Formalising Luck

Improved Probabilistic Semantics for Property-Based Generators

Diane Gallois-Wong, supervised by Cătălin Hriţcu, INRIA Paris

March-August 2016

General Context

Property-Based Testing consists in running a program on many randomly generated inputs to invalidate an executable property. Popularized by QuickCheck (Claessen and Hughes, 2000), this technique is now widely used, both in programming languages (Arts et al., 2008; Lindblad, 2007; Hughes, 2007; Pacheco and Ernst, 2007) and in proof assistants (Chamarthi et al., 2011; Bulwahn, 2012; Owre, 2006; Dybjer et al., 2003; Paraskevopoulou et al., 2015). Frequently, the tested property takes the form of an implication $p \Rightarrow q$, where p is a precondition over the input: for example, we may assume a list to be a sorted, or a program to be well-typed if we are testing a compiler. The test is then a trivial success for any input not satisfying p, leading to inefficient testing if p is *sparse* (verified by a small proportion of the possible inputs), as in the examples above. To address this issue, QuickCheck allows the user to write a *property-based generator*, returning exclusively data satisfying property p. However, generators can easily be biased toward values sharing some specific structure; and worse, they can be incomplete, sometimes covering a very small part of the values satisfying p. In these cases, testing becomes quite hopeless, as potential counterexamples involving a structure that is not covered by the generator may remain undetected. Writing complete, balanced and efficient generators for complex properties is difficult and error prone. Luck (Lampropoulos et al., 2016a) (§1) is a language aimed at addressing this challenging problem.

Research Problem

Since Luck programs are generators, it can be understood as a probabilistic programming language. Devising semantics for probabilistic programming languages is a complex problem, which has recently gathered significant attention (Gordon et al., 2014; Borgström et al., 2015). Ideally, we want a semantics for Luck to be as expressive as possible, yet simple enough to allow for easy proofs which can be machine checked by a proof assistant. Previous Luck semantics (e.g., §2.5) were too complex to mechanize, which caused errors in some of the corresponding proof sketches to remain undetected for a long time.

Moreover, Luck's mixing of random choices and constraint solving sometimes leads to *failure* states with unsatisfiable constraints. In practice, any efficient implementation requires a backtracking strategy to handle these. Lacking a formal definition, backtracking strategies are not well integrated into Luck semantics.[1] Therefore we often cannot describe well the final probability distribution of a generator in Luck, which depends on the backtracking strategy applied.

[1] One previous attempt (Xia and Hriţcu, 2015) at making backtracking strategies part of Luck's semantics was complex yet not flexible enough.

Contribution

I proposed a new big-step operational probabilistic semantics style for Luck, based on recording choices made and their probability (§2). This style leads to a semantics that is much simpler than previous Luck semantics, and at least as expressive. I also proposed a formalism for defining backtracking strategies based on Markov chains (§3), together with an extension of the aforementioned choice-recording semantics integrating them, which provides the final probability distribution of a given Luck generator under a given strategy. I suggested ways to measure how accurately a strategy reproduces the distribution that we would most intuitively expect from a generator.

I also had the privilege to contribute to a POPL submission about Luck (Lampropoulos et al., 2016a), which uses the choice-recording semantics I proposed. Finally, I disproved a conjecture I was asked to prove, and which was assumed to hold by the Luck team for a very long time (§2.4).

Arguments Supporting its Validity

The choice-recording probabilistic semantics style is simple enough that a significant part of the proofs from the POPL submission could be formalized and checked in Coq by Leonidas Lampropoulos. This was considered too difficult with the previous semantics style. The new semantics is strictly more expressive than all the previous ones, since it handles programs with potential infinite execution traces and it provides support for a wide range of backtracking strategies.

The probability distribution returned by the extended semantics integrating backtracking strategies is computable from the Luck generator and the formal definition of the strategy applied (as I show formally in §3.4). The suggested measures of accuracy of a strategy are shown to achieve their minimum for the intuitive but expensive strategy of always restarting the computation from scratch when a failure is encountered (§2.4).

Conclusion and Future Work

The new choice-recording semantics for Luck is the first one to be simple enough for a few related proofs to be checked using Coq. Continuing in this direction, we should be able to formally verify a larger part of Luck, strengthening current claims and clearing out potential remaining bugs. The extent of this semantics' expressiveness came out as a surprise: handling programs with potential infinite executions traces was not a result I was actively seeking. This opens up the possibility to study such programs more closely. Formally defining backtracking strategies and a semantics integrating them is an important result, as when we write a generator, the probability distribution that it samples from (provided by our semantics) is what we are really interested in. Even better, it comes with a computability result. Formalising backtracking strategies also allowed me to tentatively set up various requirements about them, each of which led to its own interesting property about the resulting mapping from generators to distributions. Other requirements could certainly provide important properties. By providing measures of how accurately a strategy reproduces the intuitive distribution of a generator, I give concrete values to an approximative-looking concept. This will allow for quantitative comparisons of strategies used in practice. As I discuss very briefly, it could also be very interesting to mix (for example by summing) this accuracy penalty with a time complexity term (or any other function we may want to minimise) to evaluate a strategy on a combination of criteria.

1 Overview of Luck

Luck (Lampropoulos et al., 2016a) is a domain-specific programming language dedicated to writing Property-Based Generators. We first explain its motivation (§1.1) and high-level design principles (§1.2), before introducing its syntax (§1.3) and illustrating it by an example generating binary search trees (BSTs, §1.4).

1.1 Motivation

Property-Based Testing relies on *properties* given as executable predicates, to be tested on a large amount of randomly generated inputs. This provides either a counterexample, or a greater confidence in the correctness of the property the more tests pass. However, this confidence assumes the tests to be relevant, which is actually a challenging requirement.

For example, let us consider an insertion function for BSTs. A natural property to test states that if the input tree is a BST, then the output is also a BST. A naive approach would consist in generating random binary trees, most of which would not be BSTs, trivially verifying the property independently from the code of the insert function and making the corresponding testing totally irrelevant. It would take a very long time, spent generating trees and checking whether they are BSTs, before we have run a few BSTs. Even worse, most of the randomly generated trees which would happen to be BSTs would statistically be very small: running a naive tree generator (which produces big trees with a reasonable probability) and keeping only the BSTs, (Paraskevopoulou et al., 2015) obtain 95.3% of trees with 1 node, 4.2% with 3 nodes, 0.4% with 5 nodes, and only 0.03% with 7 or 9 nodes. Therefore the whole testing process is quite hopeless, as bugs involving complex trees would extremely likely remain undetected.

This problem is solved by carefully designed Property-Based Generators. In this example, we want to sample inputs to test using a generator that produces only BSTs, has a reasonable chance to generate even complex or structurally curious ones, and is efficient in terms of generation time. However, writing generators for complex properties is often a challenging task, sometimes a research problem on its own (Hriţcu et al., 2015; Pałka et al., 2011).

Moreover, the property we want from generation often coincides with one we want to check about the output: in this example, we want to both generate BST inputs and check that outputs are BSTs too. This requires both a generator and an executable predicate for the same properties; they need to be kept synchronised, which is a common source of errors.

1.2 Chosen Approach

Luck programs take a predicate form, which is easier for the user to write and understand. In order to be used as generators, they Luck programs can be executed on *unknown* inputs (basically logic variables): execution tries to randomly determine concrete values for these unknowns which make the whole predicate evaluate to *True*.

This kind of automatic generation based on a predicate has already been well studied; Luck combines two traditional approaches, each of them having its own strength and weaknesses. One of them relies on constraint solving (Carlier et al., 2010; Seidel et al., 2015; Gotlieb, 2009; Köksal et al., 2011; Godefroid et al., 2005; Sen et al., 2005; Cadar et al., 2008; Avgerinos et al., 2014; Torlak and Bodík, 2014). The other one randomly instantiates an unknown as soon as it is used for the first time in a non-parametric way, invoking backtracking when some choices make later constraints impossible to satisfy (Antoy, 2000; Hanus, 1997; Lindblad, 2007; Tolmach and Antoy, 2003; Fischer and Kuchen, 2007; Christiansen and Fischer, 2008; Reich et al., 2011; Gligoric et al., 2010; Claessen et al., 2014; Fetscher et al., 2015). Luck's default strategy is constraint solving, yet several of the Luck constructs allow the user to control random instantiation. For

example, forcing early instantiation of an unknown that heavily affects control flow may prevent a big complexity blow-up from symbolically executing multiple branches.

A Luck program can also be directly interpreted as its underlying predicate, by providing concrete values instead of unknowns as inputs, and ignoring the aforementioned Luck constructs controlling instantiation. It is useful to allow a single program to be interpreted as either an actual predicate or a generator, since it prevents mismatching bugs in the common scenario where we are testing a preservation property, such as insertion preserving the BST property or taking steps preserving well-typedness (i.e., subject reduction of a type system).

1.3 Syntax

$$
\begin{array}{rcl}
e & ::= & () \mid x \\
& \mid & (e, e) \mid \texttt{case}\ e\ \texttt{of}\ (x, x) \to e \\
& \mid & \texttt{L}_T\ e \mid \texttt{R}_T\ e \mid \texttt{case}\ e\ \texttt{of}\ (\texttt{L}\ x \to e)\ (\texttt{R}\ x \to e) \\
& \mid & \texttt{rec}\ (f : T \to T)\ x = e \mid e\ e \\
& \mid & \texttt{fold}_T\ e \mid \texttt{unfold}_T\ e \\
& \mid & u \mid\ !e \mid e \leftarrow (e, e) \mid e\ ;\ e \\
T & ::= & X \mid 1 \mid T + T \mid T \times T \mid \mu X.\ T \mid T \to T \\
\overline{T} & ::= & X \mid 1 \mid \overline{T} + \overline{T} \mid \overline{T} \times \overline{T} \mid \mu X.\ \overline{T}
\end{array}
$$

Figure 1: Luck Expressions and Types

Standard expressions The syntax of Luck expressions is given in Figure 1. Except for the last line, it is a standard simply typed call-by-value lambda calculus with pairs, binary sums and isorecursive types. Recursive lambdas are included for convenience, although in principle they could be encoded using recursive types. The sum constructors are `L` and `R`, annotated with a type to prevent any ambiguity. `case` expressions provide destructors for pairs and sums. `fold` and `unfold` simply witness the isomorphism between a recursive type and its unfolding.

Nonstandard expressions The value u stands for an unknown drawn from a countably infinite set. The *sampling* construct $!e$ fully instantiates every unknown present in e with a uniform distribution between the possibilities, assumed to be finite. The *head instantiation* construct $e \leftarrow (n_1, n_2)$ evaluates the sum-typed expression e to an unknown and instantiates it to an `L` or a `R` form with respective probabilities $\frac{n_1}{n_1+n_2}$ and $\frac{n_2}{n_1+n_2}$. It does nothing if e does evaluates to a value that is not an unknown or to an unknown that is already instantiated. n_1 and n_2 must be positive integers and are fully instantiated beforehand as if they were in a *sample* expression. The *after* construct $e_1\ ;\ e_2$ is similar to a usual sequence construct, except that the result of the first expression is kept instead of the second one: e_1 is evaluated to a value v, then e_2 is evaluated for its side-effects only, and the whole expression yields v.

Types Types (T in Figure 1) are usual for the standard lambda calculus except for the last line (1 is the type of ()). The typing rules are exactly as expected for the standard expressions. The nonstandard expressions do not add new type constructs, but they call for the definition of a restrictive class of types: we note $\overline{T}$ for types which do not contain any function type. Unknowns are restricted to types of form $\overline{T}$, because unknowns involving functions would be much more complex to handle. We require e to be of type $\overline{T}$ in $!e$ and in $e \leftarrow (n_1, n_2)$ (this is also implicitly true of n_1 and n_2, due to the stronger condition to have natural type). Indeed,

instantiation is intended to act on unknowns, and although we allow complex expression whose unknowns are recursively affected, there is no need for the additional complexity of functions.

Booleans, integers, associated operations The standard expressions of the lambda calculus above allow usual encodings of common features such as booleans, naturals, lists, etc. Indeed, we can define $\mathit{Bool} := 1 + 1$, $\mathit{True} := \mathtt{L}_{1+1}\ ()$, $\mathit{False} := \mathtt{R}_{1+1}\ ()$. We can then define an if-then-else construct and common operators on booleans: for example a `&&` b could be `case` a `of` (`L` _ $\to$ b) (`R` _ $\to$ False). For natural numbers we can use Peano encoding: $\mathit{Nat} := \mu X.\ 1 + X$, $0 := \mathtt{L}_{1+\mathit{Nat}}\ ()$, $\mathit{Succ}\ n := \mathtt{R}_{1+\mathit{Nat}}\ n$, and define common operations and predicates such as equality or any comparison. From now on, we assume all these to be defined.

1.4 Example: Binary Search Tree Generator

```
rec (bst : Nat -> Nat -> Nat -> NatTree -> Bool)
  size low high tree =
    if size == 0 then tree == Empty
    else case (tree <- (1,size)) of
      (Empty -> True)
      (Node x l r ->
        ((low < x && x < high) ; !x)
          && bst (size / 2) low x l
          && bst (size / 2) x high r)
```

Figure 2: Binary Search Tree (Predicate and) Generator in Luck

Figure 2 shows a recursive Luck function for Binary Search Trees, understandable both as a predicate checking whether an input is a BST and as a generator for BSTs. It uses basic Luck syntax (the function definition abusively taking several arguments at once for concision) and common definitions admitted above; we also assume some elements to handle *Nat* trees to be defined: $\mathit{NatTree} := \mu X.\ 1 + \mathit{Nat} \times X \times X$, $\mathit{Empty} := \mathtt{L}_{1+\mathit{Nat}\times\mathit{NatTree}\times\mathit{NatTree}}\ ()$, etc.

We obtain a standard predicate by ignoring non-standard Luck constructs and calling bst with concrete values for size (over-approximating upper bound on the number of nodes), low and high (bounds on the integers contained in the nodes): this yields a boolean function taking a single argument tree. We obtain a parametrised BSTs generator by calling bst with concrete values for size, low and high, and with an unknown for tree: execution will instantiate it to a concrete value such that the call returns *True*.

The most interesting are the uses of non-standard Luck constructs and their intended influence on the derived generator. A head instantiation is used for the discriminee of the `case`: it ensures that the *Empty* branch (where *tree* has form $\mathtt{L}_{\_}$ ..) will be taken with probability $\frac{1}{1+\mathit{size}}$ (provided *tree* was not instantiated, otherwise this has no effect). This prevents the generator from producing single node trees too often, which would hinder testing as discussed in §1.1. The full instantiation $!x$ is carefully placed. The constraints that x is between *low* and *high* have already been gathered, whereas serious backtracking would necessary if the instantiation were to happen tight after $\mathsf{low} < x$, as would be the case in techniques that automatically instantiate variables on their first use (Claessen et al., 2014; Fetscher et al., 2015). The *after* construct is exactly what we want here: the members are evaluated in the right order, yet we keep the value from the boolean expression. On the other hand, it is best to have instantiated x before the recursive calls involving it, which would otherwise blow up constraint solving complexity, since case analysis over an unknown has to evaluate both branches and combine the produced constraints.

2 Choice-Recording Probabilistic Semantics

We present our choice-recording semantics: we explain its main components (§2.1), provide a few examples of derivation rules (§2.2) and give examples of derivations for given inputs and of how to extract probabilities distributions (§2.3). In §2.4 I explain about a conjecture that I was supposed to prove, to which I actually found a counterexample. We introduce the previously favoured collecting semantics and explain why our semantics is strictly more expressive (§2.5). Finally, we present properties of soundness and completeness (§2.6), some of which have been proven in Coq by a member of the Luck team.

A detailed presentation of this semantics can be found in (Lampropoulos et al., 2016a), including all the rules which we cannot show here. Note that there is a slight difference compared to our presentation, that we explain when we introduce traces in §2.1; it is not very relevant and the two versions are equivalent.

2.1 Overview of the Semantics

In this section we introduce a big-step, operational semantics for Luck, however, there are several ways to handle the probabilistic aspect of Luck. Previous semantics styles required a single derivation to gather every possible outcome of a program, whereas this semantics style derives only one possible outcome at a time. This leads to rules which are simpler, both to write and to handle in proofs. We rely on choice-recording *traces*, described below, to still be able to determine all possible outcomes and their respective probabilities. This semantics describes the same evaluation strategy as a former semantics devised by Leonidas Lampropoulos *et al.*, but in a simpler, more abstract way. Some characteristics that are shared with Lampropoulos' previous semantics are the use of *constraint sets* to handle the constraint solving aspect of Luck, and an intermediate *narrowing* judgment, used by the rules of the main *matching* judgment, which is the one used for Property-Based Generation.

Traces Traces record probabilistic choices made during evaluation. A *trace* t is a sequence of *choices* of the form (m, n, q) with $m, n \in \mathbb{N}$, $q \in \mathbb{Q}$, $0 \leqslant m < n$, $0 < q \leqslant 1$, where n is the number of possibilities, m is the index of the one actually taken, and q is the probability of making this choice. The *probability of a trace* t, denoted $P(t)$, is the product of the probabilities of all its choices. Traces play no role in determining how evaluation proceeds. However, they are useful after the fact in calculating the total probability of some given outcome of evaluation, which may be reached by many different derivations. They will also allow us to determine whether some of the executions are non-terminating.

The presentation of the semantics in (Lampropoulos et al., 2016a) uses traces where choices are couples (m, n), where m and n have the same signification as above. Then, instead of storing the probability of each choice in the trace, we separately record the probability of the whole trace. These approaches are equivalent.

Values Luck values are standard for the lambda calculus on which it is based: unit, lambdas and constructors applied to values, with the addition of unknowns.

$$v ::= () \mid (v, v) \mid \mathtt{L}_T\ v \mid \mathtt{R}_T\ v \mid \mathtt{rec}\ (f : T_1 \to T_2)\ x = e \mid \mathtt{fold}_T\ v \mid u$$

Constraint Sets A generic interface for *constraint sets* allows to integrate constraint solving into the semantics, while leaving a high degree of freedom regarding solver implementations. A constraint set $\kappa \in \mathscr{C}$ ranges over a finite subset of the infinite set of unknowns $\mathcal{U}$; it contains the types of its unknowns, as well as information about their possible values. A *valuation* of

a constraint set is a mapping from unknowns to unknown-free values of the specified types verifying every constraint. The *denotation* $[\![\kappa]\!]$ of a constraint set κ is the set of its valuations. Required operations on constraint sets include addition of fresh unknowns with specified types, *unification* of two values (addition of the constraint that they are equal), test of satisfiability (whether its denotation is nonempty), and full instantiation of an unknown. Note that value equality constraints are more expressive than it may seem: for example with Peano encoding of integers ($Nat := \mu X.\ 1 + X$), introducing a fresh u of type Nat and unifying x with $\mathtt{R}_{1+Nat}\ \mathtt{R}_{1+Nat}\ \mathtt{R}_{1+Nat}\ u$ forces $x \geqslant 3$.

Narrowing judgment The narrowing evaluation judgment takes as inputs an expression e and a constraint set κ. As usual, evaluating e returns a value v, but now this value depends on a constraint set κ. Evaluation also returns a new constraint set κ', as well as the trace t of random choices made:

$$e \rightleftharpoons \kappa \Downarrow_t \kappa' \vDash v$$

Everything is required to be well typed, and κ is required to be satisfiable. κ' is intuitively κ updated with new constraints recording all random choices made. As there are no prior requirements on v we can proceed with the evaluation and return whichever value we obtain, as opposed to matching judgments below. For narrowing backtracking is never needed and κ' is always satisfiable. As an example, consider evaluating the BST program given in §1.4 using the narrowing semantics: there are no inherently wrong choices, only choices leading evaluation to return *False* which is completely allowed in this semantics.

Matching judgment As explained in §1.2, to use a Luck program as a Property-Based Generator, we want to instantiate unknowns so that the whole expression evaluates to *True*. Instead of returning a value, the *matching* judgment takes as additional input a *pattern* p: a value possibly containing unknowns but not lambdas, indicating a desired form for the value that e evaluates to. Typically, p will be *True* when e is the full program, but will become more complex when executing the sub-expressions of e. Evaluation now returns an *optional* constraint set $\kappa^?$: either failure $\varnothing$ or success $\{\kappa'\}$ for some satisfiable κ'.

$$p \Leftarrow e \rightleftharpoons \kappa \Uparrow_t \kappa^?$$

If $\kappa^?$ is $\varnothing$, the choices described in the trace t make it impossible for e to evaluate to a value of form p and backtracking is needed. If $\kappa^?$ is $\{\kappa'\}$, e evaluates to a value of form p when making the choices in t which also lead to the constraints described by κ'.

The input expression e of a matching judgment always has a type of form $\overline{T}$, since e is intended to evaluate to the pattern p, which cannot contain lambdas. Handling functions and their application is one of the reasons why we need to invoke narrowing in matching rules.

Interpretation of matching judgment Coming back to the BST example from §1.4, whenever the output of matching the *True* pattern is a success $\{\kappa'\}$, the unknown referenced by the variable *tree* is fully instantiated in κ' to a BST. The probability to immediately generate (without any backtracking) a given BST can be determined by summing the probabilities of all traces leading to some κ' where *tree* is this BST. We obtain a probability distribution over the set of BSTs augmented with an element $\varnothing$ representing a need to backtrack. Getting from this to a final probability distribution over valid values by taking into account the effect of backtracking on the probabilities is the purpose of §3.

$$\textbf{M-Base}\ \frac{e = () \vee e \in \mathcal{U} \qquad \kappa' = \mathit{unify}\ \kappa\ e\ p \qquad \mathit{SAT}(\kappa')}{p \Leftarrow e \dashv \kappa \Uparrow_{\epsilon} \{\kappa'\}}$$

$$\textbf{M-Base-Fail}\ \frac{e = () \vee e \in \mathcal{U} \qquad \kappa' = \mathit{unify}\ \kappa\ e\ p \qquad \neg \mathit{SAT}(\kappa')}{p \Leftarrow e \dashv \kappa \Uparrow_{\epsilon} \varnothing}$$

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

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

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

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

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

$$\textbf{M-App}\ \frac{\begin{array}{c}e_0 \dashv \kappa \Downarrow_{t_0} \kappa_0 \vDash (\texttt{rec}\ (f : T_1 \to T_2)\ x = e_2) \\ e_1 \dashv \kappa_0 \Downarrow_{t_1} \kappa' \vDash v_1 \\ p \Leftarrow e_2[(\texttt{rec}\ (f : x \to T_1)\ T_2 = e_2)/f, v_1/x] \dashv \kappa' \Uparrow_{t_2} \kappa^?\end{array}}{p \Leftarrow (e_0\ e_1) \dashv \kappa \Uparrow_{t_0 \cdot t_1 \cdot t_2} \kappa^?}$$

$$\textbf{N-After}\ \frac{e_1 \dashv \kappa \Downarrow_{t_1} \kappa_1 \vDash v_1 \qquad e_2 \dashv \kappa_1 \Downarrow_{t_2} \kappa_2 \vDash v_2}{e_1\ ;\ e_2 \dashv \kappa \Downarrow_{t_1 \cdot t_2} \kappa_2 \vDash v_1}$$

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

Figure 3: Examples of Narrowing (**N-**) and Matching (**M-**) Rules

2.2 Example Rules

We provide a few examples of rules in Figure 3. We assume an initial type-checking phase has been performed to determine the type of everything we encounter: we write e^T when e having type T is relevant.

Selected narrowing rules **N-Base** is the base case of the evaluation relation, treating values that are not handled by other rules by returning them as-is ($\mathcal{U}$ is the set of unknowns). There is no choice to be made, so the trace is empty. To evaluate (e_1, e_2) in **N-Pair** we sequence derivations for e_1 and e_2: the constraint set produced by the first judgment is given as argument to the second one, and traces are accumulated through concatenation. This is essentially a bind in a mix of state and writer monads. Most of the rules involve this sequencing, which we will not point out any more. In **N-App**, typing ensures that the evaluation of e_0 returns a lambda. We then evaluate the argument, and lastly the body of the function with appropriate substitutions. **N-After** is similar to **N-Pair**; however as explained in §1.3, the value obtained from the first expression is returned.

Constraint set operations used in selected matching rules *fresh* takes a constraint set κ and a list of types Ts of length l, and returns (κ', us), where us is a list of l unknowns absent from κ, and κ' is the constraint set obtained by extending κ with these unknowns and their respective types from Ts. *unify* $\kappa\ v_1\ v_2$ returns κ updated with the additional constraint that the values v_1 and v_2 (potentially containing unknowns) should be equal. $SAT(\kappa)$ returns a boolean indicating whether κ is satisfiable (the set of its valuations is not empty).

Selected matching rules Recall that patterns, thus also input expressions of matching judgments, have types of form $\overline{T}$. Therefore in **M-Base**, there is no case for function unlike in **N-Base**. Moreover, we now unify v with the target pattern p, and we check the satisfiability of κ' to maintain the property that a success always contains a satisfiable constraint set: if it is not satisfiable, we return a failure in **M-Base-Fail**. In **M-Pair**, to evaluate a pair (e_1, e_2) where the components have types $\overline{T}_1$ and $\overline{T}_2$, we introduce fresh unknowns u_1 and u_2 with these types. After unification of (u_1, u_2) with the pattern p, these unknowns provide respective patterns for the evaluations of e_1 and e_2. These evaluations can be sequenced only if they are successful. Otherwise the whole evaluation of (e_1, e_2) immediately returns a failure as well, via rules such as **M-Pair-Fail**. **M-App** and **M-After** are similar to **N-App** and **N-After**. Note however the reference to the narrowing judgment in these matching rules. The evaluation of e_0 in **M-App** cannot use matching because as a function, e_0 does not have type $\overline{T}$. The evaluation of e_1 cannot either because it can also be a function. In **M-After**, we do not have any pattern to provide for the evaluation of e_2, which we carry out only for its side-effects.

2.3 Examples of Derivation and Distribution Reconstruction

As an example, we provide the main steps of the matching derivations of two given expressions against the pattern *True* in a given constraint set. We also show how to extract probability distributions about optional constraint sets from the derivations we obtain.

We are going to evaluate $A := (0 < u \texttt{ \&\& } u < 4)\ ;\ !u$ and $B := (0 < u\ ;\ !u) \texttt{ \&\& } u < 4$ against the pattern *True* in a constraint set κ, in which u is independent from other unknowns and its possible values are $0, ..., 9$.

Recall that e_1 `&&` e_2 is shorthand for `case` e_1 `of` (`L` $a \to e_2$) (`R` $b \to$ *False*), and that we are using a standard Peano encoding of natural numbers: $Nat = \mu X.1 + X$. We elide folds which

would make the expressions too heavy. $a < b$ can be encoded as $lt\ a\ b$ where

$$\begin{aligned} lt = \ & rec\ (f : Nat \to Nat \to Bool)\ x = rec\ (g : Nat \to Bool)\ y = \\ & \texttt{case}\ y\ \texttt{of}\ (\texttt{L}\ \_ \to False)\ (\texttt{R}\ y_R \to \\ & \qquad \texttt{case}\ x\ \texttt{of}\ (\texttt{L}\ \_ \to True)\ (\texttt{R}\ x_R \to f\ x_R\ y_R)\ ) \end{aligned}$$

We assume that evaluation of these constructs gives the intuitive results. This can be shown using the multiple and complex rules for `case` that we do not present, provided we accept the notation abuse explained thereafter. Many rules, included those for `case`, introduce fresh unknowns, many of which are irrelevant: they might be directly equivalent to some other unknown, or there might not exist any reference to them. We abusively use the same variable for two constraint sets which differ only in the addition of a few irrelevant variables to one of them.

Evaluation of A We derive $True \Leftarrow (0 < u) \dashv \kappa \Uparrow_\epsilon \{\kappa_0\}$, where the domain of u in κ_0 is $\{1, ..., 9\}$ (when using the actual `case` rules, we note that the refinement of the domain of u comes from an unification of u with $\texttt{R}_{1+Nat}\ u_0$ where u_0 is a fresh unknown). Then $True \Leftarrow (0 < u\ \texttt{\&\&}\ u < 4) \dashv \kappa \Uparrow_\epsilon \{\kappa_1\}$, where the domain of u in κ_1 is $\{1, 2, 3\}$. There are 3 possible narrowing derivations for $!u$: $!u \dashv \kappa_1 \Downarrow_{[(0,3,\frac{1}{3})]} \kappa_1^A \vDash u$, $!u \dashv \kappa_1 \Downarrow_{[(1,3,\frac{1}{3})]} \kappa_2^A \vDash u$ and $!u \dashv \kappa_1 \Downarrow_{[(2,3,\frac{1}{3})]} \kappa_3^A \vDash u$, where the domain of u in κ_i^A is $\{i\}$. (We have switched to narrowing judgments because of the rule **M-After**.) Therefore all the possible derivations for $A = (0 < u\ \texttt{\&\&}\ u < 4)\ ;\ !u$ matching $True$ in κ are:

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

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

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

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

$$\begin{aligned} & True \Leftarrow B \dashv \kappa \Uparrow_{[(i-1,9,\frac{1}{9})]} \{\kappa_i^B\} && \text{for } i \in \{1, 2, 3\} \\ & True \Leftarrow B \dashv \kappa \Uparrow_{[(i-1,9,\frac{1}{9})]} \varnothing && \text{for } i \in \{4, ..., 9\} \end{aligned}$$

We can compute the corresponding probability distribution:

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

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

2.4 Disproving a conjecture about the consequences of instantiation

The probability associated to $\varnothing$ in the final distribution is the probability of backtracking. In the examples from the last subsection that is 0 for A, 2/3 for B. Without surprise, $A = (0 < u \texttt{ \&\& } u < 4)\ ;\ !u$ is far better than $B = (0 < u\ ;\ !u) \texttt{ \&\& } u < 4$ in terms of backtracking: it is a far better idea to process the constraint $u < 4$ before instantiating u. B would actually have a prohibitively high probability to backtrack if the initial domain of u were not $\{0, ..., 9\}$ but a lot larger.

For a long time, the Luck team assumed the following conjecture to hold: adding an instantiation (either sampling $!e$ or head instantiation $e \leftarrow (e_1, e_2)$) can only increase the probability of backtracking (for expressions otherwise identical, evaluated against the same pattern in the same constraint set). This seems intuitively correct, as instantiation creates a risk to later encounter an incompatible constraint, which immediately triggers backtracking. This was even explicitly conjectured in a previous version of the paper.

Trying to prove the conjecture, I eventually came up with a counterexample. The following more intuitive counterexample was then provided by another member of the Luck team. If κ contains a single unknown u of type *Nat* with possible values $0, ..., 9$, then $(\mathit{True}\ ;\ !u) \texttt{ \&\& } (0 < u)$ evaluated against *True* in κ has a probability $\frac{1}{10}$ to backtrack (it backtracks exactly when u is instantiated to 0), while $(\mathit{True}\ ;\ u \leftarrow (a, b)\ ;\ !u) \texttt{ \&\& } (0 < u)$ evaluated against *True* in κ has a probability $\frac{a}{a+b}$ to backtrack, a and b being of type *Nat*. Indeed, keeping in mind that $\mathit{Nat} = \mu X.1 + X$, $u \leftarrow (a, b)$ unifies u with $\mathtt{L}_{1+\mathit{Nat}}\ u_L$ (where u_L is a fresh unknown of type 1) with probability $\frac{a}{a+b}$, forcing it to be 0, and it unifies u with $\mathtt{R}_{1+\mathit{Nat}}\ u_R$ (where u_L is a fresh unknown of type *Nat*) with probability $\frac{b}{a+b}$, forcing it to be greater than 0. In the first case, $!u$ has no effect and we need to backtrack when we evaluate $0 < u$ against *True*. In the second case, we already have that u is greater than 0 so, even taking the evaluation of $!u$ into account, we do not need to backtrack. In addition to being a counterexample to the conjecture, this is interesting as it shows that we are able to set the probability of backtracking to any arbitrary rational number between 0 and 1 by adding an instantiation.

2.5 Recovering the previous collecting semantics and handling infinite paths

The collecting semantics We call the previous semantics style of Luck the *collecting* semantics. As opposed to our new semantics, a collecting judgment gathers every possible outcome of an expression (derivation is thus deterministic). It also uses constraint sets and distinguished *narrowing* and *matching* judgments with respective forms

$$e \Rightarrow \kappa \Downarrow^{\mathit{coll}} [\kappa_i \vDash v_i \mapsto q_i]_i \quad \text{and} \quad p \Leftarrow e \Rightarrow \kappa \Uparrow^{\mathit{coll}} [\kappa_i \mapsto q_i]_i$$

The left-hand sides of the judgments are the same as in the choice-recording semantics. The right-hand sides are sub-probability distributions: the q_i are probabilities adding up to at most 1. They range over pairs of a constraint set and a value for narrowing, and just over constraint sets for matching. For example, here is the collecting narrowing rule for pairs, which is essentially the bind of the sub-probability monad.

$$\textbf{Coll-N-Pair}\ \frac{\begin{array}{c} e_1 \Rightarrow \kappa \Downarrow^{\mathit{coll}} [\kappa_i \vDash v_i \mapsto q_i]_i \\ \forall i.\ e_2 \Rightarrow \kappa_i \Downarrow^{\mathit{coll}} [\kappa_{ij} \vDash v_{ij} \mapsto q_{ij}]_j \end{array}}{(e_1, e_2) \Rightarrow \kappa \Downarrow^{\mathit{coll}} [\kappa_{ij} \vDash (v_i, v_{ij}) \mapsto q_i * q_{ij}]_{ij}}$$

Just as the choice-recording semantics never needs to backtrack, narrowing collecting judgments contain an actual probability distribution summing up to 1. On the other hand, in the collecting matching judgment the use of sub-probability distributions replaces constraint set

options: the remaining weight $1 - \sum_i q_i$ should be understood as the probability of failure and need to backtrack (corresponding to $\varnothing$).

From choice-recording semantics to sub-probability distributions For given inputs e, κ (and p for matching) we compute the corresponding (sub-)probability distribution by associating to each possible outcome the sum of the probabilities of all the traces leading to them (as we have also done in the examples in §2.3). More formally, we define, for all p, e, κ:

$$\pi_{e \dashv \kappa} := [\ (\kappa' \vDash v) \ \mapsto \sum_{t \mid e \dashv \kappa \Downarrow_t \kappa' \vDash v} P(t)\ ]$$

$$\pi_{p \Leftarrow e \dashv \kappa} := [\ \kappa^? \ \mapsto \sum_{t \mid \kappa \Leftarrow e \dashv p \Uparrow_t \kappa^?} P(t)\ ]$$

Recovering the collecting semantics *if the expression always terminates* We want to obtain the collecting semantics by computing $\pi_{e \dashv \kappa}$ or $\pi_{p \Leftarrow e \dashv \kappa}$. There is a difficulty related to infinite execution paths. Indeed, as soon as there is one possible infinite execution path, the collecting semantics does not terminate, as it tries to gather every possible path into a distribution. Conversely, the choice-recording semantics produces a judgment for each terminating path, even if there are other infinite paths. Therefore we cannot have for example: if $\pi_{e \dashv \kappa}$ exists then $e \dashv \kappa \Downarrow^{coll} \pi_{e \dashv \kappa}$. Our first approach consists in still invoking the collecting semantics, just to check whether it terminates.

Conjecture 2.5.1. If $e \dashv \kappa \Downarrow^{coll} d$ then $d = \pi_{e \dashv \kappa}$. If $p \Leftarrow e \dashv \kappa \Uparrow^{coll} sd$ then sd is obtained from $\pi_{p \Leftarrow e \dashv \kappa}$ by replacing $\{\kappa'\}$ with κ' in its domain and leaving $\varnothing$ out. (Free variables are considered to be quantified universally.)

The result above is already very interesting, as most of the programs we may want to study have no possible infinite path anyway. However, to really allow to recover the collecting semantics, ours should be able to determine on its own whether a collecting judgment will exist. To achieve this we will study sets of traces.

Recovering the collecting semantics: general case Recall that traces are sequences of choices (m, n, q): index of choice (m), number of possibilities (n so that $0 \leqslant m < n$), probability (q). We note $Tr_{e \dashv \kappa}$ or $Tr_{p \Leftarrow e \dashv \kappa}$ the set of all possible traces of judgments with corresponding inputs, $pre(S)$ the set of prefixes of traces in S:

$$Tr_{e \dashv \kappa} := \{t \mid \exists \kappa' v.\ e \dashv \kappa \Downarrow_t \kappa' \vDash v\} \qquad pre(S) := \{t \mid \exists t'.\ t \cdot t' \in S\}$$

Definition 2.5.2. A set of traces S is *closed* if for all t, m, n, q:

$$t \cdot (m, n, q) \in pre(S) \implies \forall m' \in \{0, 1, ..., n-1\}.\ \exists q'.\ t \cdot (m', n, q') \in pre(S)$$

A set of traces is *well-formed* if it is closed and finite.

Intuitively, $Tr_{e \dashv \kappa}$ is closed if for every choice made during a terminating execution of $e \dashv \kappa$, taking any of the other possibilities would also have led to a terminating execution.

Conjecture 2.5.3. If $Tr_{e \dashv \kappa}$ is well-formed then $e \dashv \kappa \Downarrow^{coll} \pi_{e \dashv \kappa}$ is the only collecting judgment with input $e \dashv \kappa$, otherwise there is no such judgment. If $Tr_{p \Leftarrow e \dashv \kappa}$ is well-formed then $p \Leftarrow e \dashv \kappa \Uparrow^{coll} sd$, where sd is obtained from $\pi_{p \Leftarrow e \dashv \kappa}$ by replacing $\{\kappa'\}$ with κ' in its domain and leaving $\varnothing$ out, is the only collecting judgment with input $p \Leftarrow e \dashv \kappa$, otherwise there is no such judgment. (Free variables are considered to be quantified universally.)

This result allows us to exactly obtain all results of the collecting semantics from the choice-recording one. This proves that the latter is at least as expressive as the former. Moreover, as our semantics provides much more precise information about programs with at least one infinite path, it is actually strictly more expressive.

2.6 Soundness and Completeness Properties

An upside of having simpler rules compared to the previous semantics is that properties are easier to prove. In particular, all properties presented here about the narrowing judgments have been proven in Coq by Leonidas Lampropoulos, while this was considered too difficult to attempt with other semantics. The properties about matching have hand-written proofs in Lampropoulos et al. (2016a).

Definitions: predicate semantics, instantiation Recall that a *valuation* of a constraint set κ is a mapping from unknowns to unknown-free values of the specified types verifying every constraint, and its *denotation* $[\![\kappa]\!]$ is the set of its valuations. We say that $e \Rightharpoonup \kappa$ *instantiates to* E, and we write $e \Rightharpoonup \kappa \rightsquigarrow E$, if there is a valuation σ of κ such that substituting the unknowns in e with their image in σ gives the unknown-free well-typed expression E. We suppose that a *predicate semantics* is defined: it evaluates any unknown-free expression E to an unknown-free value V, noted $E \Downarrow^{pred} V$, classically handling standard constructs and ignoring any non-standard Luck construct (more precisely, to evaluate $!e$, $e \leftarrow (e_1, e_2)$, or $e \mathbin{;} e_1$ we simply evaluate e).

Narrowing soundness and completeness We define soundness and completeness in comparison with the standard semantics introduced above. They can be visualised with the following diagram:

$$\begin{array}{ccc} E & \xrightarrow{\Downarrow^{pred}} & V \\ \rightsquigarrow \uparrow & & \rightsquigarrow \uparrow \\ e \Rightharpoonup \kappa & \xrightarrow{\Downarrow_t} & \kappa' \vDash v \end{array}$$

Soundness means that given the bottom and right-hand side of the diagram, we can fill in the top and left: a narrowing derivation $e \Rightharpoonup \kappa \Downarrow_t \kappa' \vDash v$ corresponds to a predicate derivation leading to V, for any V obtained by fully instantiating v in κ'. Completeness is the opposite direction on the diagram: given a predicate derivation $E \Downarrow^{pred} V$ and narrowing inputs $e \Rightharpoonup \kappa$ that describe E amongst other expressions, there is a narrowing derivation producing outputs which can be instantiated to V.

Matching soundness In order to allow experiments with different compromises between precision and speed, the requirements on constraint sets are very low. In particular, the interface contains a *union* operation whose specification is not very demanding: $[\![union\ \kappa_1\ \kappa_2\ \kappa]\!]$ should contain all the valuations of κ_1 and of κ_2 *and possibly more*, yet be included in the denotation of κ (given as a third argument to keep a reasonable bound). This imprecision makes it actually possible to generate constraint sets which do not satisfy the program understood as a predicate. This could be addressed by strengthening the requirements on *union*; however having to run a stronger constraint solver may be worse, in terms of execution speed, than adding a cheap final check that the constraint set produced satisfies the predicate, and resorting to backtracking in case it does not. The fix described is the reason why it is usually fine to assume that every output is valid. Nonetheless, typical soundness is not formally achieved. We still have a weaker

soundness result: consider a matching derivation $p \Leftarrow e \backsimeq \kappa \Uparrow_t \{\kappa'\}$ *where* κ *has a unique valuation* ($card\,[\![\kappa]\!] = 1$), then for any V verifying $v \vDash \kappa' \rightsquigarrow V$, there exists E with $e \backsimeq \kappa \rightsquigarrow E$ such that this derivation corresponds to the predicate derivation $E \Downarrow^{pred} V$.

Matching completeness Because the matching semantics explores both branches of a *case* when none of them brings a contradiction, it can fall into a loop even if there exists a terminating path for the predicate semantics. Therefore the completeness result is also weaker than usual. Given a predicate derivation $E \Downarrow^{pred} V$ and narrowing inputs verifying $e \backsimeq \kappa \rightsquigarrow E$, *if for any other instantiation* $e \backsimeq \kappa \rightsquigarrow E'$ *its predicate execution terminates:* $\exists V'.\, E' \Downarrow^{pred} V'$*, then* there is a matching derivation $p \Leftarrow e \backsimeq \kappa \Uparrow_t \{\kappa'\}$ with $\kappa' \vDash p \rightsquigarrow V$.

3 Formal investigation of backtracking strategies

In this part, we only consider Luck generators $p \Leftarrow e \backsimeq \kappa$ which have at least a successful derivation and no possible infinite execution paths. Our choice-recording semantics, just like the former collecting semantics, can produce a probability distribution about *optional* constraint sets. In practice however, we never output a failure $\varnothing$, but we backtrack to obtain a new output, which may be another failure in which case we backtrack again, and so on until we get a success $\{\kappa\}$. Therefore, taking backtracking into account, we have a probabilistic procedure whose output is always an actual constraint set and there is a corresponding probability distribution over proper constraint sets (considering only generators with at least one success, and assuming that the backtracking strategy is well designed so that it will always find a success with probability 1). For example, the simplest backtracking strategy consists in restarting from scratch upon failure; the corresponding distribution is obtained from the distribution π over constraint set options by converting every $\{\kappa\}$ to κ, removing $\varnothing$, and homogeneously scaling the total weight back to 1 by multiplying every probability by $\frac{1}{1-\pi(\varnothing)}$. This is the ideal strategy in terms of conserving similarities to the distribution over options, however it is inefficient in terms of generation speed, since we are reevaluating the expression from scratch each time we encounter a failure (Claessen et al., 2015, 2014).

Currently, backtracking strategies are often given as heuristics that have been added to the Luck interpreter, with the hope that the resulting distributions are not too far away from the ones on constraint set options, but without a formal study. A previous semantics for Luck (Xia and Hriţcu, 2015) does take a backtracking strategy as a parameter, but does not precisely define them, so it is unclear which strategies we are actually allowed to provide.

We provide a formal definition of backtracking strategies, allowing us to extend our semantics to define the probability distribution over constraint sets $\rho^{bs}_{p \Leftarrow e \backsimeq \kappa}$ corresponding to a Luck generator $p \Leftarrow e \backsimeq \kappa$ under a backtracking strategy bs (§3.3). As intermediate steps, we introduce choice trees (§3.1) and provide a model of the practical application of a strategy to a given generator using Markov chains (§3.2). We prove that the semantics we have defined is computable (§3.4), and we discuss how to evaluate a strategy bs based on comparisons of $\rho^{bs}_{p \Leftarrow e \backsimeq \kappa}$ with $\pi_{p \Leftarrow e \backsimeq \kappa}$ (§3.5).

3.1 Choice Trees

Our formalisation of backtracking strategies relies on a representation of a Luck generator that is easily derived from the choice-recording semantics: its *choice tree*. Consider a generator $p \Leftarrow e \backsimeq \kappa$ with at least one success and no infinite execution path. In particular, its set of traces $Tr_{p \Leftarrow e \backsimeq \kappa}$ is finite (otherwise we can construct an increasing sequence of infinitely appearing prefixes, whose limit is an infinite execution path). To compute the corresponding

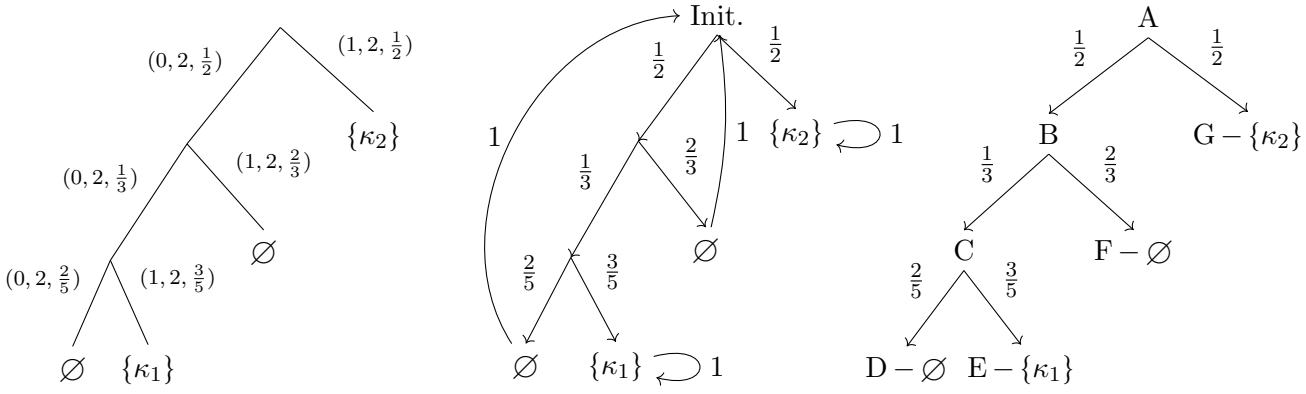

Figure 4: Choice tree, transition graph for restart-from-scratch strategy, indexing of the nodes used for a more complex strategy

choice tree, we construct the unique tree (not caring about order between siblings) whose edges are labelled with choices (m, n, q), such that the set of paths from the root to each leaf is exactly $Tr_{p \Leftarrow e \rightleftharpoons \kappa}$, and no two edges sharing the same source have the same label (we factorise as much as possible). Then we add to each leaf the result (a constraint set option) of the judgment that the trace leading to the leaf comes from. For example, if all the possible judgments with input $p \Leftarrow e \rightleftharpoons \kappa$ are $p \Leftarrow e \rightleftharpoons \kappa \Uparrow_{[(0,2,\frac{1}{2});\,(0,2,\frac{1}{3});\,(0,2,\frac{2}{5})]} \varnothing$, $p \Leftarrow e \rightleftharpoons \kappa \Uparrow_{[(0,2,\frac{1}{2});\,(0,2,\frac{1}{3});\,(1,2,\frac{3}{5})]} \{\kappa_1\}$, $p \Leftarrow e \rightleftharpoons \kappa \Uparrow_{[(0,2,\frac{1}{2});\,(1,2,\frac{2}{3})]} \varnothing$, $p \Leftarrow e \rightleftharpoons \kappa \Uparrow_{[(1,2,\frac{1}{2})]} \{\kappa_2\}$, then the corresponding choice tree is the leftmost tree in Figure 4. Once a choice tree has been build, we can actually forget the first two components m and n of the choice labels: the important parts are the structure, the probabilities q and the results in the leaves.

3.2 Application of backtracking strategies using Markov chains

We model the effect of a given backtracking strategy on the execution of a given program using time-homogeneous Markov chains with discrete time and finite state space: systems of probabilistic transitions over a finite set of states, where the probability of a transition depends only on the current state. We represent them as directed graphs with a finite set of state nodes and transition edges each labelled with a positive probability, so that the probabilities of the edges with a given source sum up to 1. Therefore we will now use *transition graph* as a synonym to time-homogeneous Markov chains with discrete time and finite state space. This can also be understood as a probabilistic automaton with no blocking node on an alphabet with a single element. We will use the following definitions: an *absorbing state* is a state which transitions to itself with probability 1 and thus cannot be left; we have an *absorbing Markov chain* if from every state we can reach an absorbing state.

First example: restart-from-scratch Let us apply the basic restart-from-scratch backtracking strategy to a program whose possible derivations are listed in the previous subsection (its choice tree is given in Figure 4). We obtain the transition graph displayed in the middle of Figure 4. The states are the nodes of the choice tree, the initial state is its root, and we keep the probabilities of its edges and direct them downward, away from the root. Starting from scratch consists in going back to the root, which happens with probability 1 from any failure node. Reaching a success state $\{\kappa\}$ is the goal of any execution: intuitively it would stop there and return κ. To obtain an actual Markov chain, we represent success states as absorbing states. Once this graph has been constructed, we read and exploit it as follows. We check

that it forms an absorbing Markov chain; we compute the probabilities to reach each individual absorbing state starting from the initial state (this step is detailed later) and gather them into a probability distribution over constraint sets: $[\kappa_1 \mapsto \frac{1}{6};\ \kappa_2 \mapsto \frac{5}{6}]$.

Second example We now consider a more complex but commonly used strategy: local backtracking with memory and bounded number of failures before restarting from the beginning of the program. More precisely (tree vocabulary refers to the choice tree): failure leaves backtrack to their parent; as soon as we know that a node has no success descendant, we add it to a set of nodes that we will never visit again; if every child of the current node is in this set, it backtracks to its own parent (and is added to the set); if we encounter more than a set number $Bmax$ of failure leaves, we start again from the root (but we can keep our set of nodes we do not want to visit, as the underlying property depends only on the tree, not on the path followed). Let us apply this strategy to the same program as before, whose choice tree is in Figure 4. As this strategy has memory, we cannot just use the nodes of the choice tree as states. Instead, we will consider states from $\mathcal{N} \times \mathcal{P}(\mathcal{N}) \times \{0, 1, ..., Bmax\}$, where $\mathcal{N} = \{A, B, ..., G\}$ is the set of the nodes of the choice tree as indexed in the right-hand side of Figure 4. In a state (X, S, n), X is the current position in the choice tree, S the set of nodes we do not want to visit again, n the number of failures seen since we last came back to the root. The initial state is $(A, \varnothing, 0)$. We give a few examples of transitions:

$(D, S, n) \xrightarrow{1} (C, S \cup \{D\}, n + 1)$ if $n < Bmax$
$(D, S, Bmax) \xrightarrow{1} (A, S \cup \{D\}, 0)$
$(C, S, n) \xrightarrow{2/5} (D, S, n);\ (C, S, n) \xrightarrow{3/5} (E, S, n)$ if $D, E \notin S$
$(C, S, n) \xrightarrow{1} (E, S, n)$ if $D \in S, E \notin S$
$(C, S, n) \xrightarrow{1} (B, S \cup \{C\}, n)$ if $D, E \in S$ (cannot happen, yet the transition exists)
$(G, S, n) \xrightarrow{1} (G, S, n)$

As in the previous example, the absorbing states are exactly the states (X, S, n) where X is a success leaf. We check that this forms an absorbing Markov chain, sum for each success leaf X the probabilities to reach states whose first component is X, and obtain a probability distribution over constraint sets.

Definition 3.2.1. We require a *correct backtracking transition graph* of a choice tree to have the following properties: its set of states has the form $\mathcal{N} \times \mathcal{M}$, where $\mathcal{N}$ is the set of nodes of the choice tree and $\mathcal{M}$ is an arbitrary *finite* set of memory states (we can of course rewrite states as $(X, (S, n))$ in the example above); there is a single initial state (R, M_0) where R is the root of the choice tree and M_0 is called the *initial memory*; the absorbing states are exactly those whose first component is a success leaf; the corresponding Markov chain is absorbing.

Realistically executable graphs There are some other properties that it makes sense to require from a transition graph obtained by applying a backtracking strategy to a choice tree. For instance, during a realistic execution we do not have access to the whole choice tree; we still follow some of its paths because from a given node we usually compute and move to one of its children. Therefore it would not be natural to allow a transition that jumps across the tree to an arbitrary node that has never been visited. We do not want to study backtracking strategies such as "when encountering a failure, we just go to the closest success leaf in the tree (in terms of shortest path) and we have our successful result": it would certainly be interesting in terms of obtaining a probability distribution that makes sense when looking at the choice tree, however it would be a lot harder to implement and worse in terms of complexity than the strategies from the example, as we may have to compute large parts of the choice tree to find success leaves and check whether one of them is the closest one.

Definition 3.2.2. A backtracking transition graph of a given choice tree is *realistically executable* if all the transitions from a state $(X, M) \in \mathcal{N} \times \mathcal{M}$ can be computed from: X (notably whether it is a failure leaf, a success leaf), M, and the children of X in the choice tree (and the probabilities on the edges leading to them).

The graph from the second example is not realistically executable because we cannot compute the transitions going back up to one's parent; however we can easily make it realistically executable by extending the memories so that we also store the path leading from the root to the current node: this allows to retrieve its parent, and is easy to maintain when moving to a parent or a child. We also store the integer *Bmax* in every memory so that we can always access it and pass it on. Note that we can have realistically executable graphs where from anywhere, we can jump to any visited node (more precisely, to at least a state whose first component is this node): we achieve this by recording every visited node in the memory. The graph from the first example needs a slight adjustment to become realistically executable: we add a singleton set of memories {Init.} to be able to jump from failures to the root Init.

Activated upon failure graphs Another sensible property to require would be that as long as no failure has been encountered, a backtracking strategy cannot influence the execution. Both examples verify this.

Definition 3.2.3. A backtracking transition graph of a choice tree is *activated upon failure* if for any state (X, M) reachable from the initial state without visiting any state containing a failure leaf, the first projection of the transitions leaving this state (we sum the probabilities of transitions leading to states with a given node component to get a total probability to go to this node) corresponds exactly to the labelled edges from X to its children in the choice tree.

3.3 Defining backtracking strategies and the related semantics

Our model of the application of a backtracking strategy to a program suggests the following definition.

Definition 3.3.1. A *backtracking strategy* is a computable function associating a transition graph to any choice tree of a Luck generator. (A backtracking strategy can easily be extended as a function taking directly a generator $p \Leftarrow e \Rightarrow \kappa$ with at least a success and no possible infinite execution path as an argument, as from $p \Leftarrow e \Rightarrow \kappa$ we can compute its choice tree.)

A backtracking strategy is *correct* if the image of any choice tree is a correct backtracking transition graph.

We usually study correct backtracking strategies, as seen in the following definition.

Definition 3.3.2 (Semantics integrating backtracking strategies)**.** Let $p \Leftarrow e \Rightarrow \kappa$ be a Luck generator with no infinite execution path and at least a success. Let *bs* be correct backtracking strategy. The transition graph obtained by applying *bs* to (the choice tree of) $p \Leftarrow e \Rightarrow \kappa$, which is correct and absorbing, defines a probability distribution over the absorbing states: this is a property of absorbing time-homogeneous Markov chains (more details are given in §3.4 and the proof of Theorem 3.4.1). Absorbing states are exactly those whose first component is a success leaf. To each κ we can associate the sum of the probabilities of the absorbing states whose first component has the label $\{\kappa\}$. This forms *the probability distribution over constraint sets* $\rho^{bs}_{p \Leftarrow e \Rightarrow \kappa}$ *of the generator* $p \Leftarrow e \Rightarrow \kappa$ *over the strategy bs.*

Definition 3.3.3. A backtracking strategy is *activated upon failure* if the image of any choice tree is an activated upon failure transition graph.

Most strategies will naturally be activated upon failure, but this is a property we may not necessarily require. For example, a strategy might immediately start exploring a few distinct paths (not all of them of course, which would mean computing the whole choice tree) and use information about how they behave to choose how to react to a failure. This could be done by alternating between nodes of these paths using the memory, and would not be activated upon failure, yet there is nothing fundamentally wrong with this.

There is still an interesting property about activated upon failure strategies:

Theorem 3.3.4. Let bs be an activated upon failure correct backtracking strategy and $p \Leftarrow e \dashv \kappa$ a Luck generator with no infinite execution path, then $\forall \kappa.\, \rho^{bs}_{p \Leftarrow e \dashv \kappa}(\kappa) \geqslant \pi_{p \Leftarrow e \dashv \kappa}(\{\kappa\})$.

Proof. Any path leading to $\{\kappa\}$ in the choice tree still exists in the transition graph, since all its nodes are reachable without visiting any failure. □

A backtracking strategy that can produce non absorbing, but otherwise correct transition graphs is usually bad, as a run in such a graph can get stuck in an infinite loop which cannot possibly be left. A simple example is a strategy where we backtrack from a failure leaf to its parent, and otherwise just follow the choice tree. Applying this to a tree containing a node whose children are all failure leaves, we can be stuck indefinitely alternating between this node and its children. Although we usually do not want such backtracking strategies, we may sometimes be interested in some of the simplest ones, when we know that we are only going to apply them to a few selected programs for which the graph will actually be absorbing.

3.4 Computability of $\rho^{bs}_{p \Leftarrow e \dashv \kappa}$

Properties of absorbing time-homogeneous Markov chains with discrete time and finite state space Such a Markov chain, even forgetting about the absorbing property, can be characterised by its transition matrix: assuming an indexing of the states from 1 to n the number of states, its elements in position (i, j) is the probability of the transition from i to j (0 if there is no such transition). Supposing now that the Markov chain is also absorbing, giving the biggest indexes to the k absorbing states gives the transition matrix the canonical form $\begin{pmatrix} Q & R \\ 0 & I_k \end{pmatrix}$ where I_k is the identity matrix of size k. A non absorbing state is called *transient*. Q is the transition relation over transient states, R from them to absorbing ones. The probability to reach the absorbing state j for the first time after $s+1$ steps starting from transient state i is the element (i, j) of $Q^s R$. Therefore the probability to be absorbed in j starting from i is $\sum_{s \geqslant 0} Q^s R = (I_{n-k} - Q)^{-1} R$.

Another property of such Markov chains is that the probability to eventually reach an absorbing state is 1.

Theorem 3.4.1. The probability distribution over constraint sets corresponding to a correct transition graph is computable.

Proof. As a correct transition graph is absorbing, the results above show that we can compute the probability to reach a given absorbing state starting from any state (it is trivially either 0 or 1 if we start from an absorbing state), in particular from the initial state. This forms a probability distribution over absorbing states, as we always reach at least, and also at most one of them with probability 1. Absorbing states are exactly the states whose first component is a success leaf. We can therefore project the probability distribution into another one over success leaves, by summing the probabilities of states with the same first component. Then we project this into a probability distribution over constraint sets: the probability of κ is the sum of the probabilities of the leaves labelled $\{\kappa\}$. □

Theorem 3.4.2 (Computability of the semantics)**.** The probability distribution over constraint sets $\rho_{p \Leftarrow e \Rightharpoonup \kappa}^{bs}$ resulting from the application of a correct backtracking strategy bs to a Luck generator $p \Leftarrow e \Rightharpoonup \kappa$ with no infinite execution path is computable.

Proof. Computability of choice trees (§3.1), of backtracking strategies by definition (3.3.1), and Theorem 3.4.1. □

3.5 Evaluation of backtracking strategies

Ideally, we want a backtracking strategy to provide a probability distribution over constraint sets that is as close as possible to the probability distribution over constraint set options. Indeed, some of the new constructs introduced in Luck give the user control over the latter, and it would be appreciated if this control could, to some extent, extend to the final distribution over constraint sets.

Let $p \Leftarrow e \Rightharpoonup \kappa$ be a Luck generator with at least one success and no infinite execution path. We want to evaluate the effect of a backtracking strategy bs on it by computing a distance between $\pi_{p \Leftarrow e \Rightharpoonup \kappa}$ and $\rho_{p \Leftarrow e \Rightharpoonup \kappa}^{bs}$: the smaller this distance is, the better the strategy. To lighten the notations we will now write π and ρ^{bs} or even ρ if their is no possible confusion over bs.

We will use the word "measure", which is not to understand in the mathematical sense but in the mundane one.

We have found in the literature two potentially interesting functions to measure the similarity between two probability distributions P and Q: the *Bhattacharyya distance* (Bhattachayya, 1943) $D_B(P,Q) = -\log \sum_i \sqrt{P(i)Q(i)}$ and the *Kullback-Leibler divergence* (MacKay, 2003) $D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$. Note that for the second one, P and Q do not have symmetric roles: Q is intuitively a model of the distribution P. Based on this interpretation, we would like to replace P with π and Q with ρ; however looking at the definition we need to have $\forall i.\, Q(i) = 0 \Rightarrow P(i) = 0$, therefore we have to use $P = \rho, Q = \pi$ as $\rho(\varnothing) = 0$ even when $\pi(\varnothing) > 0$ (to have two distributions with the same domain, we have converted the domain of ρ to constraint set options, with $\rho(\varnothing) = 0$).

These functions share a promising property: for any given $p \Leftarrow e \Rightharpoonup \kappa$, the restart-from-scratch strategy rs minimises both of them.

Theorem 3.5.1. For any Luck generator $p \Leftarrow e \Rightharpoonup \kappa$ (elided in the probability distributions) with at least one success and no infinite execution path,

$$\min_{bs} D_B(\rho^{bs}, \pi) = D_B(\rho^{rs}, \pi) \qquad \min_{bs} D_{KL}(\rho^{bs}||\pi) = D_{KL}(\rho^{rs}||\pi)$$

Proof. We quantify universally over the elided backtracking strategy argument in ρ. We write ρ^{rs} when this argument is the restart-from-scratch strategy. The symbol m stands for a constraint set option, and k for a constraint set option we know to be distinct from $\varnothing$.

$$\sum_m \sqrt{\rho(m)\pi(m)} = \sum_k \rho(k) \sqrt{\frac{\pi(k)}{\rho(k)}} \leqslant \sqrt{\sum_k \rho(k) \frac{\pi(k)}{\rho(k)}} = \sqrt{1 - \pi(\varnothing)}$$

using, in this order: $\rho(\varnothing) = 0$ to switch from m to k, the inequality between $\frac{1}{2}$-mean and arithmetic mean of the elements $\frac{\pi(k)}{\rho(k)}$ with weights $\rho(k)$ summing up to 1 (adding a square root on each side of the inequality), and $\sum_k \pi(k) = 1 - \pi(\varnothing)$.

$$D_B(\rho, \pi) = -\log \sum_m \sqrt{\rho(m)\pi(m)} \geqslant -\log \sqrt{1 - \pi(\varnothing)}$$

$$\rho^{rs}(k) = \frac{\pi(k)}{1-\pi(\varnothing)} \qquad D_B(\rho^{rs}, \pi) = -\log \sum_k \frac{\pi(k)}{\sqrt{1-\pi(\varnothing)}} = -\log \sqrt{1-\pi(\varnothing)}$$

$$\begin{aligned} D_{KL}(\rho||\pi) &= \sum_m \rho(m) \log \frac{\rho(m)}{\pi(m)} = \sum_k \rho(k) \log \frac{\rho(k)}{\pi(k)} = -\sum_k \rho(k) \log \frac{\pi(k)}{\rho(k)} \\ &\geqslant -\log \sum_k \rho(k) \frac{\pi(k)}{\rho(k)} = -\log(1-\pi(\varnothing)) = D_{KL}(\rho^{rs}, \pi) \end{aligned}$$

using concavity of log and Jensen inequality with weights $\rho(k)$ summing up to 1. □

These results of minimality are encouraging, as restarting from scratch is the most natural strategy, and the one that modifies least the distribution. These functions will allow to numerically compare backtracking strategies. As a further step, it would be very interesting to mix this with a time complexity term, or with any other function we may want to minimise, for example by summing them: this would allow to evaluate a strategy on several criteria simultaneously.

An example of term representing run speed could be, using properties of absorbing Markov chains with the notations from §3.4: the i_0-th element of the vector $(I_{n-k} - Q)^{-1}J$ where J is a column vector whose elements are all 1: this is the expectation of the number of steps before being absorbed. This however does not take into account that some transitions may be more complex than others. We could try to decorate the transition graph with complexity terms, which may or may not become too complicated to handle.

Related work

As Luck lies in the intersection of many different topics in programming languages, its related work is huge. Part of it explores Property-Based Testing and Generation and can be found in the General Context section of the introduction and in §1.1. Another part studies either constraint solving or instantiation mixed with backtracking, and appears in §1.2.

Here I will present the references which are more directly related to my internship. There is of course the POPL submission I contributed to (Lampropoulos et al., 2016a), which describes Luck and uses the choice-recording semantics. Presentations of previous Luck semantics are very relevant as well. Lampropoulos et al. (2016b) describes the collecting semantics discussed in §2.5, from which the trace-recording semantics has inherited constraint sets and the distinction between narrowing and matching. (Xia and Hriţcu, 2015) presents a semantics which has strong similarities to the choice trees we define in §3.1. It is very expressive, but as it directly computes the trees it is complex: related proofs are difficult to establish. It also takes a backtracking strategy as a parameter, however its interface for them is not very flexible. Borgström et al. (2015) presents many different styles of semantics for probabilistic programming, including a big-step operational one accumulating traces and probabilities, which has been a source of inspiration for our semantics. Claessen et al. (2015) compares the two most intuitive backtracking strategies: restarting from scratch upon failure and local backtracking. It also introduces the middle ground of locally backtracking until a set number of failures are encountered, at which point we restart from scratch; the second example in §3.2 is quite similar.

Appendix

References

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

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

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.

A. Bhattachayya. On a measure of divergence between two statistical population defined by their population distributions. *Bulletin Calcutta Mathematical Society*, 35:99–109, 1943.

J. Borgström, U. D. Lago, A. D. Gordon, and M. Szymczak. A lambda-calculus foundation for universal probabilistic programming. *CoRR*, abs/1512.08990, 2015.

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.

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.

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

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.

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

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.

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

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

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.

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.

S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*. 2007.

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.

P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2005.

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)*. May 2014.

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.

M. Hanus. A unified computation model for functional and logic programming. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1997.

C. Hriţcu, L. Lampropoulos, A. Spector-Zabusky, A. Azevedo de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis. Testing noninterference, quickly. arXiv:1409.0393; Accepted in Special Issue of Journal of Functional Programming (JFP) for ICFP 2013. To appear, July 2015.

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

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.

L. Lampropoulos, D. Gallois-Wong, C. Hriţcu, J. Hughes, B. C. Pierce, and L. Xia. Beginner's Luck: A language for random generators. Draft, arXiv:1607.05443, July 2016a.

L. Lampropoulos, B. C. Pierce, C. Hriţcu, J. Hughes, Z. Paraskevopoulou, and L. Xia. Making our own Luck: A language for random generators. Probabilistic Programming Semantics Workshop, St. Petersburg, Florida, 23 January 2016, 2016b.

F. Lindblad. Property directed generation of first-order test data. In *8th Symposium on Trends in Functional Programming*. 2007.

D. J. MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003. page 34.

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

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.

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.

Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In C. Urban and X. Zhang, editors, *6th International Conference on Interactive Theorem Proving (ITP)*. 2015.

J. S. Reich, M. Naylor, and C. Runciman. Lazy generation of canonical test programs. In *23rd International Symposium on Implementation and Application of Functional Languages*. 2011.

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.

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.

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

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.

L. Xia and C. Hriţcu. Integrating functional logic programming with constraint solving for random generation of structured data. Inria Internship Report, Sept. 2015.