

(Generative) Random (Property) Testing

Test Cases So Far

Each test relates a particular input to a particular output.

```
(test (subst-interp
      (with 'x (num 3) (id 'x))
      ' ( ) )
      3)
(test (deferred-interp
      (with 'x (num 3) (id 'x))
      ' ( ) )
      3)
```

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 evaluating a program with `subst-interp` returns `3` ,
then evaluating it with `deferred-interp` also returns `3`”

Property Violation

If we can find some FIWAE for which the property doesn't hold ...

```
(define an-flwae ...)  
(subst-interp an-flwae fundefs) ;  $\Rightarrow$  3  
(deferred-interp an-flwae fundefs (mtSub)) ;  $\Rightarrow$  0
```

... we've found a bug.

Property Testing

We can test this property in the usual style.

```
; subst=deferred? : F1WAE? -> boolean?  
; checks that subst and def return  
; the same result  
(define (subst=deferred? e) ...)  
  
(test (subst=deferred?  
      (add (num 1) (num 2))))  
      true)  
(test (subst=deferred?  
      (with 'x (num 0) (id 'x))))  
      true)
```

Expected result is always **true**, so if we had lots of **F1WAE**s, 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 : -> integer?  
(define (random-nat)  
  (random 1000))  
  
; random-symbol : -> symbol?  
(define (random-symbol)  
  (random-ref '(x y z a b c)))
```


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 $1.2 = (2/5)*0 + (3/5)*2$

Height Bound

Limit WAE size by bounding tree height.

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

(Alternatively, tweak weights.)

Displaying Generated Terms

```
; FlWAE? -> SExp
(define (unparse an-flwae)
  (type-case FlWAE an-flwae
    [num (n) n]
    [id (name) name]
    [add (l r) ` {+ , (unparse l) , (unparse r) } ]
    [sub (l r) ` {- , (unparse l) , (unparse r) } ]
    [with (name named-expr body)
      ` {with , name
          , (unparse named-expr)
          , (unparse body) } ]
    [app (f a) ` { , f , (unparse a) } ]))

(unparse (add (num 1) (num 2))) ; => ' (+ 1 2)
```

A Simple Property

Parsing and unparsing should be inverses.

Composing them should be the identity function.

```
(define (unparse-parse? a-wae)
  (equal? a-wae (parse (unparse a-wae))))
```

A Simple Property

```
(define (test-unparse-parse n-attempts h)
  (cond [(zero? n-attempts) 'success]
        [else
         (define a-wae (random-wae h))
         (if (unparse-parse? a-wae)
             (test-unparse-parse (sub1 n-attempts) h)
             (unparse a-wae))]))

(test-unparse-parse 10000 5)
; => error: parser: expected with, got:
; (with b (+ x a) 3)
```

Oops, forgot one pair of {}s.

Fixing Unparsing

```
(define (unparse an-flwae)
  (type-case F1WAE an-flwae
    ...
    [with (name named-expr body)
      ; now with {}s!
      ` {with { ,name , (unparse named-expr) }
        , (unparse body) } ]
    ...))
```

A More Interesting Property

```
; subst=deferred? : FlWAE? -> boolean?
; checks that subst and def return
; the same result
(define (subst=deferred? an-flwae)
  (equal? (subst-interp an-flwae '())
          (deferred-interp an-flwae '() (mtSub))))

(define (test-subst=deferred n-attempts h)
  ; see test-unparse-parse
  )

(test-subst=deferred 1000 5)
; => error: interp: free identifier
```


A More Interesting Property

```
(define (subst=deferred? an-flwae)
  ; if both have the same error, that's ok
  (define subst-result
    (with-handlers ([exn:fail?
                      (lambda (e) (exn-message e))])
      (subst-interp an-flwae ' ())))
  (define def-result
    (with-handlers ([exn:fail?
                      (lambda (e) (exn-message e))])
      (deferred-interp an-flwae ' (()) (mtSub))))
  (equal? subst-result def-result))
```

Now let's test some buggy interpreters.

Finding Bug #1

```
(test-subst=deferred 1000 5)
; => '(with (c 0) (+ z 0))

(subst-interp (parse '(with (c 0) (+ z 0)))
              '())
; => error: interp: free identifier

(deferred-interp (parse '(with (c 0) (+ z 0)))
                 '()
                 (mtSub))

; => 0
```

Clearly def is wrong.

Finding Bug #1

```
(define (lookup name ds)
  (type-case DefSub ds
    [mtSub () (error 'interp "free identifier")]
    ; always uses first variable. oops.
    [aSub (n val rest) val]))
```

Fixing Bug #1

```
(define (lookup name ds)
  (type-case DefSub ds
    [mtSub () (error 'interp "free identifier")]
    [aSub (n val rest)
      (if (symbol=? name n)
          val
          (lookup name rest))]))
```

Finding Bug #2

```
(test-subst=deferred 1000 5)
; => ' (- 0 1)

(subst-interp (parse ' (- 0 1))
              ' ( ) )
; => 1

(deferred-interp (parse ' (- 0 1))
                 ' ( )
                 (mtSub))
; => -1
```

Clearly subst is wrong.

Finding Bug #2

```
; interp : F1WAE? (listof FunDef) -> number?
(define (interp an-flwae fundefs)
  (type-case F1WAE an-flwae
    ...
    [add (a b) (+ (interp a fundefs)
                   (interp b fundefs))]
    [sub (a b) (+ (interp a fundefs)
                   (interp b fundefs))]
    ...))
```

Fixing Bug #2

```
; interp : F1WAE? (listof FunDef) -> number?
(define (interp an-flwae fundefs)
  (type-case F1WAE an-flwae
    ...
    [add (a b) (+ (interp a fundefs)
                   (interp b fundefs))]
    [sub (a b) (- (interp a fundefs)
                   (interp b fundefs))]
    ...))
```

The Search for Bug #3

```
(test-subst=deferred 1000 5)  
; => 'success  
(test-subst=deferred 100000 5)  
; => 'success
```

But we're not exercising function application...

Random FIWAEs

```
(define (random-flwae h)
  (case (if (= h 0) (random 2) (random 6)) ; now 6
    ...
    [(5) (app (random-symbol)
               (random-flwae (sub1 h))))]))

(define (random-fundef h)
  (fundef (random-symbol)
          (random-symbol)
          (random-flwae h)))

(define (unparse-fundef a-fundef)
  `{defun {, (fundef-fun-name a-fundef)
          , (fundef-param-name a-fundef) }
    , (unparse (fundef-body a-fundef))})
```

Generating Functions

```
(define (test-subst=deferred n-attempts h)
  (cond [(zero? n-attempts) 'success]
        [else
         (define a-fundef (random-fundef h))
         (define an-flwae (random-flwae h))
         (if (subst=deferred? an-flwae
                               (list a-fundef))
             (test-subst=def (sub1 n-attempts)
                             h)
             (list (unparse an-flwae)
                    (unparse-fundef a-fundef))))]))
```

Could generate more than one function.

Design decision.

Generating Functions

```
(define (subst=deferred? an-flwae fundefs)
  (define subst-result
    (with-handlers ([exn:fail?
                      (lambda (e) (exn-message e))] )
      (subst-interp an-flwae fundefs)))
  (define def-result
    (with-handlers ([exn:fail?
                      (lambda (e) (exn-message e))] )
      (deferred-interp an-flwae fundefs (mtSub))))
  (equal? subst-result def-result))
```

The Search for Bug #3

```
(test-subst=deferred 100000 5)
; => infinite loop on
;      '(list (x 3) (deffun (x x) (x (x 0)))))
```

```
(require racket/sandbox)
...
(define subst-result
  (with-handlers ([exn:fail?
                  (lambda (e) (exn-message e))])
    (with-limits 1 #f ; 1s max
      (subst-interp an-flwae fundefs))))
...
```

Mutatis mutandis for def.

The Search for Bug #3

```
(test-subst=deferred 100000 5)  
; => 'success
```

Still no luck.

Looking at generated terms, we see lots of errors (free identifier, undefined function).

Most of our "test cases" are almost useless!

Avoiding Free Variables

```
(define (random-flwae h ok-vars fun-name)
  (case (if (= h 0) (random 2) (random 6))
    [(0) (num (random-nat))]
    [(1) (if (empty? ok-vars)
              (num (random-nat))
              (id (random-ref ok-vars))))]
    [(2) (add (random-flwae (sub1 h)
                             ok-vars
                             fun-name)
                (random-flwae (sub1 h)
                             ok-vars
                             fun-name)))]
    ; Mutatis mutandis for `sub`
    ...))
```

Avoiding Free Variables

```
(define (random-flwae h ok-vars fun-name)
  (case (if (= h 0) (random 2) (random 6))
    ...
    [ (4)
      (define name (random-symbol))
      (with name
        (random-flwae (sub1 h)
                      ok-vars
                      fun-name)
        (random-flwae (sub1 h)
                      (cons name ok-vars)
                      fun-name)) ]
    [ (5) (app fun-name
              (random-flwae (sub1 h)
                            ok-vars
                            fun-name)) ]))
```

Avoiding Free Variables

```
(define (random-fundef h)
  (define fun-name (random-symbol))
  (define param-name (random-symbol))
  (fundef fun-name
    param-name
    (random-flwae h
      (list param-name)
      fun-name)))
```


Avoiding Free Variables

```
(define (test-subst=deferred n-attempts h)
  (cond
    [(zero? n-attempts)
     'success]
    [else
     (define a-fundef (random-fundef h))
     (define an-flwae
       (random-flwae
        h
        ' ()))
     (fundef-fun-name a-fundef)))
  (if (subst=deferred? an-flwae (list a-fundef))
      (test-subst=deferred (sub1 n-attempts) h)
      (unparse an-flwae))))
```

The Search for Bug #3

```
(test-subst=deferred 100000 5)  
; => 'success  
(test-subst=deferred 1000000 5)  
; => 'success  
(test-subst=deferred 10000000 5)  
; => 'success
```

Still no luck.

Turns out, bug #3 is in a "blind spot" of our generator.
We won't be able to generate a test case.

Buggy interpreters are / will be posted on Piazza. See if
you can find the bug! (And come up with a test case.)

Take-Away

- Random testing is a useful supplement to your usual testing regimen.
- But not a panacea! You should use it in addition to manually-crafted test cases.
- Applies beyond PL! If you can state a property and write a generator, you're good to go.