Making Our Own Luck
A Language For Random Generators

Leonidas Lampropoulos    Benjamin C. Pierce
Catalin Hritcu           John Hughes
Zoe Paraskevopoulou      Li-yao Xia

PPS, January 23rd 2016
The Problem

How to find bugs in large software artifacts (gcc, GHC)?

- Exhaustive testing?
- Random Testing?

  - CSmith – Well-formed C programs (Yang et al. ‘11)
  - Testing GHC Strictness Analyzer (Palka et al. ‘11)
  - Design IFC Machines (Hritcu et al. ‘13)
Property Based Random Testing

∀\overline{x}. q(\overline{x})

Generate \overline{x}

\forall \overline{x}. p(\overline{x}) \rightarrow q(\overline{x})

Check p(\overline{x})

If check succeeds, Test q(\overline{x})

If not, start over
Write generators that satisfy $p$ directly.

\[
\text{indist } (x_1, b_1) (x_2, b_2) = \\
\quad b_1 == b_2 && \\
\quad \text{if } b_1 \text{ then } x_1 == x_2 \text{ else True}
\]

\[
\text{genIndist } = \text{do} \\
\quad b \leftarrow \text{genBool} \\
\quad x_1 \leftarrow \text{genInt} \\
\quad x_2 \leftarrow \text{if } b \text{ then return } x_1 \\
\quad \text{else genInt} \\
\quad \text{return } ((x_1,b), (x_2,b))
\]
Custom generators and probabilistic programs specify probability distributions

$$\forall x. p(x) \rightarrow q(x)$$

```plaintext
x \sim \text{prior;}
b \leftarrow p(x);
observe(b)
```
Problem: Writing a good generator for a precondition \( p \)

- Easier than coming up with a prior satisfying observations
- Can use the full power of a language/libraries
- Success stories

All (most) generated values satisfy \( p \)

Distribution appropriate for testing

All values that satisfy \( p \) can be generated
Custom Generators

Problem: Maintainability

- Generators and predicates are distinct software artifacts
- Can get out of sync
- Rich source of bugs!

Solution: Derive generators automatically from predicates!
Narrowing (Claessen et al. ‘14, Fetscher et al. ‘15)

- Borrows from functional logic programming
- Incremental generate and test
- Instantiate every variable at its first constraint

\[
\text{indist } (x_1, b_1) \ (x_2, b_2) =
\quad b_1 == b_2 \&\& \text{if } b_1 \text{ then } x_1 == x_2 \text{ else True}
\]

- Generate equal \( b_1, b_2 \)
- Check \( b_1 \)
- Generate equal \( x_1, x_2 \)
Narrowing (Claessen et al. ‘14, Fetscher et al. ‘15)

- Borrows from functional logic programming
- Incremental generate and test
- Instantiate every variable at its first constraint

+ Very lightweight
+ Allows control over distributions
- Not always efficient...
Narrowing (Claessen et al. ‘14, Fetscher et al. ‘15)

\[
\text{bst low high tree} = \\
\text{case tree of} \\
\quad \text{Empty} \rightarrow \text{True} \\
\quad \text{Node } x \ l \ r \rightarrow \text{low < x && x < high} \\
\qquad \qquad \qquad \qquad \&\& \text{bst low x l \&\& bst x high r}
\]
Narrowing (Claessen et al. ‘14, Fetscher et al. ‘15)

bst low high tree =
    case tree of
        Empty -> True
        Node x l r -> low < x && x < high
                       && bst low x l && bst x high r

• Assume low = 0, high = 17
• At low < x, generate x (eg. 42)
• x < high is a check!
Constraint Solving

- Generating inputs directly from predicates (Carlier et al. ‘10, Seidel et al. ‘15, etc.)
- Symbolic execution (DART, KLEE, CutEr, etc.)

+ No backtracking
- No predictable control over distributions
- Potential overheads when narrowing works
Hybrid approach: Luck

Luck = Narrowing + Constraint Solving

- Choose when/where constraint solving happens
- Control over distributions in the style of QuickCheck
fun bst size low high tree =
  case tree of
  | 1 % tree -> Empty
  | size % Node x 1 r ->
    { x : low < x && x < high }
  && bst (size / 2) low x 1
  && bst (size / 2) x high r

1 / (size + 1)

size / (size + 1)
fun bst size low high tree =
  case tree of
  | 1 % tree -> Empty
  | size % Node x l r ->
    { | x : low < x && x < high |}
    && bst (size / 2) low x l
    && bst (size / 2) x high r
Assume low = 0, high = 17 and x ∈ [-42..42]
At low < x, x ∈ [1..42]
At x < high, x ∈ [1..16]
Sample from the domain of x – No backtracking!
Luck vs Probabilistic Programming

\[
\text{indist (x1:{-42..42}, b1)} \\
\text{(x2:{-42..42}, b2) =} \\
b1 == b2 && \text{if } b1 \text{ then } x1 == x2 \text{ else True}
\]
x1 ~ uniform(-42, 42);
x2 ~ uniform(-42, 42);
b1 ~ Bernoulli(0.5);
b2 ~ Bernoulli(0.5);
indist <- b1 == b2 && (!b1 || x1 == x2);
observe(indist);
Experiments

- Palka et al. ‘11 – Test GHC by generating random lambda terms
- Hritcu et al. ‘13 – Test IFC design by generating abstract machines

+ Significantly less code and effort
+ Exact bugfinding performance (tests per counterexample)
- Efficiency (up to 1 order of magnitude)
A Taste of Semantics

Recall: Control over where narrowing and where constraint solving happens

\[ \text{low} < x \land x < \text{high} \]

\[ \{ | x : \text{low} < x \land x < \text{high} \} \]

Higher chance of backtracking!

Does potentially more work
A Taste of Semantics

- Use subprobability distributions!
- “complement” probability = chance of backtracking

**Input constraints**

Set \((Vars \rightarrow Vals)\)

**Resulting distribution**

\[[e] :: K \rightarrow Pr(K)\]

Remove unsatisfying valuations from the input
A Taste of Semantics

\[ [0 < x \land x < 17] \{ x \in \{-42..42\} \} \]

- \[ [0 < x ] \{ x \in \{-42..42\} \} \]

\[ \left\{ \begin{array}{c} \{ x \in \{1\} \} \mapsto \frac{1}{42} , \ldots , \{ x \in \{42\} \} \mapsto \frac{1}{42} \end{array} \right\} \]

- \[ [x < 17 ] \{ x \in \{i\} \}, i \in \{1..42\} \]

\[ \left\{ \begin{array}{c} \{ x \in \{1\} \} \mapsto \frac{1}{42} , \ldots , \{ x \in \{16\} \} \mapsto \frac{1}{42} \end{array} \right\} \]

62% chance of backtracking!
A Taste of Semantics

\[\{\{x : 0 < x \land x < 17\}\}\{x \in \{-42..42\}\}\]

- \[0 < x \]\{x \in \{-42..42\}, \ x \text{ delayed}\}

\[\{x \in \{1..42\}\}\mapsto 1\]

- \[x < 17 \]\{x \in \{1..42\}\}\mapsto 1, \ x \text{ delayed}\}

\[\{x \in \{1..16\}\}\mapsto 1\]

\[\{x \in \{1\}\}\mapsto \frac{1}{16}, \ldots, \{x \in \{16\}\}\mapsto \frac{1}{16}\]
Conclusion

• Luck is a language for writing artifacts that serve as both a predicate and a generator

• Lightweight annotations give the user control over:
  ▪ Distribution of generated data
  ▪ Generation strategy (narrowing vs constraint solving)

Thank you!
Boltzmann Samplers – Uniform Distributions

\[ B(x) = 1 + xB^2(x) \]

fun {boltzmann: size} bst tree low high =
  case tree of
    | r % Empty -> True
    | r’ % Node x l r -> ...

• Choose parameter \( x \) systematically
• \( r \) and \( r' \) are chosen so that \( \frac{r}{r+r'} = \frac{x}{B(x)} \)
• Approximate size: Too small -> discard / too large -> stop generation
• Linear complexity (in size)!