Effective properties

John Hughes

CHALMERS

QuviQ


BITCOIN·DIGITAL·DECENTRALIZED·PEER TO PEER
MONETARY METALS
1TROYOZ 999 FINE COPPER MJB

```
newtype Coin = Coin Int
  deriving (Eq, Show)

maxCoinValue = 1000000

validCoin (Coin n) =
  0 <= n && n <= maxCoinValue

add (Coin a) (Coin b) =
  if a+b < maxCoinValue
    then Just (Coin (a+b))
    else Nothing
```

```
testNormal =
  Coin 2 `add` Coin 2
  ===
  Just (Coin 4)

testOverflow =
  Coin maxCoinValue `add` Coin 1
  ===
  Nothing
```

100% coverage of code of add

```
*Coins> quickCheck testNormal
+++ OK, passed 1 tests.
*Coins> quickCheck testOverflow
+++ OK, passed 1 tests.
```

```
prop_Normal (Coin a) (Coin b) =
  a+b < maxCoinValue ==>
    Coin a `add` Coin b
    ===
    Just (Coin (a+b))

prop_Overflow (Coin a) (Coin b) =
  a+b > maxCoinValue ==>
    Coin a `add` Coin b
    ===
    Nothing
```

```
instance Arbitrary Coin where

  arbitrary =
    Coin <$> choose (0,maxCoinValue)
```

*Don't forget to write **and test:***

```
  prop_Valid (Coin c) =
    validCoin (Coin c)
```

Lots of discarded tests

```
*Coins> quickCheck prop_Normal
+++ OK, passed 100 tests; 96 discarded.

*Coins> quickCheck prop_Overflow
+++ OK, passed 100 tests; 96 discarded.
```

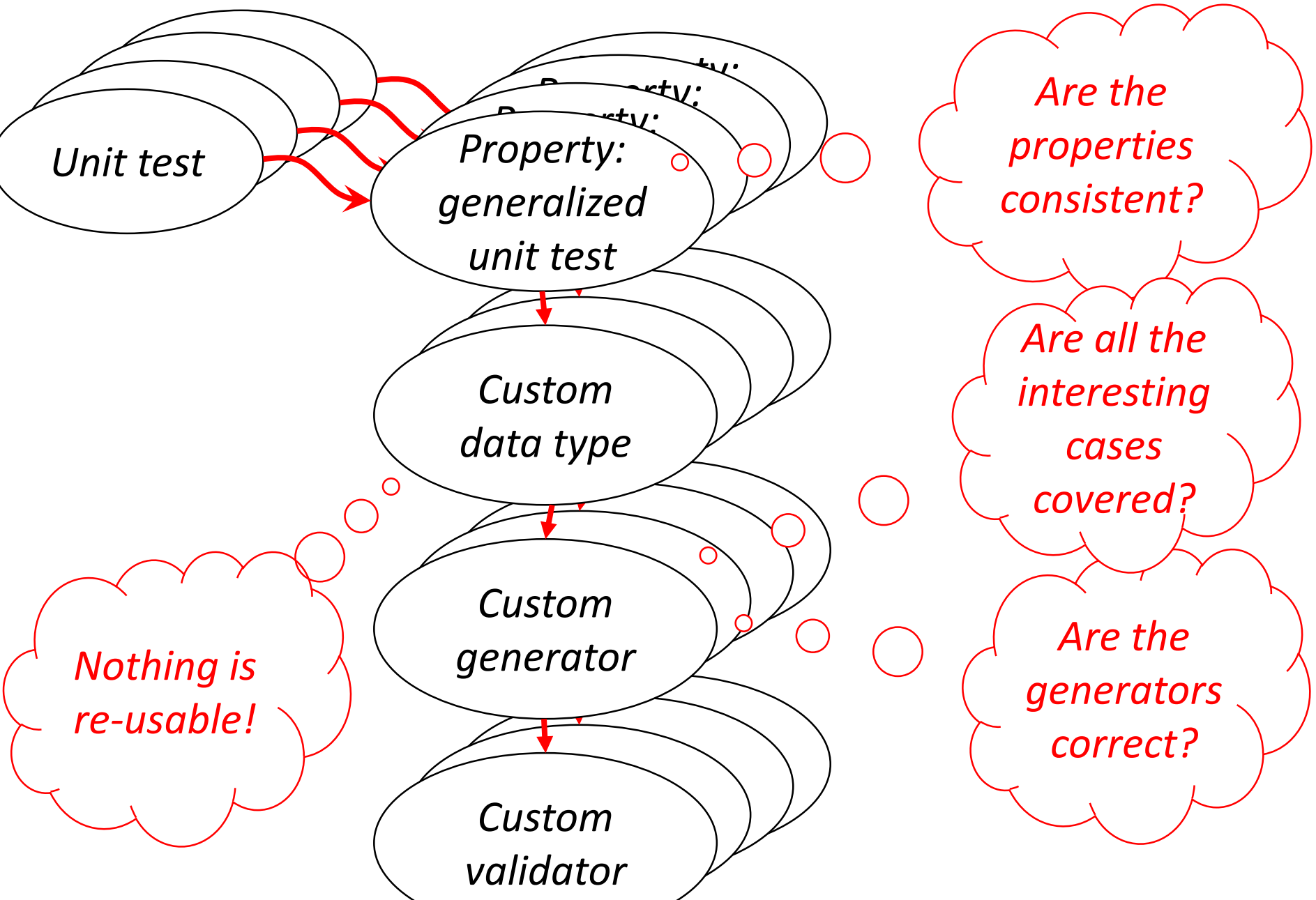
100% coverage

```
prop_Normal (Normal (Coin a) (Coin b)) =
  a+b < maxCoinValue ==>
    Coin a `add` Coin b
    ===
    Just (Coin (a+b))

data Normal = Normal Coin Coin
  deriving Show

instance Arbitrary Normal where
  arbitrary = do
    Coin a <- arbitrary
    b <- choose (0,maxCoinValue-a)
    return $ Normal (Coin a) (Coin b)
```

```
*Coins> quickCheck prop_Normal
+++ OK, passed 100 tests.
```

```
data Normal = Normal Coin Coin deriving Show

instance Arbitrary Normal where
  arbitrary = do
    Coin a <- arbitrary
    b <- choose (0,maxCoinValue-a)
    return $ Normal (Coin a) (Coin b)

data Overflow = Overflow Coin Coin deriving Show

instance Arbitrary Overflow where
  arbitrary = do
    Coin a <- arbitrary
    b <- choose (maxCoinValue-a+1,maxCoinValue)
    return $ Overflow (Coin a) (Coin b)

prop_ValidNormal (Normal c c') =
  validCoin c && validCoin c'

prop_ValidOverflow (Overflow c c') =
  validCoin c && validCoin c'
```

```
*Coins> quickCheck . withMaxSuccess maxCoinValue $ prop_ValidOverflow
*** Failed! Falsifiable (after 913693 tests):
Overflow (Coin 0) (Coin 1000001)
```


Unit test
Property: generalized unit test
Custom data type
Custom generator
Custom validator
Are the properties consistent?
Are all the interesting cases covered?
Are the generators correct?
Nothing is re-usable!

```
prop_Add (Coin a) (Coin b) =
  Coin a `add` Coin b
    ===
  if validCoin c
    then Just c
    else Nothing
  where c = Coin (a+b)
```

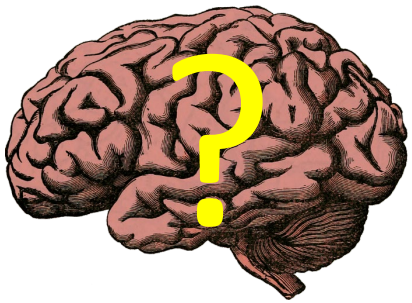
```
prop_Add (Coin a) (Coin b) =
  label (summarize (a+b)) $
  Coin a `add` Coin b
    ===
  if validCoin c
    then Just c
    else Nothing
  where c = Coin (a+b)

summarize n
  | n <= maxCoinValue = "normal"
  | n >  maxCoinValue = "overflow"
```

```
*Coins> quickCheck . withMaxSuccess 10000 $ prop_Add
+++ OK, passed 10000 tests:
50.26% normal
49.74% overflow
```

```
summarize n
   | abs (n-maxCoinValue) < 3 = "boundary"
   | n <= maxCoinValue = "normal"
   | n >  maxCoinValue = "overflow"
```

```
*Coins> quickCheck . withMaxSuccess 10000 $ prop_Add
+++ OK, passed 10000 tests:
50.61% normal
49.39% overflow
```

Not a single boundary case!



```
instance Arbitrary Coin where
  arbitrary = do
    NonNegative n <- arbitrary
    Coin <$>
      oneof [return n,
             return (maxCoinValue-n),
             choose (0,maxCoinValue)]
```

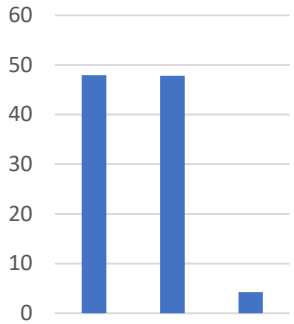
```
*Coins> quickCheck prop_Add
*** Failed! Falsifiable (after 1 test):
Coin 1000000
Coin 0
Nothing /= Just (Coin 1000000)
```

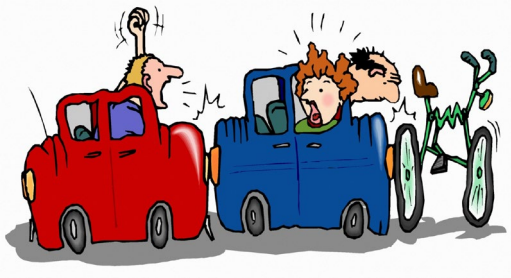
```
newtype Coin = Coin Int
  deriving (Eq, Show)

maxCoinValue = 1000000

validCoin (Coin n) =
  0 <= n && n <= maxCoinValue

add (Coin a) (Coin b) =
  if a+b < maxCoinValue
    then Just (Coin (a+b))
    else Nothing
```

```
*Coins> quickCheck . withMaxSuccess 10000 $ prop_Add
+++ OK, passed 10000 tests:
47.92% overflow
47.84% normal
 4.24% boundary
```

Unit test

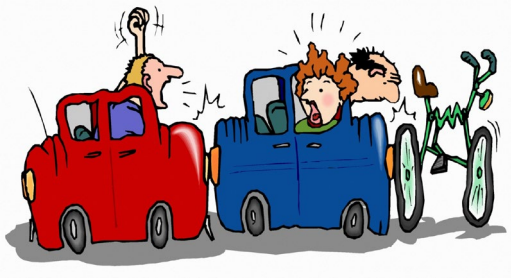
Label





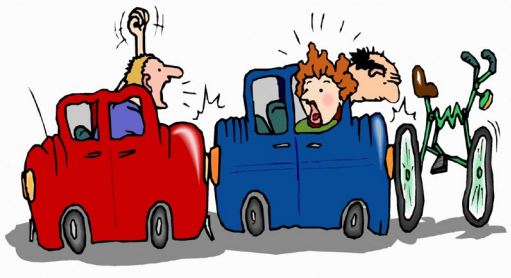
```
prop_NotElem :: Int -> [Int] -> Bool
prop_NotElem x xs = x `notElem` xs
```

```
*Coins> quickCheck prop_NotElem
*** Failed! Falsified (after 5 tests and 1 shrink):
4
[4]
```



```
prop_CoinNotElem :: Coin -> [Coin] -> Bool
prop_CoinNotElem x xs = x `notElem` xs
```

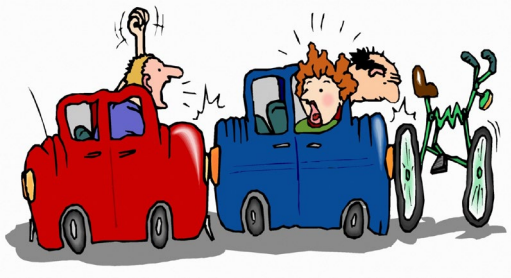
```
*Coins> quickCheck prop_CoinNotElem
*** Failed! Falsified (after 6 tests and 3 shrinks):
Coin 999997
[Coin 999997]
```



```
prop_CoinNotElem :: Coin -> [Coin] -> Bool
prop_CoinNotElem x xs = x `notElem` xs
```

```
*Coins> quickCheck . withMaxSuccess 10000 $ prop_CoinNotElem
+++ OK, passed 10000 tests.
```

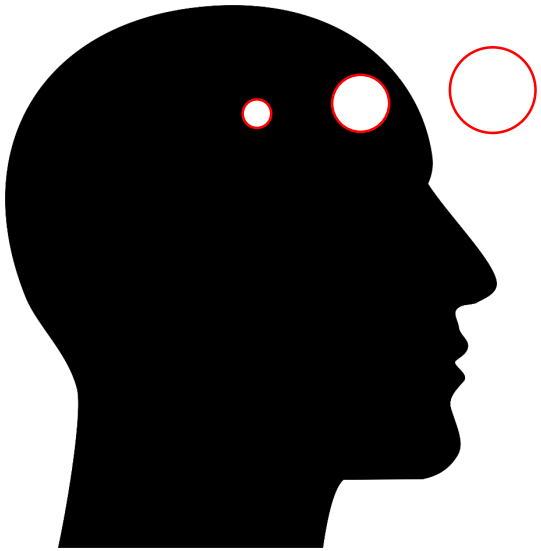
!!!



```
prop_CoinNotEqual :: Coin -> Coin -> Bool
prop_CoinNotEqual x y = x /= y
```

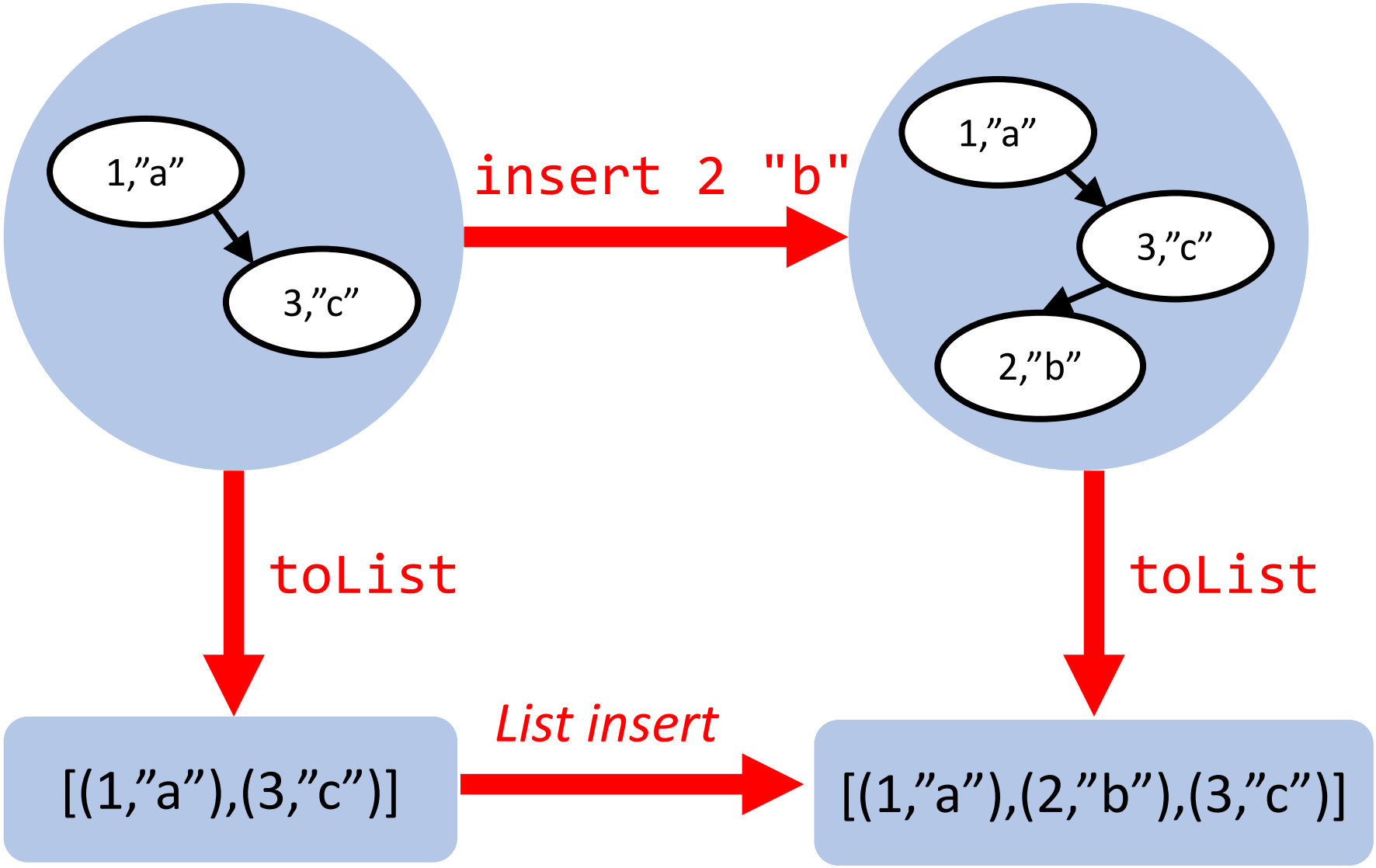
```
*Coins> quickCheck . withMaxSuccess 10000 $ prop_CoinNotEqual
+++ OK, passed 10000 tests.
```

!!!



Are collisions important test cases?

If so, choose a *type* (or *generator*) for which they are not too unlikely


1,"a"
3,"c"
insert 2 "b"
1,"a"
3,"c"
2,"b"
toList
toList
List insert
[(1,"a"),(3,"c")]
[(1,"a"),(2,"b"),(3,"c")]

```
prop_Insert :: (Int,Int) -> _
prop_Insert (k,v) t =
  toList (insert k v t)
    ===
  List.insert (k,v)
    (deleteKey k $ toList t)
```

What unit tests would you write?

```
prop_Insert :: (Int,Int) -> _
prop_Insert (k,v) t =
  label present $
  toList (insert k v t)
    ===
  L.insert (k,v) (deleteKey k $ toList t)
  where ks      = keys t
        present =
          if k `elem` ks
          then "present"
          else "absent"
```

```
*BST> quickCheck prop_Insert
+++ OK, passed 100 tests:
84% absent
16% present
```

```
prop_Insert :: (Int,Int) -> _
prop_Insert (k,v) t =
  label present $
  label position $
  toList (insert k v t)
    ===
  L.insert (k,v) (deleteKey k $ toList t)
  where ks       = keys t
        present = …
        position | all (>=k) ks = "at start"
                 | all (<=k) ks = "at end"
                 | otherwise    = "middle"
```

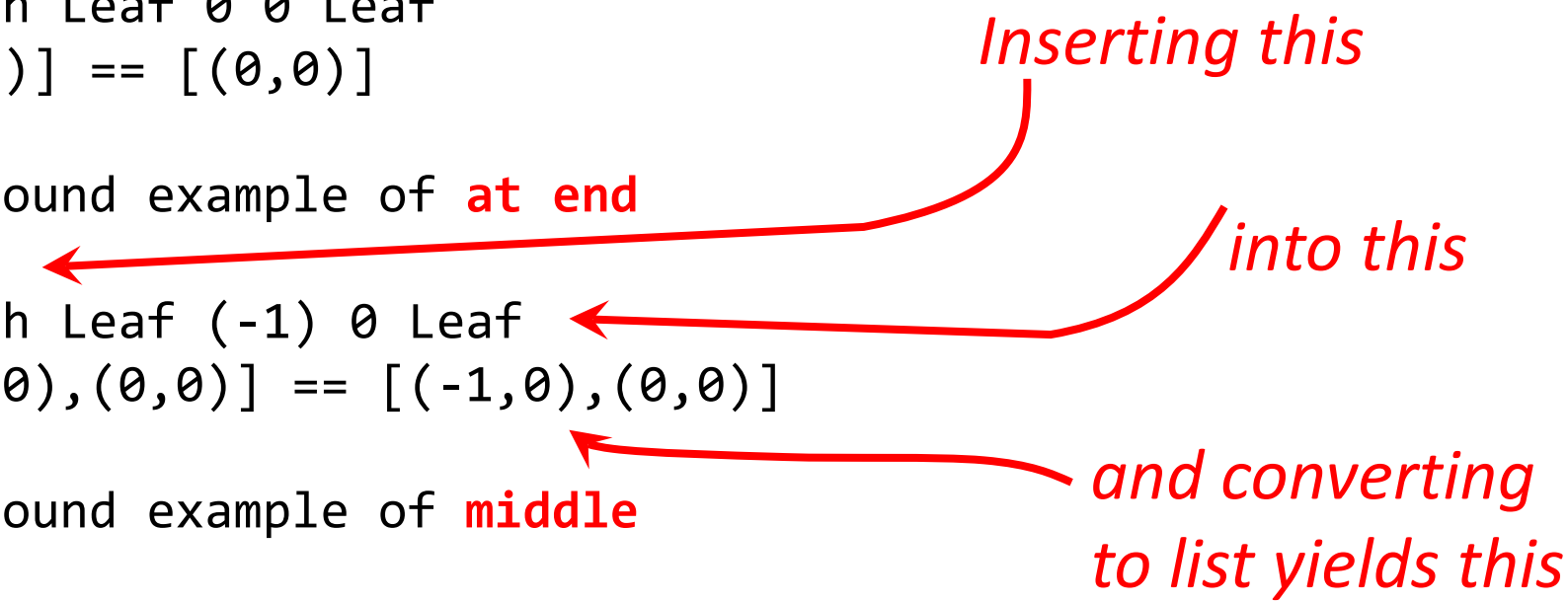
```
*BSTSpecOrig> quickCheck prop_Insert
+++ OK, passed 100 tests:
```
79% absent
21% present

69% middle
21% at start
10% at end

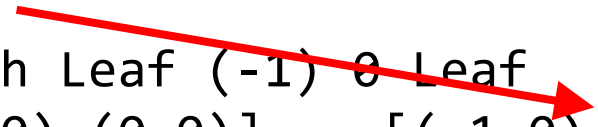
```
prop_Insert :: (Int,Int) -> _
prop_Insert (k,v) t =
  label present $
  label position $
  toList (insert k v t)
    ===
  L.insert (k,v) (deleteKey k $ toList t)
  where ks       = keys t
        present = …
        position | all (>=k) ks = "at start"
                 | all (<=k) ks = "at end"
                 | otherwise    = "middle"
```

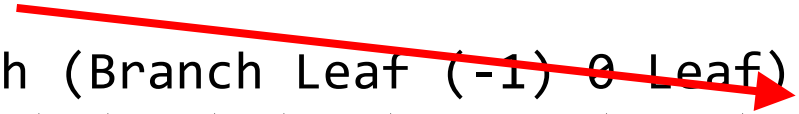
How do we know this code is right?

```
*BST> labelledExamples prop_Insert
*** Found example of absent, at start
(0,0)
Leaf
[(0,0)] == [(0,0)]

*** Found example of present
(0,0)
Branch Leaf 0 0 Leaf
[(0,0)] == [(0,0)]

*** Found example of at end
(0,0)
Branch Leaf (-1) 0 Leaf
[(-1,0),(0,0)] == [(-1,0),(0,0)]

*** Found example of middle
(0,0)
Branch (Branch Leaf (-1) 0 Leaf) 1 0 Leaf
[(-1,0),(0,0),(1,0)] == [(-1,0),(0,0),(1,0)]
```

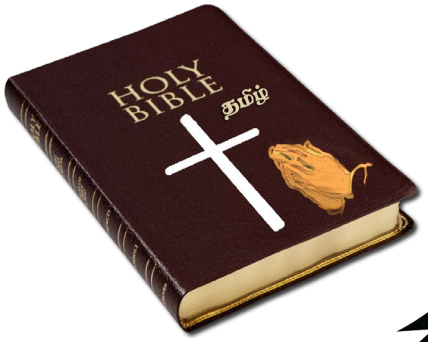


```
*BST> labelledExamples prop_Insert
*** Found example of absent, at start
(0,0)
Leaf
[(0,0)] == [(0,0)]

*** Found example of present
(0,0)
Branch Leaf 0 0 Leaf
[(0,0)] == [(0,0)]

*** Found example of at end
(0,0)
Branch Leaf (-1) 0 Leaf
[(-1,0),(0,0)] == [(-1,0),(0,0)]

*** Found example of middle
(0,0)
Branch (Branch Leaf (-1) 0 Leaf) 1 0 Leaf
[(-1,0),(0,0),(1,0)] == [(-1,0),(0,0),(1,0)]
```





```
*** Found example of at start
(0,0)
Leaf
[(0,0)] == [(0,0)]
```

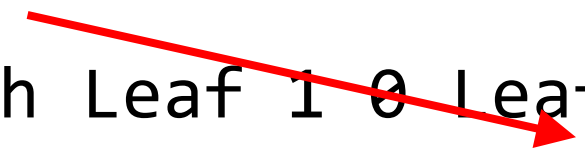


```
prop_Insert :: (Int,Int) -> _
prop_Insert (k,v) t =
  label present $
  label position $
  toList (insert k v t)
    ===
  L.insert (k,v) (deleteKey k $ toList t)
  where ks       = keys t
        present = …
        position | t == nil      = "empty"
                 | all (>=k) ks = "at start"
                 | all (<=k) ks = "at end"
                 | otherwise    = "middle"
```

```
*BST> labelledExamples prop_Insert
*** Found example of absent, empty
(0,0)
Leaf
[(0,0)] == [(0,0)]

*** Found example of at start, present
(0,0)
Branch Leaf 0 0 Leaf
[(0,0)] == [(0,0)]
```

```
prop_Insert :: (Int,Int) -> _
prop_Insert (k,v) t =
  label present $
  label position $
  toList (insert k v t)
    ===
  L.insert (k,v) (deleteKey k $ toList t)
  where ks       = keys t
        present = …
        position | t == nil      = "empty"
                 | ks == [k]     = "just k"
                 | all (>=k) ks = "at start"
                 | all (<=k) ks = "at end"
                 | otherwise    = "middle"
```

```
*** Found example of just k, present
(0,0)
Branch Leaf 0 0 Leaf
[(0,0)] == [(0,0)]

*** Found example of at start
(0,0)
Branch Leaf 1 0 Leaf
[(0,0),(1,0)] == [(0,0),(1,0)]
```



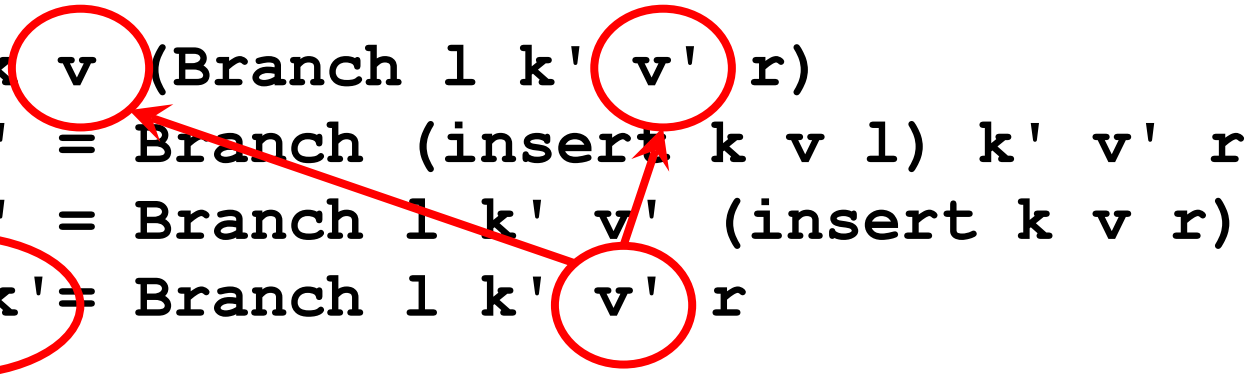
```
*BST> quickCheck . withMaxSuccess 10000 $ prop_Insert
+++ OK, passed 10000 tests:
80.06% absent
19.94% present

74.78% middle
10.01% at end
 9.53% at start
 5.28% empty
 0.40% just k
```

5 labelled examples

```
data BST k v =
  Leaf | Branch (BST k v) k v (BST k v)


insert k v Leaf =
  Branch Leaf k v Leaf

insert k v (Branch l k' v' r)
  | k<k' = Branch (insert k v l) k' v' r
  | k>k' = Branch l k' v' (insert k v r)
  | k==k'= Branch l k' v  r
```

```
data BST k v =
  Leaf | Branch (BST k v) k v (BST k v)


insert k v Leaf =
  Branch Leaf k v Leaf

insert k v (Branch l k' v' r)
  | k<k' = Branch (insert k v l) k' v' r
  | k>k' = Branch l k' v' (insert k v r)
  | k==k'= Branch l k' v' r
```

```
data BST k v =
  Leaf | Branch (BST k v) k v (BST k v)


insert k v Leaf =
  Branch Leaf k v Leaf

insert k v (Branch l k' v' r)
  | k<k' = Branch (insert k v l) k' v' r
  | k>k' = Branch l k' v' (insert k v r)
  | k==k' = Branch l k' v' r
```



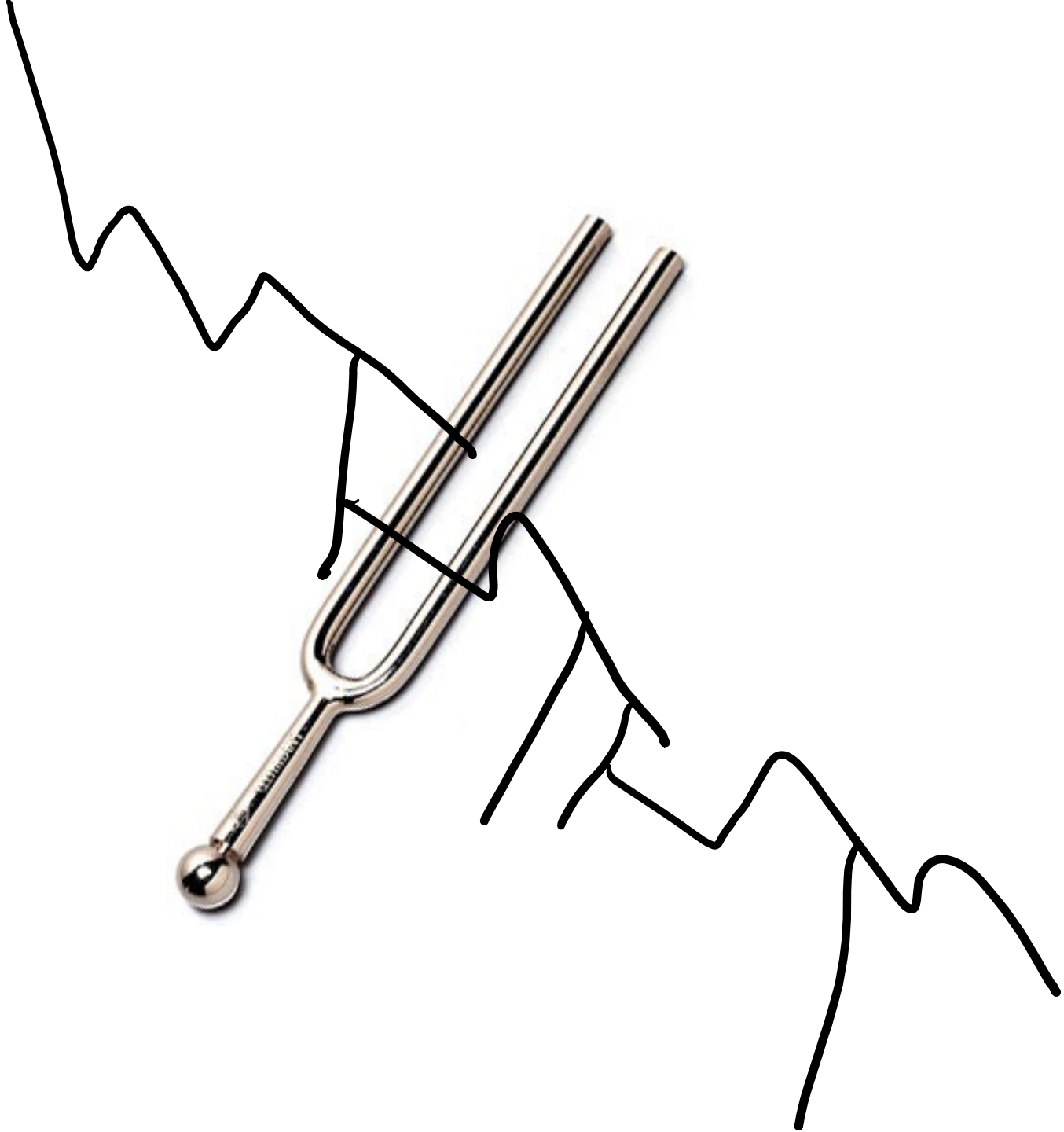
```
*BST> tests
+++ OK, passed 1 test.
+++ OK, passed 1 test.
+++ OK, passed 1 test.
+++ OK, passed 1 test.
+++ OK, passed 1 test.
```

```
*** Found example of just k, present
(0,0)
Branch Leaf 0 0 Leaf
[(0,0)] == [(0,0)]
```

```
*** Found example of just k, present
(0,0)
Branch Leaf 0 0 Leaf
[(0,0)] == [(0,0)]
```

```
*BST> quickCheck prop_Insert
*** Failed! Falsified (after 34 tests
and 12 shrinks):
(7,0)
Branch Leaf 7 1 Leaf
[(7,1)] /= [(7,0)]
```

labelledExamples



```
*Coins> quickCheck . withMaxSuccess 10000 $ prop_Add
+++ OK, passed 10000 tests:
47.92% overflow
47.84% normal
 4.24% boundary
```

```
summarize n
  | abs (n-maxCoinValue) < 3 = "boundary"
  | n <= maxCoinValue       = "normal"
  | n >  maxCoinValue       = "overflow"


prop_Add (Coin a) (Coin b) =
  cover  5 (abs (n-maxCoinValue) < 3) "boundary"$
  cover 40 (n <= maxCoinValue)        "normal"  $
  cover 40 (n >  maxCoinValue)        "overflow"$
  if validCoin (Coin n)
    then Just (Coin n)
    else Nothing
  where n = a+b
```

```
*Coins> quickCheck prop_Add
+++ OK, passed 100 tests:
55% overflow
45% normal
 5% boundary

*Coins> quickCheck prop_Add
+++ OK, passed 100 tests:
50% normal
50% overflow
 4% boundary
```

Only 4% boundary, but expected 5%

```
*Coins> quickCheck . checkCoverage $ prop_Add
*** Failed! Insufficient coverage (after 51200 tests):
50.696% normal
49.304% overflow
 4.231% boundary

Only 4.231% boundary, but expected 5.000%
```

```
*Coins> quickCheck . checkCoverage $ prop_Add
+++ OK, passed 102400 tests:
50.8828% normal
49.1172% overflow
 4.1230% boundary
```

```
*Coins> quickCheck . checkCoverage $ prop_Add
+++ OK, passed 800 tests:
50.0% normal
50.0% overflow
 3.9% boundary
```

```
*Coins> quickCheck . checkCoverage $ prop_Add
*** Failed! Insufficient coverage (after 1600 tests):
50.59% overflow
49.41% normal
 4.32% boundary

Only 4.32% boundary, but expected 10.00%
```

SEQUENTIAL TESTS OF STATISTICAL HYPOTHESES

By A. Wald

Columbia University

Table of Contents

Page

A. Introduction 117
B. Historical note 119

II. *Sequential test of a simple hypothesis against one alternative*

1. The current test procedure 122
2. The sequential test procedure: general definitions 123

2.1. Notion of a sequential test. 2.2. Efficiency of a sequential test. 2.3. Efficiency of the current procedure viewed as a particular case of a sequential test.

3. Sequential probability ratio test 125

3.1. Definition of the sequential probability ratio test. 3.2. Fundamental relations between α, β, A and B. 3.3. Determination of the values of A and B in practice. 3.4. Probability of accepting H_0 (or H_1) when some third hypothesis H is true. 3.5. Calculation of δ and η for binomial and normal distributions.

4. The number of observations required by the sequential probability ratio test 142

4.1. Expected number of observations necessary for reaching a decision. 4.2. Calculation of the quantities ξ and ξ' for binomial and normal distributions. 4.3. Saving in the number of observations as compared with the current test procedure. 4.4. The characteristic function, the moments and the distribution of the number of observations necessary for reaching a decision. 4.5. Lower limit of the probability that the sequential process will terminate with a number of trials less than or equal to a given number. 4.6. Truncated sequential analysis.

How often is it OK for a test to fail *when there is no bug?*

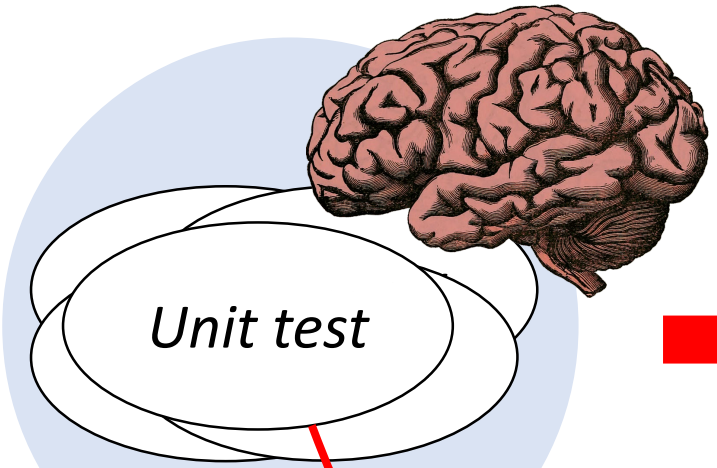

Agile Borat on Twitter: "M
Säker
https://twitter.com/AgileBorat/status/661640864880558 09
Home
About
Search Twitter
Agile Borat
@AgileBorat
Follow
My friend Azamat is very good developer, he is always have all unit test green. If unit test is fail, it is remove. Is best practice.
8:35 AM - 5 May 2011
256 Retweets
25 Likes
256
25
Agile Borat
@AgileBorat
Hello, I am Borat! Am Agile Coach, Scrum Master and Product Owner too also.
Joined April 2011
2
1

How often is it ok for a test to fail *when there is no bug?*

Never in the lifetime of the project!

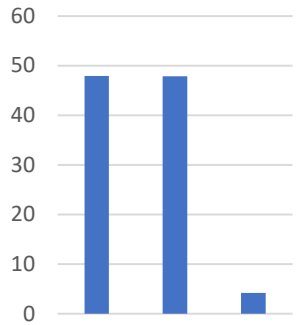
10^{-6}?

10^{-9}?



Unit test

Label



labelledExamples

checkCoverage

What haven’t we covered?

- Approaches to writing generators
  - Especially for types with complex invariants
- Defining shrinkers
  - Especially for types with complex invariants

Some further reading

John Hughes: **Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane.** A List of Successes That Can Change the World 2016: 169-186

The paper of the first lecture

John Hughes, Ulf Norell, Nicholas Smallbone, Thomas Arts: **Find more bugs with QuickCheck!** AST@ICSE 2016: 71-77

*Classify failing tests into equivalence classes provoking the "same bug", and automatically focus the search on **new** bugs*

Thomas Arts, John Hughes: **How Well are Your Requirements Tested?** ICST 2016: 244-254

Generating requirements-based test suites with QuickCheck—and a category of bugs that slip through them.

Alex Gerdes, John Hughes, Nicholas Smallbone, Stefan Hanenberg, Sebastian Ivarsson, Meng Wang: **Understanding formal specifications through good examples.** Erlang Workshop 2018: 13-24

Generating examples from a QuickCheck specification that ***convey its meaning*** *to a person.*

John Hughes, Benjamin C. Pierce, Thomas Arts, Ulf Norell: **Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service.** ICST 2016: 135-145

An approach to testing systems like Dropbox, that perform significant actions (like synchronization) in the background at unknown times—and some oddities we discovered.

Catalin Hritcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo de Amorim, Maxime Dénès, John Hughes, Benjamin C. Pierce, Dimitrios Vytiniotis: **Testing noninterference, quickly.** J. Funct. Program. 26: e4 (2016)

Using QuickCheck to test information flow security of an abstract machine

Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, Li-yao Xia: **Beginner's luck: a language for property-based generators.** POPL 2017: 114-129

A language that combines random generation and constraint solving to generate complex test data

Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, Robert Bruce Findler: **Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System.** ESOP 2015: 383-405

QuickCheck-like generation for testing programming language semantics