# Making Our Own Luck
## A Language For Random Generators

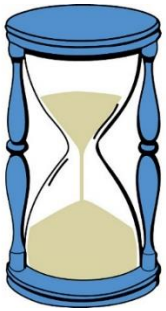
**Leonidas Lampropoulos** Benjamin C. Pierce
Catalin Hritcu John Hughes
Zoe Paraskevopoulou Li-yao Xia

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

$$\forall \overline{x}.\, q(\overline{x})$$

Generate $\overline{x}$

If not, start over

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

Check $p(\overline{x})$

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

# Custom Generators

Write generators that satisfy p directly.

```
indist (x1, b1) (x2, b2) =
   b1 == b2 &&
   if b1 then x1 == x2 else True
```

(1,True) – (1,True)
(17,False) – (42,False)

```
genIndist = do
   b   <- genBool
   x1  <- genInt
   x2  <- if b then return x1
          else genInt
   return ((x1,b), (x2,b))
```

# Random Testing and Probabilistic Programming

Custom generators and probabilistic programs specify probability distributions

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

x ~ prior;
b ← p(x);
observe(b)

# Custom Generators

All values that satisfy p can be generated

Problem: Writing a good generator for a precondition p

All (most) generated values satisfy p

Distribution appropriate for testing

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

# 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

```
indist (x1, b1) (x2, b2) =
   b1 == b2 && if b1 then x1 == x2 else True
```

Generate equal b1, b2

Check b1

Generate equal x1, x2

# 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)

```
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
```

# 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

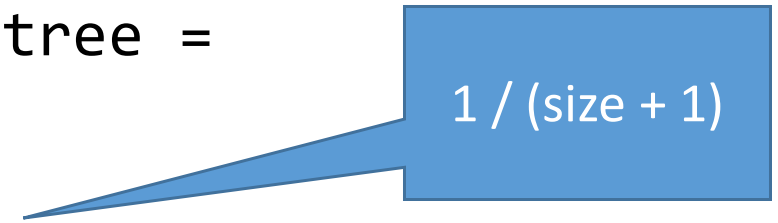
# Hybrid approach: Luck

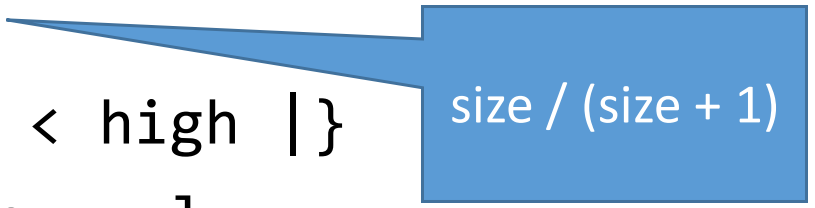
Luck = Narrowing + Constraint Solving

- Choose when/where constraint solving happens
- Control over distributions in the style of QuickCheck

# Luck by Example

```
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
```



# Luck by Example

```
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
```

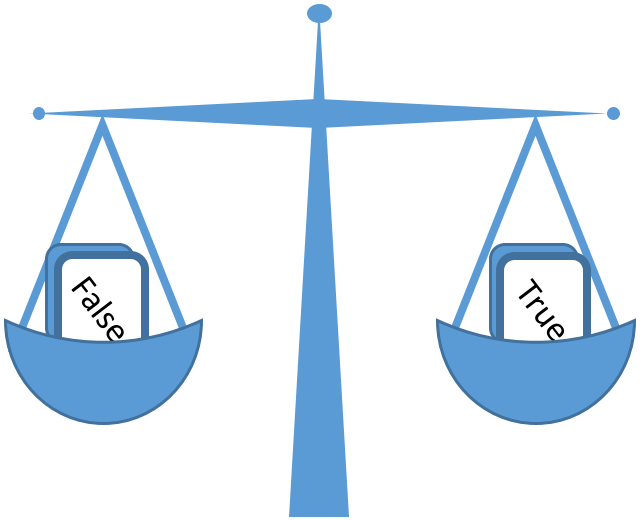
# Luck by Example

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

- 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!

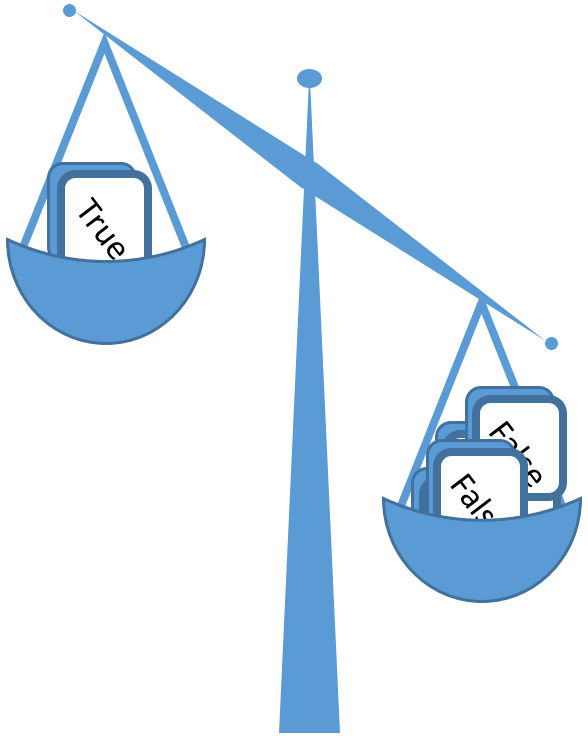
# Luck vs Probabilistic Programming

```
indist (x1:{-42..42}, b1)
         (x2:{-42..42}, b2) =
   b1 == b2 && if b1 then x1 == x2 else True
```



# Luck vs Probabilistic Programming

```
x1 ~ uniform(-42, 42);
x2 ~ uniform(-42, 42);
b1 ~ Bernoulli(0.5);
b2 ~ Bernoulli(0.5);
indist <- b1 == b2
  && (!b1 || x1 == x2);
observe(indist);
```




~~(0,T)-(-42,T) … (0,T)-(-1,T)~~ (0,T)-(0,T) ~~(0,T)-(1,T) … (0,T)-(42,T)~~

(0,F)-(-42,F) … (0,F)-(-1,F) (0,F)-(0,F) (0,F)-(1,F) … (0,F)-(42,F)

# 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

```
low < x && x < high
```

vs

```
{| x : low < x && x < 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

$$[\![e]\!] :: \mathrm{K} \rightarrow \underline{\Pr}(\mathrm{K})$$

Remove unsatisfying valuations from the input

# A Taste of Semantics

$$[\![0 < x \;\&\&\; x < 17]\!]\{x \in \{-42..42\}\}$$

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

$$\left\{\{x \in \{1\}\} \mapsto \frac{1}{42}, \ldots, \{x \in \{42\}\} \mapsto \frac{1}{42}\right\}$$

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

$$\left\{\{x \in \{1\}\} \mapsto \frac{1}{42}, \ldots, \{x \in \{16\}\} \mapsto \frac{1}{42}\right\}$$

62% chance of backtracking!

# A Taste of Semantics

$$[\![\{|x : 0 < x \;\&\&\; x < 17|\}]\!]\{x \in \{-42..42\}\}$$

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

→ $\{\{x \in \{1..42\}\} \mapsto 1\}$

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

→ $\{\{x \in \{1..16\}\} \mapsto 1\}$

→ $\left\{\{x \in \{1\}\} \mapsto \frac{1}{16}, \ldots, \{x \in \{16\}\} \mapsto \frac{1}{16}\right\}$

0% chance of backtracking!

# 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)!