Luck: A Probabilistic Language for Testing
Lampropoulos Leonidas, Benjamin C. Pierce and Li-yao Xia
University of Pennsylvania
Diane Gallois-Wong and Cătălin Hrițcu
INRIA Paris
John Hughes
Chalmers University

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. This chapter introduces a probabilistic 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 probabilistic 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.

13.1 Introduction
Since being popularized by QuickCheck (Claessen and Hughes, 2000), property-based random testing has become a standard technique for improving software quality in a wide variety of programming languages (Arts et al., 2008; Lindblad, 2007; Hughes, 2007; Pacheco and Ernst, 2007) and for streamlining interaction with proof assistants (Chamarthi et al., 2011; Bulwahn, 2012a; Owre, 2006; Dybjer et al., 2003; Paraskevopoulou et al., 2015).

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:

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

To test this property, QuickCheck generates random lists until either it finds a counterexample or a predetermined 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 \(p \implies q\), built using the implication combinator \(\implies\), 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,

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

where, as usual:

```haskell
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 generator, a probabilistic program that produces 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 (Claessen and Hughes, 2000; Hriţcu et al., 2013; Groce et al., 2012).

Custom 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...
13.1 Introduction

about random generation techniques for well-typed lambda-terms (Palka et al., 2011; Fetscher et al., 2015; Tarau, 2015) and for “indistinguishable” machine states that can be used for finding bugs in information-flow monitors (Hriţcu et al., 2013, 2016). Moreover, if we aim 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 (Hriţcu et al., 2013) is over 1500 lines of tricky Haskell, while Palka et al.’s generator for well-typed lambda-terms (Palka et al., 2011) 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 $p$, instantiating 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. It is important to carefully control the probabilities guiding this choice to avoid getting a distribution which is very skewed towards early elements.

This process resembles narrowing from functional logic programming (Antoy, 2000; Hanus, 1997; Lindblad, 2007; Tolmach and Antoy, 2003). It is attractively lightweight, admits natural control over distributions (as we will see in the next section), and has been used successfully (Fischer and Kuchen, 2007; Christiansen and Fischer, 2008; Reich et al., 2011; Gligoric et al., 2010), even in challenging domains such as generating well-typed programs to test compilers (Claessen et al., 2014; Fetscher et al., 2015).

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 given 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 list’s tail. Since the values in the tail 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 the length of ys – say, just twice as big – then 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, leading to unacceptably low rates of successful generation.

The second existing approach uses a constraint solver to generate a diverse set of valuations satisfying a predicate. This approach has been widely investigated, both for generating inputs directly from predicates (Carlier et al., 2010; Seidel et al., 2015; Gotlieb, 2009; Köksal et al., 2011) and for symbolic-execution-based testing (Godefroid et al., 2005; Sen et al., 2005; Cadar et al., 2008; Avgerinos et al., 2014; Torlak and Bodík, 2014), 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 (Chakraborty et al., 2014), but this is typically not the distribution we want in practice (see Section 13.2). 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. This chapter explores such an approach by introducing

1 Constraint solvers can, of course, be used to directly search for counterexamples to a property of interest by software model checking (Blanchette and Nipkow, 2010; Jackson, 2011; Ball et al., 2011; Ihala and Majumdar, 2009, 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.
a probabilistic domain-specific language, Luck, for writing generators via lightweight annotations on predicates. In Section 13.2 we illustrate Luck’s novel features using binary search trees as an example. We also place Luck’s design on a firm formal foundation, by defining a probabilistic core calculus and establishing key properties: the soundness and completeness of its probabilistic generator semantics with respect to a straightforward interpretation of expressions as predicates (Section 13.3).

Finally, we provide a prototype interpreter (Section 13.4) using a custom constraint solver that supports per-variable sampling. We evaluate Luck’s expressiveness on a collection of common examples from the random testing literature (Section 13.5) and on two significant case studies, demonstrating 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.

This chapter 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) (Section 13.3.3); (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) (Section 13.5); (3) an extended version of the paper this chapter is based on (Lampropoulos et al., 2017) with full definitions and paper proofs for the whole semantics (https://arxiv.org/abs/1607.05443).

13.2 Luck by Example

Figure 13.1 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 (Okasaki, 1999). It is followed by a QuickCheck generator genTree, which generates BSTs with a given maximum depth, controlled by the size parameter. This generator first checks whether low + 1 >= high, in which case it returns the only valid BST satisfying this constraint – the Empty one. Otherwise, it uses QuickCheck’s frequency combinator, which takes a list of pairs of positive integer weights and associated generators and randomly selects one of the generators using the probabilities specified by the weights. In this example, \( \frac{1}{size+1} \) 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 \langle some other property of t \rangle,” but it raises some new issues of its own. First, even for this simple example, getting the generator right is a bit tricky
Binary tree datatype (in both Haskell and Luck):

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

Test predicate for BSTs (in Haskell):

```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):

```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:

```haskell
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 13.1 Binary Search Tree tester and two generators

(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. We can’t just throw away the predicate and keep the generator because we often need them both, for example to test properties like “the insert function applied to a BST and a value returns a BST.” As predicates and generators become more complex, these issues can become quite problematic (e.g., Hriţcu et al., 2013). Enter Luck.

The bottom of Figure 13.1 shows a Luck program that represents both a BST
predicate and a generator for random BSTs. Modulo variations in concrete syntax, the Luck closely code follows the Haskell bst predicate. 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 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 Section 13.4), 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 body of the branch.

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 ↦ i, uh ↦ j, u ↦ 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 controls 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 $v$ rather than an unknown, we similarly instantiate any unknowns appearing within $v$.

As a concrete example, consider the compound inequality constraint $0 < x \land x < 4$. A generator based on pure narrowing (as in Gligoric et al., 2010), 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 $((0 < x) \land x) \land (x < 4)$. 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 $(0 < x \land x < 4) \land x$, 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 (Figure 13.1)), 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 $\frac{1}{1+\text{size}}$ of the time or partially instantiated to a Node (with fresh unknowns for $x$ and the left and right subtrees) $\frac{\text{size}}{1+\text{size}}$ 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., Hriţcu et al., 2013). 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. (2013). 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:

```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, a uniform distribution over indistinguishable pairs means that we will essentially never generate pairs with Low labels. Clearly, such a distribution cannot provide effective testing; indeed, Hriţcu et al. found that the best distribution was somewhat skewed in favor of Low labels.

We can easily validate this intuition using a probabilistic programming framework with emphasis on efficient sampling: R2 (Nori et al., 2014). We can model indistinguishability using the following probabilistic program, where labels are modeled by booleans:

```python
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. As predicted, all generated samples have their booleans set to true. 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!
13.3 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. The core omits primitive booleans and integers and replaces datatypes with binary sums, products, and iso-recursivetypes.

We begin in Section 13.3.1 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* (Section 13.3.2), 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 (Section 13.3.3). Finally, we introduce a *matching semantics*, building on the narrowing semantics, that unifies constraint solving and narrowing into a single evaluator (Section 13.3.4). The key properties of the generator semantics (both narrowing and matching versions) are soundness and completeness with respect to the predicate semantics (Section 13.3.6); 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).

13.3.1 Syntax, Typing, and Predicate Semantics

The syntax of Core Luck is given in Figure 13.2. 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-recursivetypes. 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-recursivetype 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 Section 13.3.4.

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 ($\text{Tree} = \mu X. 1 + \text{int} \times X \times X$) and naturals, plus syntactic sugar for constant naturals:

$$\text{case } (\text{unfold}_{\text{Tree}} \text{ tree } < -(1, \text{size})) \text{ of } (L \ x \to \ldots)(R \ y \to \ldots)$$

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 Figure 13.3. 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 $\Gamma = \emptyset$, we write $U \vdash e : T$ as a shorthand for $\emptyset; U \vdash e : T$. 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

Figure 13.2 Core Luck Syntax
instantiation expression  \( e \leftarrow (e_l, e_r) \) is well typed if \( e \) has sum type \( T_1 + T_2 \) and \( e_l \) and \( e_r \) are natural numbers. A sample expression \( !e \) has the (non-functional) type \( T \) when \( e \) has type \( T \).

The predicate semantics for Core Luck, written \( e \perp v \), 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 (Figure 13.4). 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 \( \text{True} \).

### 13.3.2 Constraint Sets

The rest of this section develops an alternative probabilistic generator semantics for Core Luck. This semantics will use constraint sets \( \kappa \in C \) 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 the
extended version of the chapter), simply requiring that they support the following operations:

\[
\begin{align*}
\llbracket \cdot \rrbracket &:: C \rightarrow \text{Set Valuation} \\
U &:: C \rightarrow \text{Map } \mathcal{U} \overline{T} \\
fresh &:: C \rightarrow \overline{T}^* \rightarrow (C \times \mathcal{U}^*) \\
\text{unify} &:: C \rightarrow \text{Val} \rightarrow \text{Val} \rightarrow C \\
\text{SAT} &:: C \rightarrow \text{Bool} \\
[\cdot] &:: C \rightarrow \mathcal{U} \rightarrow \text{Maybe Val} \\
\text{sample} &:: C \rightarrow \mathcal{U} \rightarrow C^*
\end{align*}
\]

Here we describe these operations informally, deferring technicalities until after we have presented the generator semantics (Section 13.3.6).

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

Many of the semantic rules will need to introduce fresh unknowns. The fresh function takes a constraint set \( \kappa \) 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 \( \kappa \) and two values, each potentially containing unknowns, \( \text{unify} \) updates \( \kappa \) to preserve only those valuations in which the values match.

\( \text{SAT} \) 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 \( \kappa \), all valuations map \( u \) to the same value \( v \), then that value is returned (written \( \{v\} \)); otherwise nothing (written \( \emptyset \)).

The \( \text{sample} \) operation is used to implement sample expressions (\( !e \)): given a constraint set \( \kappa \) and an unknown \( u \in U(\kappa) \), it returns a list of constraint sets representing all possible concrete choices for \( u \), in all of which \( u \) is completely determined – that is \( \forall \kappa \in (\text{sample } \kappa \ u). \exists v. \ \kappa[u] = \{v\} \). To allow for reasonable implementations of this interface, we maintain an invariant that the input unknown to \( \text{sample} \) will always have a finite denotation; thus, the resulting list is also finite.

### 13.3.3 Narrowing Semantics

As a first step toward a probabilistic 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 Section 13.3.4.

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

$$e \models \kappa \Downarrow q \kappa' \models v$$

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 $0 \leq m < n$, where $n$ denotes the number of possibilities chosen among and $m$ is the index of the one actually taken. We write $\varepsilon$ for the empty trace and $t \cdot t'$ for the concatenation of two traces. We also annotate the judgment with the probability $q$ of making the choices represented in the trace. Recording traces is useful after the fact in calculating the total probability of some given outcome of evaluation (which may be reached by many different derivations), but they play no role in determining how evaluation proceeds.

We maintain the invariant that both the input constraint set $\kappa$ and the input expression $e$ are well typed, the latter with respect to an empty variable context and unknown context $U(\kappa)$. Another invariant is that every constraint set $\kappa$ that appears as input to a judgment is satisfiable and the restriction of its denotation to the unknowns in $e$ is finite. These invariants are established at the top-level (see Section 13.4). The finiteness invariant ensures the output of sample will always be a finite collection and therefore the probabilities involved will be positive rational numbers. They also guarantee termination of constraint solving, as we will see in Section 13.3.4. 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 type fresh unknowns.

The narrowing semantics is given in Figure 13.5 for the standard constructs (omitting fold/unfold and $\textbf{N-R}$ and $\textbf{N-Case-R}$ rules analogous to the $\textbf{N-L}$ and $\textbf{N-Case-L}$ rules shown) and in Figure 13.6 for instantiation expressions; Figure 13.8 and Figure 13.7 give some auxiliary definitions. Most of the rules are intuitive. A common pattern is sequencing two narrowing judgments $e_1 \models \kappa \Downarrow q_1 \kappa_1 \models v$ and $e_2 \models \kappa_1 \Downarrow q_2 \kappa_2 \models v$. The constraint-set result of the first narrowing judgment ($\kappa_1$) is given as input to the second, while traces and probabilities are accumulated by concatenation ($t_1 \cdot t_2$) and multiplication ($q_1 \cdot q_2$). We now explain the rules in detail.

Rule $\textbf{N-Base}$ 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.

https://doi.org/10.1017/9781108770750.014 Published online by Cambridge University Press
13.3 Semantics of Core Luck

Figure 13.5 Narrowing Semantics of Standard Core Luck Constructs
Rule \textbf{N-Pair}: To evaluate \((e_1, e_2)\) given a constraint set \(\kappa\), we sequence the derivations for \(e_1\) and \(e_2\).

Rules \textbf{N-CasePair-P}, \textbf{N-CasePair-U}: To evaluate the pair elimination expression \(\text{case } e \text{ of } (x, y) \rightarrow e'\) in a constraint set \(\kappa\), we first evaluate \(e\) in \(\kappa\). Typing ensures that the resulting value is either a pair or an unknown. If it is a pair (\textbf{N-CasePair-P}), we substitute its components for \(x\) and \(y\) in \(e'\) and continue evaluating. If it is an unknown \(u\) of type \(\overline{T_1} \times \overline{T_2}\) (\textbf{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 \textbf{N-CasePair-U}. However, unknowns in Luck only range over non-functional
13.3 Semantics of Core Luck

\[ \text{sample } \kappa \ u = S \quad S[m] = \kappa' \]

\[ \text{sampleV } \kappa \ u \Rightarrow \{[(m,[S])] \} \bigg/ \kappa' \]

\[ \text{sampleV } \kappa \ v \Rightarrow \xi \kappa' \]

\[ \text{sampleV } \kappa \ (\text{fold}_T \ v) \Rightarrow \xi \kappa' \]

\[ \text{sampleV } \kappa \ (L_T \ v) \Rightarrow \xi \kappa' \]

\[ \text{sampleV } \kappa \ v_1 \Rightarrow \xi_{q_1} \kappa_1 \vphantom{[]} \]

\[ \text{sampleV } \kappa \ v_2 \Rightarrow \xi_{q_2} \kappa' \]

\[ \text{sampleV } \kappa \ (v_1, v_2) \Rightarrow \xi_{q_1, q_2} \kappa' \]

**Figure 13.8** Auxiliary relation sampleV

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 \( \kappa_a \).

**Rule N-L:** To evaluate \( L_{T_1+T_2} \ e \), we evaluate \( e \) and tag the resulting value with \( L_{T_1+T_2} \ e \), 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 \( u \in U \), 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 \( \kappa_l \) and \( R_{\overline{T}_l+\overline{T}_r} \ v_r \) with \( u \) to get \( \kappa_r \). We then use the auxiliary relation choose (Figure 13.7), which takes two integers \( n \) and \( m \) (here equal to 1) as well as two constraint sets (here \( \kappa_l \) and \( \kappa_r \)), to select either \( l \) or \( r \). If exactly one of \( \kappa_l \) and \( \kappa_r \) 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 $\kappa_l$ and $\kappa_r$ cannot both be unsatisfiable at the same time, since there must exist at least one valuation in $\kappa_0$ 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 $\text{rec} (f : T_1 \to T_2) \ x = e_2$ (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, $e_2[(\text{rec} (f : T_1 \to T_2) \ x = e_2)/f, v_1/x]$, 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 $\text{sampleV}$ (Figure 13.8) to completely instantiate $v$, walking down the structure of $v$. When unknowns are encountered, $\text{sample}$ 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 $m$th constraint set in $S$, for each $0 \leq m < |S|$.

Rule N-Narrow is similar to N-Case-U, modulo the “weight” arguments $e_1$ and $e_2$. These are evaluated to values $v_1$ and $v_2$, and $\text{sampleV}$ is called to ensure that they are fully instantiated in all subsequent constraint sets, especially $\kappa_e$. The relation $\text{nat}_{\kappa_e}(v_1) = n_1$ walks down the structure of the value $v_1$ (like $\text{sampleV}$) and calculates the unique natural number $n_1$ corresponding to $v_1$: when the input value is an unknown, $\text{nat}_{\kappa_e}(u) = n$ holds if $\kappa[u] = v'$ and $\llbracket v \rrbracket = n$, where the notation $\llbracket v \rrbracket$ is defined in Figure 13.4. The rest of the rule is the same as N-Case-U, but with the computed weights $n_1$ and $n_2$ given as arguments to choose to shape the distribution.

Using the narrowing semantics, we can implement a more efficient method for generating valuations than the naive generate-and-test described in Section 13.3.1: 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 (Claessen et al., 2014).

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 $\kappa$ where each unknown has type $\text{Bool}$ (i.e., $1 + 1$) and the domain associated with each contains both $\text{True}$ and $\text{False}$ ($L_{1+1}()$ 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 $\text{case } e_1 \text{ of } (L \ x \to e_2)(R \ y \to \text{False})$. If we attempt to evaluate the expression $u_1 \&\& u_2 \&\& u_3$ using the narrowing semantics,
we first apply the \textbf{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 \texttt{False} result for the entire expression 50\% of the time. If we choose to unify $u_1$ with an $L$, then we apply the \textbf{N-Case-U} rule again, returning either \texttt{False} or $u_3$ (since unknowns are values – rule \textbf{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...

### 13.3.4 Matching Semantics

In this section we present a matching semantics that takes as an additional input a \textit{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

$$ p \leftarrow e \models \kappa \upharpoonright^T \kappa^? $$

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

We keep the invariants from Section 13.3.3: the input constraint set $\kappa$ is well typed and so is the input expression $e$ (with respect to an empty variable context and $U(\kappa)$); moreover $\kappa$ 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).

The rules except for \textit{case} are similar to the narrowing semantics. Figure 13.9 shows several; the rest appear in the extended version.

Rule \textbf{M-Base}: To generate valuations for a unit value or an unknown, we unify $\nu$ and the target pattern $p$ under the input constraint set $\kappa$. Unlike \textbf{N-Base}, there is no case for functions, since the expression being evaluated must have a non-function type.

Rules \textbf{M-Pair}, \textbf{M-Pair-Fail}: To evaluate $(e_1, e_2)$, where $e_1$ and $e_2$ have types $\overline{T}_1$
and $T_2$, we first generate fresh unknowns $u_1$ and $u_2$. We unify the pair $(u_1, u_2)$ with the target pattern $p$, obtaining a new constraint set $\kappa'$. We then proceed as in N-Pair, evaluating $e_1$ against pattern $u_1$ 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 $\emptyset$, the whole computation immediately yields $\emptyset$ as well; $e_2$ is not evaluated, and the final trace and probability are $t_1$ and $q_1$.

Rules M-App, M-After: To evaluate an application $e_0 \ e_1$, we use the narrowing semantics to reduce $e_0$ to $\text{rec } f \ x = e_2$ and $e_1$ to a value $v_1$, then evaluate $e_2[(\text{rec } f \ x = e_2)/f, v_2/x]$ against the original $p$. In this rule we cannot use a pattern during the evaluation of $e_1$: we do not have any candidates! This is the main reason for introducing the sequencing operator 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.
13.3 Semantics of Core Luck

\[(\kappa_0, [u_1, u_2]) = \text{fresh } \kappa [\overline{T}_1, \overline{T}_2]\]
\[\left( \begin{array}{l}
L_{\overline{T}_1+\overline{T}_2} u_1 \iff e \vdash \kappa_0 \upharpoonright_{q_1} \{ \kappa_1 \}
\end{array} \right)
\]
\[\left( \begin{array}{l}
R_{\overline{T}_1+\overline{T}_2} u_2 \iff e \vdash \kappa_0 \upharpoonright_{q_2} \{ \kappa_2 \}
\end{array} \right)
\]

\[p \iff e_1[u_1/x_l] \vdash \kappa_1 \upharpoonright_{q_1} \kappa_a \quad p \iff e_2[u_2/y_r] \vdash \kappa_2 \upharpoonright_{q_2} \kappa_b
\]

\[\kappa' = \text{combine } \kappa_0 \kappa_a \kappa_b
\]

\[p \iff \text{case } e^{\overline{T}_1+\overline{T}_2} \text{ of } (L \ x_l \rightarrow e_1)(R \ y_r \rightarrow e_2) \vdash \kappa
\]

\[\upharpoonright_{q_1+q_2+q'_1+q'_2} \kappa'
\]

\[\text{where } \text{combine } \kappa \emptyset \emptyset = \emptyset
\]
\[\text{combine } \kappa \{ \kappa_1 \} \emptyset = \{ \kappa_1 \}
\]
\[\text{combine } \kappa \emptyset \{ \kappa_2 \} = \{ \kappa_2 \}
\]
\[\text{combine } \kappa \{ \kappa_1 \} \{ \kappa_2 \} = \text{union } \kappa_1 \text{ (rename } (U(\kappa_1)-U(\kappa)) \kappa_2)
\]

\[(\kappa_0, [u_1, u_2]) = \text{fresh } \kappa [\overline{T}_1, \overline{T}_2]\]
\[\left( \begin{array}{l}
L_{\overline{T}_1+\overline{T}_2} u_1 \iff e \vdash \kappa_0 \upharpoonright_{q_1} \emptyset
\end{array} \right)
\]
\[\left( \begin{array}{l}
R_{\overline{T}_1+\overline{T}_2} u_2 \iff e \vdash \kappa_0 \upharpoonright_{q_2} \{ \kappa_2 \}
\end{array} \right)
\]

\[p \iff e_2[u_2/y] \vdash \kappa_2 \upharpoonright_{q_2} \kappa_b
\]

\[p \iff \text{case } e^{\overline{T}_1+\overline{T}_2} \text{ of } (L \ x \rightarrow e_1)(R \ y \rightarrow e_2) \vdash \kappa
\]

\[\upharpoonright_{q_1+q_2+q'_1+q'_2} \kappa'
\]

The interesting rules are the ones for \textit{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. Instead of choosing which branch to follow with some probability (50\% in \textbf{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:

\[
\text{rename} :: \mathcal{U}^* \rightarrow C \rightarrow C
\]
\[
\text{union} :: C \rightarrow C \rightarrow C
\]

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

Two of the rules appear in Figure 13.10. (A third is symmetric to \textbf{M-Case-2}; a fourth handles failures.) We independently evaluate \( e \) against both an \textit{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 ($\text{M-Case-2}$ or the other elided rule). If both succeed ($\text{M-Case-1}$), we evaluate both branch bodies and combine the results with $\text{union}$. We use $\text{rename}$ to avoid conflicts, since we may generate the same fresh unknowns while independently computing $\kappa_a^2$ and $\kappa_b^2$. If desired, the user can ensure that only one branch will be executed by using an instantiation expression before the $\text{case}$ 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 $\text{M-Case-1}$ rule is the second place where the need for finiteness of the restriction of $\kappa$ to the input expression $e$ arises. In order for the semantics to terminate in the presence of (terminating) recursive calls, it is necessary that the domain be finite. To see this, consider a simple recursive predicate that holds for every number:

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

Even though $f$ terminates in the predicate semantics for every input $u$, if we allow a constraint set to map $u$ to the infinite domain of all natural numbers, the matching semantics will not terminate. While this finiteness restriction feels a bit unnatural, we have not found it to be a problem in practice – see Section 13.4.

### 13.3.5 Example

To show how all this works, let’s trace the main steps of the matching derivations of two given expressions against the pattern $\text{True}$ 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 $\text{True}$ in a constraint set $\kappa$, in which $u$ is independent from other unknowns and its possible values are $0, \ldots, 9$. Similar expressions were introduced as examples in Section 13.2; the results we obtain here confirm the intuitive explanation given there.

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

$$
\text{lt} = \text{rec } (f : \text{nat} \rightarrow \text{nat} \rightarrow \text{bool}) \ 
\text{case } y \ 
\text{of } (L_\rightarrow \text{False}) (R yR \rightarrow \text{case } x \ 
\text{of } (L_\rightarrow \text{True}) (R xR \rightarrow f \ xR yR))
$$

https://doi.org/10.1017/9781108770750.014 Published online by Cambridge University Press
13.3 Semantics of Core Luck

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 use the same variable for two constraint sets which differ only in the addition of a few irrelevant variables in one.

**Evaluation of A** We first derive True $\Leftarrow (0 < u) \models \kappa \models \{ \kappa_0 \}$. 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 $(L \_ \rightarrow \text{False})(R \_ \rightarrow \ldots)$. Since the domain of $u$ contains both zero and non-zero elements, unifying $u$ with $L_{\_nat} u_1$ and $R_{\_nat} u_2$ (**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 $\text{False}$, the result of the left derivation in **M-Case-1** is $\emptyset$ and in the resulting constraint set $\kappa_0$ the domain of $u$ is $\{1, \ldots, 9\}$.

Next, we turn to $\text{True} \Leftarrow (0 < u \& \& u < 4) \models \kappa \models \{ \kappa_1 \}$, where, by a similar argument following the recursion, the domain of $u$ in $\kappa_1$ is $\{1, 2, 3\}$. There are 3 possible narrowing-semantics derivations for $!u$: (1) $!u \models \kappa_1 \models \{ \kappa_i^A \} \models u$, (2) $!u \models \kappa_1 \models \{ \kappa_i^A \} \models u$, and (3) $!u \models \kappa_1 \models \{ \kappa_i^A \} \models u$, where the domain of $u$ in $\kappa_i^A$ 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 $\text{True}$ in $\kappa$ are:

$$\text{True} \Leftarrow A \models \kappa \models \{ \kappa_i^A \} \quad \text{for } i \in \{1, 2, 3\}$$

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 $\text{True} \Leftarrow A \models \kappa$ 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 $\kappa_0$ and there are 9 possibilities: $!u \models \kappa_0 \models \{ \kappa_i^B \} \models u$ for each $i \in \{1, \ldots, 9\}$, where the domain of $u$ in $\kappa_i^B$ is $\{i\}$. Then we evaluate $\text{True} \Leftarrow u < 4 \models \kappa_i^B$: if $i$ is 1, 2 or 3 this yields $\{ \kappa_i^B \}$; if $i > 3$ this yields a failure $\emptyset$. Therefore the possible derivations for $B = (0 < u ; !u) \& \& u < 4$ are:

$$\text{True} \Leftarrow B \models \kappa \models \{ \kappa_i^B \} \quad \text{for } i \in \{1, 2, 3\}$$

$$\text{True} \Leftarrow B \models \kappa \models \emptyset \quad \text{for } i \in \{4, \ldots, 9\}$$

https://doi.org/10.1017/9781108770750.014 Published online by Cambridge University Press
We can again compute the corresponding probability distribution:

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

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 \(\emptyset\) with probability \(\frac{1}{9}\), so we would not be able to compute its total probability.

The probability associated with \(\emptyset\) (0 for \(A\), 2/3 for \(B\)) is the probability of backtracking. As stressed in Section 13.2, \(A\) 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 Gallois-Wong (2016).

### 13.3.6 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 \(\kappa\), we can only remove valuations from the denotation of \(\kappa\) (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 \(\kappa\), written \(\text{dom}(\kappa)\). 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 \(\kappa\), \(u\) does not appear in the domain of \(\kappa\); it only appears in the denotation after we use it in a unification. The domain of \(\kappa\) 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 \(\sigma\) are exactly the unknowns that have bindings in \([\kappa]\), i.e., in \(\text{dom}(\kappa)\). We use the overloaded notation \(\sigma|_x\) to denote the restriction of \(\sigma\) to \(x\), where \(x\) is either a set of unknowns or another valuation.
13.3 Semantics of Core Luck

Specification of fresh

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

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

Specification of sample

\( \kappa' \in \text{sample } \kappa u \Rightarrow \begin{cases} U(\kappa') = U(\kappa) \\ \text{SAT}(\kappa') \\ \exists v. \llbracket \kappa' \rrbracket = \{ \sigma \mid \sigma \in \llbracket \kappa \rrbracket, \sigma(u) = v \} \end{cases} \)

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

\[
\begin{cases}
\sigma(u) = v \\
\sigma \in \llbracket \kappa \rrbracket
\end{cases}
\Rightarrow \exists \kappa'. \begin{cases}
\sigma \in \llbracket \kappa' \rrbracket \\
\kappa' \in \text{sample } \kappa u
\end{cases}
\]

Specification of unify

\( U(\text{unify } \kappa v_1 v_2) = U(\kappa) \)

\[
\llbracket \text{unify } \kappa v_1 v_2 \rrbracket = \{ \sigma \in \llbracket \kappa \rrbracket \mid \sigma(v_1) = \sigma(v_2) \}
\]

When we unify in a constraint set \( \kappa \) two well-typed values \( v_1 \) and \( v_2 \), the typing map remains unchanged while the denotation of the result contains the valuations from \( \kappa \) that when substituted into \( v_1 \) and \( v_2 \) make them equal.

Specification of union

\( U(\kappa_1) \mid U(\kappa_1) \cap U(\kappa_2) = U(\kappa_2) \mid U(\kappa_1) \cap U(\kappa_2) \Rightarrow \begin{cases} U(\kappa) = U(\kappa_1) \cup U(\kappa_2) \\ \llbracket \kappa \rrbracket = \llbracket \kappa_1 \rrbracket \cup \llbracket \kappa_2 \rrbracket \end{cases} \)

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.
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 $\kappa$.

**Theorem 13.1** (Decreasingness).

$$e \models \kappa \Downarrow^t \kappa' \models v \Rightarrow \kappa' \leq \kappa$$

Soundness and completeness can be visualized as follows:

$$e \models \kappa \Downarrow^q \kappa' \models v$$ $$\exists \sigma \in [\kappa]'$$

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 $e \models \kappa \Downarrow^q \kappa' \models v$ 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 $\kappa$ (or that $e$ is well typed in $\kappa$).

**Theorem 13.2** (Soundness).

$$e \models \kappa \Downarrow^q \kappa' \models v$$ $$\sigma'(v) = v_p \land \sigma' \in [\kappa']$$ $$\forall u. u \in e \Rightarrow u \in dom(\kappa)$$ $$\Rightarrow \exists \sigma \exists e_p.$$ $${\sigma'}|_{\sigma} \equiv \sigma$$ $$\sigma' \in [\kappa']$$ $$\exists \sigma(e) = e_p$$ $$e_p \Downarrow v_p$$

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 $\kappa$ such that for some valuation $\sigma \in [\kappa]$ substituting $\sigma$ in $e$ yields $e_p$, if is well typed, there is always a nonzero probability of obtaining some factoring of $v_p$ as the result of a narrowing judgment.

**Theorem 13.3** (Completeness).

$$e_p \Downarrow v_p$$ $$\sigma(e) = e_p$$ $$\sigma \in [\kappa] \land \vdash \kappa$$ $$\emptyset; U(\kappa) \vdash e : T$$ $$\Rightarrow \exists v \kappa' \sigma' q t.$$

$$\sigma'(v) = v_p$$ $$\sigma'(\sigma) \equiv \sigma \land \sigma' \in [\kappa']$$ $$\exists e' \Downarrow^q \kappa' \models v$$

https://doi.org/10.1017/9781108770750.014 Published online by Cambridge University Press
Properties of the Matching Semantics  The decreasingness property for the matching semantics is very similar to the narrowing semantics: if the matching semantics yields \{κ’\}, then κ’ is smaller than the input κ.

**Theorem 13.4** (Decreasingness).

\[ p \Leftarrow e \Vdash κ \triangleright_q \{κ’\} \Rightarrow κ’ \leq κ \]

Soundness is again similar to the matching semantics.

**Theorem 13.5** (Soundness).

\[
\begin{align*}
p &\Leftarrow e \Vdash κ \triangleright_q \{κ’\} \\
σ'(p) &= v_p \land σ' \in \llbracket κ' \rrbracket \\
\forall u. \ (u ∈ e \lor u ∈ p) &\Rightarrow u ∈ dom(κ)
\end{align*}
\]

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 input valuations result in a terminating execution.

**Theorem 13.6** (Completeness).

\[
\begin{align*}
e_p &\Downarrow v_p \land σ \in \llbracket κ \rrbracket \\
\emptyset; U(κ) &\vdash e : T \land \vdash κ \\
σ(e) &= e_p \land σ(p) = v_p \\
\forall σ' \in \llbracket κ \rrbracket. \exists v'. σ'(e) &\Downarrow v'
\end{align*}
\]

13.4 Implementation

We next describe the Luck prototype: its top level, its treatment of backtracking and its probability-preserving pattern match compiler. We refer the reader to the extended version for the constraint set implementation.

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 (\(\emptyset; U(κ) \vdash e : 1 + 1\)).

\(^2\) 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...
The interpreter matches \( e \) against \( \text{True} \) (that is, \( L_{1+1} () \)), to derive a refined constraint set \( \kappa' \):

\[
L_{1+1} () \leftarrow e \models \kappa \upharpoonright_q \{ \kappa' \}
\]

This involves random choices, and there is also the possibility that matching fails (and the semantics generates \( \emptyset \) instead of \( \{ \kappa' \} \)). In this case, 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 (Claessen et al., 2015). Our prototype backtracks locally in calls to \texttt{choose}: if \texttt{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 \( \text{True} \), all the resulting valuations \( \sigma \in \llbracket \kappa' \rrbracket \) should map the unknowns in \( \vec{u} \) to some values. However, there is no guarantee that the generator semantics will yield a \( \kappa' \) mapping every \( \vec{u} \) to a unique values. The Luck top-level then applies the \texttt{sample} constraint set function to each unknown in \( \vec{u} \), ensuring that \( \sigma|_{\vec{u}} \) is the same for each \( \sigma \) in the final constraint set. The interpreter returns this common \( \sigma|_{\vec{u}} \) if it exists, and backtracks otherwise.

**Pattern Match Compiler** In Section 13.2, we saw an example using a standard \texttt{Tree} datatype and instantiation expressions assigning different weights to each branch. While the desugaring of simple pattern matching to core Luck syntax is straightforward (Section 13.3.1), nested patterns — as in Figure 13.11 — 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.

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.
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 tl _ -> case tl of
  | 1 % Var _ -> False -- 1/18
  | 12 % Lam _ _ -> True -- 2/3
  | 1 % App _ _ -> False -- 1/18

Figure 13.11 Expanding case expression with a nested pattern and a wildcard. Comments show the probability of each alternative.

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 13.11, the False branch should have probability $\frac{1}{3}$. It is expanded into four branches, corresponding to subpatterns $\text{Var } _, \text{Lam } _, \text{App (Var } _) _, \text{App (App } _) _$. The latter two are grouped under the pattern $\text{App } _$, 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 nested branches further split that into $\frac{1}{18}$. 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.

13.5 Evaluation

To evaluate the expressiveness and efficiency of Luck’s hybrid approach to test case generation, we tested it with a number of small examples and two significant case
studies: generating well-typed lambda terms and information-flow-control machine states. The Luck code is generally much smaller and cleaner than that of existing handwritten generators, though the Luck interpreter takes longer to generate each example – around $20 \times$ to $24 \times$ for the more complex generators.

**Small Examples** The literature on random generation includes many small examples – list predicates such as `sorted`, `member`, and `distinct`, tree predicates like BSTs (Section 13.2) and red-black trees, and so on. In the extended version we show the implementation of many such examples in Luck, illustrating how to 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 (Runciman et al., 2008). 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 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 Figure 13.12. 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, Luck 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.
13.5 Evaluation

Well-Typed Lambda Terms Using our prototype implementation we reproduced the experiments of Pałka et al. (2011), 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. (2014, 2015) and Fetscher et al. (2015).

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., 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 similar to those found by Pałka et al. and Fetscher et al.. For example, the [Int] → [Int] function

```
seq (id (\a -> seq a id) undefined),
```

when fed the singleton list [undefined], yields an exception immediately with -O0 (following the semantics of seq), but prints the toplevel constructor of the result [ ] before raising the exception if compiled with -O1.

Luck’s generation speed was slower than that of Pałka’s handwritten generator. We generated 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 (Pałka, n.d.).

The derived generators of Claessen et al. (2014) achieved a 7x slowdown compared to the handwritten generator, while the Redex generators (Fetscher et al., 2015) also report a 7× 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 (Hriţcu et al., 2013). 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 $s_1$ and $s_2$ (i.e., all their low-tagged parts are identical) and run each of them to completion, then the resulting states $s_1'$ and $s_2'$ are also indistinguishable.

Hriţcu et al. 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 one must first generate one abstract machine $s$ and then vary $s$, to generate an indistinguishable one $s'$. In writing a generator for variations, one must 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. (2013) 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:

```haskell
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'
```

https://doi.org/10.1017/9781108770750.014 Published online by Cambridge University Press
We evaluated our generator on the same set of buggy information-flow analyses as in (Hrițcu et al., 2013). 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.

This prototype gives the user enough flexibility to achieve effectiveness similar to state-of-the-art generators, while significantly reducing the amount of code and effort required, suggesting that the Luck approach is promising and pointing towards the need for a real, optimizing implementation.

13.6 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; a more comprehensive treatment of related work can be found in the extended version of the paper.

Property-Based Testing The works that are most closely related to our own are the narrowing based approaches of (Gligoric et al., 2010), (Claessen et al., 2014, 2015) and (Fetscher et al., 2015). 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 (Duregård et al., 2012), a tool for functional enumeration of algebraic types, to efficiently generate near-uniform random inputs satisfying some precondition. Fetscher et al. (2015) 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 Pałka et al., 2011) by monomorphizing the polymorphic constants of the context as discussed in the previous section. However, both tools provide limited (locally, or globally, uniform) distribution guarantees, with no user control over the resulting distribution.

Another interesting related approach appears in the inspiring work of Bulwahn (2012b). In the context of Isabelle’s (Nipkow et al., 2002) QuickCheck (Bulwahn, 2012a), Bulwahn automatically constructs enumerators for a given precondition via a compilation to logic programs using mode inference. Lindblad (2007) and (Runciman et al., 2008) also provide support for exhaustive testing using narrowing-based
techniques. Instead of implementing mechanisms that resemble narrowing in standard functional languages, Fischer and Kuchen (Fischer and Kuchen, 2007) leverage the built-in engine of the functional logic programming language Curry (Hanus et al., 1995) to enumerate tests satisfying a coverage criterion. 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.

**Probabilistic programming** Semantics for probabilistic programs share many similarities with the semantics of Luck (Milch et al., 2005; Goodman et al., 2008; Gordon et al., 2014), while the problem of generating satisfying valuations shares similarities with probabilistic sampling (Mansinghka et al., 2009; Łatuszyński et al., 2013; Chaganty et al., 2013; Nori et al., 2014). For example, the semantics of PROB in the recent probabilistic programming survey of Gordon et al. (Gordon et al., 2014) 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 (Nori et al., 2014), 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 (Herbrich et al., 2006) 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 (observe true).

### 13.7 Conclusions and Future Work

In this chapter we introduced 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 and used it to repeat state-of-the-art experiments in random generation. The results showed the potential of Luck’s approach, allowing us to replicate the distribution yielded 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 ways to improve the performance of our interpreted prototype, by compiling Luck into generators in a mainstream language and by experimenting with other domain representations. We also want to investigate Luck’s equational theory, showing that the encoded logical predicates satisfy the usual logical laws. Moreover, the backtracking strategies in our implementation can be abstractly modeled on top of our notion of choice-recording trace; Gallois-Wong (Gallois-Wong, 2016) 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 (Regehr et al., 2012). It would be interesting to see if there is a counterpart to narrowing or constraint solving that allows shrinking to preserve desired properties.

References

References


https://doi.org/10.1017/9781108777075.014 Published online by Cambridge University Press
References


