Random Testing in 321
Test Cases So Far

Each test relates a particular input to a particular output.

(test (bound-ids
    (with 'x (id 'y) (id 'x))
    '(x))
(test (binding-ids
    (with 'x (id 'y) (id 'x))
    '(x))

2
Property-based Testing

But we can only write so many tests by hand.

To find additional bugs, we can automate testing.

We start with what we hope is a fact about our program.

For example,

“If \texttt{bound-ids} says \texttt{x} is bound,
then \texttt{binding-ids} says \texttt{x} is binding.”
Property Violation

If we can find some WAE for which the property doesn’t hold ...

\[
\begin{align*}
\text{(define } & \text{a-WAE } \ldots) \\
\text{(bound-ids } & \text{a-WAE)} \; \Rightarrow \; '(x) \\
\text{(binding-ids } & \text{a-WAE)} \; \Rightarrow \; '() \\
\end{align*}
\]

... we’ve found a bug.
Property Testing

We can test this property in the usual style.

; bound => binding? : WAE -> boolean
; checks if bound ids are also binding
(define (bound=>binding? e) ...)

(test (bound=>binding? (id 'x))
  true)

(test (bound=>binding?
    (with 'x (num 0) (id 'x)))
  true)

Expected result is always true, so if we had lots of WAEs, then we’d have lots of tests.
Automated Property Testing

Write a program to generate test inputs!
Random WAEs

; random-WAE: -> WAE
(define (random-WAE)
  (case (random 5)
    [(0) (num (random-nat))]
    [(1) (id (random-symbol))]
    [(2) (add (random-WAE) (random-WAE))]
    [(3) (sub (random-WAE) (random-WAE))]
    [(4) (with (random-symbol)
       (random-WAE)
       (random-WAE))]))

Watch out – that code is buggy.... (read on for why)
Random WAEs

; random-nat: -> nat
(define (random-nat)
  (case (random 2)
    [(0) 0]
    [(1) (add1 (random-nat))]))

; random-symbol: -> symbol
(define (random-symbol)
  (random-elm '(x y z a b c)))

; random-elm: (listof X) -> X
(define (random-elm xs)
  (list-ref xs (random (length xs)))))
Generation Strategy

To build a WAE,

- $1/5$ of the time, build a number
- $1/5$ of the time, build a symbol
- $3/5$ of the time, first build two more WAEs
Expected Progress

On average, we “reduce” the problem from

Generate 1 WAE.

to

Generate 1.2 WAEs.

since \(1.2 = (2/5) \times 0 + (3/5) \times 2\)
Height Bound

Limit WAE size by bounding tree height.

; random-WAE/b: nat -> WAE
(define (random-WAE/b h)
  (case (random (if (zero? h) 2 5))
    [(0) (num (random-nat))]
    [(1) (id (random-symbol))]
    [(2) (add (random-WAE/b (sub1 h))
              (random-WAE/b (sub1 h)))]
    [(3) (sub (random-WAE/b (sub1 h))
              (random-WAE/b (sub1 h)))]
    [(4) (with (random-symbol)
              (random-WAE/b (sub1 h))
              (random-WAE/b (sub1 h)))]))

(Alternatively, tweak weights.)
Property Implementation

; bound=>binding: WAE -> boolean
(define (bound=>binding e)
  (sublist? (bound-ids e) (binding-ids e)))

; sublist?: (listof X) (listof X) -> boolean
; Expects xs and ys to be sorted and have no dupes.
(define (sublist? xs ys)
  (cond [(null? xs) #t]
        [(null? ys) #f]
        [(equal? (car xs) (car ys))
          (sublist? (cdr xs) (cdr ys))]
        [else (sublist? xs (cdr ys))])))
Running Tests

; test-bound=>binding: nat nat -> (or 'passed WAE)
(define (test-bound=>binding size attempts)
  (if (zero? attempts)
    'passed
    (let ([test-input (random-WAE/b size)])
      (if (bound=>binding test-input)
        (test-bound=>binding
         size
         (sub1 attempts))
        test-input))))
(test-bound=>binding 5 1000)
HW2 Test Results

We ran random tests on a past year’s HW2 submissions.

- Received 99 submissions (includes multiple attempts from the same person)
- Tested 6 properties
- Found a bug in 53 out of those 99 submissions
Interpreter Properties

• Interpreter does not crash

• Produces same result as another implementation (e.g., DrRacket)

• Type checker accurately predicts result (later)

• Program equivalences hold
With Enclosing Example

For example, we should be able to replace any subexpression with a new variable.

\[
\{+ 1 2\} \rightarrow \{ \text{with } \{x \ 2\} \\
\{ + 1 \ x\}\}
\]
Another example:

\[
\begin{align*}
\text{with } & \{ x \{ + \ 5 \ 26 \} \} \\
\text{with } & \{ - x \ 4 \} \\
\text{with } & \{ z \{ + \ 5 \ 26 \} \} \\
\text{with } & \{ x \ z \} \\
\text{with } & \{ - x \ 4 \} \\
\end{align*}
\]
Transformation Strategy

- Generate a random path for a WAE expression tree
- Pick a subexpression based on the path to bind to a new id
- Replace subexpression with a bound occurrence of the id
Generating Random Paths

- Automatically pick leaf nodes
- Flip a coin to determine whether we move further down the tree
- Because we are "lifting" a subexpression out of its original context, we only pick expressions which will not contain free id’s
- Always pick the named-expr of the first \texttt{with} we encounter
Path Generation Implementation

; coin-flip: boolean
(define (coin-flip)
  (zero? (random 2)))

; weighted-chance: number -> boolean
(define (weighted-chance pct)
  (<= (random) (/ pct 100)))
Path Generation Implementation

; random-path: WAE -> (listof symbol)
(define (random-path wae)
  (type-case WAE wae
    [num (n) empty]
    [id (x) empty]
    [with (name named-expr body) '(left)]
    [else
      (if (weighted-chance 20)
        empty
        (if (coin-flip)
          (cons 'left
            (cons 'right
              (random-path (get-branch 'right wae)))
          (random-path (get-branch 'left wae))))])))
Path Generation Implementation

; get-branch: symbol WAE -> WAE
(define (get-branch choice wae)
  (type-case WAE wae
    [add (lhs rhs) (case choice
                      [(left) lhs]
                      [(right) rhs])]
    [sub (lhs rhs) (case choice
                   [(left) lhs]
                   [(right) rhs])]
    [with (name named-expr body)
          (case choice
            [(left) named-expr]
            [(right) body])]
    [else wae]))
Extracting the Subexpression

Given a path, we find the subexpression:

; pick-subexpr: WAE (listof symbol) -> WAE
(define (pick-subexpr wae path)
  (cond
    [(empty? path) wae]
    [else
      (pick-subexpr (get-branch (car path) wae) (cdr path))])))
Replacing with the New ID

; swap-subexpr WAE (listof symbol) symbol -> WAE
(define (swap-subexpr wae path new-id)
  (cond
   [(empty? path) (id new-id)]
   [else
     (type-case WAE wae
       [add (lhs rhs)
         (swap-in-bop path new-id add lhs rhs)]
       [sub (lhs rhs)
         (swap-in-bop path new-id sub lhs rhs)]
       [with (name named-expr body)
         (with name
           (id new-id)
           body)]
       [else wae]))]))
Replacements with the New ID

(define (swap-in-bop path new-id op lhs rhs)
  (case (car path)
    [(left) (op (swap-subexpr lhs
                  (cdr path)
                  new-id)
              rhs)]
    [(right) (op lhs
                (swap-subexpr rhs
                              (cdr path)
                              new-id))]]))
Implementing the Transformation

; rand-sym-not-in: (listof symbol) -> symbol
(define (rand-sym-not-in lst)
  (let ([leftover (remove* lst syms)])
    (list-ref leftover
      (random (length leftover))))))

; wae->with-wae: WAE -> WAE
(define (wae->with-wae wae)
  (let* ([path (random-path wae)]
         [subexpr (pick-subexpr wae path)]
         [new-id (rand-sym-not-in (binding-ids wae)])]
    (with new-id
      subexpr
        (swap-subexpr wae path new-id)))))
Putting It All Together

We generate 1000 WAE’s and compare interpreter output for the original and transformed programs:

```scheme
(for ([i (in-range 0 1000)])
  (let* ([wae (random-WAE/b 2 '())]
         [new-wae (wae->with-wae wae)])
    (test (interp wae)
          (interp new-wae))))
```
What Went Wrong?

• Our random WAE generator builds arbitrary expressions

• Probability we generate a WAE with free identifiers is very high
Eliminating Free ID’s from the Random Generator

; rand-sym-from: (listof symbol) -> symbol
(define (rand-sym-from ss)
  (list-ref ss (random (length ss))))
Eliminating Free ID’s from the Random Generator

(define (random-WAE/b h bindingids)
  (case (random (if (zero? h) 2 5))
    [(0) (num (random-nat))]
    [(1) (if (empty? bindingids)
        (num (random-nat))
        (id (rand-sym-from bindingids)))]
    [(2) (add (random-WAE/b (sub1 h) bindingids)
        (random-WAE/b (sub1 h) bindingids))]
    [(3) (sub (random-WAE/b (sub1 h) bindingids)
        (random-WAE/b (sub1 h) bindingids))]
    [(4) (let ([new-id (random-symbol)])
        (with new-id
          (random-WAE/b (sub1 h) bindingids)
          (random-WAE/b (sub1 h)
            (cons new-id
              bindingids))))])