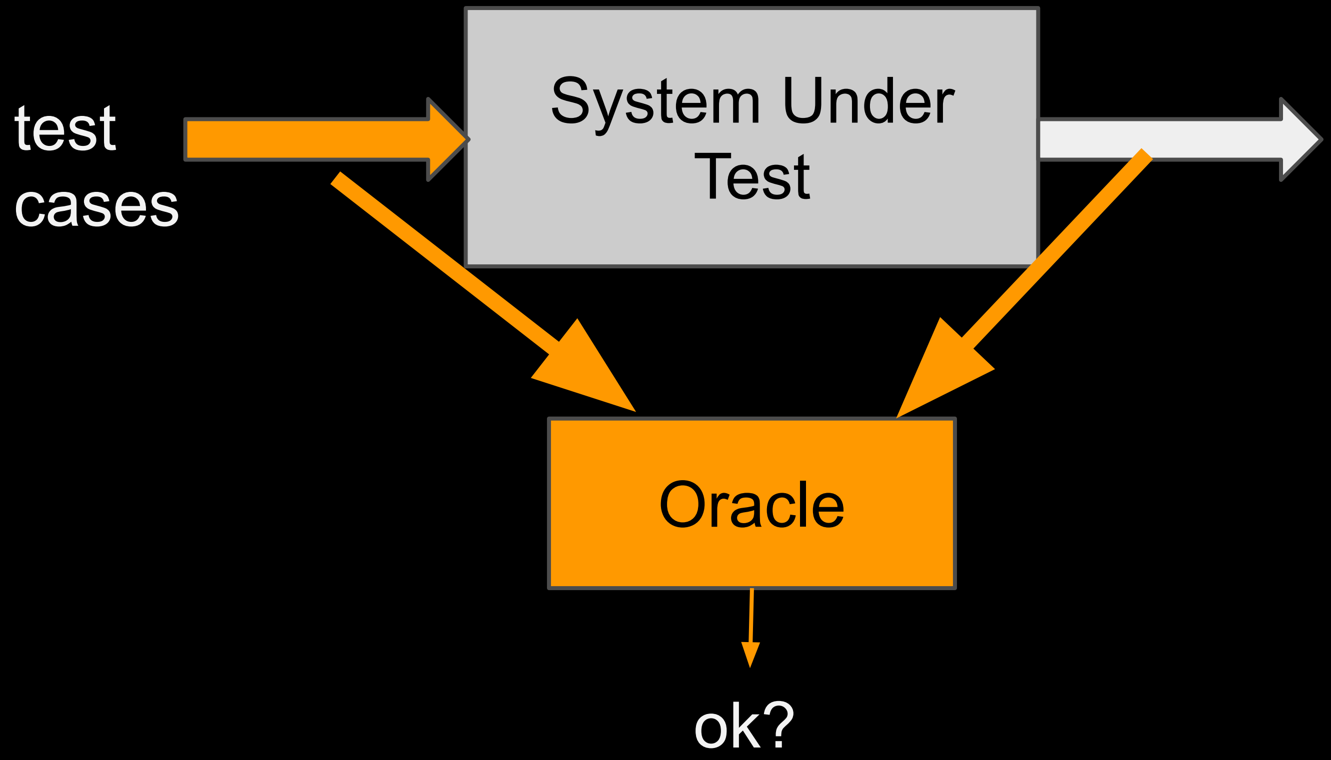
# Inductive Testing

## with applications to compiler/interpreter testing

Koen Claessen
Chalmers University of Technology
and Epic Games


test
cases
System Under
Test
Oracle
ok?

# Test Oracle: 3 Principles

- 1. Simple
  - Simpler than the implementation
- 2. Efficiently runnable
  - May need to run many tests
- 3. “Completeness”
  - For any faulty implementation, there should exist inputs that trigger the oracle to say “no”

# Example:
# 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

(a la 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
(simpler)

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

hard to test
(oracle copies implementation)

**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))
```

produce a map, two points, and a path between those points

```
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!

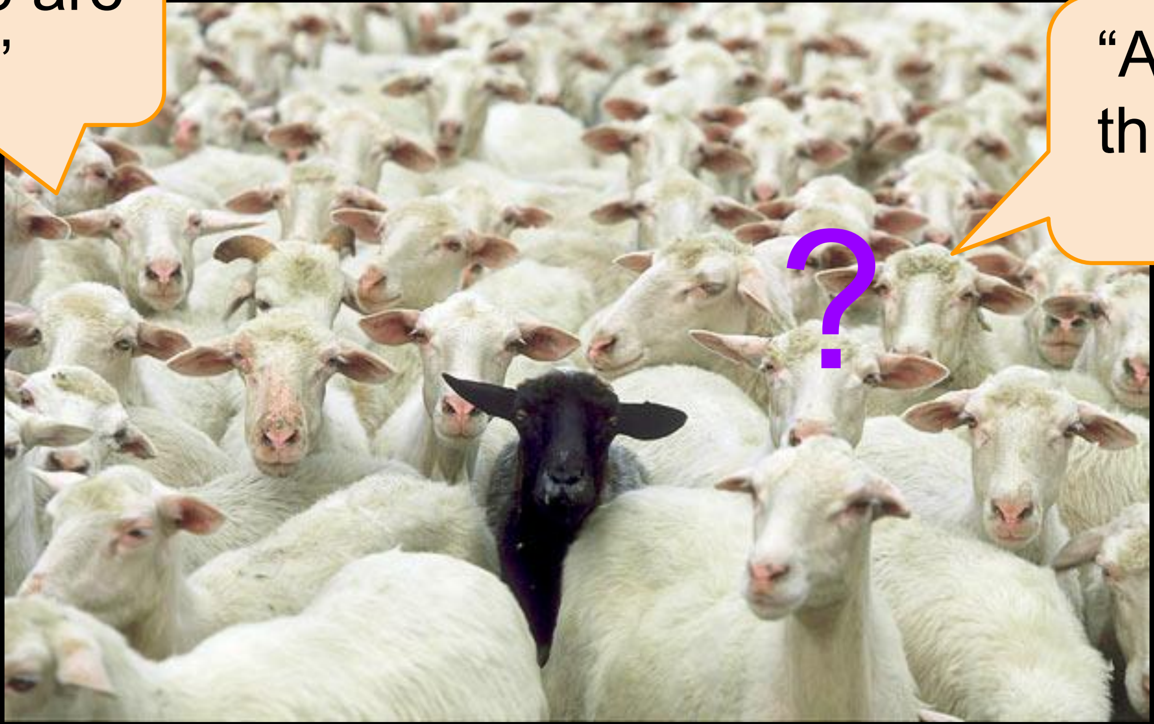
```
ForAll mp,a,b in hasPathMap .
  let Just path = shortest (mp, a, b) in
    length path <= length hasPathMap
```

make sure found path is not longer than the one we know about

# Contrapositive testing

- Change your viewpoint
  - From: Stimuli / System Under Test / Oracle
  - To: Proofs / Logical implication
- And take the **contrapositive** view to get new inspiration

# Contrapositive Testing



# Shortest Distance Algorithms

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

distance : (Map, Point, Point) -> Distance
```

hard!

**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 .
  distance(mp,a,a) == Fin 0
```

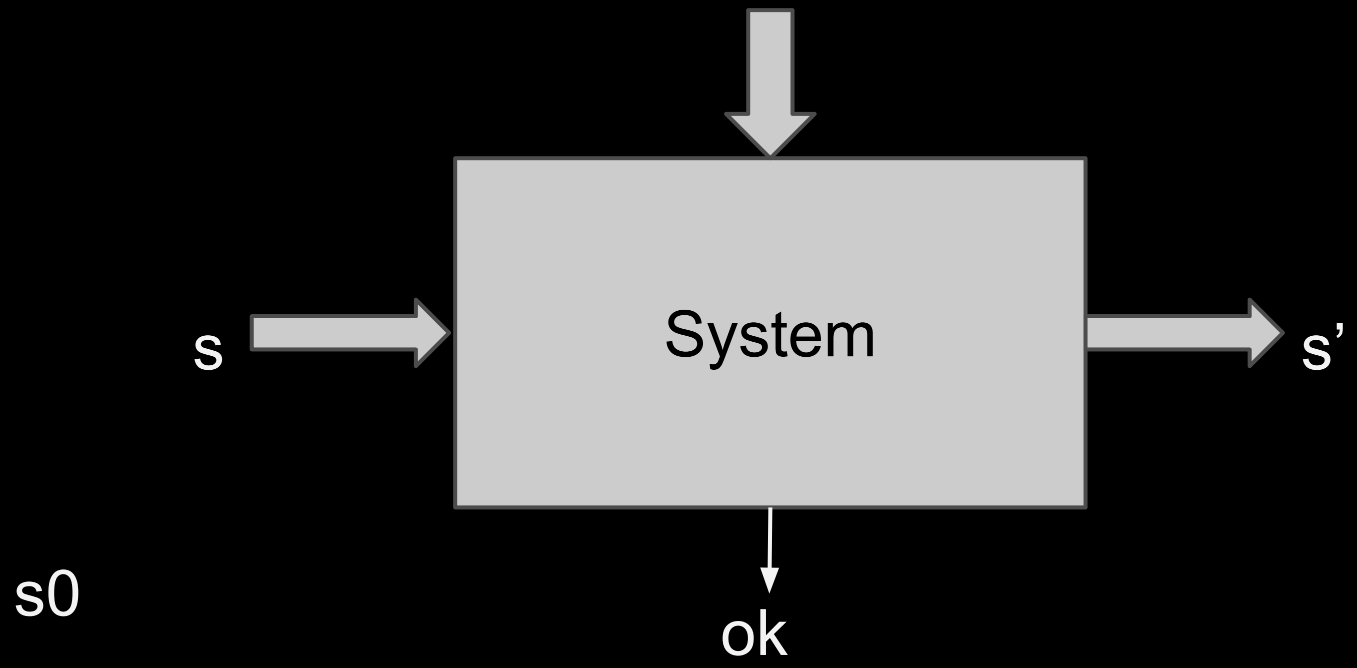
```
ForAll mp,a,b .
  distance(mp,a,b) ==
    minimum [ distance(mp,a’,b) + d
            | (a’,d) <- neighbors(mp,a)
            ]
```

# Inductive Testing

What happens to fault distribution?

- Correctness: by induction
  - soundness: induction over actual distance
  - completeness: induction over function answer
- Induction principle
  - choose this for enabling testing
  - independent of implementation (unlike proving)

# Testing Model Checkers for Safety Properties



System
s
s’
s0
ok

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

greatest fixpoint

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

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

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

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

- Efficiency
  - Model checker is called twice for each test

# **Proof-based Testing: contrapositive testing, inductive testing, coinductive testing**

- Break away from the 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?

# Inductive Testing of Compilers/Interpreters with QuickCheck

```
data Program
  = Var := Expr
  | Skip
  | Program :>>: Program
  | IfThenElse Expr Program Program
  | Decl Var Program
  ...
```

```
compileAndRun :: Program -> State -> IO State
```

```
compileAndRun2 :: Program -> State -> IO State
```

“differential testing”

```
prop_CompilersSame :: Program -> State -> IO Bool
prop_CompilersSame p s1 =
  do s2 <- compileAndRun p s1
     s2' <- compileAndRun2 p s1
     return (s2 == s2')
```

```
> quickCheck prop_CompilersSame
*** FAILED (after 17 tests and 13 shrinks):
if y then
  var x in y := 0
else
  skip
```

minimal counter example

**Library**
for writing
test data generators

recursive
generators

keep track of
test data sizes

specify
frequencies
for the cases

keep track of
invariants

**Library**
for writing 1-step
shrinking functions

**Library**
for writing 1-step
shrinking functions

replace a part with
an immediate
sub-part

*for free*

a + b ⟶ a, b

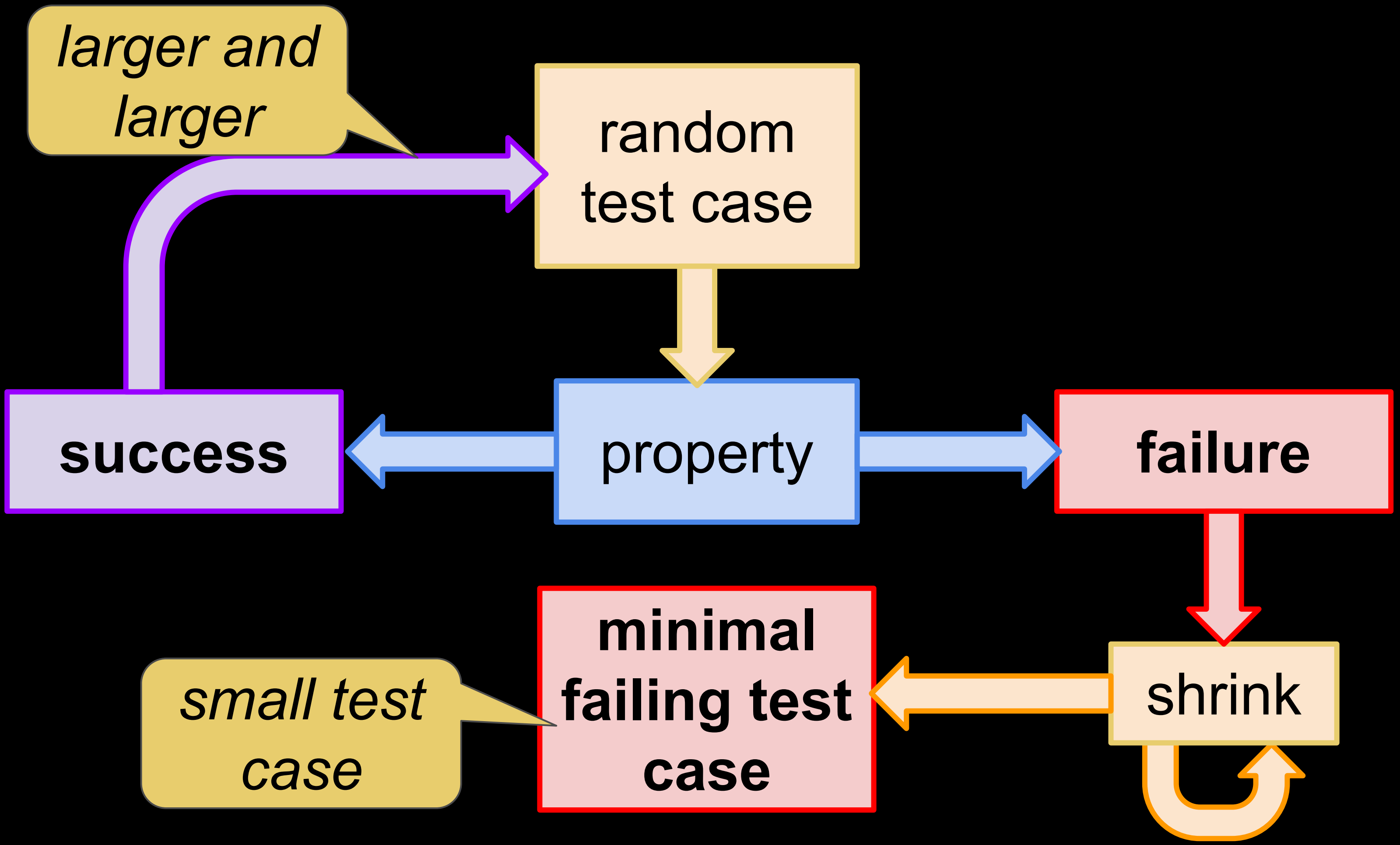
if e then p else q ⟶ p, q

custom rules

C[var x in p] ⟶
var x in C[p]

while e do p ⟶
if e then p else skip

*rules are applied repeatedly until a local minimum is found*


larger and larger
random test case
success
property
failure
shrink
minimal failing test case
small test case

situation:

A **new language**.

You only have **one** interpreter/compiler.

specification language

programming language

how to test?

simple, efficient, complete

```
data Program
  = Skip
  | Var := Expr
  | Program :>>: Program
  | If Expr Program Program
  | While Expr Program
  | ...
```

```
compileAndRun :: Program -> State -> IO State
```

structural inductive
testing of
compileAndRun

```
prop_SequentialComposition ::
                         Program -> Program -> State -> IO Bool
prop_SequentialComposition p q s1 =
  do s3 <- compileAndRun (p :>>: q) s1
     s2 <- compileAndRun p s1
     s3' <- compileAndRun q s2
     return (s3 == s3')
```

runs compiler/interpreter 3 times

"self-consistency"

```
prop_While :: Expr -> Program -> State -> IO Bool
prop_While e p s1 =
  do s2 <- compileAndRun (While e p) s1
     s2' <- compileAndRun (If e (p :>>: While e p) Skip) s1
     return (s2 == s2')
```

runs
compiler/interpreter
2 times

```
prop_Skip :: State -> IO Bool
prop_Skip s1 =
  do s1' <- compileAndRun Skip s1
     return (s1 == s1')
```

- One property for each language construct

- Specification is now **complete**
  - but do not have to specify everything
  - **incremental specification**

- Compare to making new interpreter
  - these properties are as efficient as interpreter under test
  - they can **concentrate on logic**, not efficiency

step-wise inductive testing of compileAndRun

```
step :: Program -> State -> (Program, State)
```

```
prop_Step :: Program -> State -> IO Bool
prop_Step p s1 =
  do s2 <- compileAndRun p s1
     let (p’, s1’) = step p s1
     s2’ <- compileAndRun p’ s1’
     return (s2 == s2’)
```

can also have one property for each step case

**example application 1:**
Scoria -
A language for IoT
devices

C compiler+runtime
vs. interpreter

our own language +
compiler

```
step :: Program -> State -> (Program, State)
```

```
prop_Function :: Program -> Program -> State -> IO Bool
prop_Function f body p s1 =
  do s2 <- compileAndRun (Def f body :>>: p) s1
     s2' <- compileAndRun (Def f body :>>: inline f body p) s1
     return (s2 == s2')
```

- We found bugs in the C-runtime

- We found bugs in our interpreter
  - invariants that did not hold
  - modelling optimizations we wanted to make in the compiler

- A few properties found almost all bugs (function inlining + sequential composition)

**ample application 2**:
ompiler/interpreter for
LTL+extra features

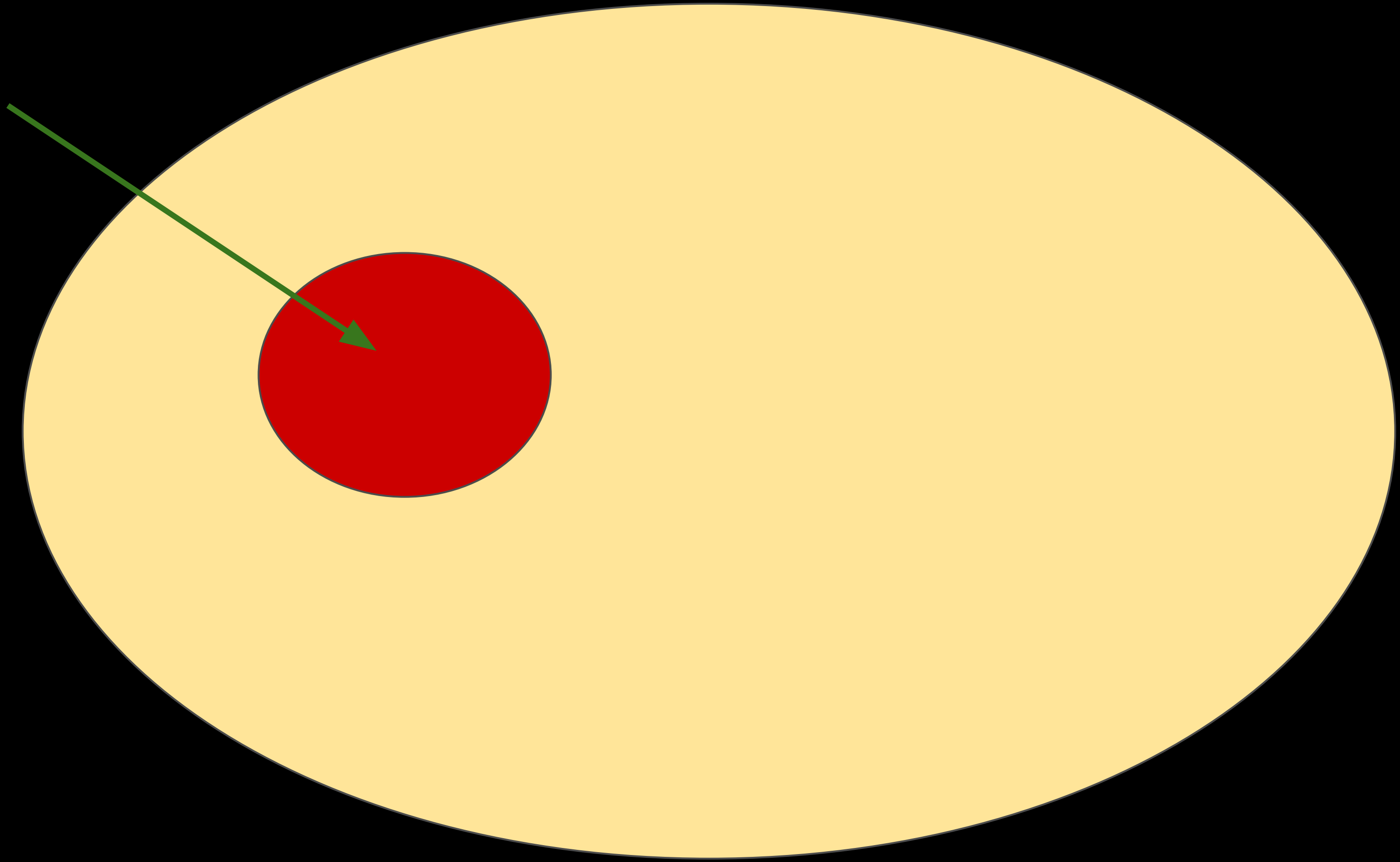
no induction over traces / step

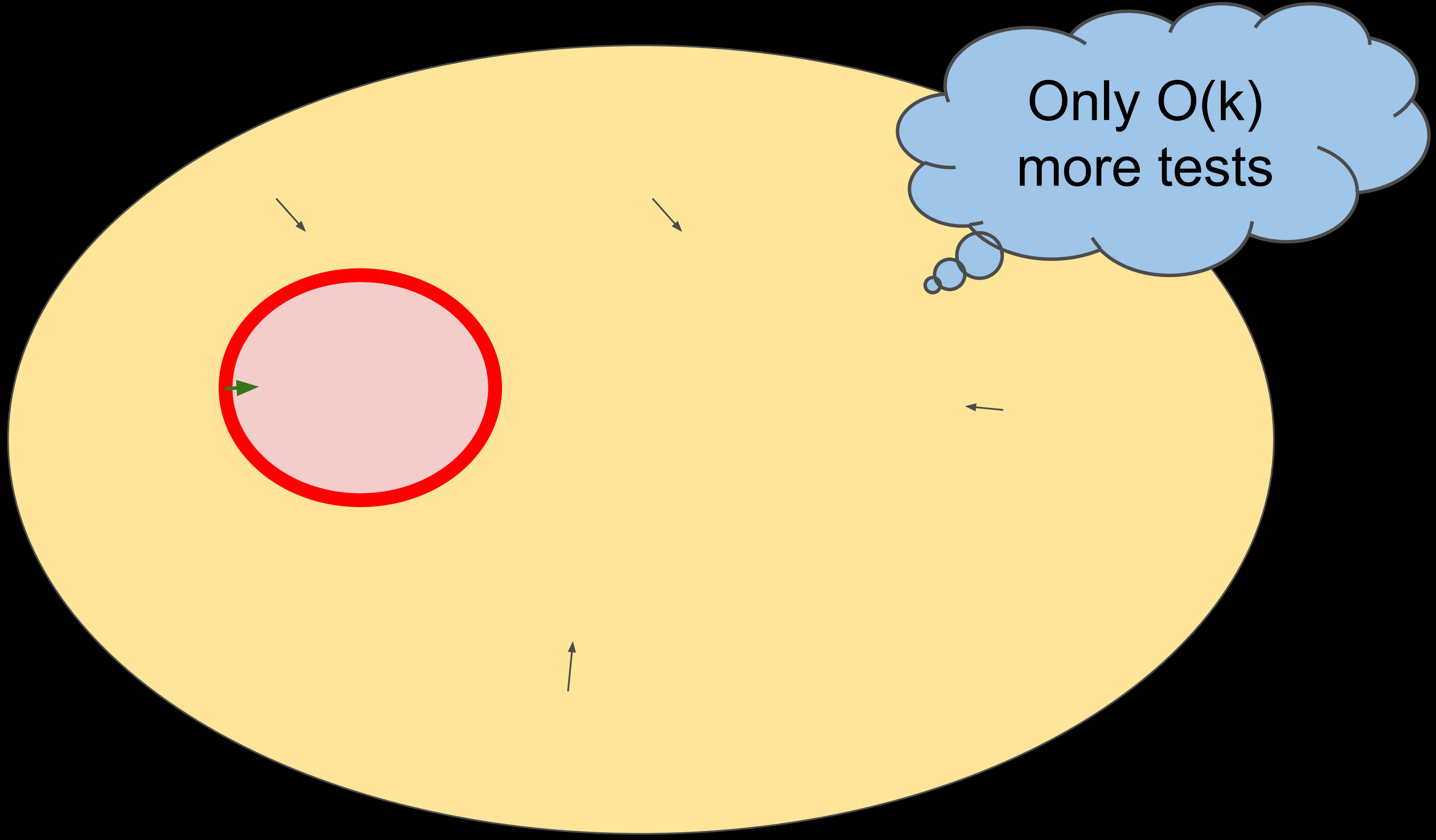
infinite traces / liveness

does not really exist implementation

```
prop_Box :: Form -> Trace -> Property
prop_Box p tr =
  do ok <- run (Box p) tr
     ok' <- forAllSteps tr (\tr' -> run p tr')
     return (ok == ok')
```

- Must fine-tune the trace generator to the property
- Flexible set-up during language design




Only O(k)
more tests

# Summary

- It’s useful to look at specifications as logical specifications and reformulate them into equivalent, but testable specifications
- simple, efficient, complete
- contrapositive testing, (co)inductive testing

# Extra Slides

# 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

# 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

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