formal specification

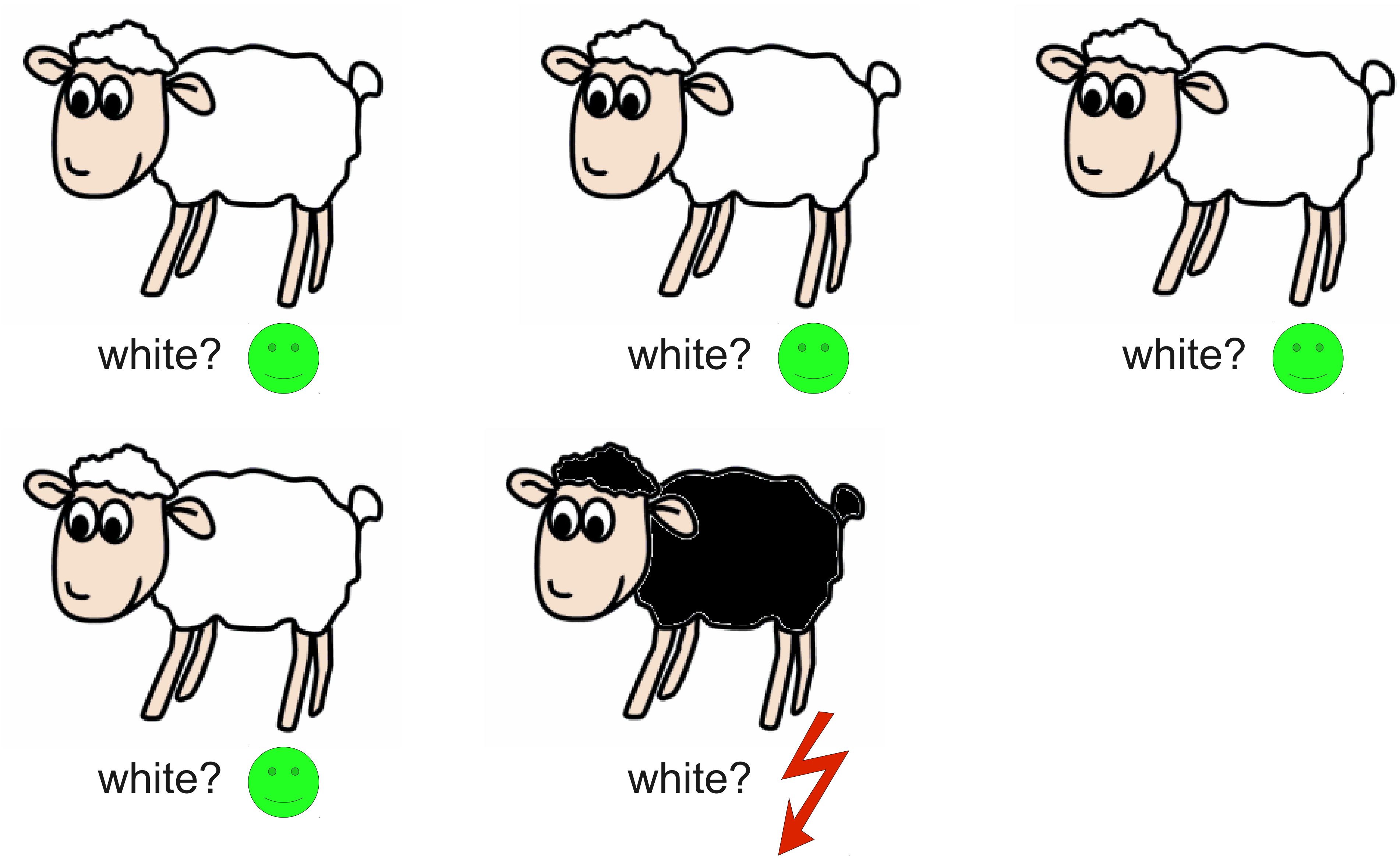
testing

Koen Lindström Claessen

APLAS 2009, Seoul

“Experimental Science”

All Sheep are White?



Testing

which inputs?

- Provide inputs to a program
- Notion of “correctness” or “what we expect”

what do we expect?

- Repeat until failing test case...

which failing case do we want to see?

- ...or until we get tired

what do we know then?

QuickCheck by Example

[Claessen, Hughes, 2000]

testing a compiler

- compile :: Program -> Code
- run :: Code -> Trace
- interpret :: Program -> Trace

- prop_Correct p =
  run (compile p) == interpret p

write a *property*
(in a simple logic)

“embedded language”
-- the programmer
already knows it

Running Tests

- Write a *generator* for random programs
  - arbitrary :: Gen Program

- quickCheck prop_Correct
  - automatically runs 100s of t random programs
  - reports the first failing case
  - Effective at finding bugs
  - Failing cases are large

```
if (x1 - (-3) >= x1 + x1 – x3)
  print (x3+(-2)+1+(-x1-0+x2))
  x2 = 1
else
  if (x1)
    x2 = 0+0+(-3)-(x2+x3)
    print (-2)
  fi
  x2 = x2
fi
print (x1-x1)
...
```

Shrinking

- Write a *shrinking function* for programs
  - shrink :: Program -> [Program]
  - structural shrinking
  - custom shrinking

```
if (0)
  print (1)
else
  print (0)
fi
```

- quickCheck prop_Correct
  - runs tests, finds failing case
  - automatically shrinks the failing case to a (local) minimum
  - effective combination!

Incomplete Specification

- simplify :: Program -> Program

- prop_SimplifyCorrect p =

  interpret p == interpret (simplify p)

- prop_SimplifySimplifies p =
  size p >= size (simplify p)

"design patterns"
-- common cases where
complete specifications
are not feasible

“Bug-Specification”

buggy...

- prop_SimplifyCorrect p =

  interpret p == interpret (simplify p)

- prop_SimplifyCorrect_noWhile p =
  noWhileLoop p ==>
  interpret p == interpret (simplify p)

implication

“bug specification”
-- allows progress
in testing process

QuickCheck Components

- **Generators**
  - structured data (trees, graphs, ...)
  - functions
  - grammar-based (automation)
- **Shrinking**
  - Some automation

- **Properties**
  - data structures
  - stateful APIs
  - concurrent APIs
  - search problems, optimization problems
  - ...

QuickCheck Success

- Conceived 2000 for Haskell
- Standard Haskell library
  - part of the community
- Other languages
  - Erlang (open source, commercial version)
  - C, C++, Java, C#, ML, F#, Mercury, Python, Perl, Google Go, ...
  - Use Haskell or Erlang to specify C, C++, Java, ...

Hypothesis

- **Formal specification**
  - forces people to think about their program
  - orthogonal
  - incremental (incomplete specifications)
- **Informal verification**
  - formal is too hard
  - no verification makes everything useless
  - sweet spot?

Experimental evidence?

Testing Logic

- forAll gen (\x -> p(x))

- b ==> p

- b

Testing Logic

- forAll gen (\x -> p(x))
  - universal quantification
- b ==> p
  - implication
- b
  - true or false

Testing Logic

- forAll gen (\x -> p(x))
  - universal quantification
  - generate an x, then test p(x)
- b ==> p
  - implication
  - if b, then test p; otherwise, discard p
- b
  - true or false
  - pass or fail

Where do properties come from?

- Write what we want, logically
- "Massage" the logical formula into a testing logic formula
  - forAll x . A(x) v B(x)

forAll gen (\x -> A(x) || B(x))

forAll gen (\x -> not A(x) ==> B(x))

forAll gen (\x -> not B(x) ==> A(x))

- Choices
  - Logically the same
  - Testing-logically different

Testing Logic

- Semantics
- Reasoning system
  - properties that are logically the same
  - testing-logical “improvement”
- What are the (expected) equalities?

```
forAll gen (\x -> A(x) ==> B(x))

vs.

forAll genA (\x -> B(x))
```

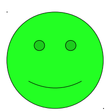
```
A ==> B

vs.

not B ==> not A
```

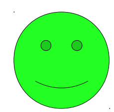
All Sheep are White?

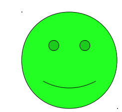
- forAll gen (\x -> Sheep(x) ==> White(x))
  - forAll sheep (\x -> White(x))

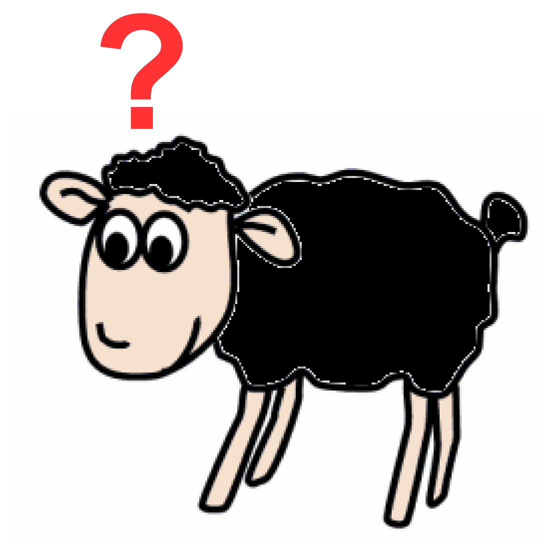
- forAll gen (\x -> not White(x) ==> not Sheep(x))
  - forAll nonWhite (\x -> not Sheep(x))

All non-White things are not Sheep?

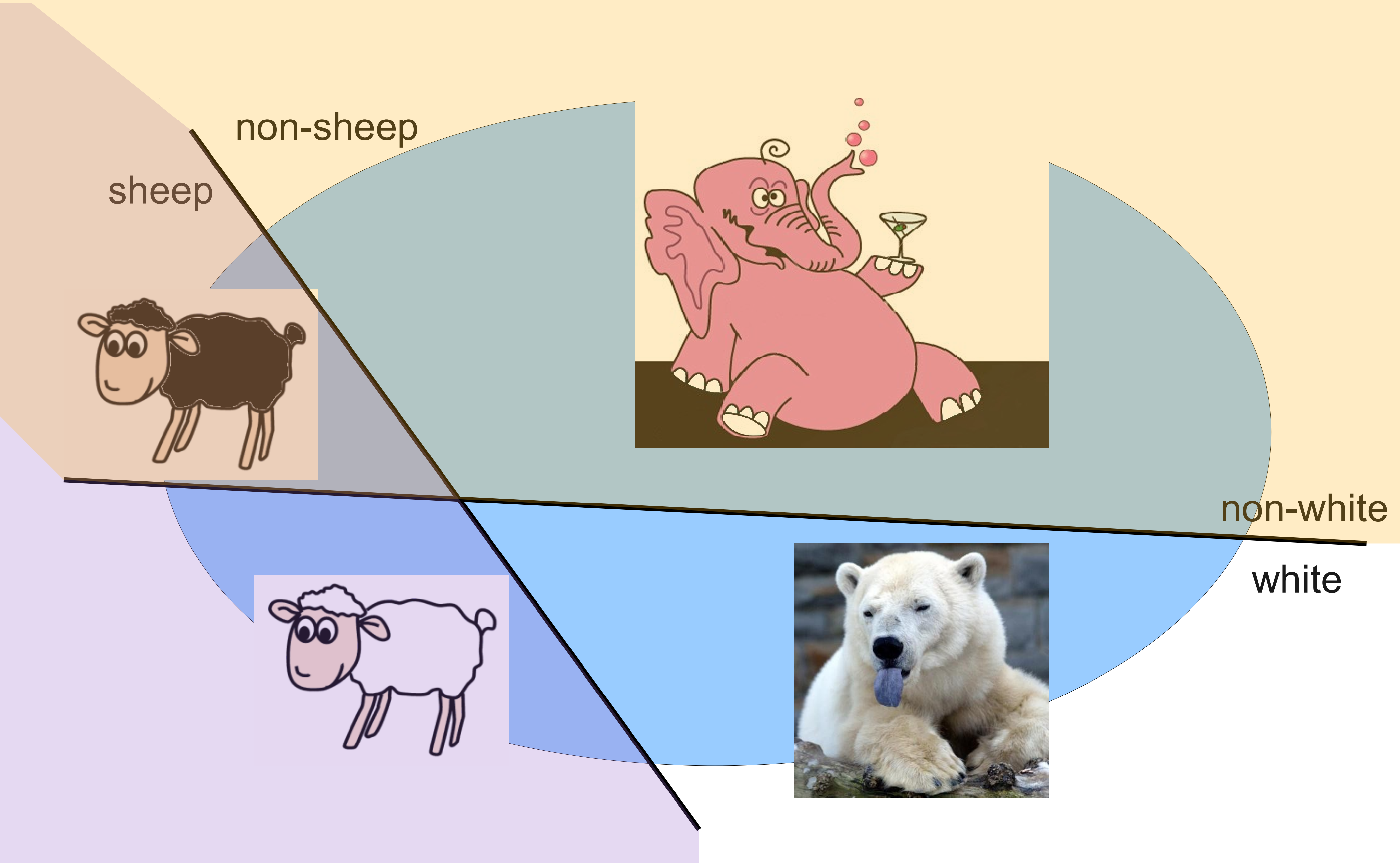
non-sheep? 

non-sheep? 

non-sheep? 



Which one is better?



Our Intuition was Right

- **sheep**: *non-white* vs. white
- **non-white**: *sheep* vs. non-sheep
- maximize amount of failing cases
- compare #non-white non-sheep against #white sheep
- A(x) ==> B(x)   v
- compare #not A,

```
prop_Complete p =
  hasSolution p ==> solve p == Yes
```

```
prop_Complete p =
  solve p == No ==> hasNoSolution p
```

Conjunction

- Adding an operator & to the testing logic
- Seems simple enough
  - Logic: A & B means conjunction
  - Testing A & B means first testing A and then testing B
- What properties does & enjoy?
  - A & A === A ? **NO**
  - A ==> (B & C) === (A==>B)&(A==>C) ? **YES**

Alternative Conjunction

A &' B :=
forAll x:Bool . if x then A else B

- Other operator &'
  - Logic: A &' B means conjunction
  - Testing A &' B means randomly choosing A or B to test
- What properties does & enjoy?
  - A &' A === A ? **YES**
  - A ==> (B &' C) === (A==>B)&'(A==>C) ? **YES**
  - A &' (B &' C) === (A &' B) &' C ? **NO**

Testing Logic

- We are defining operators
  - logical semantics
  - testing semantics
- Investigating properties
- Trying to come up with a simple set of primitives
- For *practical* use...
- ...explaining *practical* problems
- Example-driven

Hard Specifications

- Some problems are hard to specify...
- ...without reimplementing the programs
- Examples:
  - Search problems
    - path finding
    - SAT-solver
  - Optimization problems
    - shortest path
    - best solution
  - Problems with tedious specifications

“Inductive Testing”

- Specify program in terms of smaller instances of the program
  - prop_SatBaseTrue =
    sat [] == True
  - prop_SatBaseFalse =
    sat [[]] == False
  - prop_SatStep p x =
    sat p == (sat (subst x False p)
    || sat (subst x True p))

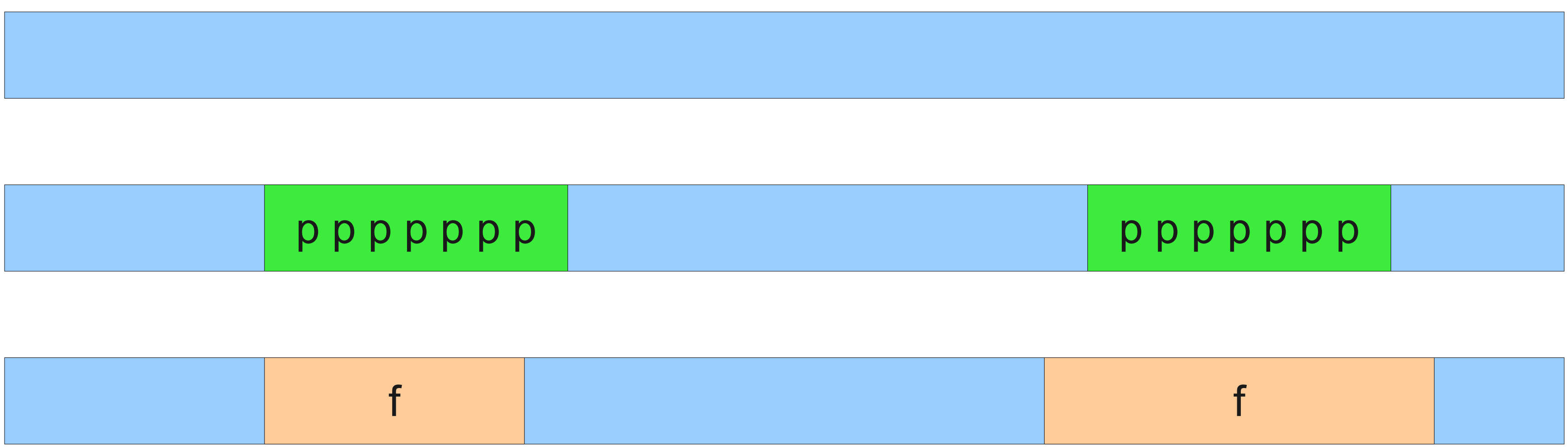
invoke the program more than once

like a naive implementation, but more efficient

Another Example

tedious specification

- anonymize :: String -> String
  anonymize s = ...
- modify :: (a -> Bool) -> ([a] -> [a]) -> [a] -> [a]
  modify p f xs = ...



Testing Modify

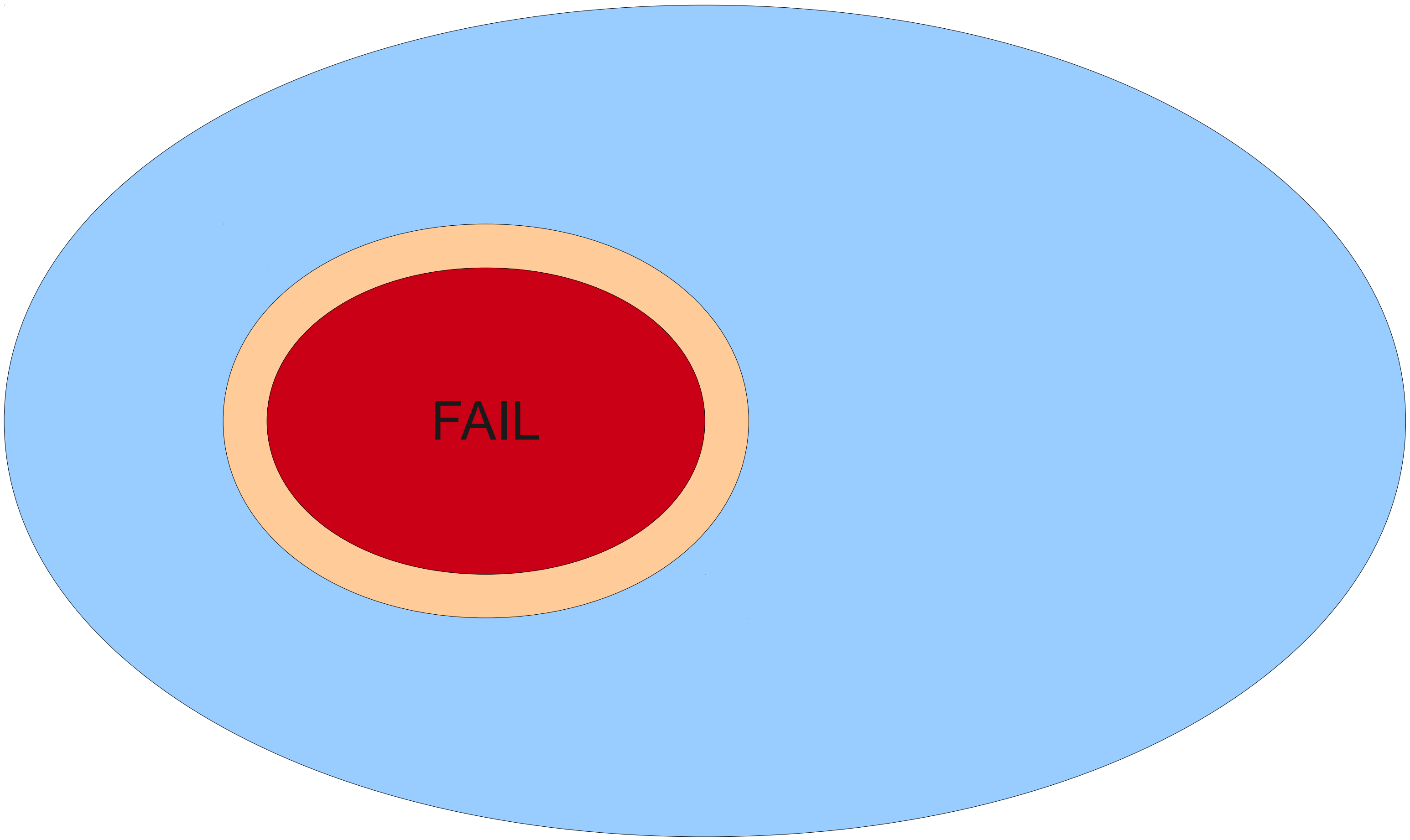
- prop_ModifyEmpty p f =
  modify p f [] == []

- prop_ModifyBase f xs =
  not (null xs) ==>
    modify (`elem` xs) f xs == f xs

- prop_ModifySep p f xs y zs =
  not (p y) ==>
    modify p f (xs ++ [y] ++ zs)
      == modify p f xs ++ [y] ++ modify p f zs

free choice
where to split

Distribution of Failing Cases



Coming Up with Specifications

- Hard to start “from scratch”
- QuickSpec:
    - Given a (compiled) module
    - API of the module
    - Test data generators
    - automatically generates an *equational specification*

Example

- **type** Map a b
- empty :: Map a b
- look :: a -> Map a b -> Maybe b
- insert :: a -> b -> Mab a b -> Map a b

- look x empty == Nothing
- look x (insert x a m) == Just a
- look x (insert y a empty) == look y (insert x a empty)
- insert x (insert x a m) == insert x a m
- insert x a (insert y a m) == insert y a (insert x a m)

What is it good for?

- Getting started with writing properties
- Exploring a module
  - Understanding
  - Improving
- Discovering strangeness
  - A law is not as general as you think, why?
  - Often, a good implementation/design has nice properties

How does it work?

- Generate a set of terms with variables
  - depth-based, ~20.000 terms
- Use testing to refine these into equivalence classes
  - when done, ~5.000 classes
  - each of which gives rise to several equations r == t
- Use *pruning* to get rid of superfluous equations
  - implied by other equations
  - hardest part!
  - ~5-30 equations

complement to QuickCheck

QuickSpec

- Joint work with Nick Smallbone, John Hughes
- Implemented for Haskell, Erlang
- Very fast (a few seconds)
- Used for
  - data structures
  - abstract data types
  - regular expression library
  - ...
- Currently working on
  - conditional equations
  - imperative programs
  - use cases

Testing Polymorphic Functions

- Suppose we are testing a property about a polymorphic function
  - reverse :: [a] -> [a]
  - filter :: (a -> Bool) -> [a] -> [a]
  - modify :: (a -> Bool) -> ([a] -> [a]) -> [a] -> [a]
- What type(s) should we pick to run the tests on?
- Standard QuickCheck practice suggests using Int
  - prop_Reverse (xs :: [Int]) =
    reverse (reverse xs) == xs

Example: reverse

- reverse :: [a] -> [a]
- the only source of a's in the result are the elements in the argument list
- we could symbolically represent these by their indices
- prop_Reverse n =
  ```
  prop_Reverse n =
     let xs = [1..n] in
       reverse (reverse xs) == xs
  ```
- It is enough to vary the *length* of the lists!

Example: filter and map

- prop_MapFilter p f xs =
  filter p (map f xs) == map f (filter p xs)
- Here:
  - p :: a -> Bool
  - f :: a -> a
  - xs :: [a]
- An a can either come (1) from the list xs, (2) from applying f

filter and map

- **data** T = X Int -- from the list
  | F T -- applying F

```
data T = X Int    -- from the list
       | F T      -- applying F
```

- prop_MapFilter p n =

```
prop_MapFilter p n =
   let xs = [X 1 .. X n]
       f  = F
    in filter p (map f xs) == map f (filter p xs)
```

- Varying the length n, and the predicate p is enough!

General Idea

- Given a property, rewrite it into the following form:
  - prop :: (F a -> a) -> (G a -> X) -> H a
  - for polynomial functors F, G, and H
- Then, the monotype T is computed as the least fixpoint of F
- The argument of type F a -> a (now F T -> T) is fixed to the initial algebra of F
- Based on parametricity

PolyTest

- Joint work with Jean-Philippe Bernardy, Patrik Jansson
- Also for arguments with properties
- Still investigating boundaries
- Paper at ESOP 2010

Summary

- QuickCheck
- Testing Logic
- Inductive Testing
- QuickSpec – generating specifications
- Testing polymorphic functions