Making Our Own Luck

A Language For Random Generators

Abstract

QuickCheck-style *property-based random testing* requires efficient generators for well-distributed random data satisfying complex logical predicates. Writing such generators by hand can be difficult and error prone.

We propose a domain-specific language, Luck, in which generators are expressed by decorating predicates with lightweight annotations controlling both the distribution of generated values and the amount of constraint solving that happens before each variable is instantiated. Generators in Luck are compact, readable, and maintainable, with efficiency close to custom handwritten generators. We give a precise denotational semantics for Luck and prove key theorems about its behavior, including the soundness and completeness of random generation with respect to a straightforward predicate semantics. We evaluate Luck on a collection of common examples from the random testing literature and on two significant case studies showing how Luck can be used for complex bug-finding tasks with comparable effectiveness and an order of magnitude reduction in testing code size, compared to handwritten generators.

1. Introduction

Since being popularized by QuickCheck [20], property-based random testing has become a standard technique for improving software quality across a wide variety of programming languages [3, 36, 41, 48] and for streamlining interaction with mechanical proof assistants [7, 17, 24, 47].

To use a property-based random testing tool, the user begins by writing *specifications* in the form of executable predicates. For example, a natural property for a list `reverse` function is that, for any list `l`, reversing `l` twice yields `l` again. In QuickCheck notation:

```
prop_reverse l  =  (reverse (reverse l) == l)
```

To test this property, the tool 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 several property combinators resembling standard logical operators. For example, a property of the form `p ==> q` will be tested automatically by generating *valuations*—i.e., maps assigning random values (of the right type) to the free variables of `p` and `q`—discarding valuations that fail to satisfy `p`, and looking for counterexamples to `q` among the ones that remain. This permits writing executable specifications that closely resemble logical ones.

Unfortunately, QuickCheck users soon discover that this default generate-and-test approach does not always 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 by `p`, so `q` is seldom exercised. Consider, for example, testing a simple property of a school database system: every student in a list `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 there are many possible student ids (e.g., because they are represented as integers), then a randomly generated id is very unlikely to be a member of `registeredStudents`, and thus almost all test cases will be discarded.

To obtain effective testing in such cases, the QuickCheck user must provide a *custom 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 streamlining the construction of such generators. These combinators also allow fine control over the *distribution* of generated values—a crucial feature in practice [20, 35] (for example, swarm testing [33] is about a specific way of changing distributions to find failures faster).

Custom generators work very well for small to medium-sized examples, but they can become challenging to write as `p` becomes more complex, sometimes to the point of becoming a research contribution on their own. For example, papers have been written about how to build QuickCheck-style random generators for well-typed lambda-terms [49] and for indistinguishable machine states suitable for finding bugs in information-flow analyses [35]. Moreover, if we are testing an *invariant* property (such as the fact that types are invariant under reduction for some programming language), then the same predicate will appear in both the precondition and the conclusion of the property, requiring that we write both a predicate `p` and a generator whose outputs all satisfy `p`. These two artifacts must then be kept in sync, which can become both a maintenance issue and a rich source of bugs. (These difficulties are not hypothetical: Hriţcu et al.'s machine-state generator [35] is over 1500 lines of tricky Haskell, while Pałka et al.'s generator for well-typed lambda-terms [49] is over 1600 lines.) 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 in the recent literature—one based on local choices and backtracking, the other on general constraint solving. Our language, Luck, synergistically combines these two approaches.

The first approach can be thought of as a kind of incremental generate-and-test: rather than generating completely random valuations and testing them against `p`, we walk over the structure of `p`, instantiating each variable `x` at the first point where we meet a constraint involving `x`. In the `member` example above, this has the effect of traversing the list of registered students and picking one of its elements: 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 process resembles *needed narrowing* from functional logic programming [2, 41, 56]. It is attractively lightweight, admits natural control over distributions (e.g., by annotating disjunctions and instantiation points), and has been used successfully in challenging domains such as generating well-typed programs for testing compilers [19, 25, 27].

However, there are cases where purely local choices lead to instantiating variables too early, before the constraints on them are known, incurring significant backtracking. For example, consider the `notMember` predicate:

```
notMember x [] = True
notMember x (h:t) = (x /= h) && notMember x t
```

Suppose that we wish to generate values for `x` such that `notMember x l` for some predetermined list `l`. When we encounter the constraint `x /= h`, we will generate a value for `x` that is not equal to the known value `h`. We will then proceed to the recursive call of `notMember`, where we will *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 we find that our choice of `x` was unlucky. If the space of possible values for `x` is, say, only twice as big as the length of `l`, 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 use this approach to generate lists satisfying `distinct`, then `notMember`'s 50% chance of backtracking will be compounded on each recursive call of `distinct`, leading to unacceptably low rates of successful generation.

The second previously explored approach leverages the power of a general constraint solver to generate a diverse set of valuations satisfying a predicate. (Constraint solvers are, of course, also widely applied to directly searching for counterexamples [5, 6, 37, 38, etc.]. We are interested here in the rather different task of quickly generating many diverse inputs, so that we can test systems like compilers whose state spaces are too large to be exhaustively explored.) This requires translating the predicate `p` from its original form—which, when the artifact under test is a functional program, may involve datatypes, pattern matching, recursive functions, etc.—into a form that can be handled by, for instance, an off-the-shelf SMT solver. This approach has been widely investigated, both for generating inputs directly from predicates [14, 32, 39, 54] and for symbolic execution-based testing [4, 10, 28, 55, 57], which additionally uses the system under test to guide generation of inputs that exercise different control-flow paths.

However, the overhead of the constraint solver can make it less efficient than the more lightweight, local approach of needed narrowing in cases when the latter does not lead to backtracking. Even more importantly, the complexity of the translation together with the complex internals of modern constraint solvers makes it hard to give the user predictable control over the distribution of generated valuations. At best, we could try to use an algorithm for SAT witness generation that guarantees uniform (or near uniform) distribution [16], but this is often not the distribution we want in practice (see §7).

The complementary strengths of local instantiation and global constraint solving suggest a hybrid approach, where limited constraint propagation is used to refine the domains of unknown variables before instantiation. As in the local instantiation approach, we sample variables one at a time, giving us a clear means of controlling the distribution in the style of QuickCheck. However, rather than instantiating *every* variable to a random value as soon as the first constraint mentioning it is encountered, we allow the user to specify a more relaxed treatment of some variables and apply lightweight constraint solving to refine the domains from which these variables are sampled.

Exploring this hybrid approach is the goal of this paper. Our main contributions are:

- We propose a domain-specific language, dubbed Luck, for writing generators via lightweight annotations on predicates. Luck combines the strengths of the local random instantiation and constraint-solving approaches to generation. Section 2 illustrates Luck using a simple binary search tree example.
- To place Luck's design on a firm formal foundation—in particular, to clarify the interactions between local instantiation and constraint solving—we introduce a core calculus into which Luck is desugared (Section 3). We give a probabilistic denotational semantics for Core Luck and prove key theorems: the soundness and completeness of the generator semantics with respect to a straightforward boolean predicate interpretation of Core Luck programs, and the fact that delaying variable instantiation reduces backtracking. We also sketch two further extensions of the core, adding datatypes and functions (Section 4).
- We detail the process by which the surface syntax of Luck—in particular, its convenient "delay brackets" and a distribution-aware syntax for nested pattern matching—is translated into Core Luck (Section 5).
- We evaluate Luck's expressiveness and efficiency (Section 6) on a collection of common examples from the random testing literature, using a prototype implementation based on the translation and denotational semantics. Two significant case studies show that Luck can be used to find bugs in an industrial compiler by randomly generating well-typed lambda terms and to help design information-flow abstract machines by generating indistinguishable machine states. Compared to hand-written generators, these experiments demonstrate comparable bug-finding effectiveness (measured in test cases generated per counterexample found) and an order-of-magnitude reduction in the size of testing code. Our current prototype is somewhat slower (time per test) than hand-written generators ($2\times$ to $50\times$), but many opportunities for optimization remain.

Section 7 discusses related work; Section 8 sketches directions for future work.

2. Luck, by example

We begin by introducing the key features of Luck, using binary search trees as a running example.

A recursive predicate that checks whether a given tree satisfies the standard BST invariant might look like this (in, say, Haskell):

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

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
```

Suppose that we want to turn this predicate into a generator. A Luck program that accomplishes this is the following:

```
fun bst size low high tree =
    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
    end
```

Modulo small differences in concrete syntax, the two programs are very similar. The important differences are: (1) the `size` parameter, which is used to annotate the branches of the `case` with relative weights, and (2) the *delay brackets* $[\![\cdot]\!]^x$ around the subexpression `low<x && x<high`. Together, these enable us to give the program both a natural interpretation as a predicate (by erasing annotations) and an efficient interpretation as a generator of well-distributed random trees. For example, evaluating the top-level expression `bst 10 0 42 tree` as a generator, where the free variable `tree` is *unknown*, will yield random binary search trees of size up to 10 with node labels between 0 and 42. (For purposes of this discussion,

we'll take `size`, `low`, and `high` to be fixed inputs, not unknowns to be generated. The case where `low` and `high` are also unknowns is discussed in Section 5. Taking `size` to be unknown doesn't make sense, because it is being used to control the distribution.)

The interpretation of the numeric annotations on the `case` branches is similar to QuickCheck's `frequency` combinator. The `Node` branch will be chosen $\frac{size}{size+1}$ of the time and the `Empty` one only $\frac{1}{size+1}$ of the time. In particular, if the `size` parameter is 0, we will only select the `Empty` branch, ensuring the desired size bound on generated trees. When the `Node` branch is chosen, we introduce new unknowns for the pattern variables `x`, `l` and `r` and we associate with each of them a set of potential values—its *domain*—and evaluate the body of the branch to instantiate them.

Informally, the annotation $[\![\cdot]\!]^{\mathtt{x}}$ delimits an area where constraint solving should be used to refine the set of possible values for `x` before instantiating it. We evaluate $[\![\texttt{low<x \&\& x<high}]\!]^{\mathtt{x}}$ by walking over the subexpression `low<x && x<high` and using the atomic constraints that we meet (`low<x` and `x<high`) to refine the domain of `x`; at the end, the possible values for `x` are those that lie between `low` and `high`. We randomly choose one of these values (with equal probability) and continue with the two recursive calls to `bst`, in which the (now fixed) value of `x` plays the role of either `high` or `low`; these yield values for the subtrees `l` and `r`, completing the generation of the `Node`.

To see why this is useful, suppose we leave off the delay brackets. A local instantiation strategy will then be used by default. However, as with `notMember` in the introduction, this strategy will perform poorly in this example. When the constraint `low < x` is encountered, a value for `x` that is larger than `low` will be generated; if the value of `x` that we generated happens to be greater than `high`, this will lead to backtracking when we reach the constraint `x < high` later on.

Section 5 discusses additional high-level features for automatically adding `size` parameters and for understanding and controlling probability distributions on generated data.

3. Semantics of Core Luck

In the previous section, we presented an example program in source Luck, illustrating syntactic constructs that allow us to improve the efficiency of local instantiation approaches by constraint solving, while controlling the resulting probability distribution. These constructs are translated to primitive operations of Core Luck. This section presents these operations, giving a denotational semantics for incrementally larger subsets of Core Luck. The most important theorems we prove are soundness and completeness: informally, whenever we use a Luck program to generate a valuation, the valuation we generate will always satisfy the predicate semantics (soundness) and we will generate *every* possible satisfying valuation with a non-zero probability (completeness). In addition, we prove that whenever we use instantiation brackets, we risk a higher probability of backtracking (though, of course, the implementation may still perform better despite the extra backtracking because of reduced bookkeeping overheads—this being the whole point of introducing programmer-controlled instantiation). Section 5 deals with the translation from source to core.

Delay brackets are translated to instantiation brackets and fixpoints (we will discuss fixpoints in Section 3.2). *Instantiation brackets* specify instantiation points for variables: for $\{p\}^{\mathtt{x}}$ we instantiate `x` using the information present in its domain *after* processing p, while for ${}^{\mathtt{x}}\{p\}$ we instantiate `x` *before* processing p.[1] Until an instantiation point for an unknown is encountered, we use constraint solving techniques to refine these domains. Our prototype implementation uses an *arc consistency* algorithm reminiscent of AC-4 and its variants [34, 45]; however, in principle any constraint-solving approach that works with per-variable state representations could be used.

In the `low < x && x < high` constraints from the `bst` predicate, we could ask for local instantiation of `x` after the first constraint as follows:

```
{ low<x }^x  && x<high
```

Suppose that the value of `low` is 0, the value of `high` is 10 and the domain associated with `x` is $\{-42..42\}$. When the constraint `low < x` is encountered, the domain of `x` will be refined to $\{1..42\}$. Then we encounter the end of the instantiation bracket for `x`, which generates a concrete value for `x`. For example, let's assume that `x` was instantiated to 5. We would then process the constraint `x < high`, which is a successful *check* that $5 < 42$. If, however, a value for `x` was chosen that was not less than 10, this check would fail and we would need to backtrack, and make another choice for `x`.

Consider now the behavior of the same example, where we have moved the instantiation point of `x` *after* both constraints:

```
{ low<x && x<high }^x
```

With the same initial assumptions as before, after processing the first inequality the domain associated with `x` is again $\{1..42\}$. However, we now encounter the constraint `x < high`, which further refines the domain of `x` to $\{1..9\}$. This time, when the instantiation point is encountered we can choose a value for `x` from a smaller set, and backtracking is no longer needed.

The other syntactic construct that is used in the `bst` example is the optional annotation of the branches of a case statement with weights. As discussed in the introduction, a local instantiation approach lends itself well to distribution control: in Core Luck we assign relative weights to disjunctions and instantiations.

Disjunctions have the form $p_1 \ {}^{w_1}||^{w_2} \ p_2$, assigning a relative weight to each disjunct. We choose the left disjunct p_1 with probability $\frac{w_1}{w_1+w_2}$, and the right one p_2 with $\frac{w_2}{w_1+w_2}$. As in Section 2, the `tree` variable will be instantiated with a `Node` $\frac{size}{size+1}$ of the time and with `Empty` $\frac{1}{size+1}$ of the time. Weighted disjunctions are given semantics in Subsection 3.1, while we will discuss weighted instantiation points in Subsection 4.1.

In the rest of this section we will give denotational semantics to Core Luck, keeping in mind two important goals: First, we need to be able to obtain a view of the semantics of a predicate as a generator that is sound and complete with respect to the standard boolean predicate semantics. The second and trickier part is to be able to differentiate between the semantics of a predicate `p` and a version of it with an instantiation annotation: $\{\mathtt{p}\}^{\mathtt{x}}$. The difference will turn out to be the probability of failure which induces the need to backtrack: intuitively, instantiating some variables early can only increase the need for backtracking.

Section 3.1 introduces a very small subset of Luck with only integral values to gently introduce the reader to the various definitions and provide more intuition for the sections that will follow. Section 3.2 scales the previous subset to the entire subset of Core Luck that deals with integers. It introduces the most interesting primitive Luck constructs in a simple setting. Section 4.1 describes the subset of Core Luck that deals with datatypes, presenting weighted instantiation brackets before dealing with the full complexity of adding functions. Finally, Section 4.2 presents the entirety of Core Luck, including recursive functions.

[1] At this point, a reader might wonder why not just have an "instantiation" construct—e.g. $!u$—and write p `&&` $!u$. The problem is that, in the presence of negation, `not`$(p$ `&&` $!u)$ is not equivalent to `not`$\{p\}^u$, since the conjunction effectively becomes a disjunction.

3.1 A simple fragment

We begin with a tiny subset of Core Luck, to introduce the basic definitions and design choices for our semantics. In the next subsection we extend this language to include instantiation brackets; the next section discusses datatypes and lambdas. To simplify the exposition, the only expressions of the language in this subsection will be bounded precision integers (n) and free variables (u), which we call *unknowns*; predicates are simple boolean expressions over integers. We associate a domain with each unknown, which is gradually refined to a value.

$$e ::= u \mid n$$
$$p ::= \texttt{True} \mid \texttt{False} \mid p \texttt{ \&\& } p \mid p \ {}^{n}|{}|^{n} \ p \mid e \texttt{ == } e$$

The most natural semantics of a QuickCheck-style generator is a probability distribution over the generated type. Probabilistic programs [31] (which share many similarities with Luck) generalize this notion by denoting probability distributions over the free variables contained in the program. In the same spirit, we introduce the notion of valuations:

Definition 3.1.1. A *valuation* is a mapping from unknowns to concrete values. For instance:

$$\{x \mapsto 42, y \mapsto 13, z \mapsto 17\}$$

Given a predicate, we should generate only *satisfying valuations*, i.e., valuations that, when substituted into the predicate, make the standard semantics `True`. Moreover, we would like *every* satisfying valuation to be generatable. These properties reflect the soundness and completeness of our generators, in the spirit of Paraskevopoulou et al. [51].

Ultimately, the meaning of a predicate in Luck is a probability distribution over satisfying valuations. However, unfortunate choices for variables instantiated too early, or wrongly taken branches in disjunctions, can cause generation to fail. We represent this semantically by using *subprobability distributions*, which may sum to a total less than one—the difference represents the probability of failing to generate a value at all. When generation fails, we will backtrack and try again from the beginning, which will eventually succeed as long as the total probability is not zero.

In order to handle the combination of constraint solving and local instantiation in the following subsections, we will use a structured representation of sets of valuations, which we call *candidate spans*.

Definition 3.1.2. A *candidate span* is a map from a collection of unknowns to nonempty finite sets of values. For each unknown, we refer to its mapping in a candidate span as its *domain*. For example:

$$\{x \mapsto \{0, 1\}, y \mapsto \{1\}\}$$

The fact that the variable domains are nonempty is a useful invariant of the semantics, and helps avoid unnecessary computation by detecting failure early. The finiteness of the domains restricts our technique a little, requiring an a-priori choice of the space from which random tests are drawn (such as ranges of integer unknowns), but it is necessary to ensure well-definedness of the semantics. Every candidate span implicitly represents a set of valuations, which we call its cover.

Definition 3.1.3. The *cover* of a candidate span σ is the set of all valuations v (with the same domain) mapping each unknown x in σ to some value in $\sigma[x]$.

$$cover(\sigma) = \{v \mid \forall\, x,\ v(x) \ \in\ \sigma[x]\}$$

For example, the cover of the candidate span above is:

$$\{\{x \mapsto 0, y \mapsto 1\}, \{x \mapsto 1, y \mapsto 1\}\}$$

$$\begin{array}{lcl}
[\![\texttt{True}]\!](\sigma) & = & \{\sigma \mapsto 1\} \\
[\![\texttt{False}]\!](\sigma) & = & \varnothing \\
[\![e_1 \texttt{ == } e_2]\!](\sigma) & = & \texttt{let } s = \sigma[e_1] \cap \sigma[e_2] \texttt{ in} \\
 & & \texttt{if } s = \varnothing \texttt{ then } \varnothing \\
 & & \texttt{else } \{\sigma[e_1, e_2 \leftarrow s] \mapsto 1\} \\
[\![p_1 \texttt{ \&\& } p_2]\!](\sigma) & = & [\![p_1]\!](\sigma) \texttt{ >>= } [\![p_2]\!] \\
[\![p_1 \ {}^{n_1}|{}|^{n_2} \ p_2]\!](\sigma) & = & \mathit{rescale}\ ([\![p_1]\!](\sigma))\ \frac{n_1}{n_1+n_2} + \\
 & & \mathit{rescale}\ ([\![p_2]\!](\sigma))\ \frac{n_2}{n_1+n_2}
\end{array}$$

Figure 1. Semantics of a subset of Luck

Definition 3.1.4. Given two candidate spans σ_1 and σ_2, we say that $\sigma_1 < \sigma_2$ if $cover\ \sigma_1 \subset cover\ \sigma_2$

We also consider the natural lifting of a candidate span to a map from predicates to sets of values. We write $\sigma[e]$ for accesses in this map. We will also *update* spans by binding new sets to an *expression*, writing $\sigma[e \leftarrow set]$; this is just a map update if e is an unknown, but if e is an integer n, then this is just the original σ, provided set is the singleton set $\{n\}$. We will ensure that we only write $\sigma[e \leftarrow set]$ when $\emptyset \neq set \subseteq \sigma[e]$, so that the update is always defined.

We give the semantics in terms of *refiners*—functions that, given a candidate span as an input, return a subprobability distribution over candidate spans.

$$[\![p]\!] :: Span \to \underline{\Pr}(Span)$$

Refiners can, in turn, be naturally lifted to transformations between subprobability distributions over valuations.

Given an initial candidate span, mapping each of the unknowns appearing free in a predicate p to some finite set of possible values, we apply the semantics of p to the span to get a resulting refined subprobability distribution on candidate spans. This induces a probability distribution over valuations from which we can sample. The semantics are given in Figure 1. We write $\{x_1 \mapsto p_1, \cdots, x_n \mapsto p_n\}$, where $0 \leq \sum p_i \leq 1$ for the subprobability distribution that contains spans $x_1, \cdots, x_n$ in its support, each one with probability p_i. We write $\varnothing$ for the trivial subprobability distribution with empty support. Given a subprobability distribution pr, $rescale\ pr\ p$ multiplies the probability of every element in the support of pr by p. Given subprobability distributions pr_1 and pr_2, we write $pr_1 + pr_2$ for the subprobability distribution that assigns a probability of $pr_1(x) + pr_2(x)$ to each element x of the *combined* support of pr_1 and pr_2 (we only use this notation when the total probability is still ≤ 1). We write $\Sigma\ prs$ for the sum of a finite set of subprobability distributions.

The semantics of `True` is the function that, given a candidate span σ, returns the singleton probability distribution where σ has probability 1. The semantics of `False` is the constant function returning the empty probability distribution. For equality constraints e_1 `==` e_2, we remove from the input span σ all valuations that we know cannot satisfy the constraint, by restricting both e_1 and e_2 to values in the intersection of $\sigma[e_1]$ and $\sigma[e_2]$. We ensure the update is defined by checking for an empty intersection; in our implementation this check allows us to avoid unnecessary computation.

As a concrete example of equality constraints in context, consider each conjunct of the following:

```
x == 0 && y == x
```

Given an input span $\{\texttt{x} \mapsto \{0..10\}, \texttt{y} \mapsto \{-5..5\}\}$, the first constraint refines this to $\{\texttt{x} \mapsto \{0\}, \texttt{y} \mapsto \{-5..5\}\}$. If we applied the second to the same starting span, it would refine it to $\{\texttt{x} \mapsto \{0..5\}, \texttt{y} \mapsto \{0..5\}\}$. But conjunction does not apply the two refiners independently: it propagates the refinements made by the first conjunct into the input to the second, resulting in a subprobabil-

ity distribution containing a single refined span, $\{\texttt{x} \mapsto \{0\}, \texttt{y} \mapsto \{0\}\}$.

In general, conjunction can be expressed as a monadic *bind* of the semantics of the two conjuncts in the (sub)probability monad. Bind is a binary operation that given a distribution on candidate spans m and a function f that assigns a distribution to every span, creates a new distribution on spans in a natural way: we first select a span σ based on the distribution m and then select from the distribution $(f\ \sigma)$.

$$\begin{array}{lcl} \texttt{>>=} & :: & \underline{\Pr}(Span) \to (Span \to \underline{\Pr}(Span)) \to \underline{\Pr}(Span) \\ m \texttt{>>=} f & = & \Sigma\, \{y \mapsto q \cdot p \mid (x,p) \in m, (y,q) \in (f\ x)\} \end{array}$$

This handling of conjunctions introduces an asymmetry between the two conjuncts: the choices in the first conjunct affect choices in the second, but not vice versa. We will alleviate this asymmetry in the next subsection.

Finally, in this simple subset, disjunction is the only construct that introduces a probabilistic choice. Given a disjunction $p_1\ {}^{n_1}||^{n_2}\ p_2$, we either choose p_1 with probability $\frac{n_1}{n_1+n_2}$, or p_2 with the complementary probability.

Properties The key properties of this semantics are *decreasingness*, *completeness*, and *soundness*; we defer formal statements and proofs to the next subsection. Intuitively, during the generation process we filter the domains associated with each unknown, thus refining the space of candidate valuations. We never add new valuations to a span (decreasingness), we only remove valuations that cannot satisfy the predicate (completeness), and if we run our semantics long enough then all the candidate spans that are left will contain at least one satisfying valuation (soundness). When we add instantiation brackets in the next subsection, we will obtain a slightly stronger definition of soundness that concerns distributions on satisfying valuations directly.

3.2 Adding instantiating brackets and fixpoints

We next extend the subset of the previous section to include instantiation annotations and fixpoints, allowing us to explore their interactions with constraint solving. The expressions are again just integer constants and unknowns, but predicates now also include negation, more integer comparisons, instantiation brackets, and a fixpoint construct. For simplicity, we don't discuss weighted instantiation brackets yet, assuming instead that sampling is based on a uniform distribution on the associated domain; they are discussed in Section 4.1.

$$\begin{array}{rcl} e & ::= & u \mid n \\ p & ::= & \texttt{True} \mid \texttt{False} \mid p\ \texttt{\&\&}\ p \mid p\ {}^{n_1}||^{n_2}\ p \\ & \mid & \texttt{not}\ p \mid e\ \texttt{==}\ e \mid e\ \texttt{/=}\ e \mid e\ \texttt{<}\ e \mid e\ \texttt{<=}\ e \mid e\ \texttt{>}\ e \mid e\ \texttt{>=}\ e \\ & \mid & \texttt{Fix}\ p \mid \{p\}^u \mid {}^u\{p\} \end{array}$$

Fixpoints are introduced to address the asymmetry in the handling of conjunctions (not to allow recursive function definitions in Luck, those will be introduced in Subsection 4.2). Informally, `Fix` p takes the fixpoint of the semantic function of p; we keep running $[\![p]\!]$, getting new subprobability distributions on candidate spans each time, until the final transformation leaves the input span unchanged.

Fixpoints introduce an interesting problem when combined with disjunction (and, later, other decision-making constructs such as pattern matching). This is best demonstrated via an example:

```
Fix (x == 0 || x == 1)
```

Given an initial span $\{x \mapsto \{0 \cdots 10\}\}$, the semantics of the disjunction produces the distribution

$$\{\{x \mapsto \{0\}\} \mapsto \tfrac{1}{2}, \{x \mapsto \{1\}\} \mapsto \tfrac{1}{2}\}$$

$$\begin{array}{lcl} [\![\texttt{True}]\!](\sigma) & = & \{(\sigma, \texttt{True}) \mapsto 1\} \\ [\![\texttt{False}]\!](\sigma) & = & \varnothing \\ [\![e_1\ \texttt{==}\ e_2]\!](\sigma) & = & \texttt{let}\ s = \sigma[e_1] \cap \sigma[e_1]\ \texttt{in} \\ & & \texttt{if}\ s = \varnothing\ \texttt{then}\ \varnothing \\ & & \texttt{else}\ \{(\sigma[e_1, e_2 \leftarrow s], e_1\ \texttt{==}\ e_2) \mapsto 1\} \\ [\![p_1\ {}^{n_1}||^{n_2}\ p_2]\!](\sigma) & = & \mathit{rescale}\ ([\![p_1]\!](\sigma))\ \frac{n_1}{n_1+n_2} + \\ & & \mathit{rescale}\ ([\![p_2]\!](\sigma))\ \frac{n_2}{n_1+n_2} \\ [\![p_1\ \texttt{\&\&}\ p_2]\!](\sigma) & = & [\![p_1]\!]\sigma\ \texttt{>>=}\ \lambda(\sigma', p_1'). \\ & & [\![p_2]\!]\sigma'\ \texttt{>>=}\ \lambda(\sigma'', p_2'). \\ & & \{(\sigma'', p_1'\ \texttt{\&\&}\ p_2') \mapsto 1\} \end{array}$$

Figure 2. Updated semantics of Core Luck

with two spans in its support—operationally, it makes a choice. But if we apply the same semantic function *again* to each span, we get

$$\{\{x \mapsto \{0\}\} \mapsto \tfrac{1}{4}, \{x \mapsto \{1\}\} \mapsto \tfrac{1}{4}\}$$

with a total probability of less than one, which arises because the second application may "make a different choice" to the first! This is pointless, and risks leading to much unnecessary backtracking in an implementation. Having made a choice once, of course we want to stick to it while computing a fixpoint.

To do so, we need to keep track of the choices we make during generation. Our refiners now become functions that, given a span as argument, return a subprobability distribution on a pair of a span and a *residue*, a new predicate reflecting the choices made:[2]

$$[\![p]\!] :: Span \to \underline{\Pr}(Span, p)$$

Figure 2 contains the semantics of the constructs of the previous section, modified to reflect the type change. In the case of `True` the residue is trivially `True`; in the case of `False` there is no residue since the result is always $\varnothing$. To handle a conjunction we run the semantics of the first conjunct, getting a span and residue. Then we run the semantics of the second conjunct on the new span, getting another new span and residue. The result is the singleton distribution containing the latter span, together with a conjunction of the residues, with probability 1. The asymmetry we described in the previous section is still present, but it can now be eliminated explicitly by wrapping conjunctions in a fixpoint construct. The semantics of a disjunction $p_1\ {}^{\cdot}||^{\cdot}\ p_2$ looks the same as before, but actually "makes the choice" between the disjuncts only once, since the residues in the result are residues of p_1 or of p_2.

Equality handling remains virtually unchanged from the original version, returning itself as the residue. Figure 3 shows the semantics of the rest of the binary constructs of integer Luck. For disequality, we can only drop values from either domain if at least one of the predicates corresponds to a singleton set. If both predicates are represented in the span by singletons, then the inequality is basically just a check. If one is a singleton we can remove it from the other's span. Ordering comparisons behave similarly. If we want the set represented by e_1 in the span to be smaller than the one by e_2, e_1 can't contain elements that are bigger than the biggest element of e_2. Similarly the other way around.

Figure 4 shows the semantics of the two most interesting constructs we have added. For instantiation brackets, we run the semantics of the predicate and pick a value for the unknown from its domain. In later sections we will allow weighted instantiation brackets, where picking the value uses a distribution that can be chosen by the user; here, for simplicity, we suppose a uniform one.

[2] Alternatively, we could return residues in the form of functions: $[\![p]\!] :: \mu\alpha.\ Span \to \underline{\Pr}(Span, \alpha)$. This representation may be a good choice for implementations, since it saves the cost of re-interpreting the residue, but keeping things syntactic simplifies proofs.

$$
\begin{array}{lcl}
[\![e_1 \texttt{ /= } e_2]\!](\sigma) & = & \texttt{if } \sigma[e1] = \{n_1\} \texttt{ then} \\
& & \ \texttt{if } \sigma[e2] = \{n_2\} \texttt{ then} \\
& & \ \ \texttt{if } n1 = n2 \texttt{ then } \varnothing \\
& & \ \ \texttt{else } \{(\sigma, e_1 \texttt{ /= } e_2) \mapsto 1\} \\
& & \ \texttt{else} \\
& & \ \ \{(\sigma[e_2 \leftarrow \sigma[e_2] \setminus \{n_1\}], e_1 \texttt{ /= } e_2) \mapsto 1\} \\
& & \texttt{else} \\
& & \ \texttt{if } \sigma[e2] = \{n_2\} \texttt{ then} \\
& & \ \ \{(\sigma[e_1 \leftarrow \sigma[e_1] \setminus \{n_2\}], e_1 \texttt{ /= } e_2) \mapsto 1\} \\
& & \ \texttt{else } \{(\sigma, e_1 \texttt{ /= } e_2) \mapsto 1\} \\
[\![e_1 \texttt{ < } e_2]\!](\sigma) & = & \texttt{let } s_1 = \{x \in \sigma[e_1] \mid x < \texttt{max}(\sigma[e_2])\} \texttt{ in} \\
& & \texttt{let } s_2 = \{x \in \sigma[e_2] \mid \texttt{min}(\sigma[e_1]) < x\} \texttt{ in} \\
& & \texttt{if } s_1 = \varnothing \vee s_2 = \varnothing \texttt{ then } \varnothing \\
& & \texttt{else} \\
& & \ \{(\sigma[e_1 \leftarrow s_1][e_2 \leftarrow s_2], e_1 \texttt{ < } e_2) \mapsto 1\} \\
[\![e_1 \texttt{ <= } e_2]\!](\sigma) & = & \texttt{let } s_1 = \{x \in \sigma[e_1] \mid x \leq \texttt{max}(\sigma[e_2])\} \texttt{ in} \\
& & \texttt{let } s_2 = \{x \in \sigma[e_2] \mid \texttt{min}(\sigma[e_1]) \leq x\} \texttt{ in} \\
& & \texttt{if } s_1 = \varnothing \vee s_2 = \varnothing \texttt{ then } \varnothing \\
& & \texttt{else} \\
& & \ \{(\sigma[e_1 \leftarrow s_1][e_2 \leftarrow s_2], e_1 \texttt{ <= } e_2) \mapsto 1\} \\
[\![e_1 \texttt{ > } e_2]\!](\sigma) & = & \texttt{let } s_1 = \{x \in \sigma[e_1] \mid x > \texttt{min}(\sigma[e_2])\} \texttt{ in} \\
& & \texttt{let } s_2 = \{x \in \sigma[e_2] \mid \texttt{max}(\sigma[e_1]) > x\} \texttt{ in} \\
& & \texttt{if } s_1 = \varnothing \vee s_2 = \varnothing \texttt{ then } \varnothing \\
& & \texttt{else} \\
& & \ \{(\sigma[e_1 \leftarrow s_1][e_2 \leftarrow s_2], e_1 \texttt{ > } e_2) \mapsto 1\} \\
[\![e_1 \texttt{ >= } e_2]\!](\sigma) & = & \texttt{let } s_1 = \{x \in \sigma[e_1] \mid x \geq \texttt{min}(\sigma[e_2])\} \texttt{ in} \\
& & \texttt{let } s_2 = \{x \in \sigma[e_2] \mid \texttt{max}(\sigma[e_1]) \geq x\} \texttt{ in} \\
& & \texttt{if } s_1 = \varnothing \vee s_2 = \varnothing \texttt{ then } \varnothing \\
& & \texttt{else} \\
& & \ \{(\sigma[e_1 \leftarrow s_1][e_2 \leftarrow s_2], e_1 \texttt{ >= } e_2) \mapsto 1\}
\end{array}
$$

Figure 3. Semantics of the rest of the binary constraints

$$
\begin{array}{lcl}
[\![\{p\}^u]\!](\sigma) & = & [\![p]\!](\sigma) \texttt{ >>= } \lambda(\sigma', p'). \\
& & \mathit{partition}\ \sigma'[u] \texttt{ >>= } \lambda v_u. \\
& & \{(\sigma'[u \leftarrow v_u], p') \mapsto 1\} \\
[\![^u\{p\}]\!](\sigma) & = & \mathit{partition}\ \sigma[u] \texttt{ >>= } \lambda v_u. \\
& & [\![p]\!](\sigma[u \leftarrow v_u]) \\
[\![\texttt{Fix } p]\!](\sigma) & = & [\![p]\!](\sigma) \texttt{ >>= } \lambda(\sigma', p'). \\
& & \texttt{if } \sigma' < \sigma \texttt{ then } [\![\texttt{Fix } p']\!](\sigma') \\
& & \texttt{else } \{(\sigma', p') \mapsto 1\}
\end{array}
$$

Figure 4. Instantiation brackets and fixpoints

$$
\begin{array}{lcl}
[\![\texttt{not } p]\!] & = & [\![p]\!]^- \\
\\
[\![\texttt{True}]\!]^- & = & [\![\texttt{False}]\!] \\
[\![\texttt{False}]\!]^- & = & [\![\texttt{True}]\!] \\
[\![p_1\ ^{n_1}\texttt{\&\&}^{n_2}\ p_2]\!]^- & = & [\![(\texttt{not } p_1)\ ^{n_1}\texttt{||}^{n_2}\ (\texttt{not } p_2)]\!] \\
[\![p_1\ ^{n_1}\texttt{||}^{n_2}\ p_2]\!]^- & = & [\![(\texttt{not } p_1)\ ^{n_1}\texttt{\&\&}^{n_2}\ (\texttt{not } p_2)]\!] \\
[\![\texttt{not } p]\!]^- & = & [\![p]\!] \\
[\![e_1 \texttt{ == } e_2]\!]^- & = & [\![e_1 \texttt{ /= } e_2]\!] \\
\cdots & & \cdots \\
[\![\{p\}^u]\!]^- & = & [\![\{\texttt{not } p\}^u]\!] \\
[\![\texttt{Fix } p]\!]^- & = & [\![\texttt{Fix } (\texttt{not } p)]\!]
\end{array}
$$

Figure 5. Negation

Fixpoint constructs keep refining the initial span using the inner predicate p's semantics, until no further refinement occurs. Termination will be guaranteed by the combination of Theorem 3.2.2 and the fact that our domains are finite.

Negation is easy to handle using De Morgan's laws. We define a different, negative, version of the underlying semantics $[\![\cdot]\!]^-$ based on the already presented ones. Every conjunction is also associated with weights, whose only purpose is to serve as weights when negated.[3] This is shown in Figure 5. In the rest of the paper, we don't show these weights for conjunctions, to simplify the presentation.

Properties The main properties of this system are: decreasingness, and correctness (completeness and soundness) with respect to the standard predicate semantics. We now state these properties carefully and give full proofs. We also give a sanity-check theorem on instantiating brackets: if we add brackets to a predicate, the only possible effect on its semantics is to increase backtracking. In all of these proofs, we assume that weights in distribution annotations are strictly positive.

For the proofs, we will need a well-founded ordering on predicates. The details of this ordering require a bit of care. While one would expect a simple subexpression ordering to work—and in fact it does allow most of the proofs to go through—it doesn't quite work because of our treatment of negation. In order for induction to succeed for the negation case, we need $\texttt{not}(p_1\ \texttt{\&\&}\ p_2)$ to be considered a "bigger" predicate than $(\texttt{not}\ p_1)\ \texttt{||}\ (\texttt{not}\ p_2)$, etc. This observation leads to the following ordering on predicates:

Definition 3.2.1. We say p_1 is smaller than p_2 if the height of (the syntax tree representing) p_1 is smaller than the corresponding height of p_2. This means that any predicate that is a subexpression of another is smaller in our ordering. In addition, to handle negation, if p_1 and p_2 share the same height, the smaller predicate is the one whose highest negation is deeper in the tree, predicates without `not` being the smallest possible predicate for a given height.

Theorem 3.2.2 (Decreasingness). Given a span σ and a predicate p, all of the spans σ' in the support of $[\![p]\!]\sigma$ are smaller (but not strictly smaller) than σ.

Proof: The proof proceeds by well-founded induction on the ordering of the spans and then nested induction on the well-founded ordering of the predicates. We then proceed by case analysis on the given predicate:

Case `True`: The resulting subprobability distribution contains only one span, which is equal to the input one.

Case `False`: The resulting distribution is empty and the result holds vacuously.

Case `==`, `/=`, `<`, `<=`, `>`, `>=` : The resulting distribution is either empty or contains just one span, which agrees with the input span on all bindings except two that have possibly been updated to a set that is a subset of the original.

Case p_1 `&&` p_2 : We can apply the inductive hypothesis on σ and p_1, (which is strictly smaller than p) to learn that all the spans in $[\![p_1]\!]\sigma$ are less than or equal to the input span. Let σ_1 be a span in the above distribution. We can now apply the inductive hypothesis on σ_1 and p_2 (recalling that either σ_1 is strictly smaller than σ or else they are equal, in which case we note that p_2 is strictly smaller that p) to learn that every span in $[\![p_2]\!]\sigma_1$ is less than or equal to σ_1, hence less than or equal to σ.

Case $p_1\ ^{w_1}\texttt{||}^{w_2}\ p_2$: We can apply the inductive hypothesis on σ and p_1 and on σ and p_2 (noting that both of the predicates are strictly smaller than p) to learn that all the spans in $[\![p_1]\!]\sigma$ and $[\![p_2]\!]\sigma$ are less or equal than the input span; these are exactly the spans that are present in the returned distribution.

Case `not` p_1 : Follows directly from the inductive hypothesis applied to the input span and the negated predicate (noting that this is always strictly smaller than the given one).

Case `Fix` p_1 : We can apply the inductive hypothesis on σ and p_1 (noting that p_1 is strictly smaller than p) to learn that all the spans

[3] The conjunction weights could potentially have a meaning: Choosing which conjunct to satisfy first. However, it is not clear that the same weights should be used in both conjunction and its negation.

in $[\![p_1]\!]\sigma$ are less than or equal to the input span. Let σ_1 be a span in the above distribution. If σ_1 is equal to the input one then we add it to the returned distribution, satisfying the requirements of the theorem. Otherwise we add $[\![\texttt{Fix}\ p_1]\!]\sigma_1$ to the returned distribution. Since σ_1 is strictly less than σ, we can apply the inductive hypothesis to show that every span in the above distribution is less than σ_1.

Case $\{p_1\}^u$: We can apply the inductive hypothesis on σ and p_1 (noting that p_1 is strictly smaller than p) to learn that all the spans in $[\![p_1]\!]\sigma$ are less or equal than the input span. Let σ_1 be a span in this distribution and let $s = \sigma_1[u]$. We *partition* s, obtaining different possible values v_u for u. For each such v_u, we add to the output distribution the span $\sigma_1[u \leftarrow v_u]$, which is less than or equal to σ_1.

Case ${}^u\{p_1\}$ Similarly with the post-instantiation brackets, let $s = \sigma[u]$. We first *partition* s, obtaining different possible values v_u for u. For each such v_u, we add to the output distribution the span $\sigma[u \leftarrow v_u]$, which is less than or equal to σ. For every span σ_1 in the resulting distribution, we apply the inductive hypothesis on p_1, which concludes the proof. □

Theorem 3.2.3 (**Completeness**). Let σ be a span, p a predicate, and v a valuation in $cover(\sigma)$. If v is not present in the cover of any of the spans in the support of $[\![p]\!]\sigma$, then it does not satisfy p.

Proof: In order to handle the `Fix` case, we prove two statements simultaneously: (1) If v is not in any of the covers of the spans σ' in the support of $[\![p]\!]\sigma$, then it does not satisfy p. (2) If v is not in any of the covers of the spans σ' in the support of $[\![\texttt{Fix}\ p]\!]\sigma$, then v does not satisfy p. The proof proceeds by well-founded induction on the ordering of spans, with a nested induction on the well-founded ordering of predicates. We then proceed by case analysis on the given predicate:

Case `True`: (1) The result holds vacuously as no valuations are dropped. (2) Since $[\![\texttt{Fix}\ \texttt{True}]\!]\sigma$ is equal to $[\![\texttt{True}]\!]\sigma$, the argument is the same.

Case `False`: (1) The result holds vacuously as the resulting distribution is empty. (2) Since $[\![\texttt{Fix}\ \texttt{False}]\!]\sigma$ is equal to $[\![\texttt{False}]\!]\sigma$, the argument is the same.

Case `==` (1) Since v does not belong to the resulting distribution, it must be that case that $v[e_1] \notin \sigma[e_1] \cap \sigma[e_2]$ and $v[e_2] \notin \sigma[e_1] \cap \sigma[e_2]$. As a result, $e_1 == e_2$ is false in v. (2) $[\![\texttt{Fix}\ (e_1 == e_2)]\!]\sigma = [\![e_1 == e_2]\!]\sigma$.

Case `/=` (1) If the resulting distribution is empty, then both $\sigma[e_1]$ and $\sigma[e_2]$ necessarily map to singleton sets $\{n_1\}, \{n_2\}$ respectively. Since the result is empty $n_1 \neq n_2$ and there is no satisfying valuation. The only two cases when the distribution is non-empty and it drops valuations from the initial span are when exactly one of $\sigma[e_1]$ and $\sigma[e_2]$ is a singleton. The two cases are symmetric; consider the case when $\sigma[e_1] = \{n_1\}$. For all the valuations v that are in $cover(\sigma)$ we have $v[e_1] = n_1$. Also, for all the valuations v that are dropped, we have $v[e_2] = n_1$. Thus, the valuations that are dropped do not satisfy the original predicate.
(2) $[\![\texttt{Fix}\ (e_1\ \texttt{/=}\ e_2)]\!]\sigma = [\![e_1\ \texttt{/=}\ e_2]\!]\sigma$.

Case `<`, `<=`, `>`, `>=` : (1) The proofs of these cases are similar, so we will only consider `<`. If a valuation is dropped then it is either the case that $v[e_1] \geq \texttt{max}(\sigma[e_2])$ or $v[e_2] \leq \texttt{min}(\sigma[e_1])$. In both cases, this valuation does not satisfy the original predicate. (2) Unrolling the definition of `Fix`, we can see that any possible recursive call to the semantics of `Fix` will be with a span σ' smaller than σ and residue the inequality itself. The result follows from the inductive hypothesis.

Case p_1 `&&` p_2: (1) Running the semantics of p_1 on the input span σ yields a distribution on pairs of spans and residues. For every such pair (σ', p_1'), we know that no valuation in $cover(\sigma) \backslash cover(\sigma')$ satisfies p_1, by the induction hypothesis. Therefore it doesn't satisfy p either. Similarly, running the semantics of p_2 on σ' yields a probability distribution on pairs of spans σ'' and residues p_2'. By the induction hypothesis, anything removed from σ' during this process doesn't satisfy p_2, and therefore also not p. (2) Unrolling the definition of `Fix`, after running the semantics of p_1 `&&` p_2 we get a probability distribution on pairs of spans σ' and residues p'. If $\sigma' = \sigma$, then `Fix` will not make a recursive call and the conclusion follows. If however, $\sigma' < \sigma$, then we can apply the inductive hypothesis to the span σ' and the predicate `Fix`p' since $\sigma' < \sigma$.

Case $p_1\ {}^{w_1}||^{w_2}\ p_2$: (1) From the inductive hypothesis, we know that valuations v not in the covers of the spans in the support of $[\![p_1]\!]\sigma$ or $[\![p_2]\!]\sigma$ do not satisfy p_1 or p_2, respectively. If v is removed from the spans of both distributions (as it must be, to fail to appear in the final resulting distribution), then it doesn't satisfy either p_1 and p_2, meaning it doesn't satisfy p. (2) Unrolling the definition of the semantics for `Fix`, after running the semantics of p we get a distribution on pairs of spans σ' and residues p', where p' is the residue of either p_1 or p_2. Without loss of generality, assume p' is a residue of p_1. If $\sigma' = \sigma$, then there will be no recursive call, and there is nothing to prove. Assume $\sigma' < \sigma$ and let v dropped from σ' during the recursive call; by the inductive hypothesis, v will not be satisfying for p_1. If v is not satisfying for p_2 as well, then it is not satisfying for the original p as well. If v is satisfying for p_2 then by the induction hypothesis, it wouldn't be dropped from $[\![p_2]\!]\sigma$, which is a subdistribution of $[\![p_1\ {}^{w_1}||^{w_2}\ p_2]\!]\sigma$.

Case `not` p (1) From the inductive hypothesis applied on the input span and the negated predicate p^-, which is always strictly smaller than the input one, we get that every valuation dropped by $[\![p^-]\!]\sigma$, which by definition is equal to $[\![\texttt{not}\ p]\!]\sigma$, is not satisfying. We can easily prove that the standard boolean semantics of p^- are equal to those of `not` p, and the result follows immediately. (2) We can easily prove that the semantics of $[\![\texttt{Fix}\ (\texttt{not}\ p)\ ]\!]\sigma$ coincide with $[\![\texttt{Fix}\ p^-\ ]\!]\sigma$. The result then follows from the induction hypothesis, as above.

Case `Fix` p: (1) Follows directly from the second inductive hypothesis. (2) For all the spans σ' and residues p' in the support of $[\![\texttt{Fix}\ p]\!]\sigma$ we know that $[\![\texttt{Fix}\ p']\!]\sigma' = \{(\sigma', p') \mapsto 1\}$. Therefore $[\![\texttt{Fix}(\texttt{Fix}\ p)]\!] = [\![\texttt{Fix}\ p]\!]$ and the conclusion follows as before.

Case $\{p\}^u$, ${}^u\{p\}$: Since instantiation brackets do not remove valuations from the input span, the conclusion follows from the inductive hypothesis on p. □

For soundness, we get an intuitive guarantee: given a predicate p, for every non-instantiated unknown u present in p we can run the semantics of p until we reach a fixpoint, instantiate u and continue with the rest. This process leaves only satisfying valuations in the resulting probability space. First, a simple observation about residues.

Lemma 3.2.4. Given a predicate p and a span σ, all valuations satisfying a residue in $[\![p]\!]\sigma$ also satisfy p.

Proof: Straightforward induction on p. □

Theorem 3.2.5 (**Soundness**). Given a predicate p and a span σ, let $u_1, \ldots, u_n$ be the unknowns that appear free in p and whose binding in σ is a non-singleton set. Then all of the valuations in all of the spans in the support of $[\![\texttt{Fix}\{\texttt{Fix}\ldots\{\texttt{Fix}\ p\}^{u_1}\ldots\}^{u_n}]\!]\sigma$ are satisfying.

Proof: We unroll the definition of the semantics for `Fix`, applying $[\![\{\texttt{Fix}\ldots\{\texttt{Fix}\ p\}^{u_1}\ldots\}^{u_n}]\!]$ on the input span σ, which yields a subprobability distribution over pairs of spans σ' and residues p'. Because our semantics is decreasing and the semantics of instantiation brackets set the binding of an unknown u to a singleton set, σ' can only contain a single valuation v. We will then apply $[\![\texttt{Fix}\ p']\!]$ to σ'.

Using Lemma 3.2.4, we can see that, if a valuation v satisfies `Fix` p', then it will also satisfy $[\![\{\texttt{Fix}\ldots\{\texttt{Fix}\ p\}^{u_1}\ldots\}^{u_n}]\!]$, and

therefore also the original predicate p. It thus suffices to show that, for any predicate p, if σ is a span containing a singleton valuation v, then either $[\![p]\!]\sigma$ will be empty or else the only spans in its support is σ itself and v is a satisfying valuation for p. We proceed by induction on the well-founded ordering on the predicate p.

Case `True`: The result holds trivially since all valuations satisfy `True`.

Case `False`: The result hold vacuously since the resulting distribution is empty.

Case `==`, `/=`, `<`, ... Since σ only contains the valuation v in its cover, there exist n_1, n_2 such that $\sigma[e_1] = \{n_1\}$ and $\sigma[e_2] = \{n_2\}$ Then the semantics of every binary constraint *op* reduces to a simple check: If n_1 *op* n_2, then the resulting distribution contains only σ in its support and v is satisfying. Otherwise the result holds trivially since the resulting distribution is empty.

Case p_1 `&&` p_2: By the inductive hypothesis, either $[\![p_1]\!]\sigma$ is empty, or all spans in its support are equal to σ. Similarly, either $[\![p_2]\!]\sigma$ is empty, or spans in its support are equal to σ. Since conjunction acts as a bind in the subprobability monad, the result follows.

Case $p_1 \ {}^{w_1}||^{w_2} \ p_2$: By the inductive hypothesis, either $[\![p_1]\!]\sigma$ is empty, or spans in its support are equal to σ. Similarly, either $[\![p_2]\!]\sigma$ is empty, or spans in its support are equal to σ. The distribution $[\![p_1 \ {}^{w_1}||^{w_2} \ p_2]\!]\sigma$ will consist of the combination of $[\![p_1]\!]\sigma$ and $[\![p_2]\!]\sigma$.

Case `not` p Since the negated predicate is always strictly smaller than p, the result follows directly from the inductive hypothesis.

Case `Fix` p: Since σ consists of a single valuation, $[\![p]\!]\sigma$ will either be empty, in which case the theorem holds vacuously, or, by the inductive hypothesis, it will only contain spans equal to σ in its support and v is satisfying. In this case, the fixpoint will not make a recursive call and the result follows.

Case $\{p\}^u, {}^u\{p\}$: Since the input span σ contains singleton bindings for all unknowns, instantiation brackets leave it unaffected. The conclusion follows from the inductive hypothesis on p. □

Luck programs in this fragment can be viewed as standard predicates by ignoring the fix and instantiation constructs. Given a predicate p and a valuation v, we can derive a boolean function from the refiner semantics: We apply $[\![p]\!]$ to the span that contains only v in its cover. If the result is an empty distribution we return `False`. Otherwise we return `True`.

Corollary 3.2.6. The boolean function described above agrees with the standard predicate semantics on all valuations.

Proof: Follows from the proof of soundness (3.2.5).

In addition to the soundness and completeness results, now that we have added explicit instantiation constructs we can quantify their effect when used. Intuitively, even though instantiating early can sometimes be much faster because it avoids carrying along heavy representations of constrained domains, it can also lead to more backtracking than necessary.

Consider a predicate `p && x < 5`, where `p` is some sub-predicate containing the unknown `x`. Suppose that, after running the semantics of `p` on some input span, we get a subprobability distribution `S` containing a span σ with probability k. Assume that σ maps `x` to the set `{0..9}`. The semantics for conjunction will next refine σ using the second conjunct, `x < 5`. This filters out the values greater than five, transforming σ to a span σ' in which `x` is bound to `{0..4}`. The probability of that span in the resulting distribution is still k—the probability mass corresponding to the values `{5..9}` in σ has been redistributed among the remaining possibilities.

By contrast, consider the predicate $\{\texttt{p}\}^{\texttt{x}}$ `&& x < 5`, where instantiating brackets have been wrapped around `p`. In this case, starting from the same assumptions, after passing the input span to the semantics of `p`, we will split the resulting span σ into ten smaller ones with probability $\frac{k}{10}$, each containing a singleton binding for `x` to one of the values from `0` to `9`. Each of these spans will be passed separately to the second conjunct. Half of them will lead to an unsatisfiable constraint, meaning the final probability of successfully generating a value is only $\frac{k}{2}$ (and the probability of needing to backtrack is increased by $\frac{k}{2}$).

This example can be formalized and generalized as follows:

Theorem 3.2.7 (Effect of instantiation brackets). Let p be a predicate containing p' as a subpredicate, and let p'' be $p[\{p'\}^u/p']$, where p' is replaced by itself wrapped in instantiation brackets, for some unknown u appearing in p'. Then, for any span σ, the total probability of all the refinements in $[\![p]\!]\sigma$ is at least as much as the total probability in $[\![p'']\!]\sigma$.

For the proof, we will first need a few auxiliary definitions and lemmas.

Definition 3.2.8. A *candidate space* is a subprobability distribution over pairs of candidate spans and predicates (residues).

We can view candidate spaces as sets of triples (candidate spans, residues, and probabilities) where the sum of all the probabilities is less than or equal to one. Adopting this view, we can define the following ordering:

Definition 3.2.9. We will say that for two candidate spaces S_1, S_2, $S_1 \leq S_2$ if there exists a partitioning of the two spaces into collections of triples $\cup_{i=1}^{n}(\sigma_{1i}, p_{1i}, q_{1i}) = S_1, \cup_{i=1}^{n}(\sigma_{2i}, p_{2i}, q_{2i}) = S_2, n \leq m$ such that for every pair of triples $(\sigma_{1i}, p_{1i}, q_{1i}) \in S_1$ and $(\sigma_{2i}, p_{2i}, q_{2i}) \in S_2$, $\sigma_{1i} \leq \sigma_{2i}$, $p_{1i} = p_{2i}$ and $0 < q_{1i} \leq q_{2i}$.

We can use the semantics of a predicate p to transform a candidate space S: for every triple $(\sigma, , q)$, $[\![p]\!]\sigma$ gives rise to a new candidate space. We can then scale every element of this space by q and combine the resulting spaces. This is equivalent to the bind operation of the subprobability monad, described earlier in this paper. We will denote this transformation by $[\![p]\!]_S$.

Lemma 3.2.10. Monotonicity Given a predicate p and two candidate spaces $S_1 \leq S_2$, $[\![p]\!]_{S_1} \leq [\![p]\!]_{S_2}$.

Proof: By induction on the well-founded ordering on predicates.

Case `True`: The theorem holds trivially since `True` leaves candidate spaces unchanged.

Case `False`: The theorem holds vacuously, since both resulting candidates spaces are empty.

Case `==`, `/=`, `<`, `<=`, `>`, `>=`:

Let $\cup_{i=1}^{n}(\sigma_{1i}, p_{1i}, q_{1i}) = S_1, \cup_{i=1}^{n}(\sigma_{2i}, p_{2i}, q_{2i}) = S_2, n \leq m$ be a partitioning of S_1, S_2 such that for every pair of triples $(\sigma_{1i}, p_{1i}, q_{1i}) \in S_1$ and $(\sigma_{2i}, p_{2i}, q_{2i}) \in S_2$, $\sigma_{1i} \leq \sigma_{2i}$, $p_{1i} = p_{2i}$ and $0 < q_{1i} \leq q_{2i}$. Consider the triples $(\sigma_{1i}, p_{1i}, q_{1i})$ and $(\sigma_{2i}, p_{2i}, q_{2i})$. It is enough to prove that, as spaces, $[\![p]\!]\sigma_{1i} \leq [\![p]\!]\sigma_{2i}$.

If $[\![p]\!]\sigma_{1i} = \varnothing$, this is trivially true. If not, then it can only contain a single span σ'_{1i} with probability 1. Since $\sigma_{1i} \leq \sigma_{2i}$, we know that $\sigma_{1i}[e_1] \subseteq \sigma_{2i}[e_1]$ and $\sigma_{1i}[e_2] \subseteq \sigma_{2i}[e_2]$.

For the equality case, it follows that $\sigma_{1i}[e_1] \cap \sigma_{1i}[e_2] \subseteq \sigma_{2i}[e_1] \cap \sigma_{2i}[e_2]$ and that $[\![p]\!]\sigma_{2i}$ also contains a single span σ'_{2i}, a superset of σ'_{1i} with probability 1, which concludes the proof.

For the disequality case, the only valuation that can be removed from σ_{2i} would need to be removed from σ_{1i} as well. Therefore $[\![p]\!]\sigma_{2i}$ also contains a single span σ'_{2i}, a superset of σ'_{1i} with probability 1, which concludes the proof.

For the inequalities, we know that if a valuation v was dropped from σ_{2i} then $v[e_1] \geq \texttt{max}(\sigma_{2i}[e_2])$ or $v[e_2] \leq \texttt{min}(\sigma_{2i}[e_1])$. But then $v[e_1] \geq \texttt{max}(\sigma_{1i}[e_2])$ or $v[e_2] \leq \texttt{min}(\sigma_{1i}[e_1])$, which means they would also be dropped from σ_{1i}. $[\![p]\!]\sigma_{2i}$ also contains a single span σ'_{2i}, a superset of σ'_{1i} with probability 1, which concludes the proof.

Case p_1 `&&` p_2: By the inductive hypothesis, $[\![p_1]\!]_{S_1} \leq [\![p_1]\!]_{S_2}$. Using the inductive hypothesis again for these spaces with predicate p_2 concludes the proof.

Case p_1 `||` p_2: By the inductive hypothesis, $[\![p_1]\!]_{S_1} \leq [\![p_1]\!]_{S_2}$ and $[\![p_2]\!]_{S_1} \leq [\![p_2]\!]_{S_2}$. Therefore $[\![p_1]\!]_{S_1} \cup [\![p_2]\!]_{S_1} \leq [\![p_1]\!]_{S_2} \cup [\![p_2]\!]_{S_2}$ which concludes the proof.

Case `not` p The conclusion follows directly from the inductive hypothesis, since the negated predicate is always strictly smaller than p.

Case `Fix` p: We unroll the definition of the semantics of `Fix`, running the semantics of p once. Using the inductive hypothesis, $[\![p]\!]_{S_1} \leq [\![p]\!]_{S_2}$. Let $\cup_{i=1}^{n}(\sigma_{1i}, p_{1i}, q_{1i}) = [\![p]\!]_{S_1}, \cup_{i=1}^{n}(\sigma_{2i}, p_{2i}, q_{2i}) = [\![p]\!]_{S_2}, n \leq m$ be a partitioning of $[\![p]\!]_{S_1}, [\![p]\!]_{S_2}$ such that for every pair of triples $(\sigma_{1i}, p_{1i}, q_{1i})$ and $(\sigma_{2i}, p_{2i}, q_{2i})$, $\sigma_{1i} \leq \sigma_{2i}$, $p_{1i} = p_{2i}$ and $0 < q_{1i} \leq q_{2i}$. Consider the triples $(\sigma_{1i}, p', q_{1i})$ and $(\sigma_{2i}, p', q_{2i})$. If $p' = p$ then the fixpoint will not make a recursive call. Otherwise, it is enough to prove that, as spaces, $[\![\texttt{Fix}p']\!]\sigma_{1i} \leq [\![\texttt{Fix}p']\!]\sigma_{2i}$. This is true by using the induction hypothesis with predicate `Fix`p' and spaces $\{(\sigma_{1i}, p', 1)\} \leq \{(\sigma_{2i}, p', 1)\}$. □

Now we can go back to proving what effect instantiation brackets have on a predicate. Let p be a predicate containing p' as a subpredicate, and let p'' be $p[\{p'\}^u/p']$, where p' is replaced by itself wrapped in instantiation brackets, for some unknown u appearing in p'. Then, for any span σ, the total probability of all the refinements in $[\![p]\!]\sigma$ is at least as much as the total probability in $[\![p'']\!]\sigma$. We will prove the stronger claim that, as candidate spaces, $[\![p]\!]\sigma \leq [\![p'']\!]\sigma$.

First note that p' is p itself, then the result follows from the fact that instantiation brackets leave the total probability mass of the space unaffected.

We will also use the following lemma:

Lemma 3.2.11. Given a candidate space S, an instantiation of an unknown u yields a smaller candidate space S'. For every (σ, p, q) in S, where $\sigma[u] = \{n_1, ..., n_m\}$, S' contains the triples $(\sigma[u \leftarrow \{n_i\}], p, \frac{q}{m})$. We can split each triple in S to m equal triples, from which the ordering is apparent.

We now proceed by induction on the well founded ordering of predicates.

Case `True`, `False`, `==`, `/=`, `<`, ...: There is no subexpression of p other than p itself.

Case p_1 `&&` p_2: We identify four possible locations of instantiation brackets. (1) If p' is p_1, then the lemma above states that $[\![p']\!]\sigma \leq [\![p_1]\!]\sigma$. Applying Monotonicity with p_2 to these spaces yields the result. (2) If p' is a subexpression of p_1, then using the induction hypothesis, $[\![p_1[\{p'\}^u/p']]\!]\sigma [\![p_1]\!]\sigma$. Applying Monotonicity with p_2 to these spaces yields the result. (3) If p' is p_2, then the result follows immediately from the lemma. (4) If p' is a subexpression of p_2, the result follows from the inductive hypothesis, applied to p_2 and every span σ' in $[\![p_1]\!]\sigma$.

Case p_1 ^{w_1}`||`^{w_2} p_2: We identify four possible locations of instantiation brackets. (1) If p' is p_1, then the lemma above states that $[\![p']\!]\sigma \leq [\![p_1]\!]\sigma$. Since $[\![p_1]\!]\sigma \leq [\![p_1]\!]$ `||` $[\![p_2]\!]$, the result follows. (2) If p' is a subexpression of p_1, then using the induction hypothesis, $[\![p_1[\{p'\}^u/p']]\!]\sigma \leq [\![p_1]\!]\sigma$. Since $[\![p_1]\!]\sigma \leq [\![p_1]\!]$ `||` $[\![p_2]\!]$, the result follows. (3) If p' is p_2, then the lemma above states that $[\![p']\!]\sigma \leq [\![p_2]\!]\sigma$. Since $[\![p_2]\!]\sigma \leq [\![p_1]\!]$ `||` $[\![p_2]\!]$, the result follows. (4) If p' is a subexpression of p_2, then using the induction hypothesis, $[\![p_2[\{p'\}^u/p']]\!]\sigma \leq [\![p_2]\!]\sigma$. Since $[\![p_2]\!]\sigma \leq [\![p_1]\!]$ `||` $[\![p_2]\!]$, the result follows.

Case `not` p The conclusion follows directly from the inductive hypothesis, since the negated predicate is always strictly smaller than p.

Case `Fix` p_1: We identify two cases. (1) If p' is p_1, then the lemma above states that $[\![p']\!]\sigma \leq [\![p_1]\!]\sigma$. (2) If p' is a subexpression of p_1, $[\![p_1[\{p'\}^u/p']]\!]\sigma \leq [\![p_1]\!]\sigma$ follows from the inductive hypothesis. In both cases, the result follows from Monotonicity applied to `Fix` p.

Case $\{p_1\}^{u'}$ We identify two cases. (1) If p' is p, then the lemma above states that $[\![p']\!]\sigma \leq [\![p_1]\!]\sigma$. (2) If p' is a subexpression of p, $[\![p_1[\{p'\}^u/p']]\!]\sigma \leq [\![p_1]\!]\sigma$ follows from the inductive hypothesis. In both cases, the result follows from Monotonicity applied to `Fix` p.

Case $^{u'}\{p_1\}$ By applying the inductive hypothesis to p_1 and the space that is the result of handling the instantiation annotation. □

4. Extensions

In this section, we sketch two significant extensions of the core language described in Section 3—datatypes and pattern matching, and functions—yielding an appropriate intermediate form for the Luck surface language described in 5. We have not completely formalized these extensions; however, we expect the key properties of the core language to continue to hold.

4.1 Adding datatypes

We first remove integers from our previous subset and instead introduce datatypes and pattern matching. Users can write datatype declarations in the same form as Haskell's basic `data` declarations. Values in this calculus are unknowns or constructor applications over values:

$$v ::= u \mid C\ \overline{v}$$

We will call any value that contains no unknowns *rigid*; valuations are maps from unknowns to sets of rigid values.

Unknowns are associated with domains of a specific type.[4] To reflect type information we introduce the notion of a *rectangular map*. Given a type τ whose data constructors are $C_1..C_n$, each with (potentially zero) arguments of types $\tau_{i1}..\tau_{im_i}$, a rectangular map for τ is a map from data constructors C_i of type τ to tuples of rectangular maps of types $\tau_{i1}, \cdots, \tau_{im_i}$. In candidate spans, we will represent the domain of an unknown with a rectangular map for the corresponding type. Candidate spans become maps from unknowns of some type to rectangular maps of the same type. For example, consider the three datatypes `A`, `B`, and `C`:

```
data A = A1 B C | A2
data B = B1 | B2
data C = C1 | C2
```

The set of values $\{\{A_1\ B_i\ C_j \mid i, j \in \{1, 2\}\}, A_2\}$ can be represented by the rectangular map $\{A_1 \mapsto (\{B_1, B_2\}, \{C_1, C_2\}), A_2\}$, where we write B_1 instead of $B_1 \mapsto \{\}$ for brevity. This representation is somewhat restrictive: certain sets of valuations like $\{A_1\ B_1\ C_1, A_1\ B_2\ C_2\}$ cannot be represented. However, it allows us to greatly simplify semantics for instantiation brackets and pattern matching and it supports an efficient direct implementation while at the same time retaining enough flexibility to facilitate scaling Luck generators for complex artifacts like type systems (Section 6). The full definition of predicates in core Luck with datatypes is shown in Figure 6, where the weights in the weighted disjunctions range over (Peano-style) naturals.

In place of binary boolean constraints on integers, we introduce pattern matching as a primitive constraint. Patterns in a match introduce the need for creating fresh unknowns; for this, we add

[4] In the implementation, we support polymorphic datatypes. However, the predicates for which we want to generate satisfying valuations need to be monomorphized, meaning every individual unknown has a monomorphic type.

$$
\begin{aligned}
pat &::= u \mid C\ p \\
p &::= \texttt{True} \mid \texttt{False} \mid p\ \texttt{\&\&}\ p \mid p\ {}^{n}||^{n}\ p \mid \texttt{not}\ p \\
&\mid \{p\}^{a} \mid {}^{a}\{p\} \mid \texttt{Fix}\ p \\
&\mid \texttt{case}\ pat\ [(pat, p)] \\
a &::= u \leftarrow \{\overline{C \mapsto n}\} \\
n &::= O \mid S\ n
\end{aligned}
$$

Figure 6. Grammar of Luck with Datatypes

$$
\begin{aligned}
[\![\{p\}^{x \leftarrow \{\overline{C_i \mapsto n_i}\}}]\!]\ st\ \sigma &= ([\![p]\!]\ st\ \sigma) \gg\!= \lambda(\sigma', f).(st, \\
&\quad \left\{ \begin{array}{l} (\sigma[x \leftarrow C_1 \cdots], f) \mapsto \frac{n_1}{\sum n_i} \\ \cdots \\ (\sigma[x \leftarrow C_m \cdots], f) \mapsto \frac{n_m}{\sum n_i} \end{array} \right\}) \\
[\![^{x \leftarrow \{\overline{C_i \mapsto n_i}\}}\{p\}]\!]\ st\ \sigma &= \left\{ \begin{array}{l} \sigma[x \leftarrow C_1 \cdots] \mapsto \frac{n_1}{\sum n_i} \\ \cdots \\ \sigma[x \leftarrow C_m \cdots] \mapsto \frac{n_m}{\sum n_i} \end{array} \right\} \\
&\quad \gg\!= ([\![p]\!] st) \\
[\![\{p\}^{a}]\!]^{-} &= [\![\{\texttt{not}\ p\}^{a}]\!] \\
[\![^{a}\{p\}]\!]^{-} &= [\![^{a}\{\texttt{not}\ p\}]\!]
\end{aligned}
$$

Figure 7. Weighted instantiation brackets

a bit of state and thread it around in our semantics. Thus, the type of refiners becomes

$$[\![p]\!] :: State \to Span \to (State, \underline{\Pr}(Span, p)).$$

Semantics for most of the operations is almost identical to those of the previous section: the only difference is passing around the state. The only two constructs which need to be addressed are instantiation brackets and `case` statements. In Luck with datatypes we re-introduce the weighted form for instantiation brackets from the beginning of Section 3. Weight annotations come in the form of maps from data constructors C to natural numbers n denoting relative weights. Figure 7 shows the semantics of these weighted forms. Post-refiner instantiation brackets for an unknown `u` run the semantics of the wrapped refiner `p` and then partially instantiate `u` based on the weight annotations. We pick some constructor C_i with probability $\frac{n_i}{\sum n_i}$, and partially instantiate `u` to C_i, refining its domain to the rectangular map whose sole key is C_i. Any potential arguments to C_i are not constrained further—they will be handled by some case expression later on. For example, assuming the span of an unknown u is $\{A_1 \mapsto \{\{B_1, B_2\}, \{C_1, C_2\}\}, A_2\}$, then the weight annotation $u \leftarrow \{A_1 \mapsto 3, A_2 \mapsto 1\}$ would mean that the resulting span would be $\{A_1 \mapsto \{\{B_1, B_2\}, \{C_1, C_2\}\}\}$ 75% of the time, and $\{A_2\}$ 25% of the time. Pre-refiner instantiation brackets also partially instantiate `u` based on the weight annotations; however, this happens before running the semantics of `p`, which is given the result of the instantiation as input. Negation is just propagated to the wrapped refiner.

The primitive constraint in Luck with datatypes is `case`, which can be viewed as a kind of disjunction followed by handling the selected branch. We execute all branches in parallel, keeping the ones whose resulting distributions are not empty and combining the results. Figure 8 shows the semantics of `case`, using monadic syntax to capture state. In more detail, for every branch $C_i\overline{u_i}$ of the match we rename the unknowns $\overline{u_i}$ present, introducing fresh unknowns $\overline{u_i'}$ in the process. The binding of each of these unknowns is calculated by accessing the rectangular map that the scrutinee represents in the input span σ. For example, consider the following predicate:

$$
\begin{aligned}
&branch\ \sigma\ pat\ (C_i\ \overline{u_i}, p_{body}) = \overline{u_i'} \leftarrow fresh\ \overline{u_i} \\
&\quad \texttt{let}\ \sigma' = \sigma[u_{ij}' \leftarrow \sigma[pat][C_i][j]] \\
&\quad (\sigma'', p_{body}') \leftarrow [\![p_{body}]\!]\sigma' \\
&\quad \sigma_F \leftarrow unify\ pat\ (C_i\ \overline{u_i'})\sigma'' \\
&\quad \texttt{return}\ \{(\sigma_F \backslash \overline{u_i}, (C_i\ \overline{u_i}, p_{body}')) \mapsto 1\} \\
&combine\ p\ []\ \sigma\ alts = \{(\sigma, \texttt{case}\ p\ alts) \mapsto 1\} \\
&combine\ p\ (h : t)\ \sigma\ alts = (\sigma', h') \leftarrow h \\
&\quad combine\ p\ t\ (\sigma \cup \sigma')\ (h' : alts) \\
&[\![\texttt{case}\ p\ \texttt{of}\ brs]\!]\ \sigma = opts \leftarrow \texttt{mapM}\ (branch\ \sigma\ p)\ brs \\
&\quad \texttt{case}\ (\texttt{filter}(= \varnothing)\ opts)\ \texttt{of} \\
&\quad\quad [] \to \varnothing \\
&\quad\quad opts \to combine\ opts\ p\ \sigma_{empty}\ []
\end{aligned}
$$

Figure 8. Delayed pattern matching

```
case x of
| O  → True
| S x' → False
end
```

Suppose that the initial binding of `x` represents $\{0, 2, 3\}$:

$$\{O, S \mapsto \{S \mapsto \{O, S \mapsto \{O\}\}\}\}$$

The binding of x' in the body of the $S\ x'$ branch will then be $\{1, 2\}$, or, in rectangular map form, $\{S \mapsto \{O, S \mapsto \{O\}\}\}$. Afterwards, we run the body of the branch getting back a subprobability distribution on spans and residues, using unification again to propagate information back to the discriminee. To conclude the treatment of the branch, the bindings of the introduced unknowns are deleted. All branches are handled in parallel; afterwards, we filter out the branches that returned a probability distribution with an empty support. In the end, we return a combination of the remaining results, getting a (potentially overapproximating) union of the spans and a residue that corresponds to the `case` of the residues.

If we model positive integers using Peano numerals, we can construct an example similar to the one above (`p && x > 0`), where we have unfolded the inequality constraints to pattern matching.

```
p && case x of O → False | S _ → True end
```

Assume that the domain of `x` after refining with `p` contains all representations of naturals from 0 to 10. Using the delaying semantics of cases, the domain of `x` will be filtered to not contain O, and every value in the resulting domain for `x` satisfies the conjunction with zero backtracking probability.

Imagine now adding a prefix instantiation bracket on `x` (with uniform probability) around the second conjunct (which is exactly what a local instantiation strategy at the point of the first constraint on a variable would correspond to). The binding for `x` will become a singleton rectangular map containing either O or S as its only key, with S mapping to the rectangular map representation of $\{0..9\}$. Thus, the `case` expression will now only succeed with probability 50%.

Properties The statement of the theorems from the previous subsection remains mostly unchanged, as do the vast majority of the proofs. The only difference is in handling the `case` construct.

Theorem 4.1.1. ***Decreasingness***

Proof: Similarly with before, the proof proceeds by well-founded induction on the well founded ordering of the predicates and the by nested induction on the ordering of spans. The only interesting case is `case`. For that we will need the following lemmas:

Lemma 4.1.2. Given a span σ, a discriminee pat, and a pattern $(C_i\overline{u_i}, p_{body})$, then all of the spans σ' in the support of $branch\ \sigma\ pat\ (C_i\overline{u_i}, p_{body})$ are smaller (but not strictly smaller) than σ.

Proof: First we introduce a bunch of fresh unknowns u_i', creating a new span σ', whose restriction to the domain of σ is equal to σ. Using the inductive hypothesis (since p_{body} is smaller than our original p) we know that σ'' is smaller than σ'. With *unify* we create a span σ_F, whose restriction to the domain of σ is smaller than σ. Finally, we remove the fresh u_i's, forming σ_F, which proves the lemma.

Returning to the proof of the theorem, the result of each of the branches either has in its support a single span that is smaller (but not strictly) than σ, or is an empty subprobability distribution. If all are empty the theorem holds vacuously. If not, then we take the union of the existing spans that are smaller than σ, which preserves the ordering.

Theorem 4.1.3. ***Completeness***

Proof: Again, in order to handle the `Fix` case, we prove two statements ysimultaneously: (1) If v is not in any of the covers of the spans σ' in the support of $[\![p]\!]\sigma$, then it does not satisfy p. (2) If v is not in any of the covers of the spans σ' in the support of $[\![\texttt{Fix}\ p]\!]\sigma$, then v does not satisfy p.

First we will again handle the branch case as well:

After we introduce the fresh unknowns, for a valuation to be dropped from the result of the unification it means, that it didn't belong to the set of valuations represented by pat or that represented by $C_i\overline{u_i'}$. From the inductive hypothesis we know then that it was dropped during the call to $[\![p_{body}]\!]$ because it didn't satisfy it. Therefore the branch itself can't be satisfied.

Back to the entire case:

(1) For a valuation to not be present in the cover of any of the spans in the support of the result, it means it was dropped by all of the branches (since we take the union of the resulting spans). Therefore it can't satisfy the entire case. (2) Unrolling the definition of the semantics of `Fix`, after running the semantics of p we get a distribution on pairs of spans σ' and residues p', where the alternatives in p' are residues of each of the bodies of the alternatives. If $\sigma' = \sigma$ then there is no recursive call and there is nothing to prove. So let $\sigma' < \sigma$ and let v be dropped from this span. Since the resulting span σ' is the union of all of the spans that result from succesfully handling branches, it means that it was dropped from all branches. But then none of these branches could be satisfied and therefore neither could the entire $case$.

Theorem 4.1.4. ***Soundness***

Proof: The proof is largely identical to the soundness proof of the previous section. By inducting again on the well founded relation of predicates, we only need to prove that given a span σ containing only a single valuation v, for any discriminee pat, and branches brs, $[\![\texttt{case}\ pat\ \texttt{of}\ brs]\!]\sigma$ will either be empty or the spans in its support are equal to σ itself and v is a satisfying valuation.

Handling first each branch on its own, let $C_i\overline{u_i}$ be the patterns and p_{body} the body of a branch. Either σ'' is empty and we are done, or σ still contains the singleton valuation v. By the inductive hypothesis, v satisfies p_{body}, and will not be dropped by the unification (otherwise, σ would already be empty). Since σ contains only one valuation, only one branch can possible be satisfying. Therefore combine will be a trivial call and the result follows.

4.2 Adding functions

To introduce functions into Core Luck we split the language into two fragments. The first one is a standard, fully deterministic, simply typed lambda calculus with datatypes that is used for actual computations (like the ones for `size` in the `bst` example of Section 2). The other one—the probabilistic fragment—only includes predicates and is an extension of what we presented so far, extending the grammar of the previous subsection to include fully applied function calls. For this second fragment we assume we have an environment mapping function identifiers to *terms*, that can, in addition, contain free variables that will be substituted in function applications. The entire grammar is shown in figure 9, where we use the subscript $_d$ to refer to the deterministic fragment. The expressions e_d that are used in place of patterns for the probabilistic fragment are required to be of a non-arrow type.

$$
\begin{aligned}
pat_d &::= C\ \overline{pat_d} \mid x \\
e_d &::= \lambda x.e_d \mid (e_d\ e_d) \mid \texttt{case}\ e_d\ \texttt{of}\ \overline{(pat_d, e_d)} \\
&\mid\ \texttt{let}\ x\ \overline{x} = e_d\ \texttt{in}\ x \\
pat &::= u \mid C\ \overline{pat} \mid e_d \\
p &::= \texttt{True} \mid \texttt{False} \mid \cdots \\
&\mid\ \texttt{case}\ pat\ \texttt{of}\ \overline{(pat, p)} \mid (f\ \overline{pat}) \\
term &::= p \mid x
\end{aligned}
$$

Figure 9. Syntax of Core Luck with functions

We can now give semantics for the probabilistic fragment. They are similar to the previous section, with one difference: we need to reduce a pattern pat that is the scrutinee of a `case` statement, using the evaluation function of the deterministic fragment. Semantics for a function application $(f\ \overline{pat})$ are equally simple: we reduce all arguments $\overline{pat}$ to values $\overline{v}$, ask the environment for the body $t(\overline{x})$ of f and return the semantics of the body where its free variables have been substituted $t[\overline{x}/\overline{v}]$.

$$
\begin{aligned}
[\![(f\ \overline{pat})]\!] \quad = \quad & (\lambda\overline{x}.p) \leftarrow ask\ f; \\
& \overline{v} \leftarrow reduce\ \overline{pat}; \\
& \texttt{return}\ [\![p[\overline{x}/\overline{v}]]\!]
\end{aligned}
$$

Here, we use a monadic notation to hide the handling of the state and function environment. An interesting observation is that evaluating the arguments of every application leads to an eager language, even though narrowing is generally associated with laziness. However, the laziness of the approach is encoded in our handling of unknowns as values.

If the function environment includes arbitrary functions, then termination is clearly not guaranteed. Consider for example a function $f :: Nat \to Nat$ whose binding is $\lambda x.f\ (S\ x)$. Any application of f will lead to an infinite computation. We will characterize a large class of terminating terms that we can use to populate well-behaved function environments for the probabilistic fragment of Luck. This class includes, for example, structurally recursive predicates over datatypes and we can obtain guarantees similar to the previous section. First, we require that every function f identifier is also associated with some well founded ordering that operates possible arguments to f in some context σ: $wf :: (Span, \overline{pat}) -> (Span, \overline{pat}) -> Bool$. For example, we could say $(\sigma_1, \overline{p_{1i}})$ is smaller than $(\sigma_2, \overline{p_{2i}})$ using a lexicographic ordering of the tuples, where a projection p_{1i} is smaller than another p_{2i} if the set of valuations p_{1i} represents in σ_1 is smaller than the corresponding set of p_{2i} in σ_2. We also exclude mutually recursive functions, which allows us to have a total ordering of the functions in the environment. We could remove this restriction by associating the same well founded ordering to a class of mutually recursive functions, but we keep it to simplify the presentation.

With these assumptions we can define *terminating predicates* with respect to a span σ.

- Predicates without function applications are always terminating with respect to σ. This is true because of the finiteness of σ and the decreasingness of the semantics in Section 4.1.
- A predicate containing a function application $(f\ \overline{pat})$ is terminating with respect to σ, if for all calls $([\![f'\ \overline{pat'})\ state'\ \sigma'$ that are invoked during the evaluation of $[\![(f\ \overline{pat})]\!]\ st\ \sigma$, either

$f' < f$ based on the ordering of functions in the environment or $f = f'$ and $(\sigma', \overline{pat'})$ is smaller than $(\sigma, \overline{pat})$ based on the well founded ordering associated with f.

Terminating predicates over a span σ have a maximum number of substitutions that can be performed. Theorems about our predicates can now be proved by simple induction on the number of substitutions performed and reduced to the theorems in Section 4.1.

5. Surface Language

So far, we presented the semantics of Core Luck, using explicit instantiation annotations to control the location where each unknown will be instantiated. This approach is very expressive, giving the programmer very fine grained control over the generator aspect of a refiner, while admitting a clean probabilistic denotational semantics like the one presented.

On the other hand, it would quickly get cumbersome to add instantiation points explicitly for all variables, especially if the outermost-needed narrowing strategy of instantiating a variable `x` when we first encounter a constraint on `x` is what we want. Noting that this is indeed usually the case, we return to the source version of Luck from Section 2 where—by default—every variable is instantiated at its first occurrence. The user, instead of providing explicit instantiation point annotations, writes *delay brackets*, delimiting areas where instantiation is postponed in favor of constraint solving.

The `bst` refiner from Section 2 contains a constraint of the form `low < x && x < high`, wrapped in a delay bracket:

```
⟦ low < x && x < high ⟧x
```

In Core Luck this corresponds to:

```
{ {low<x}low && {x<high}high }x
```

The translation to Core Luck also has instantiation points for `low` and `high`, meaning we can drop the assumption of Section 2 that these variables are instantiated prior to invoking `bst`. More concretely, suppose the domains of `low`, `x`, and `high` are all $\{0..10\}$ initially. When we encounter the constraint `low < x`, we filter the domains of both `low` and `x` to reflect that constraint. In our example, the resulting domain of `low` would be $\{0..9\}$ and that of `x` would be $\{1..10\}$, since the rest of the possible values could never satisfy the constraint. Then we encounter the instantiation annotation $\{\cdot\}^{\texttt{low}}$ and randomly pick a value for `low` from its domain; then we use the constraint `low < x` again to further filter the domain of `x`. If, for example, `low` was instantiated with 2, then the domain of `x` would further filter down to to $\{3..10\}$. Then we propagate the constraint `x < high`, which leads to the domains being $\{3..9\}$ and $\{4..10\}$ for `x` and `high` respectively, before generating a value for `high`. If, for example, the value picked for `high` was 7, the domain of `x` would end up being $\{3..6\}$. Finally, when we exit $\{\cdot\}^{\texttt{x}}$ we sample `x` from the filtered domain.

From Source to Core Intuitively, one would expect a fairly simple translation process from source Luck to Core. Throughout the translation, we keep a context containing all the unknowns that have already been delayed. Every time a constraint for an unknown `x` is encountered we check the context: if `x` is not delayed we add an instantiation bracket for `x`; if `x` is delayed, then we don't. To handle delay brackets, when we encounter $[\![p]\!]^{\texttt{x}}$, we add `x` to the context, translate `p` and wrap an instantiation bracket for `x` around this translation.

However, the handling of function calls requires special care, since we would expect the function to behave differently depending on whether its argument has been delayed or not. Consider for example the `notMember` function from the introduction, now used in the context of a predicate `distinct`, as shown in Figure 10.

```
sig notMember :: Int → [Int] → Bool
fun notMember x l =
    case l of
    | [] → True
    | (h:t) → h /= x && notMember x t
    end

sig distinct :: [Int] → Bool
fun distinct l = aux [] l

sig aux :: [Int] → [Int] → Bool
fun aux acc l =
    case l of
    | [] → True
    | y:ys → ⟦ notMember y acc ⟧y
             && aux (y:acc) ys
    end
```

Figure 10. `notMember` and `distinct` refiners

As discussed in the introduction, we would not want to instantiate `x` inside the body of `notMember`, if list `l` is already fixed. We can express this in Luck using the delay bracket for `x` around the call to `notMember`. At the same time, we would expect that not adding this delay bracket would have the opposite effect of instantiating `x` to a value not equal to the head of the list. To achieve such an effect in Core Luck, we would have to code two separate versions of the `notMember` function, one with and one without the annotation, invoking each one where appropriate. Source Luck allows us to write only one version of `notMember`, the non-annotated version above, and get different behavior depending on the arguments. To achieve this, the source-to-core transformation lazily translates each function a number of times: given a function f with n arguments, we obtain 2^n different versions, 2 for each potential status of every argument.

5.1 Pattern compiler

We expand case expressions with nested patterns—as in Fig. 11—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 is to simply share the weight equally with all duplicated branches. The probability of a single branch then depends on the total number of expanded branches that come from the same source: that can be hard for users to determine, and may vary widely even between sets of patterns that appear similar.

Instead, Luck provides a different default weighing strategy, illustrated by Figure 11, with the following characterization: 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.

5.2 Size erasure and elaboration

Once again, consider the `bst` predicate of Section 2. Programs in Luck are meant to serve as both generators and predicates. However, in the case of `bst`, we use sizes to control distribution and termination. While when viewed as a generator this makes sense, when viewed as a boolean predicate this excludes valid binary search trees of larger sizes. We address this by providing

```
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
  end

sig isRedex :: T → Bool          -- Expansion:
fun isRedex t =
  case t of
  | 1 % Var _ → False          -- 1/9
  | 1 % Lam _ _ → False        -- 1/9
  | 1 + 6 % App t1 _
    → case t1 of
      | 1 % Var _ → False     -- 1/18
      | 12 % Lam _ _ → True   -- 2/3
      | 1 % App _ _ → False   -- 1/18
      end
  end
```

Figure 11. Case expression with a nested pattern and a wildcard and its expansion. Comments show the probability of each alternative.

an automatic way to *erase* size information, which in this example induces predicate semantics corresponding directly to the Haskell boolean predicate at the beginning of Section 2.

Moreover, we also provide a translation taking structurally recursive boolean predicates on datatypes and *elaborating* them into Luck predicates automatically. We use a simple process, similar to the handling of recursion in PropEr [50], which adds an additional `size` argument, skews non-base branches using this size parameter, and adjusts the recursive calls to divide `size` by the number of recursive calls in the branch (or subtract 1 if it is the only call). Given the `bst` predicate at the beginning of Section 2 this produces exactly the distribution annotations presented in Section 2.

These transformations between Luck programs give us a lot of flexibility when programming. When we want a simple generator, we can obtain weight annotations automatically from a predicate. If we want more control over the distribution—if, for example, we would like our trees to be left-skewed—we can annotate distributions by hand and obtain a predicate for testing via erasure.

5.3 Controlling distributions in Luck

Luck leverages established techniques from the random testing community to provide control over posterior distributions. Even though in practice a uniform distribution does not always lead to effective testing (in fact, it can instead lead to very low assurance, as we will see at the end of Section 7), it is a fairly common concern in the literature. Luck users can obtain uniform generators by turning to Boltzmann samplers [22]. Boltzmann samplers provide an efficient way of drawing samples from combinatorial structures of an approximate size `n`—linear in time in `n`—where any two objects with the same size have an equal probability to be generated.

Boltzmann samplers for algebraic datatypes work by making appropriately weighted local choices based on a control parameter. In the binary tree case, a Boltzmann sampler would look similar to a simplistic generator for trees: flip a (potentially biased) coin, generating a `Leaf` or a `Node` based on the outcome; then recurse (Figure 12). The distribution this generator induces can be easily implemented by a trivial Luck predicate, also in Figure 12. A Boltzmann sampler is a slightly modified version of this simple approach. First of all, the bias in the local choices must be systematically calculated (or approximated with numerical methods).

```
randomTree :: Gen Tree
randomTree =
  frequency [(1, return Leaf)
            ,(1, Branch <$> randomTree
                        <*> randomTree)]

sig randomTree :: Tree -> Bool
fun randomTree tree =
    case tree of
    | 1 % Leaf -> True
    | 1 % Branch l r ->
          randomTree l && randomTree r
    end
```

Figure 12. Simple tree generator (Haskell vs Luck)

This bias depends on the convergence of a generating function associated with the data type's recursive specification. Then, we start sampling using the computed bias for each choice. If at any point we reach a size bigger than $n(1+\epsilon)$, we stop the generation and try again. If the generation stops with a term $n(1-\epsilon)$, we throw away the generated term and try again. The theory behind Boltzmann samplers guarantees that this approach will terminate in expected linear time in n (including discards!), and the result will be uniformly selected among other elements of the same size.

Using Boltzmann samplers is a natural fit for our setting and requires two things: a way to compute the control parameter for the structure that is being generated, and a way to reject samples that are outside the neighborhood of the desired size. Both of these can be handled automatically, as shown by Canou and Darrasse [13].

The presence of additional constraints on generated data could skew the posterior distributions, negating any uniformity guarantees. But if, every time we reach an unsatisfiable constraint, we backtrack to the beginning of the generation, uniformity is preserved since we are only concerned with uniformity in the set of satisfying valuations. Unfortunately, throwing away all progress is not efficient; it is usually a lot faster to backtrack to a more recent choice instead of the beginning. Such an algorithm still gives us some assurance about the distribution, similarly to Claessen et al. [19]: the least likely value generated will be at most a constant factor less likely than the most likely one, where the factor is the amount of local backtracking allowed.

On the other hand, the user might wish to skew posterior distributions. For this task, Luck provides the same primitive as QuickCheck: `collect`. In QuickCheck, the user can wrap properties inside a `collect x`, and, during testing, QuickCheck will gather information on `x`, group all equal `x`'s and provide an estimate of the posterior distribution that is being sampled. This information can be used to succesfully increase bug finding rates, as shown in [35].

Note that none of these techniques is an actual contribution of this paper. Luck instead builds upon established techniques from the community to provide fine ways to tune posterior distributions.

6. Evaluation/Case Studies

In this section we will show the expressiveness and efficiency of our approach. We developed a prototype implementation of source Luck, interpreting Luck programs in Haskell using the semantics described in Section 3. We show that it can easily handle the usual kind of examples in the property-based random testing literature. Moreover, we present two complex case studies—generating well-typed lambda terms and information flow control machine states. Our prototype lets us write much smaller and cleaner code than the existing state-of-the-art handwritten generators, though the code is somewhat slower (between $1.75\times$ and $50\times$).

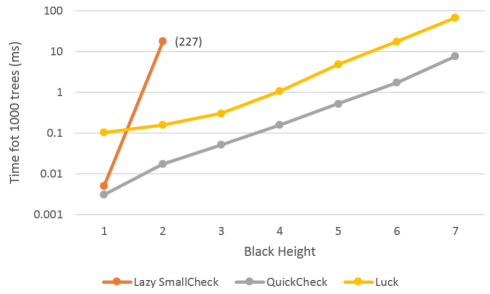

Figure 13. Red Black Tree experiment

6.1 Evaluation on small examples

The literature on generation of constrained random inputs uses many small examples, including various list-based predicates such as `sorted`, `member` and `distinct`, as well as predicates on trees like binary search trees (already presented in Section 2) and red black trees. In Appendix A.1 we show the implementation of all of those examples in Luck, showing how we can get both efficient generators and predicates with minimal effort. While the practicality of our approach is better explored by the two larger case studies that follow, we use red black trees to showcase our tool's behavior with respect to automatic tools like Lazy SmallCheck [53]. We attempted to generate 1000 red black trees with a specific black height bh, meaning the depth of the tree can be as large as $2 \cdot bh + 1$ for the purposes Lazy SmallCheck. Results are shown in Figure 13. 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 needed exploration of trees up to depth 7, Lazy SmallCheck was unable to generate 1000 red black trees within a 5 minute time limit. At the same time, the Luck implementation lies consistently within an order of magnitude of a very efficient handwritten generator in QuickCheck.

6.2 Lambda term generation

Using our prototype implementation we reproduced the experiments of Pałka et al. [49]. We encoded a model of simply typed lambda calculus in Luck, with an extensive environment of standard functions present in the Haskell Prelude to generate more interesting well-typed terms. Afterwards, we translated the generated terms into Haskell syntax and applied each one 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.

A straightforward implementation of a type checker for 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. [25], seeding the environment with monomorphic versions of possible constants and using a separate rule for `seq`. In the end, we discovered bugs similar to those found by Pałka et al. and Fetscher et al. under custom inspection. It appears that, after shrinking (which our Luck prototype currently lacks—see Section 8), the minimal counterexamples would be equivalent.

The generation speed was only a little slower than that of the handwritten generator. We were able to generate terms of average size 70 (number of internal nodes), and, grouping terms together in batches of 1000, we got a total time of generation, unparsing, compilation and execution of around 30-35 seconds per batch. This is a slowdown of 1.75x compared to that of Pałka's. On the other hand, our implementation of STLC with monomorphic constants was a total of 85 lines of fairly simple code, while the handwritten development is 1684 lines, accompanied by the phrase "[...] the code is difficult to understand, so reading it is not recommended" in its distribution page [1].

6.3 Information flow control

For our second large case study, we turn to recent work on generating information-flow control machine states [35]. Given an abstract stack machine with data and instruction memories, a stack and a program counter, we attach *labels*—security levels—to runtime values, propagating them during execution and restricting potential flows of information from *high*—secret—to *low*—publicly available—data. The desired security property, *termination insensitive noninterference*, states that if we start with two indistinguishable abstract machines `s1` and `s2` and we run each of them to completion, then the resulting states `s1'` and `s2'` are also indistinguishable. Two states are indistinguishable if all of their low tagged data is identical.

Hriţcu et al. [35] discovered that they could get efficient testing in one of 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 discovering a strong invariant (strong enough to *prove* noninterference), generating two arbitrary indistinguishable states and then running for a single step (they call this approach *SSNI*, single step noninterference).

In both cases, there is a generation 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 the generator `vary`, one must effectively reverse the indistinguishability predicate between states and keep the two artifacts in sync.

Borrowing their stronger property (SSNI), we encoded an indistinguishability predicate in Luck, using our prototype to generate small, indistinguishable pairs of states. In 193 lines of code we are able to describe both the predicate and the generator for indistinguishable machines. The same piece of software required more than 1000 lines of complex Haskell code in its handwritten version. Performancewise, the handwritten generator was reported to generate an average of 18400 tests per second, while the Luck prototype generates an average of 3160 tests per second, 5.8 times slower.

The real promise of Luck, however, became apparent when we turned to LLNI. Hriţcu et al. [35] 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 around two months of work to code, debug and optimize, resulting in more than 100 additional lines of code. The same effect was achieved in Luck by the 10 intuitive lines shown in figure 14, where we just put the previous explanation in code!

We used this approach to find the same set of bugs as in Hriţcu et al. [35]. We were able to find all of the same bugs, with the same effectiveness: Figure 15 shows a comparison of the numbers of bugs found per 100 tests (bigger is better) both for our Luck generator and the artifact from Hriţcu et al. However, the Luck generator was about fifty times slower. We expect to be able to greatly improve this result with a more efficient implementation that compiles Luck programs to QuickCheck generators directly, instead of interpreting them in an unoptimized prototype.

The success of the prototype in maintaining similar effectiveness and significantly reducing the amount of effort required, suggests that the approach Luck takes is extremely promising and points towards the need for a real, optimizing implementation.

```
sig runsLong :: Int -> AS -> Bool
fun runsLong len st =
    if len <= 0 then True
    else case st of
      | AS m i s (Atom addr lab) ->
         wellFormedInstr i addr s &&
         case step (AS m i s (Atom addr lab)) of
           | Just st' -> runsLong (len - 1) st'
           | Nothing  -> True
        end
    end
```

Figure 14. Declarative skewing towards long runs

Bug	Luck	TNI
ArithNoTaint	7.6	11.3
PushNoTaint	60.4	65.3
PopPopsReturns	0.03	0.07
LoadNoTaint	14.2	6.2
StoreNoValueTaint	23.4	20.1
StoreNoPointerTaint	3.0	0.9
StoreNoPcTaint	0.07	0.2
JumpNoRaisePc	26.6	27.4
JumpLowerPc	3.1	2.9
CallNoRaisePc	1.8	2.3
ReturnNoTaint	0.4	0.2
WriteDownHighPtr	6.9	3.3
WriteDownHighPc	0.1	0.4

Figure 15. Comparison of number of bugs found per 100 tests

7. Related Work

Luck lies in the crossroads of many different topics in programming languages; thus, the potentially related literature is huge. Here we select and present the most relevant related work.

Local choices and backtracking The works that are most closely related to our own are these of Claessen et al. [19] and Fetscher et al. [25]. Claessen et al. exploit the laziness of Haskell, combining a needed-narrowing-like technique with FEAT [23], a tool for functional enumeration of algebraic types, to efficiently generate uniformly distributed random inputs satisfying a precondition. While their use of FEAT allows them to get uniformity by default, it is not clear how user control over the resulting distribution could be achieved. Fetscher et al. [25] 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 [49]) using the "trick" of monomorphizing the polymorphic constants of the context. They present two different strategies for making local choices: uniformly at random, or by ordering branches based on their branching factor. While both of these strategies seem reasonable (and somewhat complementary), there is no way of providing different distributions if needed.

An interesting and related approach appears in the inspiring work of Bulwahn [8]. In the context of Isabelle's QuickCheck [7], 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 [41] and Runciman et al. [53] also provide support for exhaustive testing using narrowing-based techniques. Similarly, Christiansen and Fischer [18] get data enumerators for free, exploiting the already present narrowing mechanism of Curry. While exhaustive testing is useful and has its own merits and advantages over random testing in some domains, we turn to random testing because the complexity of our applications—testing noninterference or optimizing compilers—makes exhaustive testing infeasible in practice.

Constraint-solving On the other side of the spectrum, 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. [16] stands out because of its efficiency and its guarantees of approximate uniformity. However, there is no support—and no obvious way to add it—for controlling distributions. In addition, their efficiency relies crucially on the fact that the *independent support*[5] for the examples in their domain is small relatively to the entire space. While true for usual SAT instances, this is not the case for random testing properties like non-interference. In fact, a minimal independent support for indistinguishable machines includes one entire machine state and the low parts of another, meaning the benefit from their heuristics would be minimal. Finally, they require SAT formulae as inputs, which would require a rather heavy translation from a higher-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 [14, 32], the most recent and efficient for Haskell being Target [54]. In Target, they translate from preconditions in the form of refinement types, and use a constraint solver to generate a satisfying valuation for testing. Then they introduce 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 their tool, pointing further towards the need to combine the two approaches as in Luck. Moreover, their 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 aiming to achieve higher coverage [4, 9–12, 28, 29, 42, 55]. Recently, in the context of Rosette [57], symbolic execution was used to successfully find bugs in the same information flow control case study.

Semantics for narrowing-based solvers This year, Fowler and Hutton [26] attempted to 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." With the added benefit of constraint solving, we successfully addressed this issue in Luck.

Probabilistic programming Semantics for probabilistic programs share many similarities with the denotations of Luck [30, 31, 44], while the problem of generating satisfying valuations shares similarities with probabilistic sampling [15, 40, 43, 46]. For example, the semantics of the language PROB in the recent probabilistic programming survey of Gordon et al. [31] also take the form of probability distributions over valuations. In addition, in probabilistic programs, the notion of observation serves a similar role as preconditions in random testing, creating problems for simplistic probabilistic samplers that use *rejection testing*—i.e., generate and test.

[5] The *support* X of a boolean formula $\mathtt{p}$ is the set of variables appearing in $\mathtt{p}$. A subset D of X is called an *independent support* of $\mathtt{p}$ if there are no two satisfying assignments for $\mathtt{p}$ that differ only in $X \setminus D$.

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

Figure 16. R2 probabilistic program for a single atom

Recent advances in this domain, like the work on Microsoft's R2 Markov Chain Monte Carlo sampler [46], have shown promise in providing more efficient sampling, using pre-imaging transformations in analyzing programs.

We did a simple experiment with R2, using the probabilistic program shown in figure 16 to model indistinguishability of tagged values. The result was surprising at first, since all the generated samples were tagged secret, which would clearly lead to ineffective testing. However, keeping in mind that the purpose of sampling in probabilistic programs is to provide an estimation of the posterior distribution, such a result makes sense: Given our modeling of the values of atoms as `doubles`, for every low indistinguishable pair of atoms there exist 2^{64} high ones! This further serves to motivate that uniform distributions are not what a testing tool should rigidly provide; in fact uniform distributions in cases like this will lead to extremely low assurance!

8. Conclusions and Future Work

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

The local distribution controls provided by Luck can be used to obtain uniformly distributed test data belonging to complex algebraic datatypes by applying the technique of *Boltzmann samplers* [22] to calculate appropriate local weight annotations. If uniform distribution turns out to be a frequent requirement, it may be worth adding annotations to Luck's surface syntax to tell the compiler to generate these annotations automatically.

In the future it will be interesting to explore compilation of Luck into generators in a language like Haskell, for example using Haskell's QuickCheck combinators to improve the performance of our interpreted prototype. Another interesting direction is to transfer ideas from Luck to generate inductive datatypes for testing in the Coq proof assistant [21, 51].

Another potential direction for this work is automatically deriving smart shrinkers. The notion of shrinking, or delta-debugging, is crucial in property-based testing, and may also require a lot of user effort and domain specific knowledge to be efficient [52]. It would be very interesting to see if there is a counterpart to narrowing or constraint solving that allows shrinking to stay within the space of the preconditions.

References

[1] Testing an optimising compiler by generating random lambda terms. http://www.cse.chalmers.se/~palka/testingcompiler/.

[2] S. Antoy. A needed narrowing strategy. In *Journal of the ACM*. 2000.

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

[4] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with Veritesting. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014.

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

[6] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *First International Conference on Interactive Theorem Proving (ITP)*. 2010.

[7] L. Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In *2nd International Conference on Certified Programs and Proofs (CPP)*. 2012.

[8] L. Bulwahn. Smart testing of functional programs in Isabelle. In *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 2012.

[9] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX conference on Operating systems design and implementation*. 2008.

[11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *13th ACM conference on Computer and communications security*. 2006.

[12] 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. In *33rd International Conference on Software Engineering*. 2011.

[13] B. Canou and A. Darrasse. Fast and sound random generation for automated testing and benchmarking in objective caml. Workshop on ML, 2009.

[14] M. Carlier, C. Dubois, and A. Gotlieb. Constraint reasoning in FocalTest. In *5th International Conference on Software and Data Technologies*. 2010.

[15] A. T. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics (AISTATS)*, 2013.

[16] S. Chakraborty, K. S. Meel, and M. Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proceedings of the 51st Annual Design Automation Conference*. 2014.

[17] H. R. Chamarthi, P. C. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. In *10th International Workshop on the ACL2 Theorem Prover and its Applications*, 2011.

[18] J. Christiansen and S. Fischer. EasyCheck – test data for free. In *9th International Symposium on Functional and Logic Programming (FLOPS)*. 2008.

[19] K. Claessen, J. Duregård, and M. H. Pałka. Generating constrained random data with uniform distribution. In *Functional and Logic Programming*. 2014.

[20] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2000.

[21] M. Dénès, C. Hriţcu, L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. QuickChick: Property-based testing for Coq. The Coq Workshop, 2014.

[22] 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.

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

[24] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. 2003.

[25] 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. In *24th European Symposium on Programming*. 2015.

[26] J. Fowler and G. Hutton. Towards a theory of reach. Draft of March 2015, 2015.

[27] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *32nd ACM/IEEE International Conference on Software Engineering*. 2010.

[28] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 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. In *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, 2008.

[31] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *International Conference on Software Engineering (ICSE Future of Software Engineering)*. 2014.

[32] A. Gotlieb. Euclide: A constraint-based testing framework for critical C programs. In *ICST 2009, Second International Conference on Software Testing Verification and Validation, 1-4 April 2009, Denver, Colorado, USA*, 2009.

[33] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 2012.

[34] P. V. Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artif. Intell.*, 57(2-3):291–321, 1992.

[35] C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2013.

[36] J. Hughes. QuickCheck testing for fun and profit. In *9th International Symposium on Practical Aspects of Declarative Languages (PADL)*. 2007.

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

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

[39] A. S. Köksal, V. Kuncak, and P. Suter. Scala to the power of Z3: integrating SMT and programming. In *23rd International Conference on Automated Deduction*. 2011.

[40] 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.

[41] F. Lindblad. Property directed generation of first-order test data. In *8th Symposium on Trends in Functional Programming (TFP)*, 2007.

[42] R. Majumdar and K. Sen. Hybrid concolic testing. In *29th international conference on Software Engineering*. 2007.

[43] V. K. Mansinghka, D. M. Roy, E. Jonas, and J. B. Tenenbaum. Exact and approximate sampling by systematic stochastic search. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, AISTATS 2009, Clearwater Beach, Florida, USA, April 16-18, 2009*, 2009.

[44] B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: probabilistic models with unknown objects. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, 2005.

[45] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artif. Intell.*, 28(2):225–233, 1986.

[46] A. V. Nori, C.-K. Hur, S. K. Rajamani, and S. Samuel. R2: An efficient mcmc sampler for probabilistic programs. In *AAAI Conference on Artificial Intelligence (AAAI)*. 2014.

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

[48] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems And Applications*. 2007.

[49] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*. 2011.

[50] M. Papadakis and K. F. Sagonas. A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*, 2011.

[51] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. To appear in ITP 2015, 2015.

[52] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, 2012.

[53] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *1st ACM SIGPLAN Symposium on Haskell*. 2008.

[54] E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, 2015.

[55] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 2005.

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

[57] E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014.

A. Appendix

A.1 Luck examples

In this part of the appendix we present the Luck programs that serve as both predicates and generators for the small examples of Section 6.

```
sig sorted :: [Int] → Bool
fun sorted l =
    case l of
    | (x:y:t) → x < y && sorted (y:t)
    | _ → True
    end

sig member :: Int → [Int] → Bool
fun member x l =
    case l of
    | h:t → x == h || member x t
    | _ → False
    end

sig distinctAux :: [Int] → [Int] → Bool
fun distinctAux l acc =
    case l of
    | []  → True
    | h:t → ⟦ not (member h acc)⟧^h
            && distinctAux t (h:acc)
    end

sig distinct :: [Int] → Bool
fun distinct l = aux l []
```

In order to obtain lists of a specific size, we could skew the distribution towards the `cons` case using numeric annotations on the branches, or, we can use the conjunction of such a predicate with the following simple length predicate (which could be greatly simplified with some syntactic sugar).

```
sig length :: [a] → Int → Bool
fun length l n =
    if n == 0 then
      case l of
      | [] → True
      | _  → False
      end
    else case l of
         | h:t → length t (n-1)
         | _ → False
         end
```

Finally, the Luck program that generates red black trees of a specific height is:

```
data Color = Red | Black
data RBT a = Leaf | Node Color a (RBT a) (RBT a)

fun isRBT h low high c t =
  if h == 0 then
      case (c, t) of
        | (_, Leaf) -> True
        | (Black, Node Red x Leaf Leaf) ->
          [| x | low < x && x < high |]
        | _ -> False
      end
  else case (c, t) of
         | (Red, Node Black x l r) ->
             [| x | low < x && x < high |]
             && isRBT (h-1) low x  Black l
             && isRBT (h-1) x high Black r
         | (Black, Node Red x l r) ->
             [| x | low < x && x < high |]
             && isRBT h low x  Red l
             && isRBT h x high Red r
  | (Black, Node Black x l r) ->
      [| x | low < x && x < high |]
      && isRBT (h-1) low x  Black l
      && isRBT (h-1) x high Black r
  | _ -> False
end
```