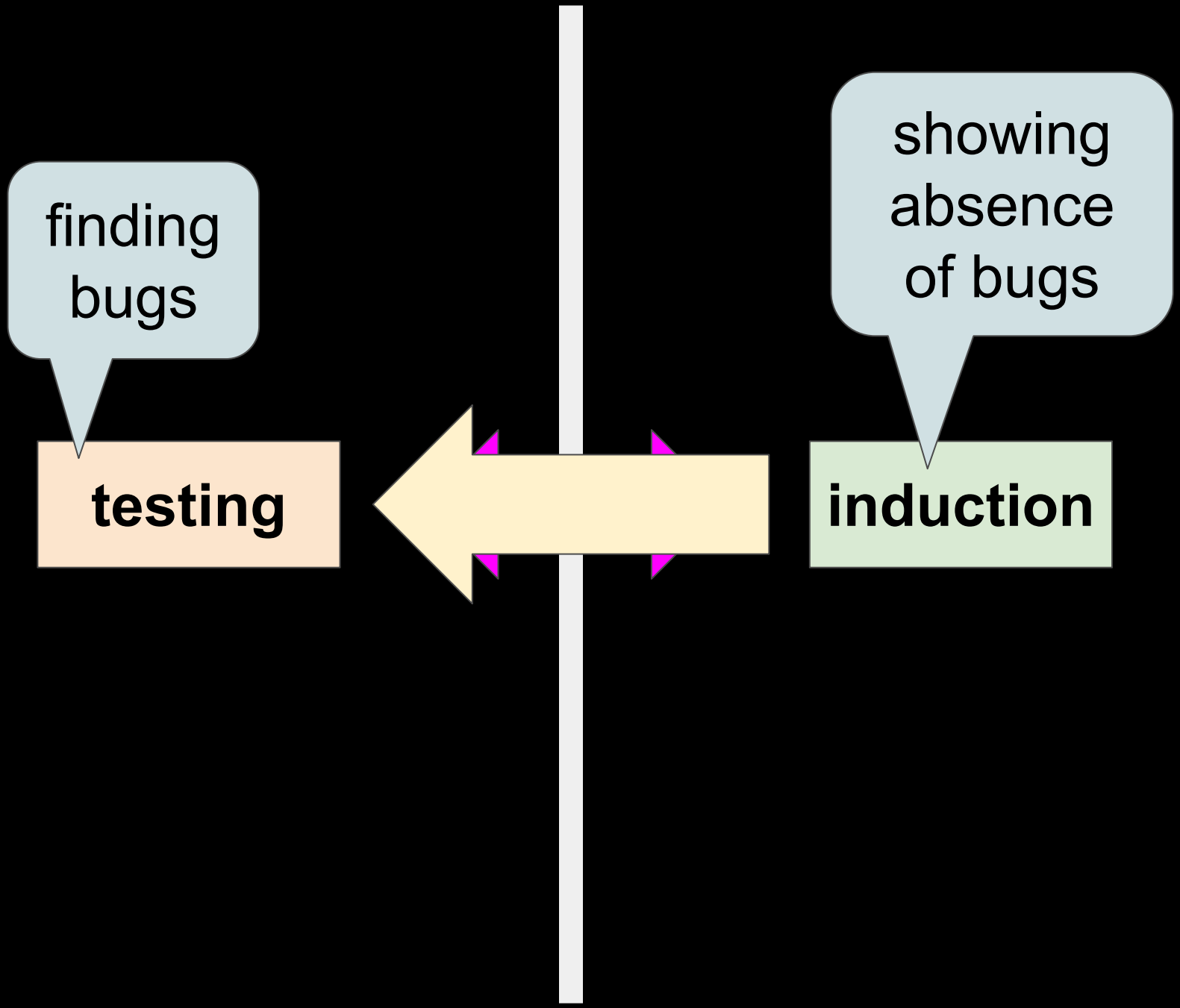

type
families

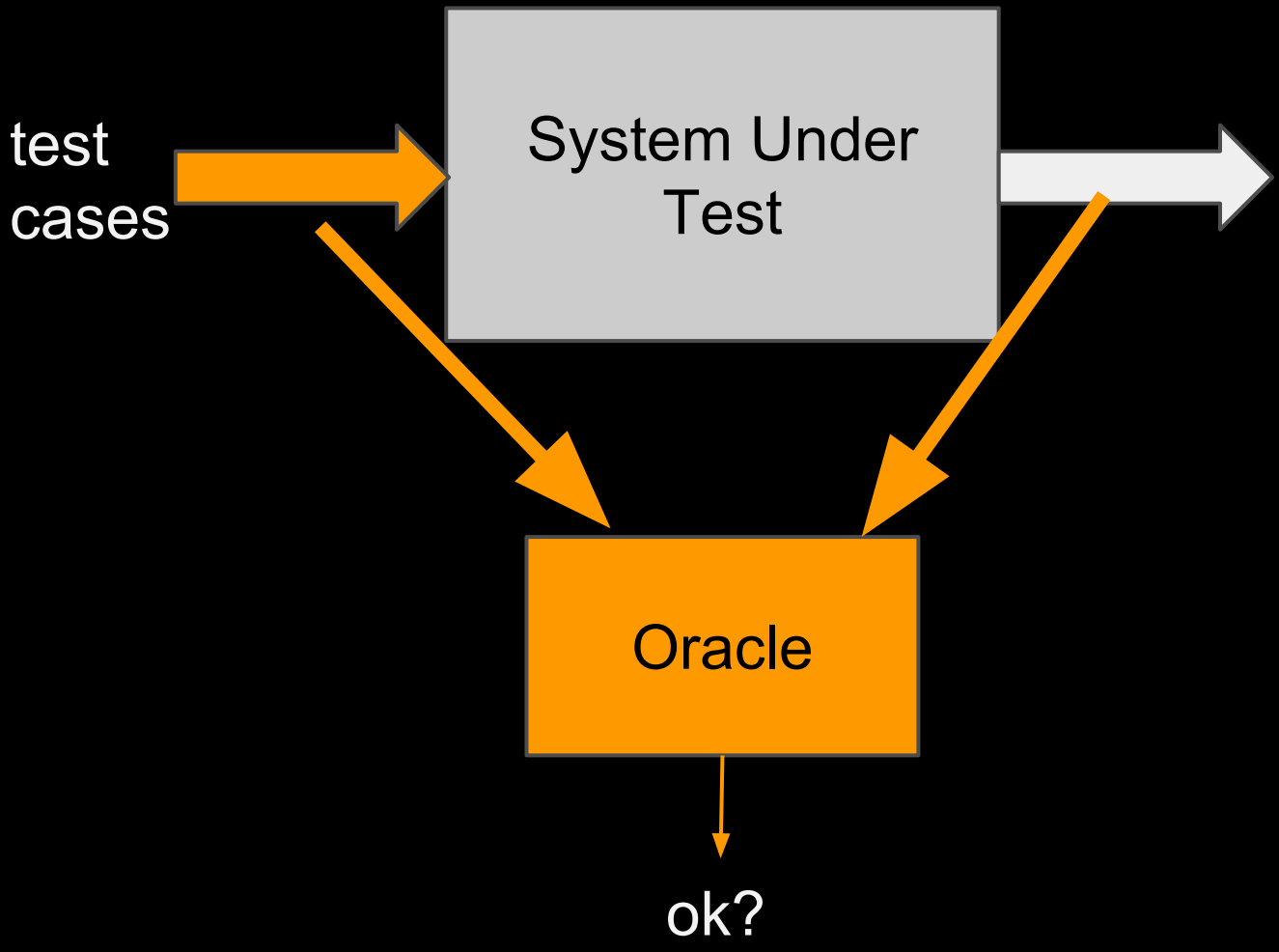
Testing and Induction

Koen Claessen
Chalmers University of Technology
Gothenburg, Sweden


finding bugs
showing absence of bugs
testing
induction

Proof-based Testing

contrapositive testing,
inductive testing,
and *co-inductive testing*

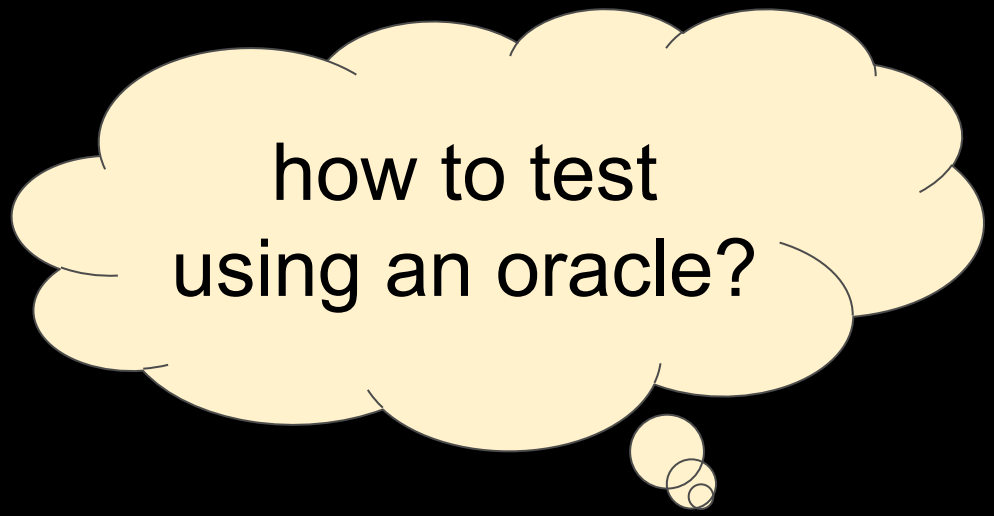

test
cases
System Under
Test
Oracle
ok?

Oracle

- Simple
  - Simpler than the implementation
  - (environment)
- Practically runnable
  - May need to run many tests
- Oracle should be “complete”
  - For any faulty implementation, there should exist inputs that trigger the oracle to say “no”

Shortest Path Algorithms

```
type Map
type Point
type Path
```




```
shortest :: Map -> Point -> Point -> Maybe Path
```

```
( solve :: Problem -> Maybe Solution )
```

Problem

- The oracle needs to know what the shortest path is

- We can be **simple**, but it is **too slow**
  - Not practical when testing
  - (Non-termination!)

- We can be **fast**, but it is **too complex**
  - We may not trust our test results

Property-based Testing

(in the style of QuickCheck)

Sound - If an answer is produced, it should be an actual solution

Complete - If no answer is produced, there indeed was no actual solution

Optimal - If an answer is produced, there is no actual solution that is better

Sound - If an answer is produced, it should be an actual solution

(easy to test)

Complete - If no answer is produced, there indeed was no actual solution

logically equivalent

Complete' - If there is a solution, some answer will be produced

testable

ForAll x . A(x) ==> B(x)

ForAll x in “A”. B(x)

```
ForAll mp,a,b .
    hasPath mp a b ==>
        isJust (shortest mp a b)
```

```
ForAll (mp,a,b) in hasPathMap .
        isJust (shortest mp a b)
```

Optimal - If an answer is produced, there is no actual solution that is better

logically equivalent

Optimal' - If there is a solution, then no worse answer will be produced

testable ?

Contrapositive testing

- Change your viewpoint
  - From: Stimuli / System Under Test / Oracle
  - To: Logical implication
- And take the contrapositive view to get new inspiration
- Sometimes, you have a choice! (How to make it?)

Contrapositive Testing

Shortest Distance Algorithms

```
type Map
type Point
data Distance = Inf | Fin Int


distance :: Map -> Point -> Point -> Distance
```

Sound - If an answer is produced, it should be an actual solution

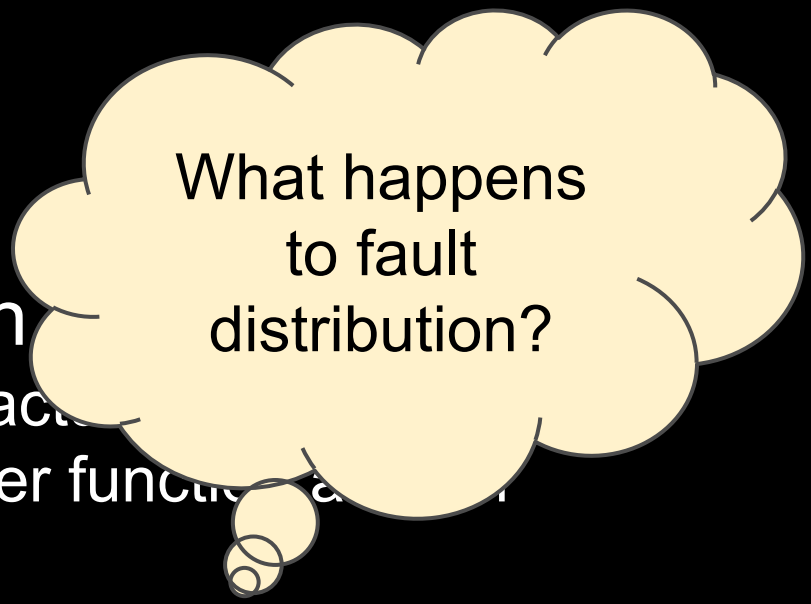
Complete - If no answer is produced, there indeed was no actual solution

Optimal - If an answer is produced, there is no actual solution that is better

```
ForAll mp,a,b .
  distance mp a b ==
    minimum [ distance mp a' b + d
            | (a',d) <- neighbors mp a
            ]
```

```
ForAll mp,a,a .
  distance mp a a == Fin 0
```

Inductive Testing



- Correctness: by induction
  - soundness: induction over act
  - completeness: induction over functi a
- Induction principle
  - choose this for enabling testing
  - independent of implementation (unlike proving)
- Induction vs. recursion in implementation
  - too slow to use directly (even non-terminating)
  - Plotkin induction

Testing SAT-solvers

Testing SAT-solvers

- If model and proof are generated
  - Direct soundness
  - Direct completeness
- If only model is generated when found
  - Direct soundness
  - Contrapositive testing for completeness
- If only yes/no answer
  - Inductive testing
  - Base case: no variables
  - Step case: branch on a variable

Testing Sorting

Testing sorting functions

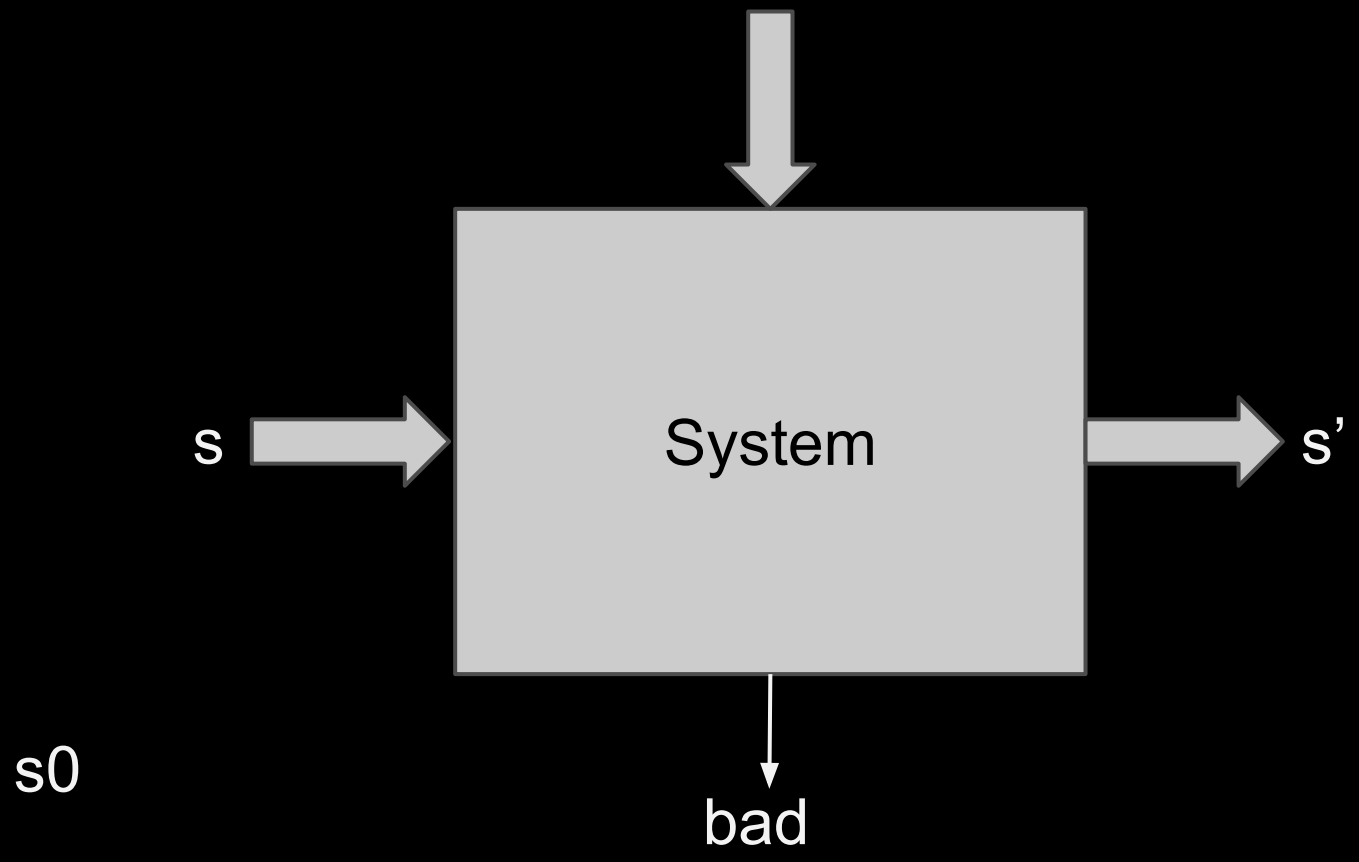
- Write down the simplest sorting function you can think of
  - *You trust this code*
- Show that the function you want to test has the same behavior
  - *How?*

Testing FFT implementations

Testing FFT

- Base case
  - vectors [0,..,0,1,0,..,0]
- Step cases
  - a * fft v = fft (a*v)
  - fft v + fft w = fft (v + w)
- Using exact arithmetic
  - Implementation is still fast
  - Specification is extremely slow

Testing Model Checkers for Safety Properties



s0

False: The system is **not safe**; often produces a **trace**

```
check :: State -> Circuit -> Bool
```

True: The system is **safe**; (produces nothing)

```
step :: State -> System -> Input -> (Bool, State)
```

```
safe(s, C) =
    ForAll inp .
        let (ok, s') = step s C inp in
            ok && safe(s', C)
```

greatest fixpoint

cannot simply compute

```
ForAll s, C .
    check s C ==>
        safe(s, C)
```

cannot simply compute

```
ForAll s, C .
    check(s, C) ==>
        ForAll inp .
            let (ok, s') = step(s, C, inp) in
                ok && check(s', C)
```

- Correctness
  - Safety is defined as *greatest fixpoint*
  - Most natural is to use *coinduction*

$$\frac{a \leq F(a)}{a \leq \mathrm{gfp}\ x\ .\ F(x)}$$

  - Can also use induction (over the length of the shortest missed countertrace)

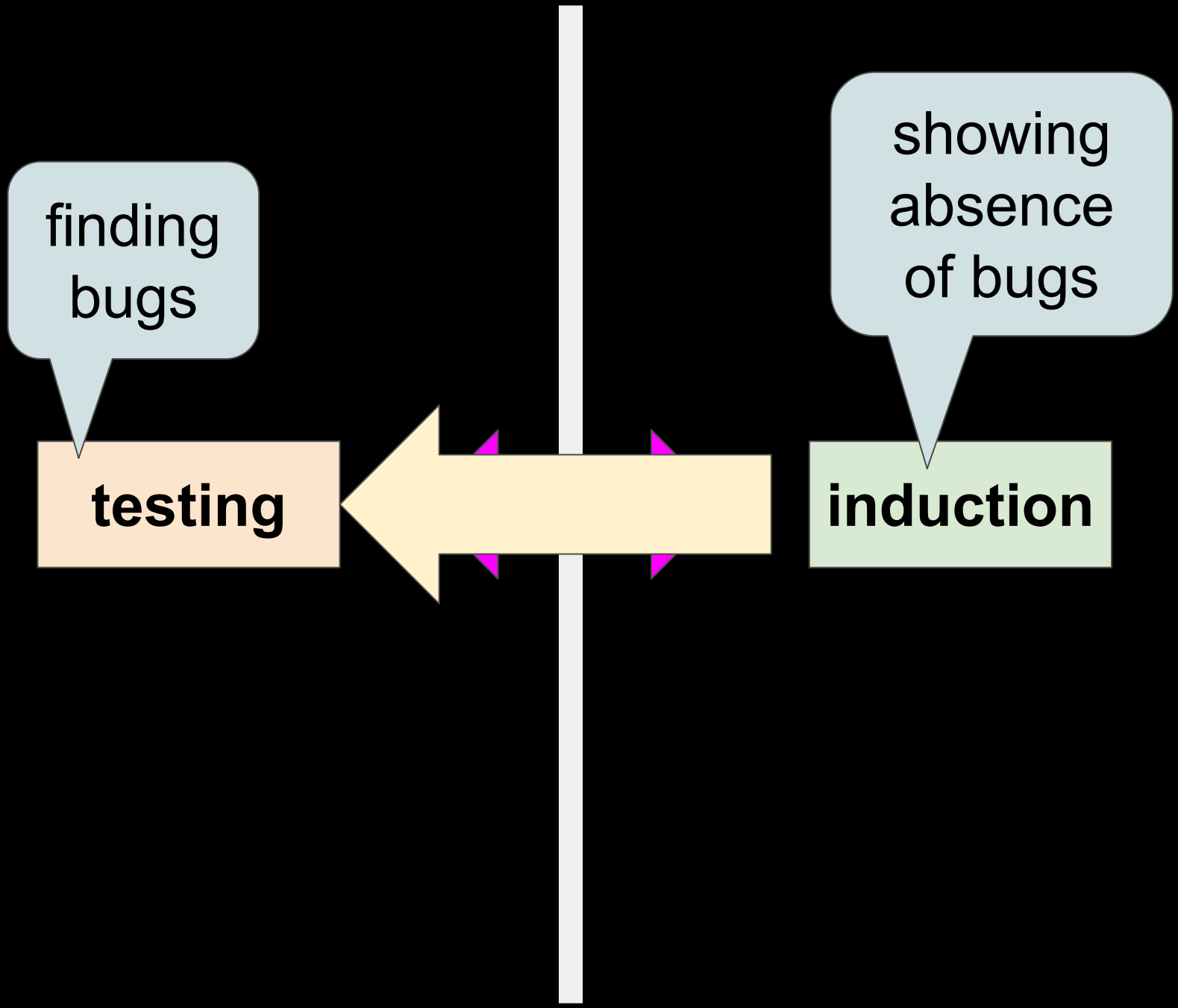
- Efficiency
  - Model checker is called twice for each test

Inductive Testing

- Break away from the stimuli / system under test / oracle view
- Look at the logical meaning of the property
- Use proof techniques to “break up” into smaller properties
  - Together, they imply the original property
  - They may be easier to test
  - The system may be run several times
- What happens to the distribution of faulty test cases?

More...

- More examples
  - Testing compilers / interpreters
  - Theorem provers for decidable logics
  - Theorem provers for semi-decidable logics
  - Model checkers for liveness properties
  - Unification algorithm
  - Distributed systems
  - …
- Developing “testing logic”
  - Logical equivalence
  - Testing non-equivalence
  - Cost of testing
  - Predict which testing ways are most effective


finding bugs
showing absence of bugs
testing
induction

Automating Induction

```
rev :: [a] -> [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

```
rev (rev xs) =?= xs
```

functions are *total* and *terminating*

```
rev :: [a] -> [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

BASE: `rev (rev []) = []`

STEP:
```
rev (rev as) = as  ==>
rev (rev (a:as)) = a:as
```

```
rev :: [a] -> [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

```
ASSUME: rev (rev as) = as

 PROVE: rev (rev (a:as)) = a:as

   <=>  rev (rev as ++ [a]) = a:as

        <stuck>
```

```
rev :: [a] -> [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

```
rev (xs ++ ys) =?= rev ys ++ rev xs
```

```
BASE: rev ([] ++ ys) = rev ys ++ rev []

 <=>  rev ys = rev ys ++ []

 <=>  rev ys = rev ys
```

```
rev :: [a] -> [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

```
rev (xs ++ ys) =?= rev ys ++ rev xs
```

ASSUME: `rev (as ++ ys) = rev ys ++ rev as`

PROVE: `rev ((a:as)++ys) = rev ys ++ rev (a:as)`

<=> `rev (a:(as++ys)) = rev ys ++ (rev as++[a])`

<=> `rev (as++ys)++[a] = rev ys++(rev as++[a])`

<=> `(rev ys++rev as)++[a] = rev ys++(rev as++[a])`

<=> `rev ys++(rev as++[a]) = rev ys++(rev as++[a])`

```
rev :: [a] -> [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

```
ASSUME: rev (rev as) = as

 PROVE: rev (rev (a:as)) = a:as

   <=>  rev (rev as ++ [a]) = a:as

   <=>  rev [a] ++ rev (rev as) = a:as

   <=>  a : rev (rev as) = a : as

   <=>  a : as = a : as
```

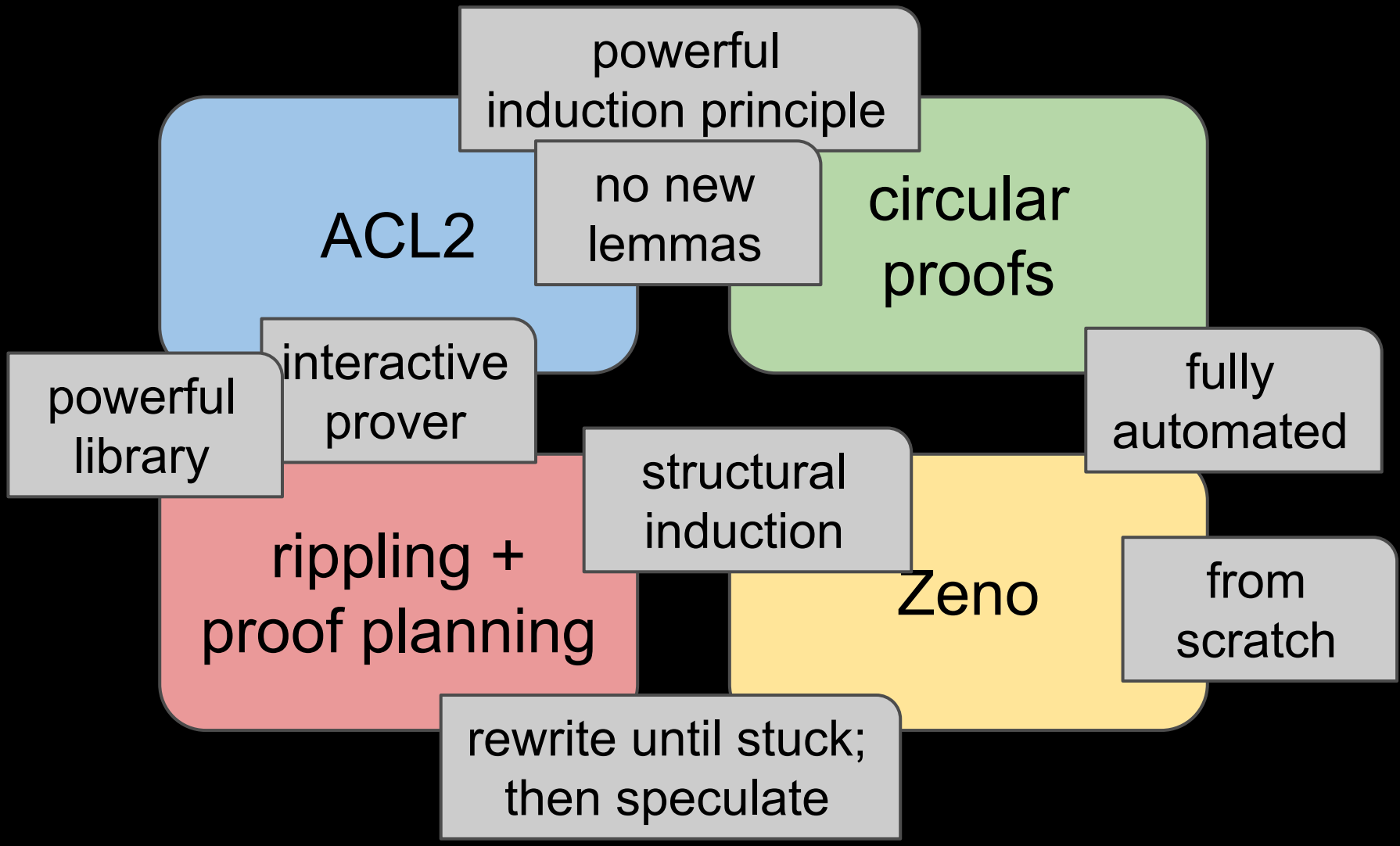
```
rev (xs ++ ys) =?= rev ys ++ rev xs
```

```
xs ++ [] =?= xs
```

```
xs ++ (ys ++ zs) =?= (xs ++ ys) ++ zs
```

Activities

- Choose which variable(s) to do induction on ✔ ☐
- Choose what kind of induction ✔ ☐
- Equational reasoning ✔ ☐
- Inventing new lemmas ☐ ✖ ☐

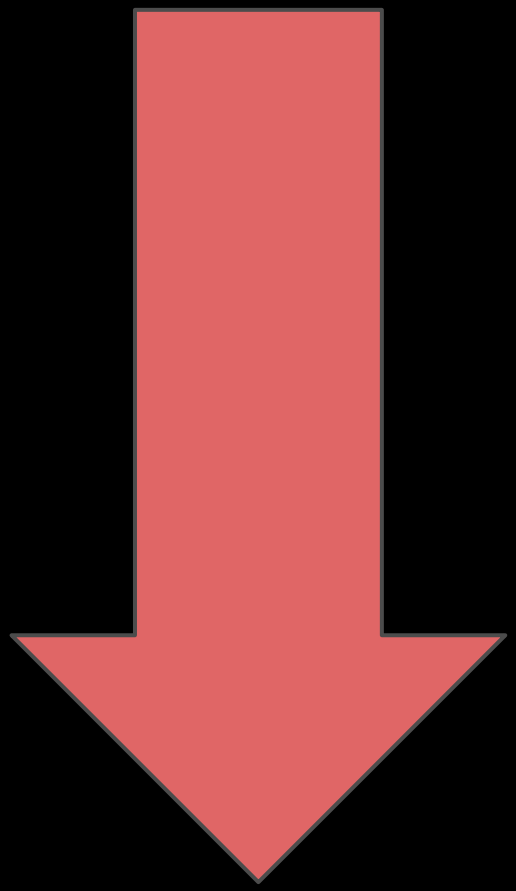

powerful induction principle
ACL2
no new lemmas
circular proofs
interactive prover
powerful library
fully automated
structural induction
rippling + proof planning
Zeno
from scratch
rewrite until stuck; then speculate

rewrite until stuck;
then speculate

```
rev :: [a] -> [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

```
ASSUME: rev (rev as) = as

 PROVE: rev (rev (a:as)) = a:as

   <=>  rev (rev as ++ [a]) = a:as

   <=>  rev (rev as ++ [a]) = a:rev(rev as)
```

```
rev (xs ++ [a]) =?= a:rev xs
```



existing lemma speculation works *top-down...*

...starting from what you **want to prove**, arrive at something that is **needed**...

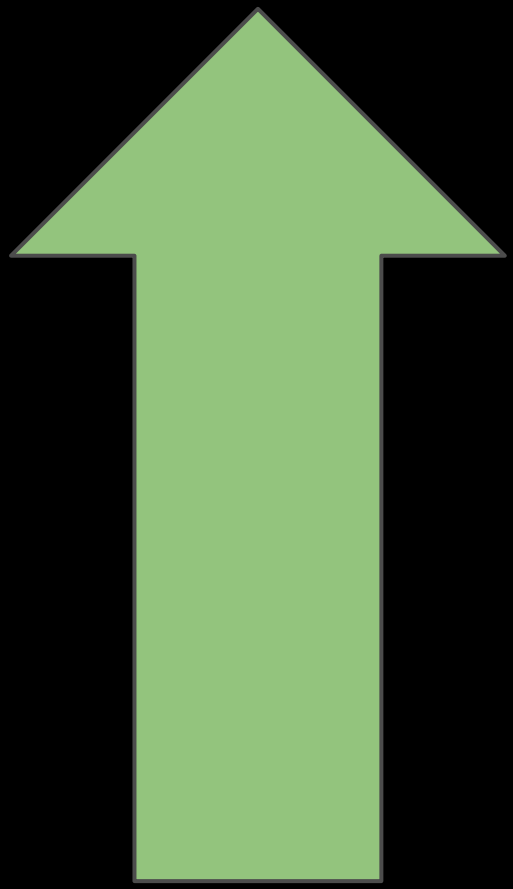
...hopefully ending up with things that are provable.

...hopefully helping in the proof of the properties!

how?

...looking at your program, it **calculates** and **summarizes** all equational lemmas that hold...

HipSpec works *bottom-up...*



QuickSpec

```haskell
import QuickSpec

main = quickSpec [
  con "0" (0 :: Int),
  con "1" (1 :: Int),
  con "+" ((+) :: Int -> Int -> Int),
  con "*" ((*) :: Int -> Int -> Int) ]
```

```
== Laws ==
  1. x * y = y * x
  2. x + y = y + x
  3. x * 0 = 0
  4. x * 1 = x
  5. x + 0 = x
  6. (x * y) * z = x * (y * z)
  7. x * (y + y) = y * (x + x)
  8. (x + y) + z = x + (y + z)
  9. x * (y + 1) = x + (x * y)
 10. x * (y + (y + y)) = y * (x + (x + x))
 11. (x * y) + (x * z) = x * (y + z)
```

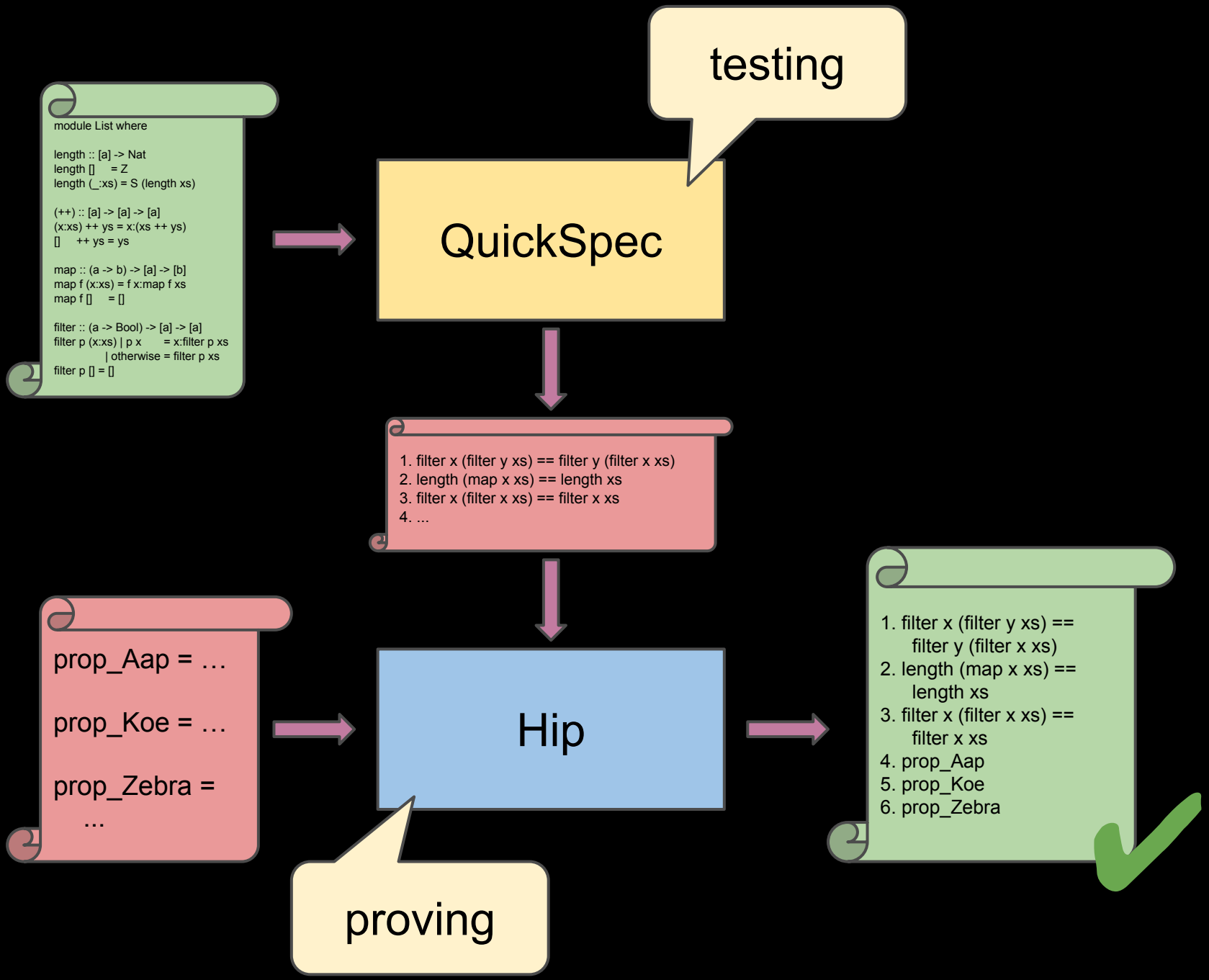
only tested

```
import QuickSpec

main = quickSpec [
  con "reverse" (reverse :: [A] -> [A]),
  con "++" ((++) :: [A] -> [A] -> [A]),
  con "[]" ([] :: [A]),
  con "map" (map :: (A -> B) -> [A] -> [B]),
  con "length" (length :: [A] -> Int),
  con "concat" (concat :: [[A]] -> [A]),

  -- Add some numeric functions to get more laws about length.
  arith (Proxy :: Proxy Int) ]
```

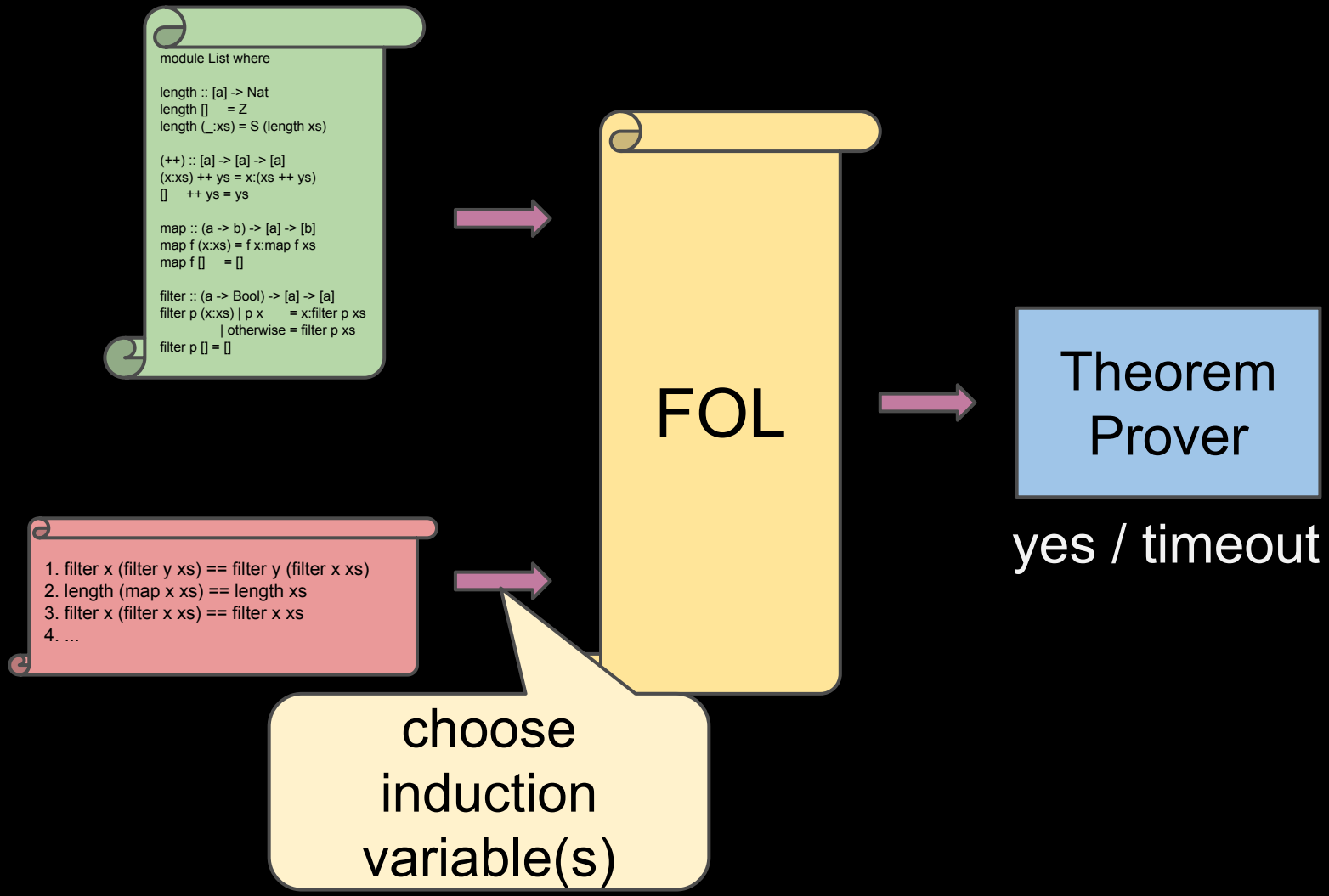
```
== Laws ==
  1. concat [] = []
  2. length [] = 0
  3. reverse [] = []
  4. xs ++ [] = xs
  5. [] ++ xs = xs
  6. length (reverse xs) = length xs
  7. map f [] = []
  8. reverse (reverse xs) = xs
  9. length (xs ++ ys) = length (ys ++ xs)
 10. length (map f xs) = length xs
 11. reverse (map f xs) = map f (reverse xs)
 12. length (concat (reverse xss)) = length (concat xss)
 13. concat (map reverse xss) = reverse (concat (reverse xss))
 14. map ((++) []) xss = xss
 15. (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
 16. length xs + length ys = length (xs ++ ys)
 17. concat xss ++ concat yss = concat (xss ++ yss)
 18. reverse xs ++ reverse ys = reverse (ys ++ xs)
 19. concat (map (map f) xss) = map f (concat xss)
 20. map reverse (map reverse xss) = xss
 21. map (map f) (map reverse xss) = map reverse (map (map f) xss)
 22. map f xs ++ map f ys = map f (xs ++ ys)
 23. length (concat (map f (concat (reverse xss)))) = length (concat (m
 24. length (concat (map ((++) (reverse xs)) xss)) = length (concat (ma
 25. map ((++) xs) (map ((++) ys) xss) = map ((++) (xs ++ ys)) xss
```

```
== Laws ==
  1. concat [] = []
  2. length [] = 0
  3. reverse [] = []
  4. xs ++ [] = xs
  5. [] ++ xs = xs
  6. length (reverse xs) = length xs
  7. map f [] = []
  8. reverse (reverse xs) = xs
  9. length (xs ++ ys) = length (ys ++ xs)
 10. length (map f xs) = length xs
 11. reverse (map f xs) = map f (reverse xs)
 12. length (concat (reverse xss)) = length (
 13. concat (map reverse xss) = reverse (conca
 14. map ((++) []) xss = xss
 15. (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
 16. length xs + length ys = length (xs ++ ys
```

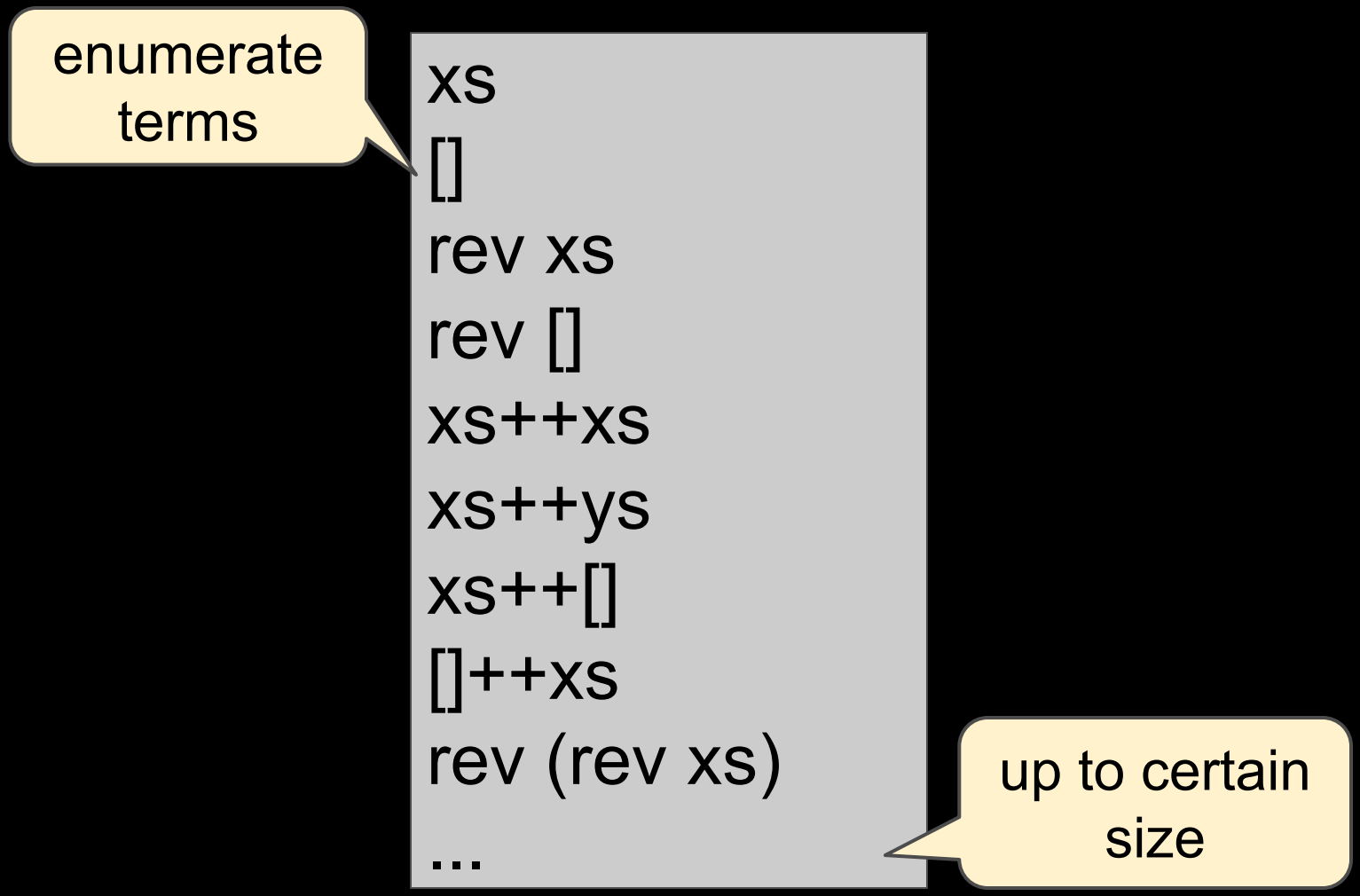
```
 9. length (xs ++ ys) = length (ys ++ xs)
10. length (map f xs) = length xs
11. reverse (map f xs) = map f (reverse xs)
12. length (concat (reverse xss)) = length (c
13. concat (map reverse xss) = reverse (conca
14. map ((++) []) xss = xss
15. (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
16. length xs + length ys = length (xs ++ ys)
17. concat xss ++ concat yss = concat (xss ++
18. reverse xs ++ reverse ys = reverse (ys ++
19. concat (map (map f) xss) = map f (concat
20. map reverse (map reverse xss) = xss
21. map (map f) (map reverse xss) = map rever
22. map f xs ++ map f ys = map f (xs ++ ys)
23. length (concat (map f (concat (reverse xs
24. length (concat (map ((++) (reverse xs)) x
25. map ((++) xs) (map ((++) ys) xss) = map (
```


testing
module List where
length :: [a] -> Nat
length [] = Z
length (_:xs) = S (length xs)
(++) :: [a] -> [a] -> [a]
(x:xs) ++ ys = x:(xs ++ ys)
[] ++ ys = ys
map :: (a -> b) -> [a] -> [b]
map f (x:xs) = f x:map f xs
map f [] = []
filter :: (a -> Bool) -> [a] -> [a]
filter p (x:xs) | p x = x:filter p xs
| otherwise = filter p xs
filter p [] = []
QuickSpec
1. filter x (filter y xs) == filter y (filter x xs)
2. length (map x xs) == length xs
3. filter x (filter x xs) == filter x xs
4. ...
prop_Aap = …
prop_Koe = …
prop_Zebra =
...
Hip
proving
1. filter x (filter y xs) ==
filter y (filter x xs)
2. length (map x xs) ==
length xs
3. filter x (filter x xs) ==
filter x xs
4. prop_Aap
5. prop_Koe
6. prop_Zebra

Hip (Haskell Inductive Prover)



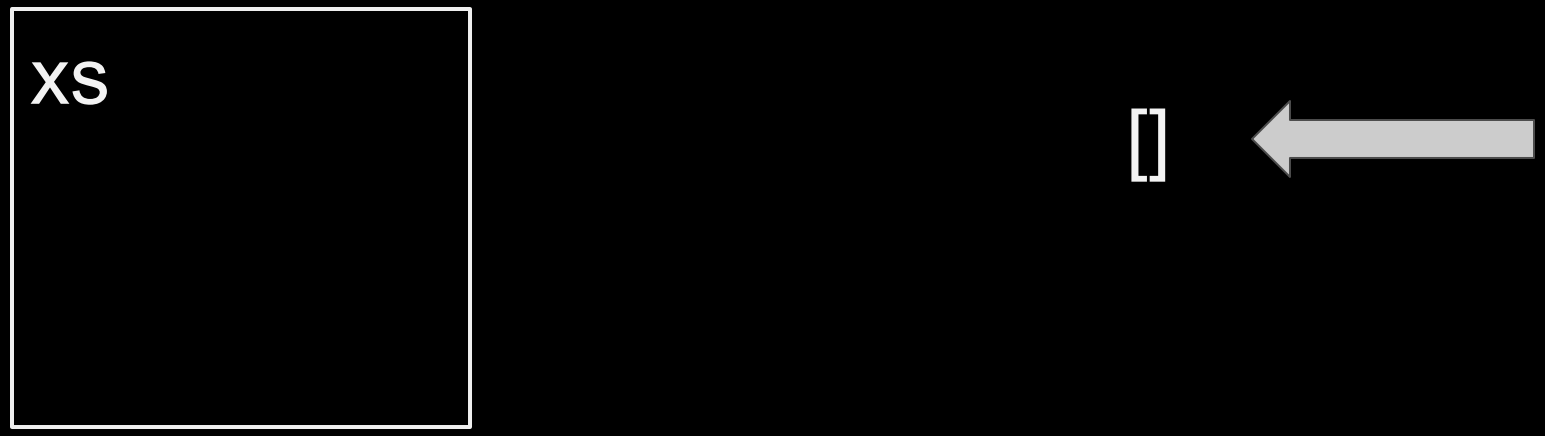
QuickSpec - how does it work?



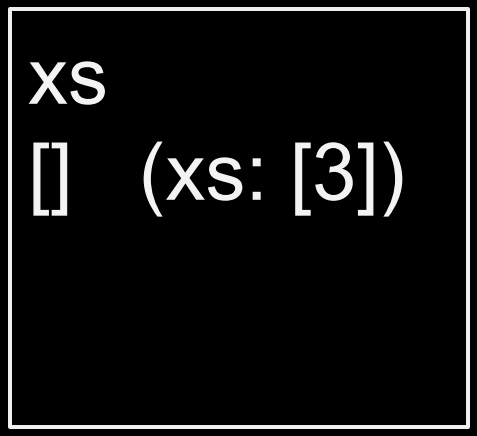
QuickSpec - algorithm

xs 

QuickSpec - algorithm



QuickSpec - algorithm



rev xs ⟵

QuickSpec - algorithm

xs
[] (xs: [3])
rev xs (xs: [4,1])

[] = rev []

QuickSpec - algorithm

xs
[] (xs: [3])
rev xs (xs: [4,1])

xs = xs++[]

QuickSpec - filtering

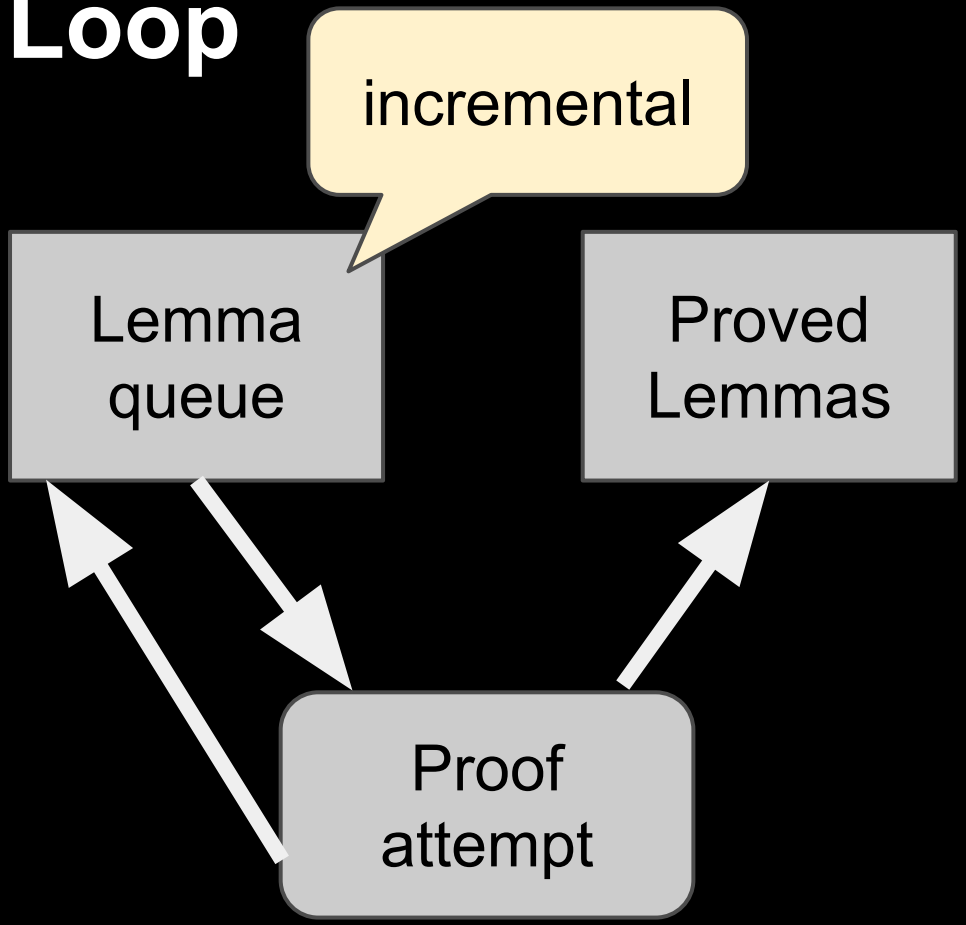
```
xs
[]          (xs: [3])
rev xs    (xs: [4,1])
...
```

quickly check if already provable

before testing

rev xs = rev (xs++[])

Proving Loop



Showcase

```
rev (drop i xs) = take (len xs - i) (rev xs)
rev (take i xs) = drop (len xs - i) (rev xs)
```

No	Conjecture
1	`len (drop x xs) = len xs-x`
2	`len xs = len (rev xs)`
3	`xs = take x xs++drop x xs`
4	`rev (ys++xs) = rev xs++rev ys`
5	`xs = take (len xs) (xs++ys)`

Extensions

- conditional properties
  - simulate with functions
  - simulate with types
  - small set of predetermined conditions

User Interaction

- add more functions
  - more properties
  - simulate conditional properties
- limit the number of functions to study
  - scalability

Semantics

- everything is total, and terminates
  - simplest
- everything terminates, not necessarily total
  - Isabelle/HOL semantics
  - special change to QuickSpec required
- no totality assumptions whatsoever
  - full domain theory with _|_
  - fixpoint induction

Summing Up

- Proof techniques can help testing
  - contrapositive testing
  - inductive testing
- Testing can help proving
  - QuickSpec
  - Hip