Making the Best of a Bad Situation

Inferring Random Generators for Numerical Properties with Multi-Armed Bandits

Joseph W. Cutler
jwc@seas.upenn.edu
University of Pennsylvania
USA

ABSTRACT

Property-Based Testing in the style of QuickCheck has proven to be a very powerful and useful generalization of traditional software testing techniques such as unit testing. This power comes at the cost of requiring users to occasionally write *generators*: programs which emit random data satisfying the invariants assumed by the program under test. While some work exists to derive these generators directly from the invariants in question, this work is mostly focused on highly structured data, and often fails to handle the kinds of numerical invariants that occur commonly in systems programming tasks. In this extended abstract, we present an approach to inferring generators for a restricted class of numerical properties.

1 MOTIVATION

Property-Based Testing (PBT) [2] is a program testing method which tests logical properties of functions by randomly generating thousands of inputs, and ensuring that the the properties hold at each concrete value. Given a property $\forall x.P(x) \implies Q(x)$, we generate values satisfying P, and check that they satisfy Q. While this process is highly automated and requires no user input, users must occasionally write *generators*: programs which emit random data that satisfies the premise P.

Much of the time, this burden can be mitigated by automation, such as when the data structure being generated is composed entirely of structures that already have generators, or P is described as an inductive relation [7]. However, the case of inferring generators for *numerical* properties—those involving constriants over numbers— is not well handled by prior work. Numerical properties appear regularly in systems programming tasks [5] where structured data described by inductive invariants is rare, inputs regularly take the form of pointers and integers, and many function preconditions amount to bounds-checking or ensuring that certain fields in different inputs are equal.

In this extended abstract, we develop a method for inferring random generators for a small class of numerical properties akin to the kind commonly found in systems programming. Because manual generator-writing for numerical properties is especially tedious, our goal is to infer generators that are comparable to ones that a user might write as a first cut. The method proceeds by developing a number of generator "candidates", expressed in a DSL we call `ALuck`, described in Section 2. These candidates mimic the way that a human might write a generator for numerical preconditions: by randomly choosing values one variable at a time, constraining subsequent choices by the previous values chosen. In Section 3, we show how candidates are chosen from the space of possible `ALuck` generators. As we will see, it is sometimes the case that *none* of the candidates are all that good. In this case, we must make the best of a bad situation, and discover the best candidate among the set. A method for this based on the multi-armed bandits problem [8] is presented in Section 4. Finally, in Section 5, we describe our implementation of this algorithm and present benchmarks.

Figure 1: Syntax of Propositions and Generator Scripts

Expressions	e	::=	$x \mid e+e \mid -e \mid e*e$
Relations	R	::=	$= \mid \leq \mid <$
Propositions	P	::=	$P \wedge P \mid eRe$
Generator Actions	a	::=	$!x \mid eRe$
Generator Scripts	s	::=	$[] \mid a :: s$

2 SYNTAX OF PROPOSITIONS AND ALUCK

The syntax of propositions that we handle are shown in Figure 1. The variables range over the arguments of the function under test. The allowable expressions in these inequalities are essentially multilinear functions, where each variable occurs with degree at most one. While this form is restrictive, we have found empirically that this covers a wide range of preconditions in systems verification. Moreover, there appears to be no inherent difficulty to extending the technique to handle other numerical operators (div, mod), and constraints over structured types like lists: we present the algorithm as-is to simplify the discussion.

To make the generator inference problem tractable, we fix the syntactic form of the generators. Our generators are written in a language called `ALuck` (for *arithmetic* `Luck`) inspired by `Luck` [6]. Generators written `ALuck` run by sequentially *constraining* and *concretizing* variables. Every variable in an `ALuck` generator begins as a symbolic variable. Constraints over these variables are then added. Variables can then be concretized, wherein they are replaced by a random value satisfying the constraints on that variable that have been accumulated thusfar. The final result of the generator is a map from variables to their (randomly) chosen values.

More concretely, generators in `ALuck` are sequences of "concretize" operations, written $!x$, and "constrain" operations, written simply as the constraint to be added. This syntax is shown in Figure 1. These sequences are then evaluated from left to right while maintaining a pair of mappings: one from concretized variables to their values, and the other from yet-unconcretized variables to the set of constraints that have accumulated on them. When a "constrain" operation c is encountered, the constraint c is added to the constraint sets of all of the variables it mentions. If a constraint mentions no variables, it is checked for valdity: if the constraint is not valid, the generator fails.

When a "concretize" operation $!x$ is encountered, a value is randomly sampled from the uniform distribution on the set of possible values[1] denoted by the constraints on x. This semantics is shown

[1] Because integers are bounded by machine word lengths, the "uniform distribution" does make sense here, even for unconstrained variables.

Figure 2: Generation Semantics

$$\frac{v \sim \mathcal{U}\left(\llbracket C(x) \rrbracket\right) \qquad V\{x \mapsto v\}, C \vdash s \Downarrow V', C'}{V, C \vdash !x :: s \Downarrow V', C'}$$

$$\frac{C'(x) = C(x) \cup \{c\} \qquad V, C' \vdash s \Downarrow V', C''}{V, C \vdash c(x) :: s \Downarrow V', C''} \quad \frac{V \vDash c \qquad V, C \vdash s \Downarrow V', C'}{V, C \vdash c :: s \Downarrow V', C'}$$

in Figure 2, where the judgment $V, C \vdash s \Downarrow V', C'$ means that the script s evaluates under concretized-variable-map V and constraint map C, and returns updated variable and constraint maps V' and C'. We note that this semantics is partial, as generators can fail when attempting to concretize a variable whose constraints are unsatisfiable. This is crucial: different generators for the same property can fail more or less often, and we would like to infer generators which fail infrequently.

To make the sampling step tractable, we enforce that our `ALuck` programs be "well-concretized": when $!x$ occurs, all variables which occur in constraints with x must already have been concretized.

3 GENERATOR CANDIDATE INFERENCE

The first step in our algorithm is to infer a set of generators from a given predicate. We begin by noting that every ordering of the variables in a property immediately determines a well-concretized generator: this procedure is shown in the Appendix in Algorithm 1. In essence, the procedure works by placing all of the constraints that could possibly appear before a concretization immediately before it.

Because of this, to infer generators for a property P, it suffices to generate orderings of its variables. Unfortunately, for a property P with n free variables, there are $n!$ such orderings. We can prune this search space by only looking for "relevant" orderings. If some variables are not related to others, the well-concretization condition won't care about the relative order in which they're generated.

To operationalize this insight, we build a graph $G(P)$ whose nodes are variables with an edge (x, y) when x and y both occur in one of the conjuncts of P. In this graph, "unrelated" variables live in different connected components. Then, to generate a concretization ordering, we depth-first search $G(P)$, randomly choosing the next neighbor to explore, and list variables in the order that they're visited in the graph. Since different connected components are listed separately, concretizations of unrelated variables will not occur together.

To generate our set of generator candidates, we repeatedly run this random DFS procedure. This may give repeated generators, and so we filter the result for uniqueness. The number of generators in our set requires a careful balance. Too few and the set may not contain a generator which succeeds often enough to rapidly generate our desired number of unique inputs. Too many and the learning algorithm will converge too slowly to the best generator in the set. Empirically, we have found that n^2 (where n is the number of variables in P) is a number of generators to take.

4 GENERATOR LEARNING WITH BANDITS ALGORITHMS

With our bag of generators in hand, we now need to find the best one. The approach we take will be inspired by the Multi-Armed Bandits [4] problem from the theory of reinforcement learning. In short, the Multi-Armed Bandits problem describes a game where an algorithm is repeatedly presented a fixed set of choices. Each choice gives a different (random) reward, and the goal of the game is to maximize the total reward. Solving this optimization problem online is too difficult in an adversarial setting, so algorithms for the multi-armed bandits problem aim to instead minimize *regret*: how much worse their total reward was than the total reward from the best single choice in hindsight. To achieve this, bandits algorithms learn which actions have historically given more reward, and play those more frequently. In our setting, the "choices" are our generator candidates, and the rewards are given by success or failure of a generator to yield a value. Under this analogy, an algorithm for the Multi-Armed Bandits problem will let us learn which generators give the best results *while simultaneously* generating a stream of valid inputs for the function.

The algorithm we will use is called UCB1 [1]. While more sophisticated bandits algorithms exist, this one is sufficient for our purposes. When given a list of generators $g_1, \ldots g_k$ and a number of rounds T to run for, UCB1 runs the generators g_i in a way that attempts to learn which generator succeds the most frequently, while also attempting to not waste time by running failing generators too much. In the end, UCB1 emits the $m \leq T$ successfully-generated values over the course of its run. The main upshot is that the ratio success ratio $\frac{m}{T}$ of this derived generator should, in the limit, be no worse than the *best* generator the algorithm was given.

Theorem 4.1. *Fix a property P and generators $g_1, \ldots, g_K$, where g_i succeeds with probability at least p_i. Then,*

$$\lim_{T \to \infty} \mathbb{E}\left[\frac{1}{T}\left|\mathsf{ucb1}(g_1, \ldots, g_K, P, T)\right|\right] \geq \max_i p_i$$

In other words, the stream of values emitted by an "infinite run" of UCB1 acts like[2] a generator for P which succeeds with probability at least $\max_i p_i$.

5 IMPLEMENTATION, RESULTS, AND FUTURE WORK

I have implemented the generator inference algorithm described in this abstract, as well as a deductive program verifier (a la Dafny or Frama-C [3, 9]) for the `IMP` language which uses these generators as a verification backend. The code is available here.

In Table ??, we present some results. In each trial, we run the generration algorithm on the specified constraint five times for $T = 1000$ iterations. The largest issue is the performance differences between the runs on the proposition $0 \leq x \leq y \leq 100$ with and without explicitly including the implicand $x \leq 100$. The "best" generator for the proposition without it samples an arbitrary $x \geq 0$, which is very unlikely to leave room for some y satisfying $x \leq y \leq 100$. Including the added constraint ensures that we leave space with our initial choice of x. We leave resolving this to future work.

[2] In theory, the difference $\left|\frac{1}{T}\mathbb{E}\left[|\mathsf{ucb1}|\right] - \max_i p_i\right|$ converges like $O\left(\frac{\log T}{T}\right)$ [1, Theorem 1]. Empirically, we have found this to be quite rapid.

Constraint	K	R	U
$x = y + z \wedge x, y, z \geq 0$	5	506	494
$x = y + z \wedge x \geq y \wedge x, y, z \geq 0$	4	48.6	951.4
$x - y \leq 5 \wedge x - y \leq 10 \wedge y - z \leq 2$	3.8	26.2	831
$0 \leq x \leq y \leq 100$	2	993.4	6.6
$0 \leq x \leq y \leq 100 \wedge x \leq 100$	2	11	831

Table 1: Results. K **= average number of unique scripts,** R **= average number of failed draws,** U **= average number of unique values generated.**

REFERENCES

[1] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[2] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, sep 2000.

[3] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods*, pages 233–247, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[4] John C Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society: Series B (Methodological)*, 41(2):148–164, 1979.

[5] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 114–129, 2017.

[7] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[8] Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

[9] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

APPENDIX

Algorithm 1 Generator from an ordering

function CONSTRUCTGENERATOR(xs,P)
  $P_{const} \leftarrow$ conjuncts in P mentioning only one variable
  $P \leftarrow P$ without P_{const}
  $s \leftarrow P_{const}$
  $ys \leftarrow []$
  for $x \in xs$ **do**
    $P' \leftarrow$ conjuncts in P mentioning x and variables in ys
    $s \leftarrow$ append(s,append(c',!x))
    $P \leftarrow P$ without P'
    $ys \leftarrow$ append(ys,x)
  end for
  $s \leftarrow$ append(s,P)
  return s
end function

Bandits and UCB1

The Multi-Armed Bandits problem is described as the following repeated game: at each round t, the player plays an action $a_t \in [K]$, and receives a reward $X_{a_t,t}$, which is a $\{0,1\}$-valued random variable. The random variables $X_{i,t}$ are IID for fixed i, and independent for fixed t. We write the mean of the ith reward variable (for all t) μ_i. The goal of the game is to maximize one's reward, and so the goal of a bandits learning algorithm is to learn an adaptive policy, which takes a history of play up until state t (all actions $a_{t'}$ and received rewards $X_{a_{t'},t'}$ for $t' < t$), and produces a new action a_t. The metric by which we compare bandits algorthms is *regret*: how much worse they do than the best policy in hindsight.

DEFINITION 1 (REGRET). *Define $i_\star = \arg\max_i \mu_i$, and write $\mu_\star = \mu_{i_\star}$. The regret $R(A)$ of an algorithm A over T rounds is defined as*

$$R(A) = T\mu_\star - \mathbb{E}\left[\sum_{t=1}^{T} X_{A(t),t}\right]$$

THEOREM 4.1. *Fix a property P and generators $g_1, \ldots, g_K$, where g_i succeeds with probability at least p_i. Then,*

$$\lim_{T\to\infty} \mathbb{E}\left[\frac{1}{T}\left|\mathsf{ucb1}(g_1,\ldots,g_K,P,T)\right|\right] \geq \max_i p_i$$

PROOF. We begin by noting that $r_t = 1$ in an iteration if and only if an element is added to the output list in stage t. Therefore, the length of the output $|\mathsf{ucb1}(g_1,\ldots,g_K,P,T)|$ is precisely the sum of the ($\{0,1\}$-valued) r_t, $\sum_{t=1}^{T} r_t$. In the notation of the bandit problem, r_t is the (revealed) value of the random variable $X_{A(t),t}$, and so

$$\mathbb{E}\left[|\mathsf{ucb1}(g_1,\ldots,g_K,P,T)|\right] = \mathbb{E}\left[\sum_{t=1}^{T} r_t\right] = \mathbb{E}\left[\sum_{t=1}^{T} X_{\mathsf{ucb1}(t),t}\right]$$

.

But then by the definition of regret,

$$\mathbb{E}\left[\sum_{t=1}^{T} X_{\mathsf{ucb1}(t),t}\right] = T\mu_\star - R(\mathsf{ucb1})$$

Algorithm 2 Learn a Generator

function UCB1(generators $g_1, \ldots, g_K$, property P, rounds T)
  for $i = 1 \ldots K$ **do**
    $x_i \leftarrow$ sample(g_i)
    $\hat{\mu}_i \leftarrow 1$ if $P(x_i)$, 0 otherwise
    $n_i \leftarrow 1$
  end for
  $X \leftarrow []$
  for $t = 1 \ldots T$ **do**
    $j \leftarrow \arg\max_i \hat{\mu}_i + \sqrt{\frac{2\log t}{n_i}}$
    $x \leftarrow$ sample(g_j)
    if $P(x)$ **then**
      $r_t \leftarrow 1$
      $X \leftarrow$ snoc(X,x)
    else
      $r_t \leftarrow 0$
    end if
    $\hat{\mu}_j \leftarrow \hat{\mu}_j + r_t$
    $n_j \leftarrow n_j + 1$
  end for
end function

The regret bound for UCB1 [8, Theorem 7.2] states that $R(\mathsf{ucb1}) \in O\left(\sqrt{KT\log T}\right)$.

Then, dividing through by T, using the regret bound for UCB1, and the fact that $p_i \geq \mu_i$, we have that:

$$\begin{aligned}\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} X_{\mathsf{ucb1}(t),t}\right] &= \mu_\star - \frac{R(\mathsf{ucb1})}{T} \\ &\geq \mu_\star - O\left(\sqrt{\frac{K\log T}{T}}\right) \\ &\geq \max_i p_i - O\left(\sqrt{\frac{K\log T}{T}}\right)\end{aligned}$$

which approaches $\max_i p_i$ as $T \to \infty$. □