Making Our Own Luck

A Language for Random Generators

(Extended Abstract)

Leonidas Lampropoulos[1] Benjamin C. Pierce[1] Cătălin Hriţcu[2]
John Hughes[3] Zoe Paraskevopoulou[4] Li-yao Xia[2,5]

[1]University of Pennsylvania [2]INRIA Paris [3]Chalmers University [4]Princeton [5]ENS Paris

Abstract

QuickCheck-style *property-based random testing* [4] 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 semantics for Luck, reminiscent of those for probabilistic languages [7], and prove key theorems about its behavior, including the soundness and completeness of random generation with respect to a straightforward predicate semantics.

Extended Abstract

Since being popularized by QuickCheck [4], PBT has become a standard technique for improving software quality across a wide variety of programming languages and for streamlining interaction with mechanical proof assistants [2, 9, etc].

Property-based testing tools require *specifications* of software artifacts in the form of executable predicates, that are used to check the artifact's behavior with respect to large numbers of randomly generated test cases. Generating good random tests data is of paramount importance. Tools like QuickCheck help automate this process leveraging type information. However, for *conditional properties* of the form $p \rightarrow q$, the default approach of generating random inputs according to some fixed distribution and filtering using p can become quite unsatisfactory, especially when p is a sparce property satisfied by only a small fraction of possible inputs.

Consider, for example, the property of noninterference of information flow control systems: Given two machine states that are *indistinguishable* to an external observer (i.e. they only differ in a few locations which are tagged "secret"), if they both take a step they should remain indistinguishable. Trying to test this property by generating completely arbitrary machine states and hoping that they will only differ in non-observable locations is bound to fail [8].

Moreover, controlling the *distribution* of test data is also critical. Consider indistinguishability of atoms (values tagged "secret" or not, where `true` denotes "secret"):

```
indist :: (Double,Bool) -> (Double,Bool) -> Bool
indist (x1,l1) (x2,l2) =
  l1 == l2 && if l1 then true else x1 == x2
```

The space of data that satisfy `indist` is vastly skewed in favor of "secret" atoms. In fact, for every non-secret pair of indistinguishable observable atoms, there are 2^{64} atoms that are "secret". Clearly, a uniform distribution would lead to inefficient testing.

To overcome these challenges, a QuickCheck user must provide a *custom generator* for inputs satisfying `p`. While QuickCheck provides a library of combinators for streamlining this task, writing such custom generators becomes increasingly challenging as `p` becomes more complex. 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 predicates appears in both the precondition and the conclution, 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. 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, closely related with the concept of *needed narrowing* from functional logic programming [1]: 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`. However,

there are cases where purely local choices lead to instantiating variables too early, before the constraints on them are known, incurring significant backtracking.

The other 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. 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. 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.

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.

- 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.
- 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. We give a probabilistic 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.

Experiments We evaluate Luck's expressiveness and efficiency on a collection of common examples from the random testing literature, using a prototype implementation based on the translation and 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 however slower (time per test) than hand-written generators ($2\times$ to $20\times$), but many opportunities for optimization remain.

Related work Luck lies in the crossroads of many different topics in programming languages; thus, the potentially related literature is huge. The works that are most closely related to our own are these of Claessen et al. [3] and Fetscher et al. [6]. Claessen et al. exploit the laziness of Haskell, combining a needed-narrowing-like technique with FEAT [5], 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. [6] 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. 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.

In the probabilistic programming setting, the closest work is arguably that on the R2 system [10]. The denotational semantics of Luck can be viewed as a generalization of the semantics of PROB to account for giving the user more control over constraint solving. While in principle one could use R2 to sample inputs satisfying a precondition by adding an `observe(True)` at the end, doing so for complicated programs would require a lot of effort. For one, coming up with invariants to allow the PRE transformation to work for loops is comparable to proving for cases like indistinguishability and type safety. In addition, one would need to fine tune the priors to obtain good testing behavior; when using Luck this process is facilitated by the established methodology of `collecting` statistics a-la QuickCheck.

References

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

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

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

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

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

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

[7] 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.

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

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

[10] 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.