Testing with Properties

John Hughes

CHALMERS QuviQ

Anatomy of a property

*We need to specify the **type**, to determine what data will be generated*

- A function that should always return True*

x and xs are the test case, which will be randomly generated

```
prop_Delete :: Int32 -> [Int32] -> Bool
prop_Delete x xs =
  not (x `elem` delete x xs)
```

Once an element has been deleted from a list, it should no longer be present

*Not quite—the truth is a bit more general

Testing a property

```haskell
import Test.QuickCheck

prop_Delete :: Int32 -> [Int32] -> Bool
prop_Delete x xs =
  not (x `elem` delete x xs)
```

```
*Example> quickCheck prop_Delete
+++ OK, passed 100 tests.
*Example> quickCheck prop_Delete
+++ OK, passed 100 tests.
*Example> quickCheck prop_Delete
+++ OK, passed 100 tests.
```

Testing a property

```
*Example> quickCheck prop_Delete
*** Failed! Falsifiable (after 9 tests and 1 shrink):
2
[2,2]
```

```
prop_Delete x xs =
  not (x `elem` delete x xs)
```

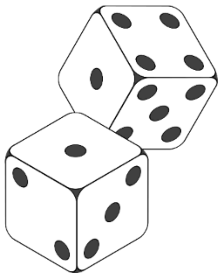
```
*Example> delete 2 [2,2]
[2]
```

More examples

```
*Example> quickCheck prop_Delete
*** Failed! Falsifiable (after 9 tests and 2 shrinks):
-12
[-12,-12]
*Example> quickCheck prop_Delete
*** Failed! Falsifiable (after 7 tests and 1 shrink):
-7
[-7,-7]
*Example> quickCheck prop_Delete
*** Failed! Falsifiable (after 4 tests):
1
[1,1]
```

shrinking discarded list elements that don't affect the failure

different examples (random)

 +  = Easy debugging

Everything in the reported test case is relevant to the failure!

2
[2,2]

*We know **we must have both elements** for the test to fail; otherwise QuickCheck would shrink it to*

2
[2]

Where is the bug?

- We have a failing test!

```
*Example> quickCheck prop_Delete
*** Failed! Falsifiable (after 10 tests and 4 shrinks):
6
[6,6]
```

- Is delete in Data.List wrong?

What shall we do about it?

- The property fails for repeated values in the list.

```
prop_Delete x xs =
  xs == nub xs ==>
  not (x `elem` delete x xs)
```

```
*Example> quickCheck . withMaxSuccess 10000 $ prop_Delete
+++ OK, passed 10000 tests.
```

- Is this reasonable?

A common approach…

- Find a bug with QuickCheck
- Characterize the situations in which the bug appears as a predicate
- Add a precondition:

  not (buggy x xs) ==> ...

We don’t think anybody will do that!

We don’t think anybody *should* do that!

Our code isn’t *supposed* to work in that case!

Is this a *documented* restriction?

Are we *sure* the code is never used in this way?

Can we at least *check* that this is the case?

Preconditions

Don't test this—
we know it
doesn't work!

Another criticism

```
prop_Delete x xs =
  not (x `elem` delete x xs)
```

- The property is *weak!*

```
delete x xs = []
```

passes this test!

IDEA: construct a test case with a *predictable* result

```
prop_Delete :: Int -> _
prop_Delete x xs ys =
  delete x (xs++[x]++ys) == xs++ys
```

```
*Example> quickCheck prop_Delete
*** Failed! Falsifiable (after 19 tests and 8 shrinks):
6
[6,0]
[]
```

IDEA: construct a test case with a *predictable* result

```
prop_Delete :: Int -> _
prop_Delete x xs ys =
  delete x (xs++[x]++ys) === xs++ys
```

```
*Example> quickCheck prop_Delete
*** Failed! Falsifiable (after 19 tests and 8 shrinks):
6
[6,0]
[]
[0,6] /= [6,0]
```

delete 6 [6,0,6]

A good property

```
prop_Delete :: Int -> _
prop_Delete x xs ys =
  not (x `elem` xs) ==>
  delete x (xs++[x]++ys) === xs++ys
```

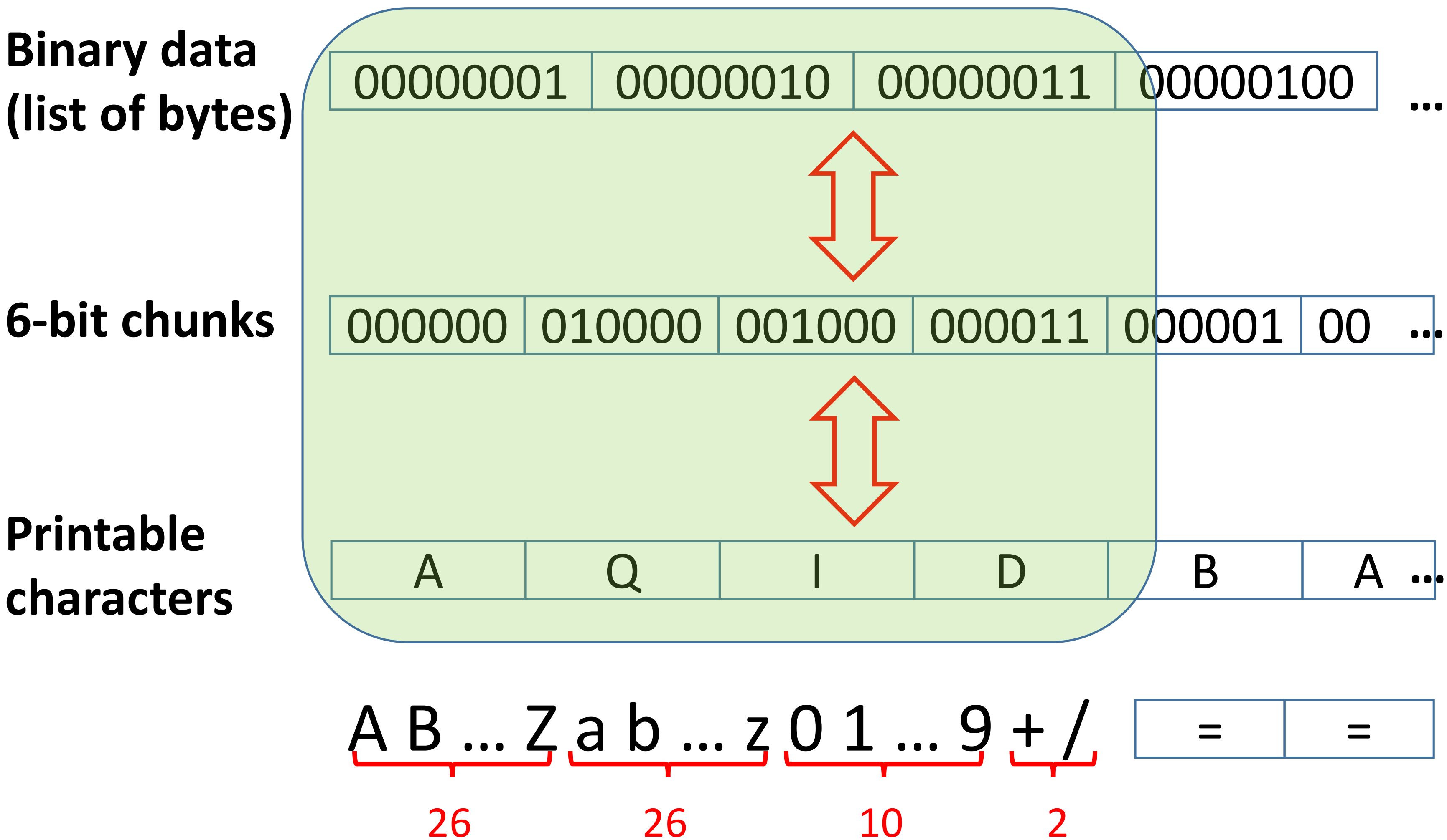
```
*Example> quickCheck . withMaxSuccess 10000 $ prop_Delete)
+++ OK, passed 10000 tests.
```

- Precisely characterizes the behaviour of delete, in all cases when x occurs in the argument.

Unit tests for Base64 encoding

```
import Codec.Binary.Base64.String

testTwoPads =
  encode "Aladdin:open sesame"
    ===  "QWxhZGRpbjpvcGVuIHNlc2FtZQ=="

testOnePad =
  encode "Hello World"
    ===  "SGVsbG8gV29ybGQ="

testNoPad =
  encode "Aladdin:open sesam"
    ===  "QWxhZGRpbjpvcGVuIHNlc2Ft"

testSymbols =
  encode "0123456789!@#0^&*();:<>,. []{}"
    ===  "MDEyMzQ1Njc4OSFAIzBeJiooKTs6PD4sLiBbXXt9"
```

Base64 encoding



Unit tests for Base64 encoding

```
import Codec.Binary.Base64.String

testTwoPads =
  encode "Aladdin:open sesame"
    ===  "QWxhZGRpbjpvcGVuIHNlc2FtZQ=="

testOnePad =
  encode "Hello World"
    ===  "SGVsbG8gV29ybGQ="

testNoPad =
  encode "Aladdin:open sesam"
    ===  "QWxhZGRpbjpvcGVuIHNlc2Ft"

testSymbols =
  encode "0123456789!@#0^&*();:<>,. []{}"
    ===  "MDEyMzQ1Njc4OSFAIzBeJiooKTs6PD4sLiBbXXt9"
```

How do we write a property?

```
prop_Base64 bs =
  encode bs === ???
```

*Must we **reimplement** base64 encoding in the test?*

Expensive!

Low value!

Another possibility…

String = [Char]

Unicode!

```
prop_RoundTrip s =
  decode (encode s) === s
```

```
*Tests> quickCheck prop_RoundTrip
*** Failed! (after 4 tests and 2 shrinks):
Exception:
  toChar: Can't happen: Bad input: 69452
…
```

Another possibility…

```
prop_RoundTrip ws =
  decode (encode s) === s
  where s = map w8tochar ws

w8tochar :: Word8 -> Char
w8tochar = chr . fromIntegral
```

What does this test?

```
prop_RoundTrip ws =
  decode (encode s) === s
  where s = map w8tochar ws
```

- A bug in the encoder or decoder is *certain* to be found (e.g. wrong table entry)
- A *misunderstanding* of base 64 encoding will *not* be found
- This property + unit tests == quite effective testing!

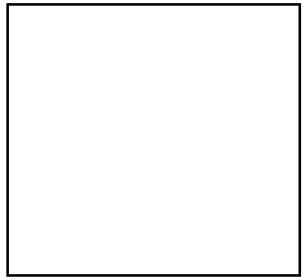
Other properties?

- The length of an encoding is a multiple of 4
- Every character in an encoding belongs to the base 64 alphabet
- Groups of three bytes are encoded independently

  encode s === encode (take 3 s) ++ encode (drop 3 s)
- The encoding represents the *same bit string* as the original

Example: Binary Search Trees

```
module BST where

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

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
```



vs.

- Tests are *inside* the abstraction boundary
- Can refer to the *representation*
- Properties important to the *developer*

- Tests are *outside* the abstraction boundary
- Can refer only to the exported API
- Properties important to the *user*

- Binary search trees have an important *invariant*:

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

keys t = map fst (toList t)

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

Validity properties

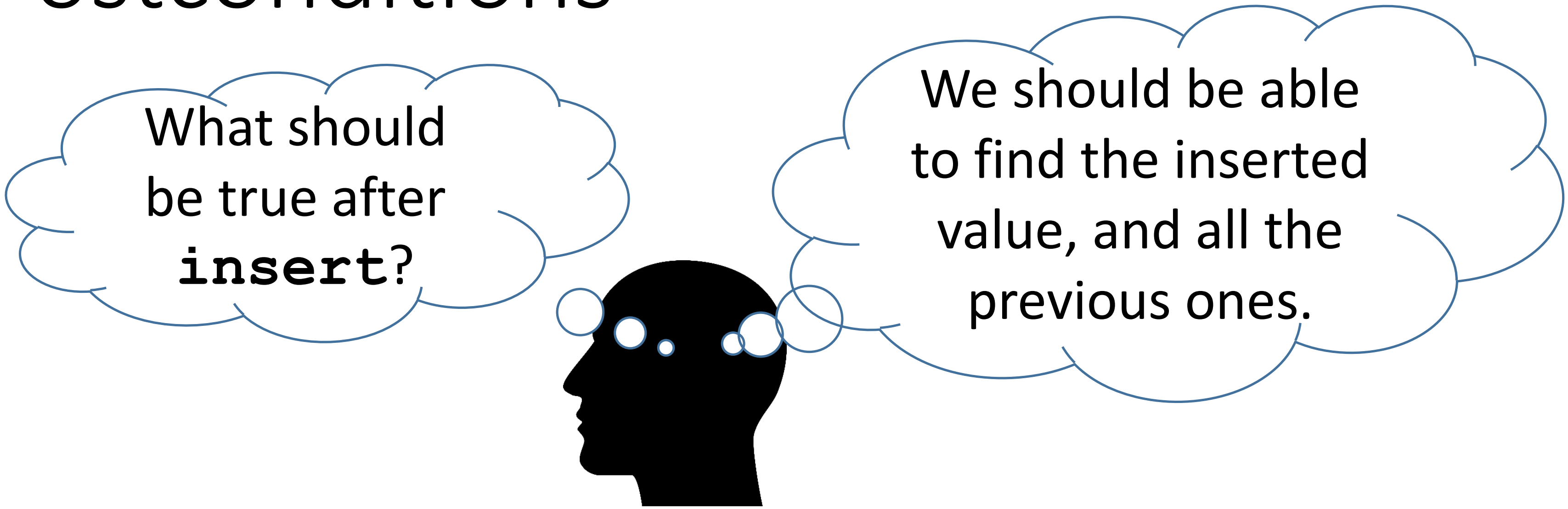
```
prop_InsertValid :: Int -> Int -> _
prop_InsertValid k v t = valid (insert k v t)
```

...and so on for all the ways to create a BST

```
prop_ArbitraryValid :: BST Int Int -> _
prop_ArbitraryValid t = valid t
```

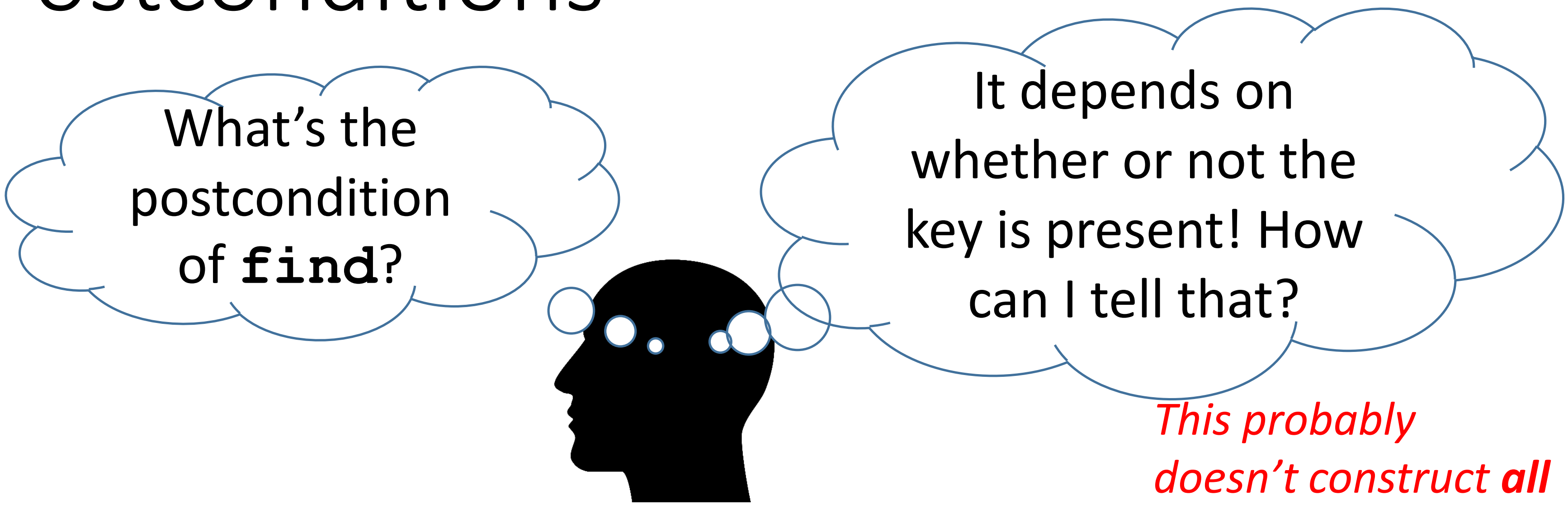
Why do we need to test this?

Postconditions




```
prop_InsertPost :: Int -> Int -> _
prop_InsertPost k v t k' =
  find k' (insert k v t)
  ===
  if k==k' then Just v else find k' t
```

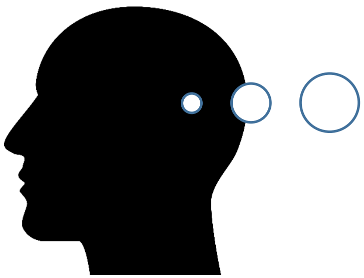
Postconditions



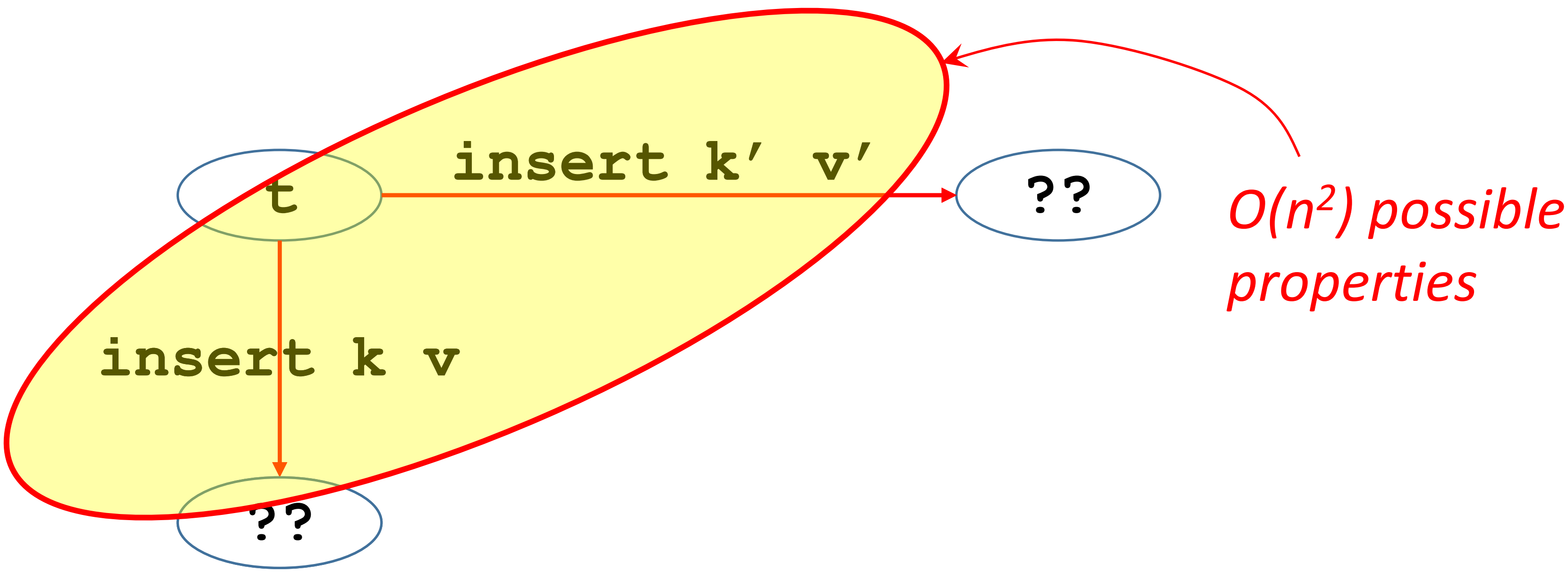
*This probably doesn't construct **all possible** trees in which* **k** *can be found*

```
prop_FindPostPresent :: Int -> Int -> _
prop_FindPostPresent k v t =
  find k (insert k v t) === Just v

prop_FindPostAbsent :: Int -> BST Int Int -> _
prop_FindPostAbsent k t =
  find k (delete k t) === Nothing
```

Metamorphic Testing



If I change the *input* to a function , can I predict the *change* to the output?



Metamorphic Testing

```
prop_InsertInsert :: Int -> Int -> _
prop_InsertInsert k v k' v' t =
  insert k v (insert k' v' t)
  ===
  insert k' v' (insert k v t)
```

```
*BSTSpec> quickCheck prop_InsertInsert
*** Failed! Falsifiable (after 4 tests and 8 shrinks):
0
0
0
1
Leaf
Branch Leaf 0 0 Leaf /= Branch Leaf 0 1 Leaf
```

Metamorphic Testing

```
prop_InsertInsert :: Int -> Int -> _
prop_InsertInsert k v k' v' t =
  insert k v (insert k' v' t)
  ===
  insert k' v' (insert k v t)
```

```
*BSTSpec> quickCheck prop_InsertInsert
*** Failed! Falsifiable (after 4 tests and 8 shrinks):
0
0
0
1
Leaf
Branch Leaf 0 0 Leaf /= Branch Leaf 0 1 Leaf
```

We inserted the ***same key*** *twice!*

Of course we store the ***most recent*** *value!*

Metamorphic Testing

```
prop_InsertInsert :: Int -> Int -> _
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)
```

```
*BSTSpec> quickCheck prop_InsertInsert
*** Failed! Falsifiable (after 12 tests and 8 shrinks):
0
0
1
0
Leaf
Branch (Branch Leaf 0 0 Leaf) 1 0 Leaf /=
Branch Leaf 0 0 (Branch Leaf 1 0 Leaf)
```

Metamorphic Testing

[redacted]

```
prop_InsertInsert :: Int -> Int -> _
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)
```

```
*BSTSpec> quickCheck prop_InsertInsert
*** Failed! Falsifiable (after 12 tests and 8 shrinks):
0
0
1
0
Leaf
Branch (Branch Leaf 0 0 Leaf) 1 0 Leaf /=
Branch Leaf 0 0 (Branch Leaf 1 0 Leaf)
```

Different keys this time; different failing test.

Metamorphic Testing

```
prop_InsertInsert :: Int -> Int -> _
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)
```

```
*BSTSpec> quickCheck prop_InsertInsert
*** Failed! Falsifiable (after 12 tests and 8 shrinks):
0
0
1
0
Leaf
Branch (Branch Leaf 0 0 Leaf) 1 0 Leaf /=
Branch Leaf 0 0 (Branch Leaf 1 0 Leaf)
```

*The **contents** of the trees are the same; only the **shape** is different!*

Abstraction

- The order of insertions affects the tree *shape*, but not the *semantics*

- Compare trees ”up to shape”

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

`toList` *abstracts away the shape, leaving only the contents*

Metamorphic Testing: Success!

```
prop_InsertInsert :: Int -> Int -> _
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)
```

```
*BSTSpec> quickCheck prop_InsertInsert
+++ OK, passed 100 tests.
```

Recall our postcondition test for **`insert`**…

```
prop_InsertPost :: Int -> Int -> _
prop_InsertPost k v t k' =
  find k' (insert k v t)
  ===
  if k==k' then Just v else find k' t
```

This is just a metamorphic test for **`find`***!*

A very simple metamorphic property

```
prop_SizeInsert :: Int -> Int -> _
prop_SizeInsert k v t =
  size (insert k v t) >= size t
```

Even such simple properties can find bugs!

T.Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T.H. Tse, and Z.Q. Zhou, "Metamorphic testing: A review of challenges and opportunities", *ACM Computing Surveys* 51 (1): 4:1-4:27 (2018).

Inductive Testing

- How would we *prove* that **`union`** works?
  - By *induction* on the size of the argument!

- Base case:

  `union nil t`

- Inductive case:

  `union (insert k v t) t′`

  (assuming **`union t t′`** works)

- If union works in both these cases, it works for *all* inputs, by induction!

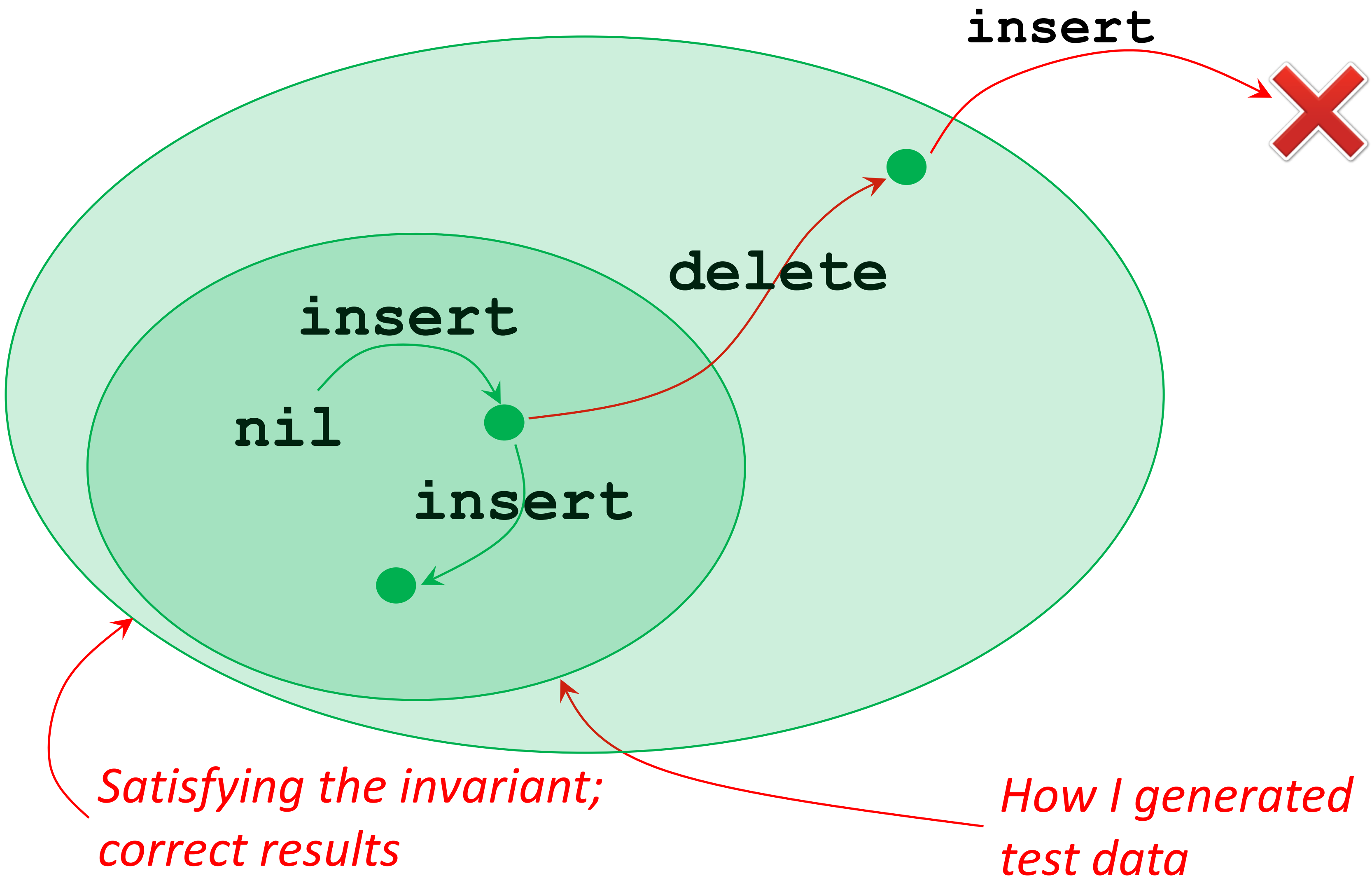
Inductive tests for **`union`**

Is this ***complete?***

```
prop_UnionBaseCase :: BST Int Int -> _
prop_UnionBaseCase t =
  union nil t === t

prop_UnionInductionStep :: Int -> Int -> _
prop_UnionInductionStep k v t t' =
  union (insert k v t) t' =~= insert k v (union t t')
```

- Could make an *inefficient definition* of **`union`** (if **`insert`**/**`nil`** were constructors); makes an *efficient test*
- Many applications—e.g. graph algorithms, search algorithms, SAT solvers…

Can every BST be built with just **`insert`** and **`nil`**?



Is there a sequence of insertions to build an arbitrary tree?

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

valid' t =
   t === foldl (flip $ uncurry insert) nil
                (insertions t)
```

*Note we require exactly the same **structure.***

```
prop_ArbitraryValid' :: BST Int Int -> _
prop_ArbitraryValid’ t = valid’ t
```

*Except, of course, that we only **generate** trees built by* **`insert`***!*

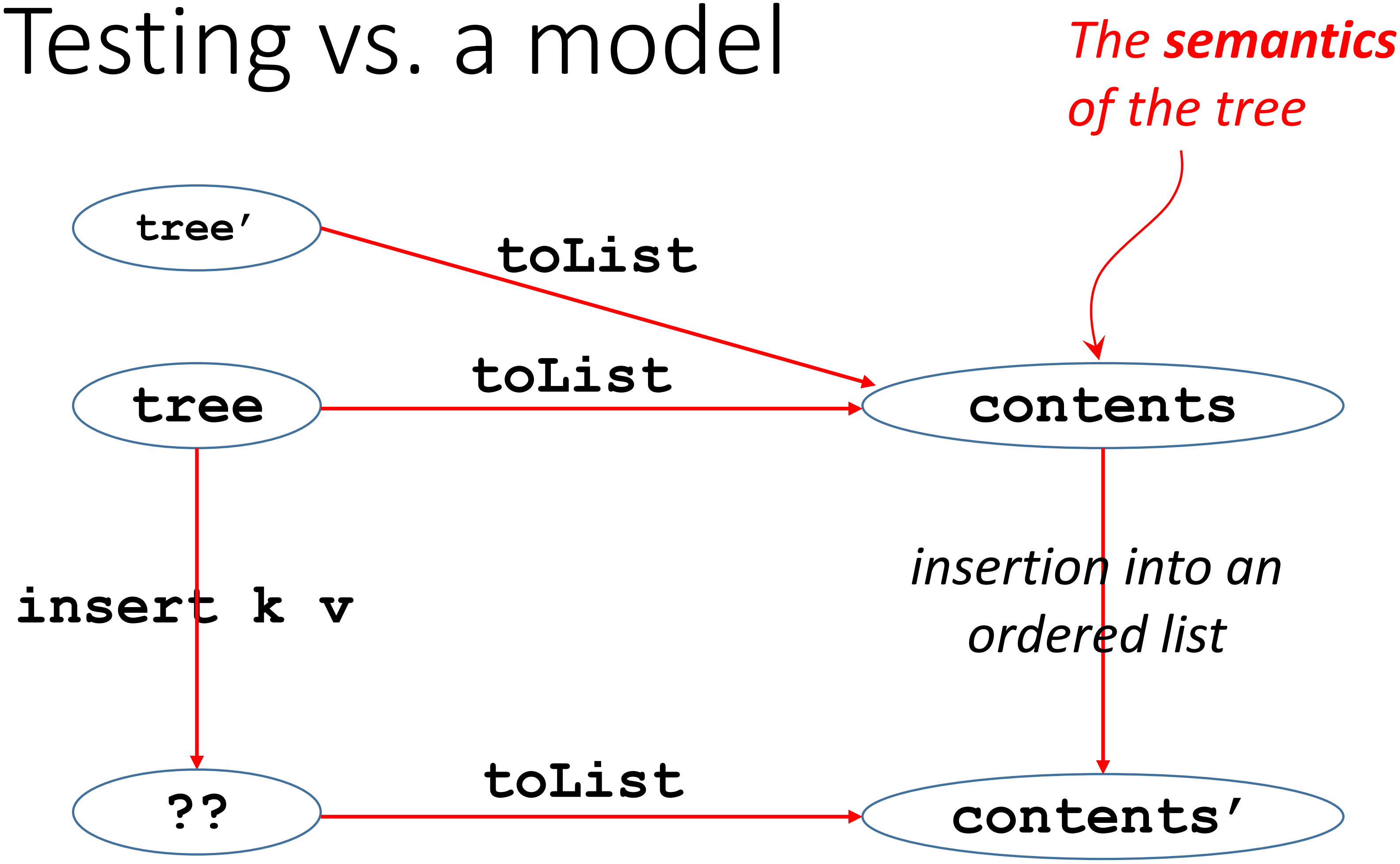
Additional properties…

```
prop_NilValid’            = valid' (nil :: BST Int Int)

prop_InsertValid' :: Int -> Int -> _
prop_InsertValid' k v t = valid' (insert k v t)

prop_DeleteValid' :: Int -> BST Int Int -> _
prop_DeleteValid' k t   = valid' (delete k t)

prop_UnionValid' :: BST Int Int -> _
prop_UnionValid' t t’   = valid' (union t t')
```

All the ways of building trees result in trees that *could* be built with `insert`.

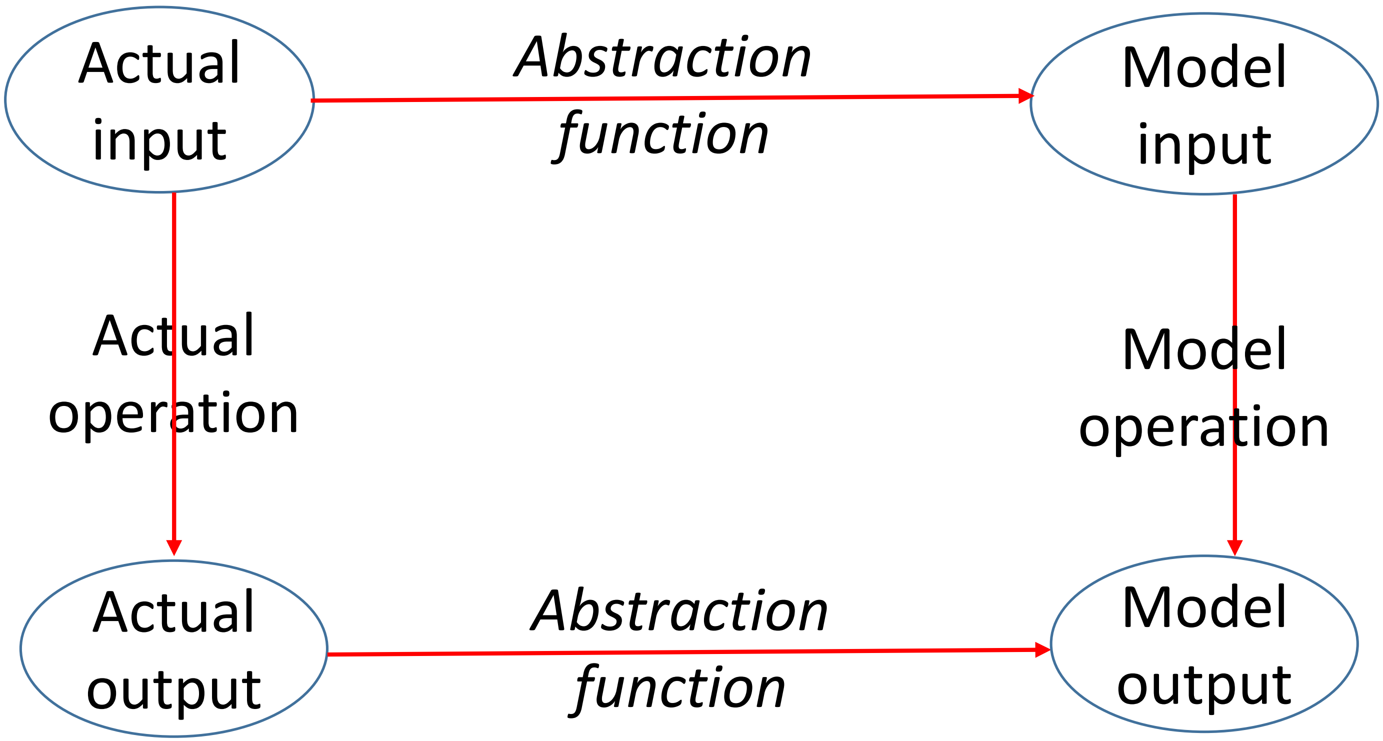
A new *invariant* on trees, testing our tests!

Testing vs. a model



We can implement the *same* API using the model instead, to serve as a *specification* for the real code

The Basic Principle



Model based test of **insert**

```
import qualified Data.List as L
…
prop_InsertModel :: Int -> Int -> _
prop_InsertModel k v t =
  toList (insert k v t)
    ===
  L.insert (k,v) (toList t)
```

```
*BSTSpec> quickCheck prop_InsertModel
*** Failed! Falsifiable (after 6 tests and 5 shrinks):
0
0
Branch Leaf 0 0 Leaf
[(0,0)] /= [(0,0),(0,0)]
```

List insertion does not replace existing keys

Model based test of **insert**

```
import qualified Data.List as L
…
prop_InsertModel :: Int -> Int -> _
prop_InsertModel k v t =
  toList (insert k v t)
    ===
  L.insert (k,v) (deleteKey k $ toList t)
```

```
*BSTSpec> quickCheck prop_InsertModel
+++ OK, passed 100 tests.
```

Acta Informatica 1, 271—281 (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-

Summary of property types

- Validity
- Postconditions
- Metamorphic
- Inductive
- Model-based

- Auto-generated

JFP **27**, e18, 49 pages, 2017.
doi:10.1017/S0956796817000090

Quick specifications for the busy programmer

NICHOLAS SMALLBONE, MOA JOHANSSON,
KOEN CLAESSEN and MAXIMILIAN ALGEHED
Chalmers University of Technology, Gothenburg, Sweden
(*e-mails:* nicsma@chalmers.se, moa.johansson@chalmers.se, koen@chalmers.se,
algehed@chalmers.se)

Abstract

QuickSpec is a theory exploration system which tests a Haskell program to find equational properties of it, automatically. The equations can be used to help understand the program, or as lemmas to help prove the program correct. QuickSpec is largely automatic: the user just supplies the functions to be tested and QuickCheck data generators. Previous theory exploration systems, including earlier versions of QuickSpec itself, scaled poorly. This paper describes a new architecture for theory exploration with which we can find vastly more complex laws than before, and much faster. We demonstrate theory exploration in QuickSpec on problems both from functional programming and mathematics.

1 Introduction

Formal specifications are a powerful tool for understanding programs. For example,

QuickSpec: Property Discovery

- Explore *equations* satisfied by an API

```
type BSTII = BST Int Integer

main = quickSpec [
  monoType (Proxy :: Proxy BSTII),

  con "nil"    (nil    :: BSTII),
  con "find"   (find   :: Int -> BSTII -> Maybe Integer),
  con "insert" (insert :: Int -> Integer -> BSTII -> BSTII)
  ]
```

Explore equations at this type...

...involving these functions

...at these types

== Functions ==

nil :: BST Int Integer

find :: Int -> BST Int Integer -> Maybe Integer

insert :: Int -> Integer -> BST Int Integer -> BST Int Integer

== Laws ==

1. find x nil = find y nil
2. find x (insert x y z) = find x (insert x y w)
3. find x (insert x y z) = find w (insert w y z)
4. find x (insert y z nil) = find
5. insert x y (insert x z w) =

find x nil = Nothing

find x (insert x y z)
= Just x

Extend the vocabulary

```
con "Nothing" (Nothing :: Maybe Integer),
con "Just"    (Just    :: Integer -> Maybe Integer),
```

== Laws ==

1. find x nil = Nothing
2. find x (insert x y z) = Just y
3. find x (insert y z nil) = find y (insert x z nil)
4. insert x y (insert x z w) = insert x y w

Now in the expected form

Inserting a key twice just keeps the second value

Finding equations ”up to equivalence”

- Recall

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

```
instance (Ord k, Ord v) =>
            Observe () [(k,v)] (BST k v) where
  observe () = toList

main = quickSpec [
         monoTypeObserve (Proxy :: Proxy BSTII),
```

observe = toList

5. insert x y (insert z y w) = insert z y (insert x y w)

Conditional equations

```
predicate "/=" ((/=) :: Int -> Int -> Bool),
```

3. x /= y => find x (insert y z w) = find x w

...

7. z /= x => insert x y (insert z w x2) = insert z w (insert x y x2)

The Effect of a Bug

1. find x nil = Nothing
2. find x (insert x y z) = Just y
3. x /= y => find x (insert y z w) = find x w
4. find x (insert y z nil) = find y (insert x z nil)
5. insert x y (insert x z w) = insert x y w
6. insert x y (insert z y w) = insert z y (insert x y w)
7. z /= x => insert x y (insert z w x2) = insert z w (insert x y x2)

4. find x (insert x y nil) = Just y
5. insert x y (insert x z w) = insert x z w

Why do we get this ***specific instance*** *of (2) above?*

What???

Key takeaway

- The ***fundamental problem*** in property-based testing is ***coming up with properties*** that are:
  - inexpensive to write
  - effective as tests

- This lecture explains many useful ideas for tackling this problem