. 2014.

[15] H. R. Chamarthi, P. C. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. *ACL2*, 2011.

[16] J. Christiansen and S. Fischer. EasyCheck – test data for free. *FLOPS*. 2008.

[17] K. Claessen, J. Duregård, and M. H. Pałka. Generating constrained random data with uniform distribution. *FLOPS*. 2014.

[18] K. Claessen, J. Duregård, and M. H. Palka. Generating constrained random data with uniform distribution. *J. Funct. Program.*, 25, 2015.

[19] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ICFP*. 2000.

[20] P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability & Computing*, 13(4-5):577–625, 2004.

[21] J. Duregård, P. Jansson, and M. Wang. Feat: Functional enumeration of algebraic types. *Haskell Symposium*. 2012.

[22] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. *TPHOLs*. 2003.

[23] B. Fetscher, K. Claessen, M. H. Palka, J. Hughes, and R. B. Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. *ESOP*. 2015.

[24] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. *PPDP*. 2007.

[25] J. Fowler and G. Huttom. Towards a theory of reach. *TFP*. 2015.

[26] D. Gallois-Wong. Formalising Luck: Improved probabilistic semantics for property-based generators. Inria Internship Report, 2016.

[27] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. *ICSE*. 2010.

[28] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *PLDI*. 2005.

[29] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: whitebox fuzzing for security testing. *ACM Queue*, 10(1):20, 2012.

[30] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. *UAI*, 2008.

[31] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. *FOSE*. 2014.

[32] A. Gotlieb. Euclide: A constraint-based testing framework for critical C programs. *ICST*, 2009.

[33] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. *ISSTA*. 2012.

[34] M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: A truly functional logic language. *ILPS*, 1995.

[35] M. Hanus. A unified computation model for functional and logic programming. *POPL*. 1997.

[36] R. Herbrich, T. Minka, and T. Graepel. Trueskill$^{\text{tm}}$: A bayesian skill rating system. *NIPS*, 2006.

[37] C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. *ICFP*. 2013.

[38] C. Hriţcu, L. Lampropoulos, A. Spector-Zabusky, A. Azevedo de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis. Testing noninterference, quickly. *JFP*, 26:e4 (62 pages), 2016. Technical Report available as arXiv:1409.0393.

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

[40] D. Jackson. *Software Abstractions: Logic, Language, and Anlysis*. The MIT Press, 2011.

[41] T. P. Jensen. Disjunctive program analysis for algebraic data types. *ACM Trans. Program. Lang. Syst.*, 19(5):751–803, 1997.

[42] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[43] G. Kaki and S. Jagannathan. A relational framework for higher-order shape analysis. *ICFP*. 2014.

[44] A. S. Köksal, V. Kuncak, and P. Suter. Scala to the power of Z3: integrating SMT and programming. *CADE*. 2011.

[45] K. Łatuszyński, G. O. Roberts, and J. S. Rosenthal. Adaptive gibbs samplers and related mcmc methods. *The Annals of Applied Probability*, 23(1):66–98, 2013.

[46] F. Lindblad. Property directed generation of first-order test data. *TFP*. 2007.

[47] A. K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.

[48] R. Majumdar and K. Sen. Hybrid concolic testing. *ICSE*. 2007.

[49] V. K. Mansinghka, D. M. Roy, E. Jonas, and J. B. Tenenbaum. Exact and approximate sampling by systematic stochastic search. *AISTATS*, 2009.

[50] B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: probabilistic models with unknown objects. *IJCAI*, 2005.

[51] F. Nielson and H. R. Nielson. Tensor products generalize the relational data flow analysis method. In *4th Hungarian Computer Science Conference*, 1985.

[52] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[53] A. V. Nori, C.-K. Hur, S. K. Rajamani, and S. Samuel. R2: An efficient mcmc sampler for probabilistic programs. *AAAI*. 2014.

[54] C. Okasaki. Red-black trees in a functional setting. *JFP*, 9(4):471–477, 1999.

[55] S. Owre. Random testing in PVS. In *Workshop on Automated Formal Methods*, 2006.

[56] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. *OOPSLA*. 2007.

[57] M. H. Pałka. Testing an optimising compiler by generating random lambda terms. `http://www.cse.chalmers.se/~palka/testingcompiler/`.

[58] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. *AST*. 2011.

[59] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. *ITP*. 2015.

[60] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. *PLDI*, 2012.

[61] J. S. Reich, M. Naylor, and C. Runciman. Lazy generation of canonical test programs. *IFL*. 2011.

[62] A. Rodriguez Yakushev and J. Jeuring. Enumerating well-typed terms generically. In U. Schmid, E. Kitzelmann, and R. Plasmeijer, editors, *Approaches and Applications of Inductive Programming*, volume 5812 of *Lecture Notes in Computer Science*, pages 93–116. Springer Berlin Heidelberg, 2010.

[63] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. *Haskell Symposium*. 2008.

[64] E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing. *ESOP*, 2015.

[65] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *ESEC/FSE*. 2005.

[66] P. Tarau. On type-directed generation of lambda terms. *ICLP*, 2015.

[67] A. P. Tolmach and S. Antoy. A monadic semantics for core Curry. *Electr. Notes Theor. Comput. Sci.*, 86(3):16–34, 2003.

[68] E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. *PLDI*. 2014.