

# How to Specify it!

A guide to writing properties of pure functions.

John Hughes



# Imagine testing **reverse**...

```
test_Reverse =  
reverse [1,2,3] == [3,2,1]
```

*Comparison operator  
displaying a message  
if not equal*

*Function  
under test*

*Sample  
argument*

*Expected  
result*

# Imagine testing **reverse**... with QuickCheck

*Tell QuickCheck to  
generate random  
lists of integers*

```
prop_Reverse :: [Int] -> Property
prop_Reverse xs =
  reverse xs == ???
```

*A random  
argument*

*But what do we  
put here?*

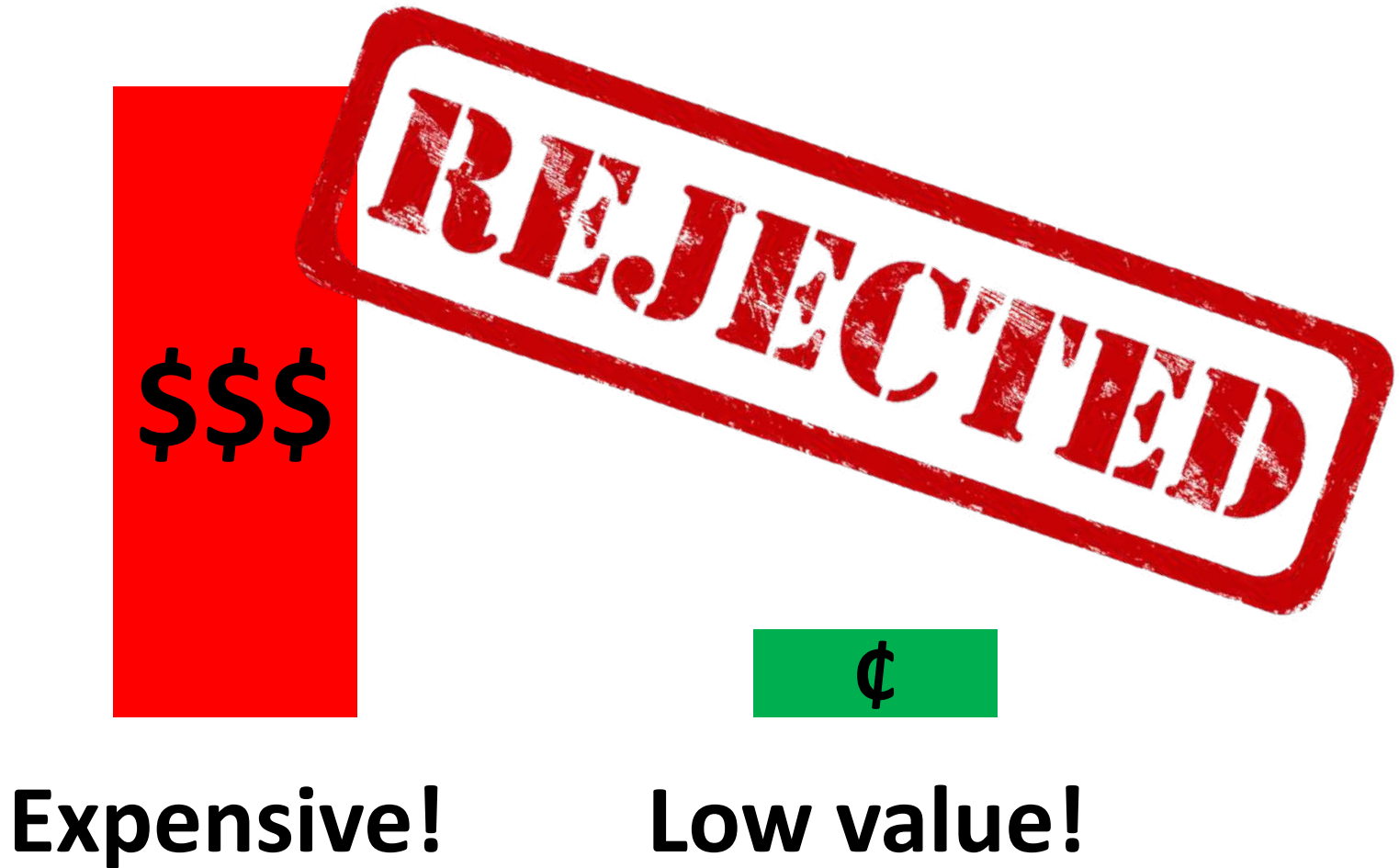
# Imagine testing **reverse**... with QuickCheck

```
prop_Reverse :: [Int] -> Property
prop_Reverse xs =
  reverse xs == predictReverse xs
```

*This is **not easier**  
to write than  
**reverse!***

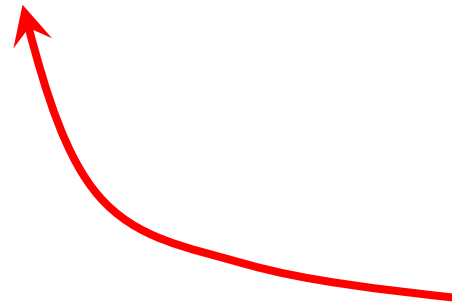
*We're likely to  
make the **same**  
**mistakes***

Replicating the code in the tests...



# What can we do instead?

```
prop_Reverse :: [Int] -> Property
prop_Reverse xs =
  reverse (reverse xs) === xs
```



*Check a **property**  
of the return  
value instead*

```
*Reverse> quickCheck prop_Reverse  
+++ OK, passed 100 tests.
```

*100 random lists!*



```
reverse xs = xs
```

*Passes even if we  
defined...*



```
*Reverse> quickCheck test_Reverse  
*** Failed! Falsified (after 1 test):  
[1,2,3] /= [3,2,1]
```

```
prop_Wrong :: [Int] -> Property
prop_Wrong xs = reverse xs == xs
```

```
*Reverse> quickCheck prop_Wrong
*** Failed! Falsified (after 3 tests and 3 shrinks):
```

```
[0,1]
```

```
[1,0] /= [0,1]
```

*Counterexample:  
Almost always [0,1],  
sometimes [1,0]*

## Shrinking

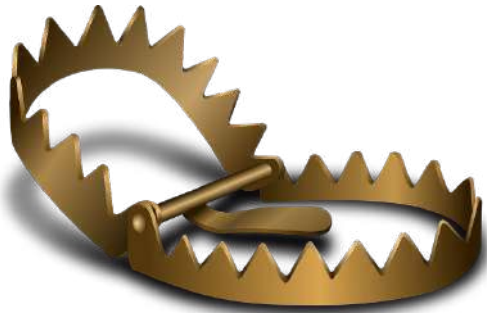
- Discards unnecessary list elements (we need at least two)
- Replaces integers by smaller integers (we need *distinct* integers, {0,1} are the two smallest)



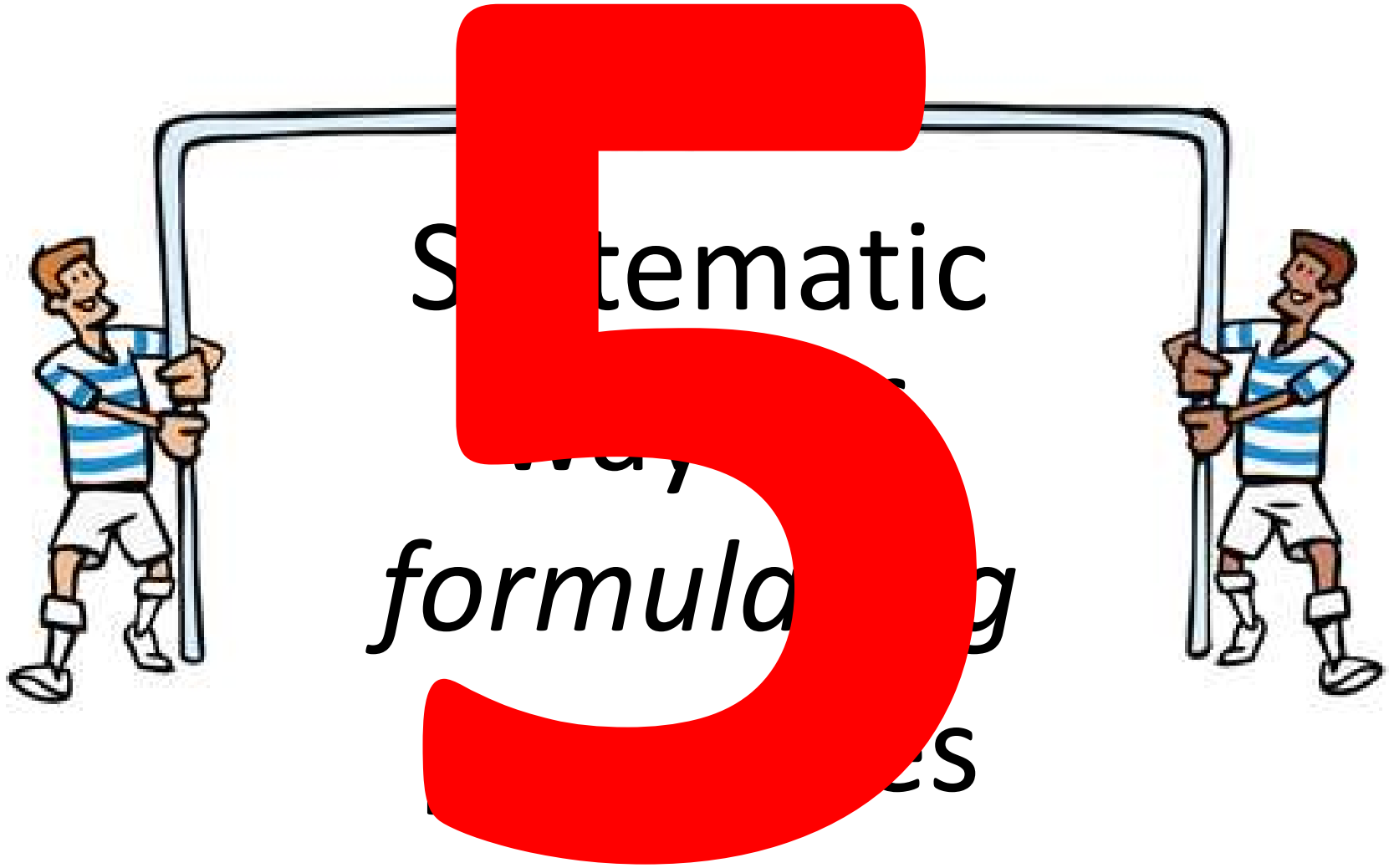
# Property Based Testing



- Random generation of *lots* of test cases
- Shrinking results in *minimal* counterexamples—easy to debug



- *Replicating code in the tests* is tempting, but expensive, and low value



# Example



```
data BST k v = Leaf
              | Branch (BST k v) k v (BST k v)
  deriving (Eq, Show, Generic)
```

**-- the operations under test**

```
find    :: Ord k => k          -> BST k v -> Maybe v
```

```
nil     ::                               BST k v
```

```
insert  :: Ord k => k -> v      -> BST k v -> BST k v
```

```
delete  :: Ord k => k          -> BST k v -> BST k v
```

```
union   :: Ord k => BST k v -> BST k v -> BST k v
```

**-- auxiliary operations**

```
toList  :: BST k v -> [(k, v)]
```

```
keys    :: BST k v -> [k]
```

# Generator and shrinker

```
instance (Ord k, Arbitrary k, Arbitrary v) =>
  Arbitrary (BST k v) where
```

```
-- generator
```

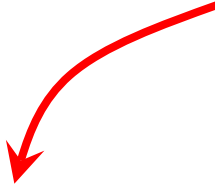
```
arbitrary =
```

```
  do kvs <- arbitrary
```

```
  return $
```

```
    foldr (uncurry insert) nil (kvs :: [(k,v)])
```

*Generate by  
inserting random  
key-value pairs*



```
-- shrinker
```

```
shrink = genericShrink
```

*Shrink using a  
generic QuickCheck  
mechanism*



# 1

Is there an *invariant*?

```
valid :: Ord k => BST k v -> Bool
```

```
valid Leaf = True
```

```
valid (Branch l k v r) =  
  valid l && valid r &&  
  all (<k) (keys l) && all (>k) (keys r)
```

# Invariant properties

```
prop_NilValid = valid (nil :: Tree)
```

```
prop_InsertValid :: Key -> Val -> Tree -> Bool  
prop_InsertValid k v t = valid (insert k v t)
```

```
prop_DeleteValid :: Key -> Tree -> Bool  
prop_DeleteValid k t = valid (delete k t)
```

```
prop_UnionValid :: Tree -> Tree -> Bool  
prop_UnionValid t t' = valid (union t t')
```

```
type Key = Int  
type Val = Int  
type Tree = BST Key Val
```



# insert

=== prop\_InsertValid from BSTSpec.hs:19 ===

\*\*\* Failed! Falsified (after 6 tests and 8 shrinks):

0

0

Branch Leaf 0 0 Leaf

=== prop\_DeleteValid from BSTSpec.hs:22 ===

\*\*\* Failed! Falsified (after 8 tests and 7 shrinks):

0

Branch Leaf 1 0 (Branch Leaf 0 0 Leaf)

=== prop\_UnionValid from BSTSpec.hs:25 ===

\*\*\* Failed! Falsified (after 7 tests and 9 shrinks):

Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)

Leaf



# insert

```
=== prop_InsertValid from BSTSpec.hs:19 ===  
*** Failed! Falsified (after 6 tests and 8 shrinks):  
0  
0  
Branch Leaf 0 0 Leaf
```

```
=== prop_DeleteValid from BSTSpec.hs:22 ===  
*** Failed! Falsified (after 8 tests and 7 shrinks):  
0  
Branch Leaf 1 0 (Branch Leaf 0 0 Leaf)
```

```
=== prop_UnionValid from BSTSpec.hs:25 ===  
*** Failed! Falsified (after 7 tests and 9 shrinks):  
Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)  
Leaf
```



# Testing our tests

```
prop_ArbitraryValid t = valid t
```

```
prop_ShrinkValid t =  
  all valid (shrink t)
```

Branch Leaf 0 0 (Branch Leaf 0 1 Leaf)

→ Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)

# Testing our tests

```
prop_ArbitraryValid t = valid t
```

```
prop_ShrinkValid t =  
  all valid (shrink t)
```

Branch Leaf 0 0 (Branch Leaf 0 1 Leaf)

→ Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)

# Testing our tests

```
prop_ArbitraryValid t = valid t
```

```
prop_ShrinkValid t =  
  all valid (shrink t)
```

Branch Leaf ~~0~~ 0 (Branch Leaf ~~0~~ 1 Leaf)

→ Branch Leaf ~~0~~ 0 (Branch Leaf ~~0~~ 0 Leaf)

# Testing our tests

```
prop_ArbitraryValid t = valid t
```

```
prop_ShrinkValid t =  
  valid t ==>  
    all valid (shrink t)
```

Branch Leaf 0 0 (Branch Leaf 1 0 Leaf)

→ Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)

# Fixing shrinking

```
shrink = genericShrink .
```

# 2

What is the *postcondition*?

“After calling **insert**, we should be able to **find** the key inserted, and any other keys present beforehand”

```
prop_InsertPost k v t k' =  
  find k' (insert k v t)  
  ==  
  if k==k' then Just v else find k' t
```

*Rarely true?*

# What is the postcondition of **find**?

"After calling **find**,

—if the key is present in the tree, the result is **Just value**

—if the key is not present, the result is **Nothing**"



*How can we  
tell this?*

# By construction!

*Don't test for  
presence—ensure by  
construction*

*Can **every** tree containing  
**k** be expressed in this  
form?*

```
prop_FindPostPresent k v t =  
  find k (insert k v t) === Just v
```

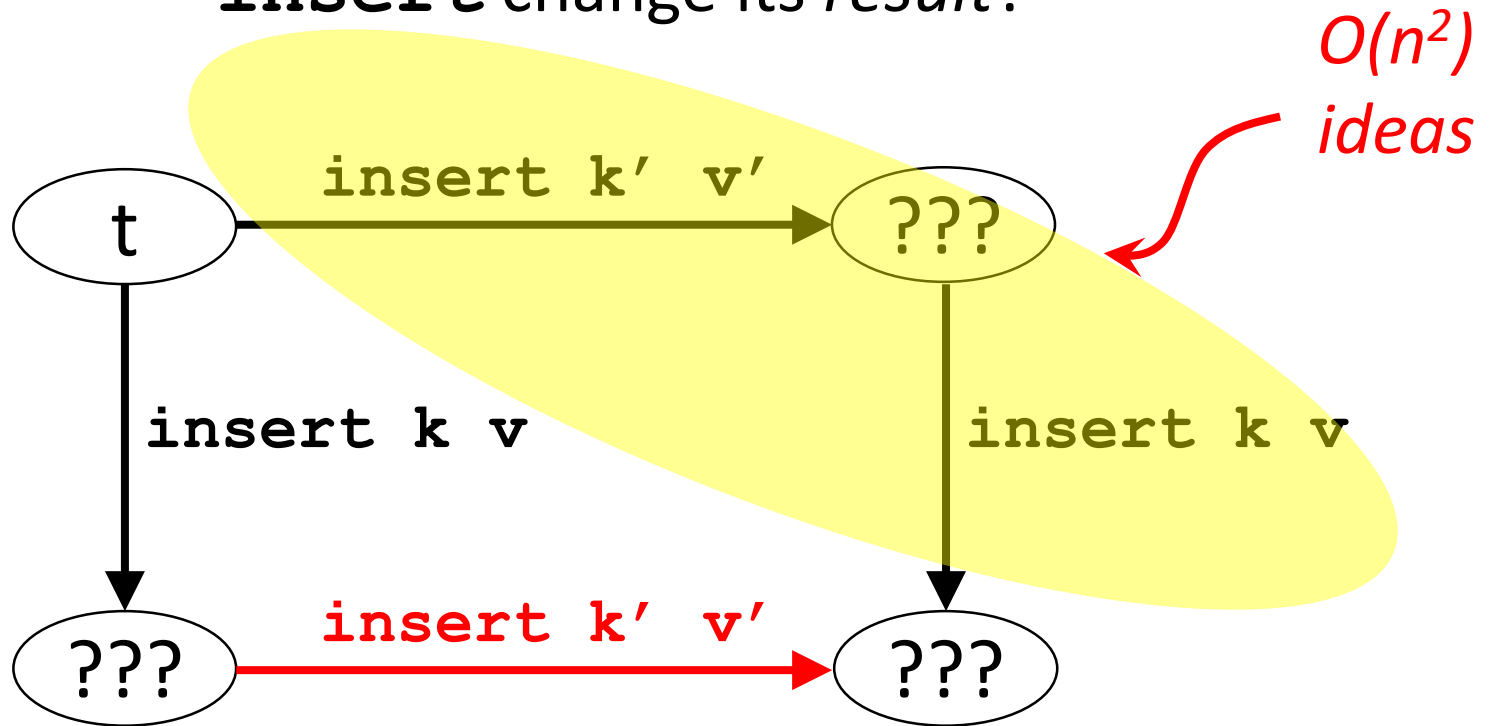
```
prop_FindPostAbsent k t =  
  find k (delete k t) === Nothing
```



# 3

## Metamorphic properties

"How does changing the *input* of **insert** change its *result*?"



*Modified call*

`prop_InsertInsert (k,v) (k',v') t =`

`insert k v (insert k' v' t)`

`===`

`insert k' v' (insert k v t)`

*Original call*

*Relationship*

```
prop_InsertInsert (k,v) (k',v') t =
  insert k v (insert k' v' t)
===
  insert k' v' (insert k v t)
```

 *Is this really true?*



```
=== prop_InsertInsert from BSTSpec.hs:78 ===
*** Failed! Falsified (after 2 tests and 5 shrinks):
```

**(0,0)**

**(0,1)**

 *Last insertion wins*

Leaf

Branch Leaf 0 0 Leaf /= Branch Leaf 0 1 Leaf

```
prop_InsertInsert (k,v) (k',v') t =
  insert k v (insert k' v' t)
===
  if k==k' then insert k v t else
  insert k' v' (insert k v t)
```

```
=== prop_InsertInsert from BSTSpec.hs:78 ===
*** Failed! Falsified (after 2 tests):
(1,0)
(0,0)
Leaf
Branch Leaf 0 0 (Branch Leaf 1 0 Leaf) /=
Branch (Branch Leaf 0 0 Leaf) 1 0 Leaf
```

# Equivalence for trees

```
t1 =~= t2 =  
  toList t1 == toList t2
```

```
prop_InsertInsert (k,v) (k',v') t =  
  insert k v (insert k' v' t)  
  =~=  
  if k==k' then insert k v t else  
  insert k' v' (insert k v t)
```

MET 19



# MET 2019

4th International Workshop on

**Metamorphic Testing**

In conjunction with ICSE 2019

**Montréal, QC, Canada**

**May 26, 2019**

# 4

## Inductive properties



```
prop_UnionNil t =  
  union nil t === t
```

```
prop_UnionInsert t t' (k,v) =  
  union (insert k v t) t'  
  ==~=  
  insert k v (union t t')
```


***union** must be correct, by  
induction on the first argument*

# Completeness: can every tree be constructed just using **insert**?

```
insertions Leaf = []  
insertions (Branch l k v r) =  
  (k,v):insertions l++insertions r
```

```
prop_InsertComplete t =  
  t  
  ===  
  foldl (flip $ uncurry insert) nil (insertions t)
```

*Structurally equal to!*



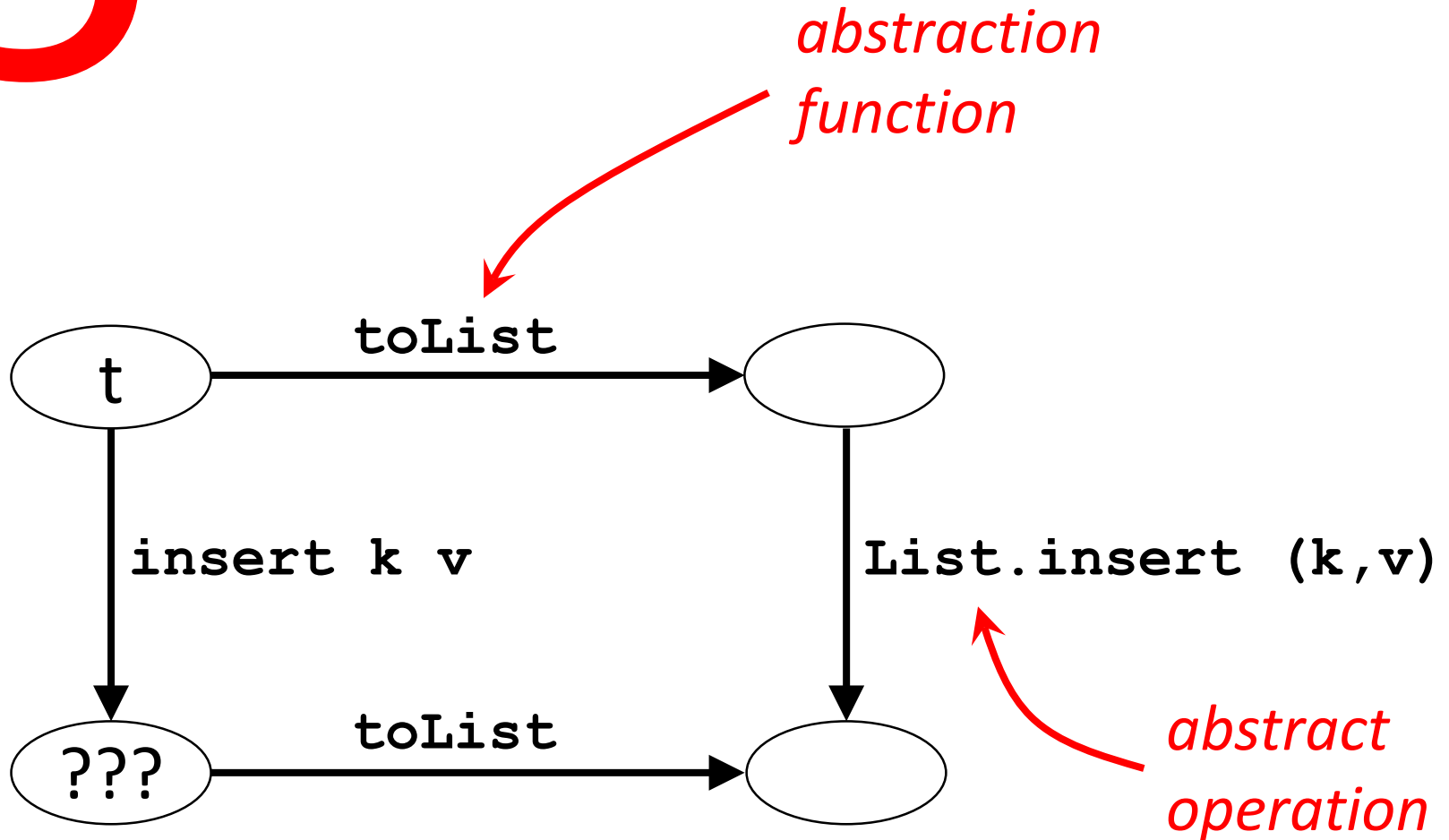
```
prop_InsertCompleteForDelete k t =  
  prop_InsertComplete (delete k t)
```

```
prop_InsertCompleteForUnion t t' =  
  prop_InsertComplete (union t t')
```



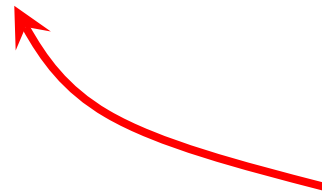
# 5

## Model-based properties



```
prop_InsertModel (k,v) t =
  toList (insert k v t)
  ==
  L.insert (k,v) (toList t)
```

```
*BSTSpec> quickCheck prop_InsertModel
*** Failed! Falsified (after 13 tests and 7 shrinks):
(1,0)
Branch Leaf 1 0 Leaf
[(1,0)] /= [(1,0),(1,0)]
```

 *duplicated key*

```
prop_InsertModel (k,v) t =  
  toList (insert k v t)
```

```
===
```

```
L.insert (k,v) (deleteKey k $ toList t)
```

Acta Informatica 1, 271—281 (1972)  
© by Springer-Verlag 1972

## Proof of Correctness of Data Representations

C. A. R. Hoare

Received February 16, 1972

*Summary.* A powerful method of simplifying the proofs of program correctness is suggested; and some new light is shed on the problem of functions with side-effects.

### 1. Introduction

In the development of programs by stepwise refinement [1-4], the programmer is encouraged to postpone the decision on the representation of his data until after he has designed his algorithm, and has expressed it as an "abstract" program operating on "abstract" data. He then chooses for the abstract data some convenient and efficient concrete representation in the store of a computer; and finally programs the primitive operations required by his abstract program in terms of this concrete representation. This paper suggests an automatic method of accomplishing the transition between an abstract and a concrete program, and also a method of proving its correctness; that is, of proving that the concrete representation exhibits all the properties expected of it by the "abstract" pro-

Type of property	Number of properties
Invariant	4
Postcondition	5
Metamorphic	16
Model-based	5

**insert**



**delete**



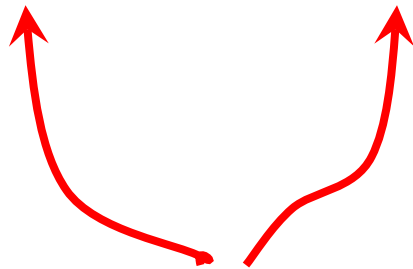
**union**



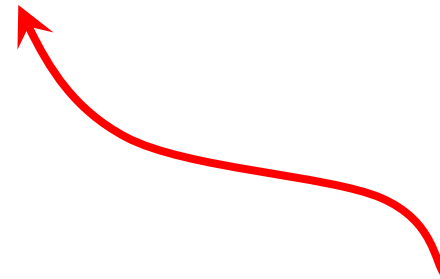
Type of property	Number of properties	Bugs missed
Invariant	4	5
Postcondition	5	0
Metamorphic	16	0
Model-based	5	0

# Effectiveness

```
prop_FindPostPresent k v t =  
  find k (insert k v t) == Just v
```



*May find  
bug in  
find or  
insert*



*Will **not** find bugs  
in **delete** or  
**union***



Effectiveness

2/3

```
prop_FindPostPresent k v t =  
  find k (insert k v t) == Just v
```

insert



delete



union



Type of property	Number of properties	Bugs missed	Effectiveness
Invariant	4	5	<b>38%</b>
Postcondition	5	0	<b>79%</b>
Metamorphic	16	0	<b>90%</b>
Model-based	5	0	<b>100%</b>

```
=== prop_UnionPost from BSTSpec.hs:75 ===  
Mean time to failure: 50.04595404595405
```

```
=== prop_InsertUnion from BSTSpec.hs:117 ===  
Mean time to failure: 15.5374626
```

```
=== prop_DeleteUnion from BSTSpec.hs:145 ===  
Mean time to failure: 4.696303694
```

```
=== prop_UnionDeleteInsert from BSTSpec.hs:167 ===  
Mean time to failure: 7.32767233
```

```
=== prop_UnionUnionAssociate from BSTSpec.hs:185 ===  
Mean time to failure: 8.95104895
```

```
=== prop_FindUnion from BSTSpec.hs:206 ===  
Mean time to failure: 1.72827
```

```
=== prop_UnionModel from BSTSpec.hs:290 ===  
Mean time to failure: 8.368631368631368
```



Logically equivalent!

```
prop_UnionPost t t' k =  
  find k (union t t')  
  ===  
  (find k t <|> find k t')
```

```
prop_UnionModel t t' =  
  toList (union t t')  
  ===  
  List.sort  
    (List.unionBy  
      ((==) `on` fst)  
      (toList t)  
      (toList t'))
```

# Mean time to failure

Property type	Min	Max	Mean
Postcondition	9.7	160	68
Metamorphic	1	401	61.6
Model-based	5	6.5	5.8

Averaged over seven bugs, and all properties of each type that detect the bugs



## **Model-based**

- Easier to think of than postconditions
- Require fewer properties than metamorphic approach
- Are the most effective properties
- Find bugs fastest
- *Complete* specification

## **Metamorphic**

- *Do not require a model*
- Easiest to write
- Good effectiveness