# Static Analysis

Martin Kellogg

# Static Analysis

Today's agenda:

- **Finish slides on build systems**
- Reading Quiz
- Motivations for static analysis
- Basics of dataflow analysis

# Incrementalization: hashing

# Incrementalization: hashing

- Compute hash codes for inputs to each task
- When about to execute a task, check input hashes - if they match the last time the task was executed, skip it!

# How to speed up builds?

- **Incrementalize** - only rebuild what you have to
- Execute many tasks in **parallel**

# How to speed up builds?

- **Incrementalize** - only rebuild what you have to
- Execute many tasks in **parallel**
  - some build system tasks are ***embarrassingly parallel***: they can be reordered without explicit synchronization
    - is this true of **all** tasks?

# How to speed up builds?

- **Incrementalize** - only rebuild what you have to
- Execute many tasks in **parallel**
  - some build system tasks are ***embarrassingly parallel***: they can be reordered without explicit synchronization
    - is this true of **all** tasks? **No**: some tasks depend on each other. The problem of scheduling tasks with no unbuilt dependencies is embarrassingly parallel, though.

# How to speed up builds?

- **Incrementalize** - only rebuild what you have to
- Execute many tasks in **parallel**
  - some build system tasks are ***embarrassingly parallel***: they can be reordered without explicit synchronization
    - is this true of **all** tasks? **No**: some tasks depend on each other. The problem of scheduling tasks with no unbuilt dependencies is embarrassingly parallel, though.
- **Cache** artifacts in the cloud

# How do build systems differ

# How do build systems differ

- Scheduling algorithm

# How do build systems differ

- Scheduling algorithm
  - We've already seen topological scheduling (used by e.g. make), which is a **static** scheduling algorithm

# How do build systems differ

- Scheduling algorithm
  - We've already seen topological scheduling (used by e.g. make), which is a **static** scheduling algorithm
  - **Dynamic** scheduling algorithms are also possible

# How do build systems differ

- Scheduling algorithm
  - We've already seen topological scheduling (used by e.g. make), which is a **static** scheduling algorithm
  - **Dynamic** scheduling algorithms are also possible
    - **Key idea**: compute what dependencies are necessary as you go

# How do build systems differ

- Scheduling algorithm
  - We've already seen topological scheduling (used by e.g. make), which is a **static** scheduling algorithm
  - **Dynamic** scheduling algorithms are also possible
    - **Key idea**: compute what dependencies are necessary as you go
    - this is how e.g., Bazel actually schedules tasks

# How do build systems differ

- Rebuilding strategy

# How do build systems differ

- Rebuilding strategy
  - We’ve seen two:

# How do build systems differ

- Rebuilding strategy
  - We've seen two:
    - a ***dirty bit*** strategy (make's timestamps)

# How do build systems differ

- Rebuilding strategy
  - We've seen two:
    - a ***dirty bit*** strategy (make's timestamps)
    - a ***verifying trace*** strategy (storing hashes of each object)

# How do build systems differ

- Rebuilding strategy
  - We've seen two:
    - a ***dirty bit*** strategy (make's timestamps)
    - a ***verifying trace*** strategy (storing hashes of each object)
  - Other options:
    - ***constructive traces***: store all intermediate objects (usually in the cloud) along with the hashes of the **inputs** used to produce them. If we ever see the same input hashes again, just return the intermediate object

# How do build systems differ

- How are tasks expressed?

# How do build systems differ

- How are tasks expressed?
  - traditionally **declarative** (e.g., make, Ant, Maven)
    - “declarative” = you tell the build system what you want, it figures out how to build that thing
    - call back to languages: programming languages can also be from the ***declarative paradigm*** (e.g., Prolog)

# How do build systems differ

- How are tasks expressed?
  - traditionally **declarative** (e.g., make, Ant, Maven)
    - “declarative” = you tell the build system what you want, it figures out how to build that thing
    - call back to languages: programming languages can also be from the ***declarative paradigm*** (e.g., Prolog)
  - most modern build systems have **scripting languages**
    - e.g., Groovy in Gradle, Starlark in Bazel, etc.
    - enables us to write tasks as if they are other code

# How to choose a build system

# How to choose a build system

**High level idea**: same rules apply to choosing a language

# How to choose a build system

**High level idea**: same rules apply to choosing a language

- **don’t change what’s already there** unless there is a good reason

# How to choose a build system

**High level idea**: same rules apply to choosing a language

- **don't change what's already there** unless there is a good reason
- **follow convention** and prefer the tooling that's "idiomatic" to your language
  - e.g., use Gradle or Maven when working in Java

# When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem

# When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:

# When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven's per-module incremental compilations)

# When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven's per-module incremental compilations)
    - lack of support for artifact caching (= **cloud builds**)

# When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven's per-module incremental compilations)
    - lack of support for artifact caching (= **cloud builds**)
    - build has become too complex for a declarative task language

# When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven's per-module incremental compilations)
    - lack of support for artifact caching (= **cloud builds**)
    - build has become too complex for a declarative task language
  - most projects keep the same build system **forever**

# Best practices

- Automate everything

# Best practices

- Automate everything
- Always use a build tool

# Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)

# Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)

Your CI server is a good place to test that your build is hermetic. **Standard practice**: spin up a new CI server for **each build**.

# Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)
- Don’t depend on anything that’s not in the build file (hermetic)

# Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)
- Don’t depend on anything that’s not in the build file (hermetic)
- Don’t break the build

# Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)
- Don’t depend on anything that’s not in the build file (hermetic)
- Don’t break the build

A **common mistake to avoid**: allowing the CI server to fail for a long time because “we know what the problem is.” Don’t do this: leads to complacency, missing real bugs.

# Static Analysis

Today's agenda:

- Finish slides on build systems
- **Reading Quiz**
- Motivations for static analysis
- Basics of dataflow analysis

# Reading quiz: static analysis

Q1: **TRUE** or **FALSE**: the goal of the FindBugs tool described in the reading is to find as many different kinds of bugs as possible. For example, it supports finding possible null-pointer dereferences, possible out-of-bounds array accesses, and possible resource leaks.

Q2: **TRUE** or **FALSE**: the AWS team that wrote the second article found that one of the most effective means of selling developers on the utility of formal verification was for the verification team to not only identify bugs but provide code patches for them.

# Reading quiz: static analysis

Q1: **TRUE** or **FALSE**: the goal of the FindBugs tool described in the reading is to find as many different kinds of bugs as possible. For example, it supports finding possible null-pointer dereferences, possible out-of-bounds array accesses, and possible resource leaks.

Q2: **TRUE** or **FALSE**: the AWS team that wrote the second article found that one of the most effective means of selling developers on the utility of formal verification was for the verification team to not only identify bugs but provide code patches for them.

# Reading quiz: static analysis

Q1: **TRUE** or **FALSE**: the goal of the FindBugs tool described in the reading is to find as many different kinds of bugs as possible. For example, it supports finding possible null-pointer dereferences, possible out-of-bounds array accesses, and possible resource leaks.

Q2: **TRUE** or **FALSE**: the AWS team that wrote the second article found that one of the most effective means of selling developers on the utility of formal verification was for the verification team to not only identify bugs but provide code patches for them.

# Motivations for static analysis

# Motivations for static analysis

- Quality assurance is critical to software engineering

# Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:

# Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:
  - **code review**, the most common **static** QA technique

# Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:
  - **code review**, the most common **static** QA technique
  - **linting**, the second-most common static QA technique

# Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:
  - **code review**, the most common **static** QA technique
  - **linting**, the second-most common static QA technique
  - **testing**, the most common **dynamic** QA technique

# Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:
  - **code review**, the most common **static** QA technique
  - **linting**, the second-most common static QA technique
  - **testing**, the most common **dynamic** QA technique
- We've seen that both code review and testing have **significant limitations** in practice:

# Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:
  - **code review**, the most common **static** QA technique
  - **linting**, the second-most common static QA technique
  - **testing**, the most common **dynamic** QA technique
- We've seen that both code review and testing have **significant limitations** in practice:
  - code review is limited by human error

# Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:
  - **code review**, the most common **static** QA technique
  - **linting**, the second-most common static QA technique
  - **testing**, the most common **dynamic** QA technique
- We've seen that both code review and testing have **significant limitations** in practice:
  - code review is limited by human error
  - testing is limited by your choice of tests (Dijkstra again)

# Motivations for static analysis

- Quality assurance is critical to
- We've already covered three i
  - **code review**, the most com
  - **linting**, the second-most c
  - **testing**, the most common
- We've seen that both code rev
  **limitations** in practice:
  - code review is limited by human error
  - testing is limited by your choice of tests (Dijkstra again)

**Today's goal:** discuss other **automated** static analysis techniques that complement testing and code review in a quality assurance process

# Motivation: many defects are hard to test for

# Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths

# Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing

# Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)

# Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)
- We want to learn about "**all possible runs**" of the program for particular properties

# Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)
- We want to learn about "**all possible runs**" of the program for particular properties
  - Without actually running the program!

# Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)
- We want to learn about "**all possible runs**" of the program for particular properties
  - Without actually running the program!
  - Bonus: we don't need test cases!

# Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyz
  such defects is **not feasible** (cf.
- We want to learn about "**all po**
  particular properties
  - Without actually running t
  - Bonus: we don't need test ca

This is especially true for certain kinds of hard-to-test-for defects that might not be apparent even if you do exercise them, such as **resource leaks**

# What does static analysis do well?

# What does static analysis do well?

- Defects that result from inconsistently following **simple**, mechanical design **rules**

# What does static analysis do well?

- Defects that result from inconsistently following **simple**, mechanical design **rules**
  - Security: buffer overruns, input validation
  - Memory safety: null pointers, initialized data
  - Resource leaks: memory, OS resources
  - API Protocols: device drivers, GUI frameworks
  - Exceptions: arithmetic, library, user-defined
  - Encapsulation: internal data, private functions
  - Data races: two threads, one variable

# What does static analysis do well?

- Defects that result from inconsiste
  mechanical design **rules**
  - Security: buffer overruns, input
  - Memory safety: null pointers, in
  - Resource leaks: memory, OS resources
  - API Protocols: device drivers, GUI frameworks
  - Exceptions: arithmetic, library, user-defined
  - Encapsulation: internal data, private functions
  - Data races: two threads, one variable

There are **rules** for doing each of these things **correctly**, and a static analysis can automate those rules.

# What is a static analysis?

# What is a static analysis?

**Definition**: ***static analysis*** is the systematic examination of an abstraction of program state space

# What is a static analysis?

**Definition**: ***static analysis*** is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program

# What is a static analysis?

**Definition**: ***static analysis*** is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program
  - in contrast to a **dynamic analysis**, such as testing, which does execute the program

# What is a static analysis?

**Definition**: ***static analysis*** is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program
  - in contrast to a **dynamic analysis**, such as testing, which does execute the program
- an **abstraction**, in this context, is a **selective representation** of the program that is simpler to analyze

# What is a static analysis?

**Definition**: ***static analysis*** is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program
  - in contrast to a **dynamic analysis**, such as testing, which does execute the program
- an **abstraction**, in this context, is a **selective representation** of the program that is simpler to analyze
  - **key idea**: the abstraction will have fewer states to explore
    - hopefully, many fewer!

# Typical static analysis: dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program

# Typical static analysis: dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program
  - Dataflow analysis is the **core idea** behind many static analyses

# Typical static analysis: dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program
  - Dataflow analysis is the **core idea** behind many static analyses
- We first abstract the program to an AST or CFG

# Typical static analysis: dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program
  - Dataflow analysis is the **core idea** behind many static analyses
- We first abstract the program to an AST or CFG
- We then abstract what we want to learn (e.g., to help developers) down to a small set of ***abstract values***

# Typical static analysis: dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program
  - Dataflow analysis is the **core idea** behind many static analyses
- We first abstract the program to an AST or CFG
- We then abstract what we want to learn (e.g., to help developers) down to a small set of ***abstract values***
- We finally give **rules** for computing those abstract values

# Typical static analysis: dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program
  - Dataflow analysis is the **core idea** behind many static analyses
- We first abstract the program to an AST or CFG
- We then abstract what we want to learn (e.g., to help developers) down to a small set of ***abstract values***
- We finally give **rules** for computing those abstract values
  - Dataflow analyses take **programs as input**

# Example dataflow analyses

Two examples of dataflow analyses:

# Example dataflow analyses

Two examples of dataflow analyses:

1. an analysis for finding **definite** null-pointer dereferences

   “Whenever execution reaches *ptr at program location L, ptr will be NULL”

# Example dataflow analyses

Two examples of dataflow analyses:

1. an analysis for finding **definite** null-pointer dereferences

   “Whenever execution reaches *ptr at program location L, ptr will be NULL”

2. an analysis for finding **potential** secure information leaks

   “We read in a secret string at location L, but there is a possible future public use of it”

# Definite vs potential

A “**definite**” null-pointer dereference exists if and only the pointer is NULL on **every** program execution

A “**potential**” secure information leak exists if and only if the secure information leaks on **any** program execution

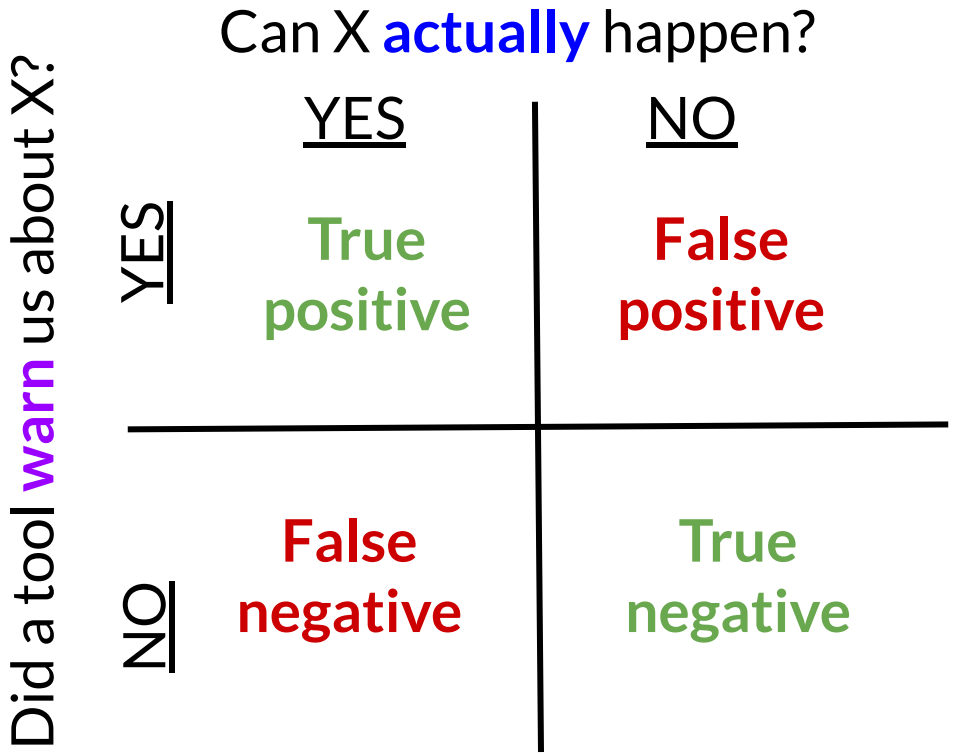
# Definite vs potential

A “**definite**” null-pointer dereference exists if and only the pointer is NULL on **every** program execution

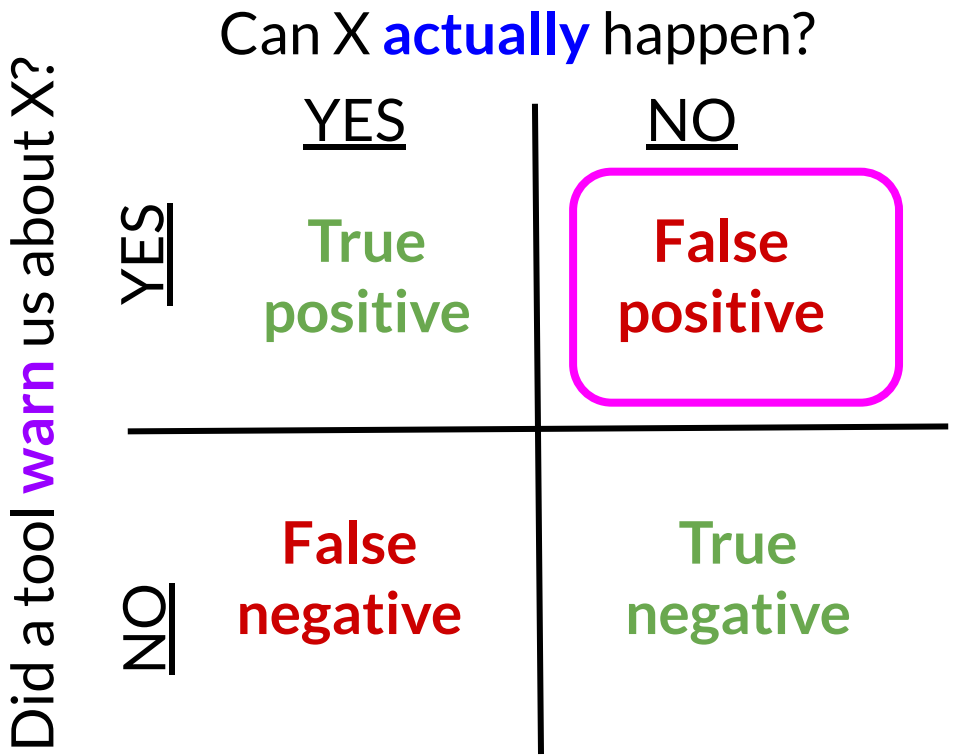
A “**potential**” secure information leak exists if and only if the secure information leaks on **any** program execution

The use of “**every**” and “**any**” here guarantee that we must reason about **all paths** through the program!

# Definite vs potential = false positives vs negatives

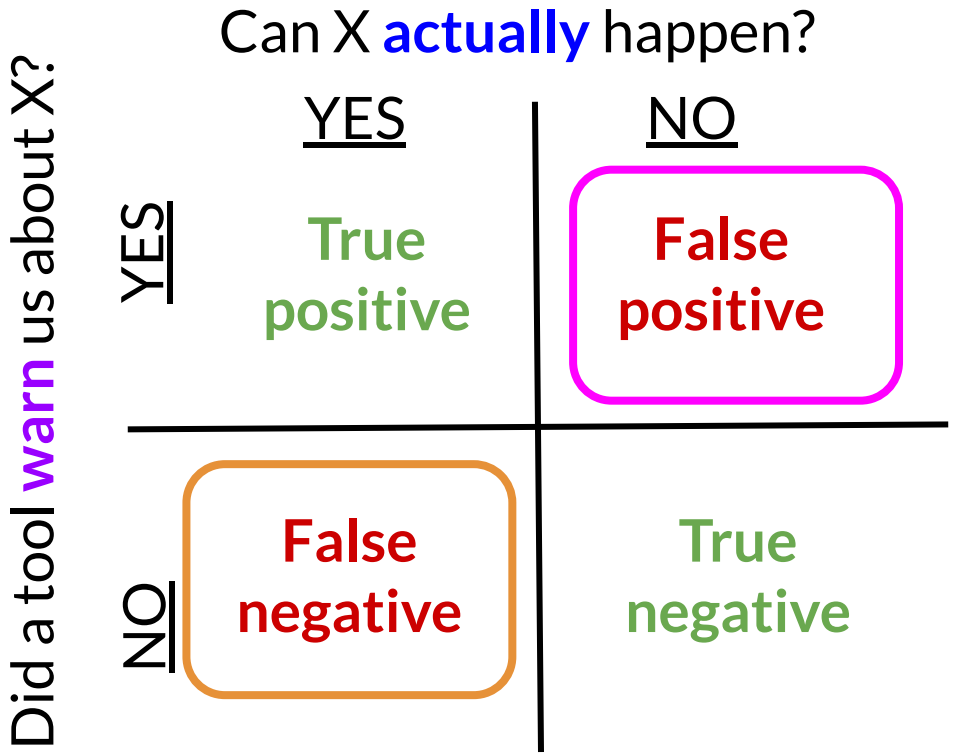


# Definite vs potential = false positives vs negatives




checking for "potential" properties usually comes with false positives

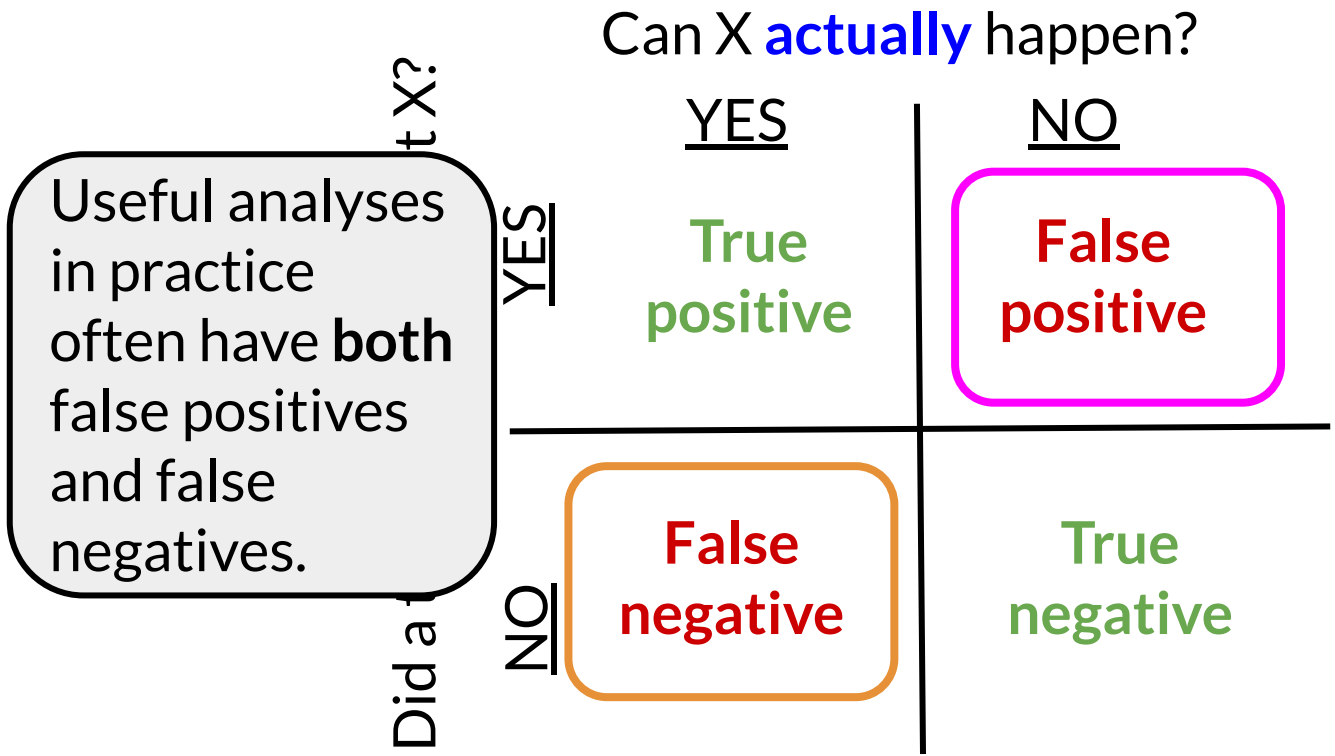
# Definite vs potential = false positives vs negatives




checking for "potential" properties usually comes with false positives

checking for "definite" properties usually comes with false negatives

# Definite vs potential = false positives vs negatives




Useful analyses in practice often have **both** false positives and false negatives.

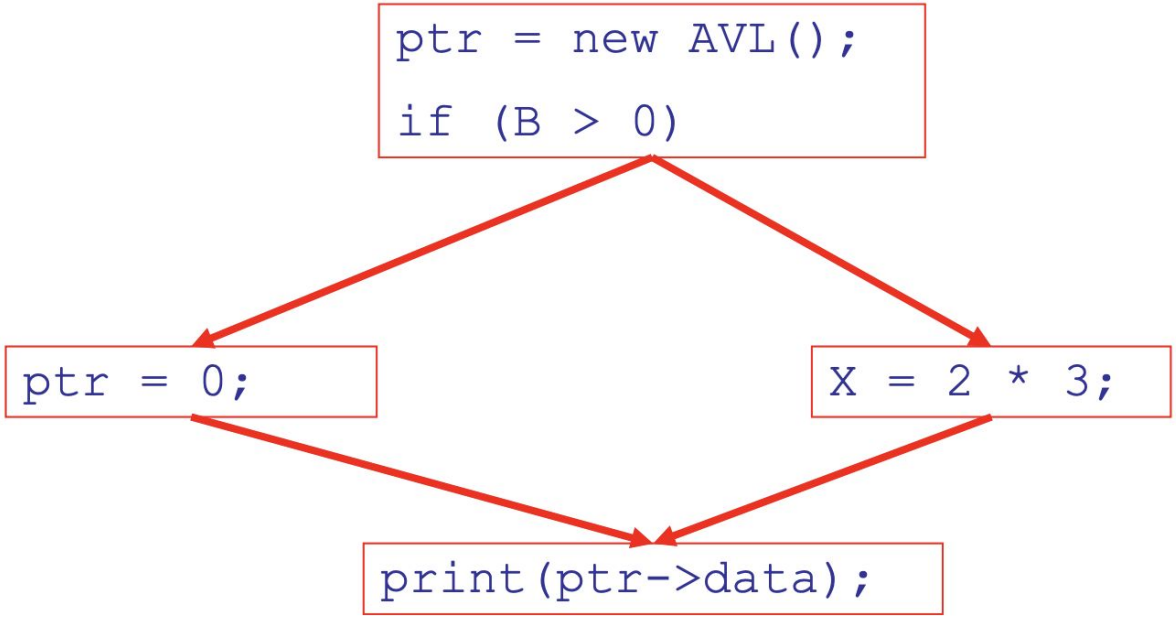
checking for "potential" properties usually comes with false positives

checking for "definite" properties usually comes with false negatives

# Null-pointer analysis example

# Null-pointer analysis example

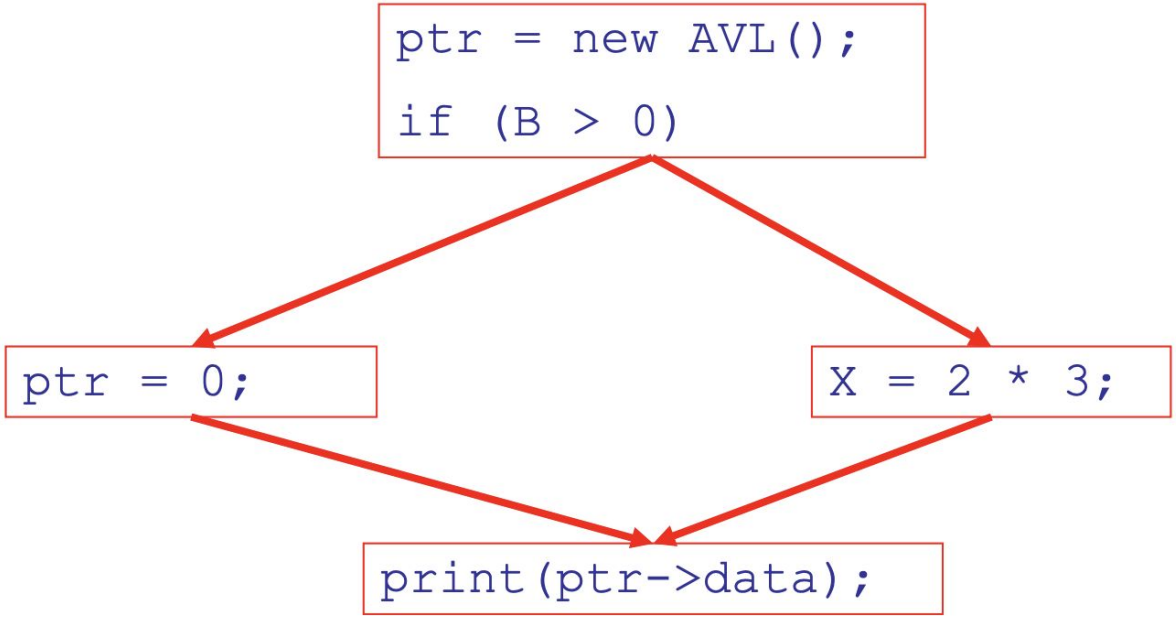
Question: is `ptr` ***always*** null when it is dereferenced?



# Null-pointer analys

**Q:** what does "`ptr` always null" actually require about assignments to `ptr`?

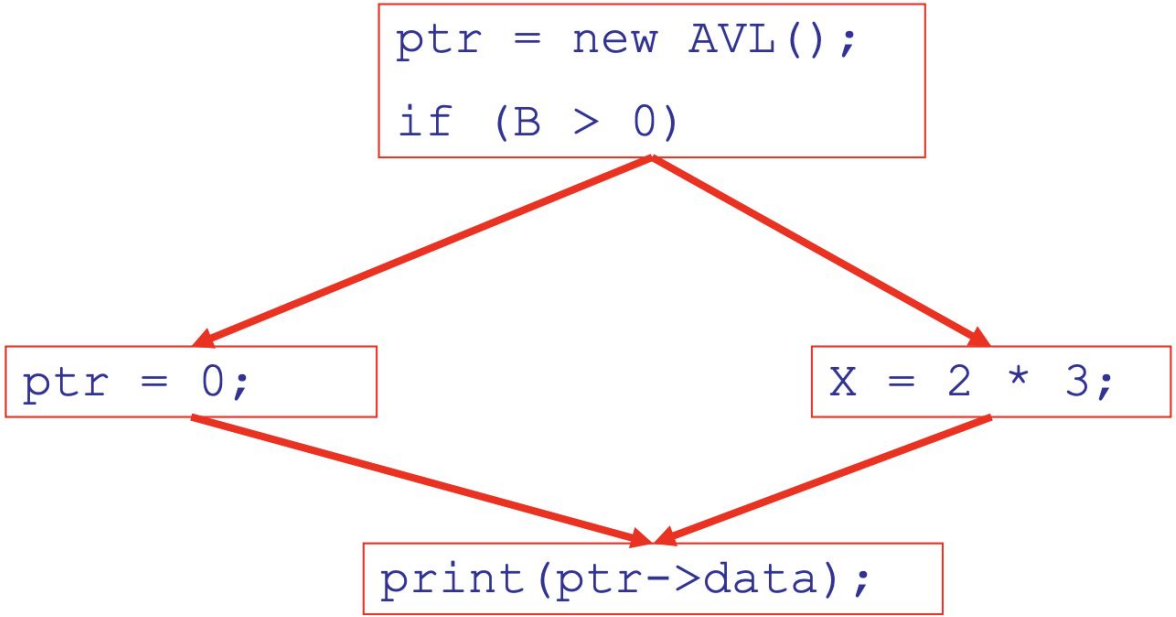
Question: is `ptr` ***always*** null when it is dereferenced?



# Null-pointer analys

**Q:** what does "`ptr` always null" actually require about assignments to `ptr`?
**A:** on all paths, the **last assignment** to `ptr` must have been null (= 0 in C)

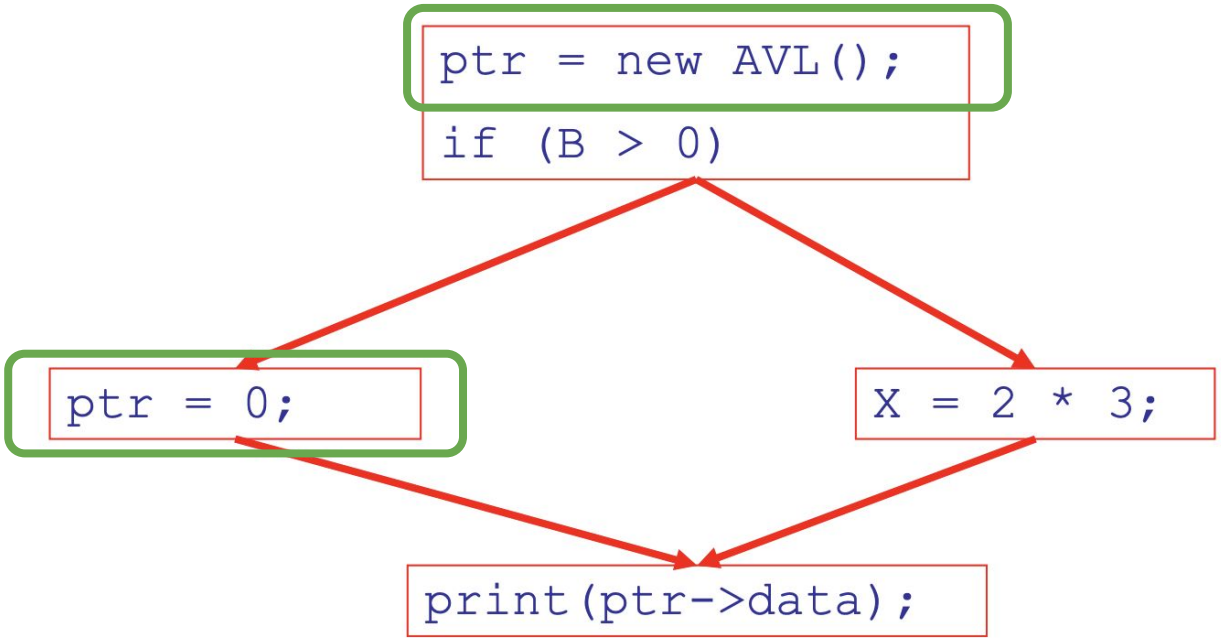
Question: is `ptr` ***always*** null when it is dereferenced?



# Null-pointer analys

**Q:** what does "`ptr` always null" actually require about assignments to `ptr`?
**A:** on all paths, the **last assignment** to `ptr` must have been null (= 0 in C)

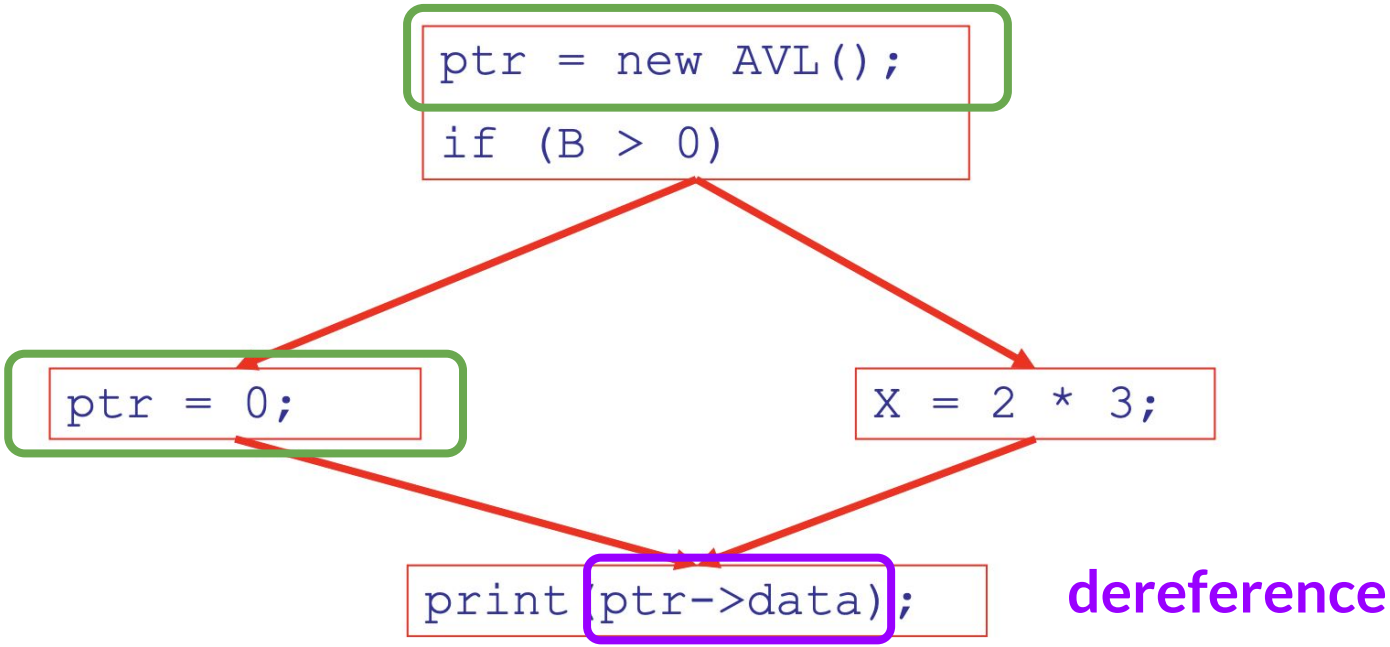
Question: is `ptr` ***always*** null when it is dereferenced?



# Null-pointer analys

**Q:** what does "`ptr` always null" actually require about assignments to `ptr`?
**A:** on all paths, the **last assignment** to `ptr` must have been null (= 0 in C)

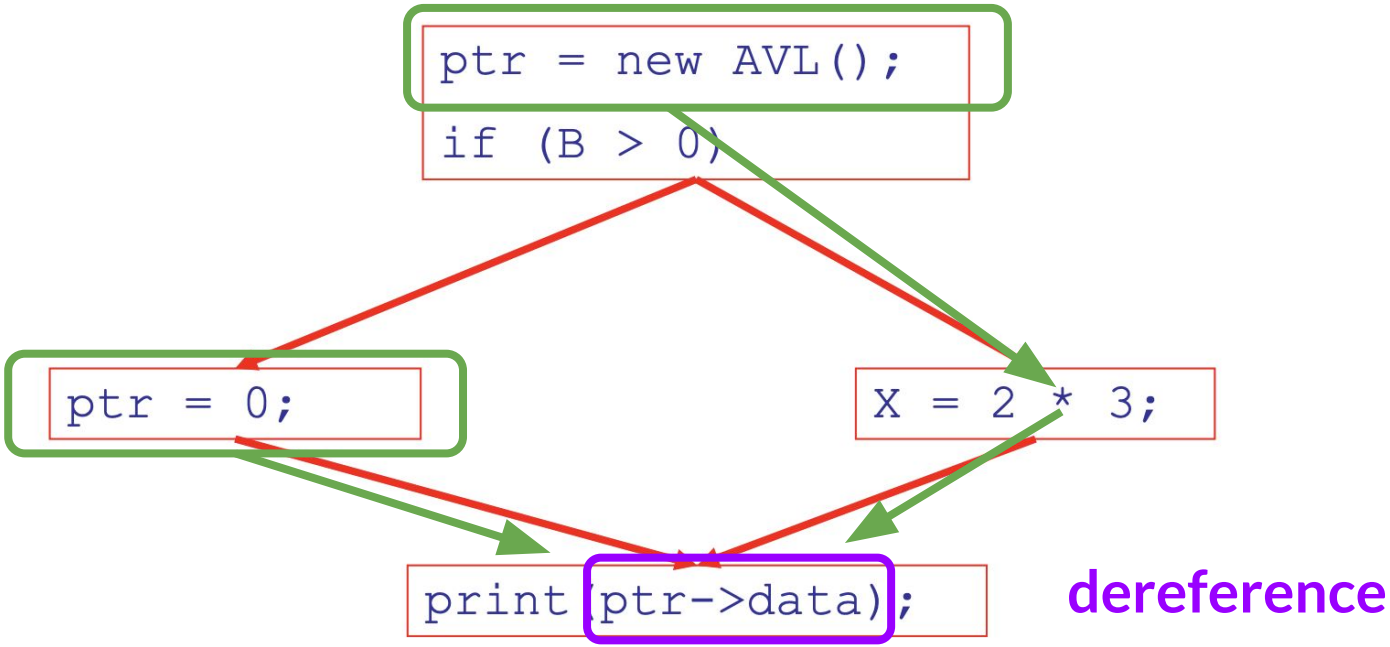
Question: is `ptr` ***always*** null when it is dereferenced?



# Null-pointer analys

**Q:** what does "`ptr` always null" actually require about assignments to `ptr`?
**A:** on all paths, the **last assignment** to `ptr` must have been null (= 0 in C)

Question: is `ptr` ***always*** null when it is dereferenced?



# Common traits of dataflow analysis

# Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution

# Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution
  - for “definite” analyses: **for all** executions, is P true at this point?

# Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution
  - for “definite” analyses: **for all** executions, is P true at this point?
  - for “potential” analyses: does **there exist** an execution for which P is true at this point?

# Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution


  - for “definite” analyses: **for all** executions, is P true at this point?
  - for “potential” analyses: does **there exist** an execution for which P is true at this point?

# Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution
  - for “definite” analyses: **for all** executions, is P true at this point?
  - for “potential” analyses: does **there exist** an execution for which P is true at this point?

# Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution
  - for “definite” analyses: **for all** executions, is P true at this point?
  - for “potential” analyses: does **there exist** an execution for which P is true at this point?
- Knowing P at any specific program point usually requires knowledge of the entire method body

# Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution
  - for “definite” analyses: **for all** executions, is P true at this point?
  - for “potential” analyses: does **there exist** an execution for which P is true at this point?
- Knowing P at any specific program point usually requires knowledge of the entire method body
- Property P is typically **undecidable**

# Undecidability of program properties

# Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:

# Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:

**"interesting"** in this context means "not trivial", i.e., not uniformly true or false for all programs

# Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**

# Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?

# Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?
    - *Assume* we can answer this question precisely

# Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?
    - *Assume* we can answer this question precisely
    - Oops: We can now solve the halting problem.

# Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?
    - *Assume* we can answer this question precisely
    - Oops: We can now solve the halting problem.
    - Take function H and find out if it halts by testing function `F(x) = { H(x); return 1; }` to see if it has a positive result

# Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?
    - *Assume* we can answer this question precisely
    - Oops: We can now solve the halting problem.
    - Take function H and find out if it halts by testing function `F(x) = { H(x); return 1; }` to see if it has a positive result
    - Contradiction!

# Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program hal
    - This is called the h
  - Is the result of a funct
    - *Assume* we can ans
    - Oops: We can now
    - Take function H an
      `F(x) = { H(x); return 1; }` to see if it has a positive result
    - Contradiction!

**Rice's theorem caveats:**

- only applies to **semantic** properties (syntactic properties are decidable)
- "programs" only includes programs **with loops**

# Loops

- Almost every important program has a **loop**
  - Often based on user input

# Loops

- Almost every important program has a **loop**
  - Often based on user input
- An **algorithm** always terminates (remember your theory class!)
  - So a dataflow analysis algorithm must terminate even if the input program loops

# Loops

- Almost every important program has a **loop**
  - Often based on user input
- An **algorithm** always terminates (remember your theory class!)
  - So a dataflow analysis algorithm must terminate even if the input program loops
- This is one source of **imprecision**
  - "**imprecision**" = "not always getting the right answer"
  - Suppose you dereference the null pointer on the 500th iteration but we only analyze 499 iterations

# Conservative program analysis

- Because our analysis must run on a computer, we need the analysis itself to be decidable

# Conservative program analysis

- Because our analysis must run on a computer, we need the analysis itself to be decidable
- But, because of Rice's Theorem, we know that finding the right answer all the time is undecidable :(

# Conservative program analysis

- Because our analysis must run on a computer, we need the analysis itself to be decidable
- But, because of Rice's Theorem, we know that finding the right answer all the time is undecidable :(
- **Solution**: when in doubt, allow the analysis to answer "I don't know"

# Conservative program analysis

- Because our analysis must run on a computer, we need the analysis itself to be decidable
- But, because of Rice's Theorem, we know that finding the right answer all the time is undecidable :(
- **Solution**: when in doubt, allow the analysis to answer "I don't know"
    - this is called ***conservative*** analysis

# Conservative program analysis

- It's **always correct** to say "I don't know"

# Conservative program analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible

# Conservative program analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible

**Definition**: a ***sound*** program analysis has no false negatives

# Conservative program analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible

**Definition**: a ***sound*** program analysis has no false negatives

- always answers "I don't know" if there is a **potential** bug

# Conservative program analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible

**Definition**: a ***sound*** program analysis has no false negatives

- always answers "I don't know" if there is a **potential** bug

**Definition**: a ***complete*** program analysis has no false positives

# Conservative program analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible

**Definition**: a ***sound*** program analysis has no false negatives

- always answers "I don't know" if there is a **potential** bug

**Definition**: a ***complete*** program analysis has no false positives

- always answers "I don't know" if there isn't a **definite** bug

# Soundness vs completeness

- Building a sound or complete analysis is **easy**

# Soundness vs completeness

- Building a sound or complete analysis is **easy**
  - trivially sound analysis: report a bug **on every line**

# Soundness vs completeness

- Building a sound or complete analysis is **easy**
  - trivially sound analysis: report a bug **on every line**
  - trivially complete analysis: **never** report a bug

# Soundness vs completeness

- Building a sound or complete analysis is **easy**
  - trivially sound analysis: report a bug **on every line**
  - trivially complete analysis: **never** report a bug
- Building a sound and ***precise*** (= “few false positives”) analysis or a complete analysis with ***high recall*** (= “few false negatives”) is **very hard**

# Soundness vs completeness

- Building a sound or complete analysis is **easy**
  - trivially sound analysis: report a bug **on every line**
  - trivially complete analysis: **never** report a bug
- Building a sound and ***precise*** (= “few false positives”) analysis or a complete analysis with ***high recall*** (= “few false negatives”) is **very hard**
  - “sound and precise” analyses are my research area :)

# Soundness vs completeness

- Building a sound or complete analysis is **easy**
  - trivially sound analysis: report a bug **on every line**
  - trivially complete analysis: **never** report a bug
- Building a sound and ***precise*** (= “few false positives”) analysis or a complete analysis with ***high recall*** (= “few false negatives”) is **very hard**
  - “sound and precise” analyses are my research area :)
  - also relevant in practice: “fast”, “easy to use”, etc.

# Soundness vs completeness

- Which is more important: **soundness** or **completeness**?

# Soundness vs completeness

- Which is more important: **soundness** or **completeness**?
- Answer: **it depends**!

# Soundness vs completeness

- Which is more important: **soundness** or **completeness**?
- Answer: **it depends**!
  - Are you writing a bug-finding analysis for websites that show pictures of cats? False positives waste time, so choose completeness.

# Soundness vs completeness

- Which is more important: **soundness** or **completeness**?
- Answer: **it depends**!
  - Are you writing a bug-finding analysis for websites that show pictures of cats? False positives waste time, so choose completeness.
    - “I don’t know” = don’t issue a warning

# Soundness vs completeness

- Which is more important: **soundness** or **completeness**?
- Answer: **it depends**!
  - Are you writing a bug-finding analysis for websites that show pictures of cats? False positives waste time, so choose completeness.
    - “I don’t know” = don’t issue a warning
  - Are you writing a bug-finding analysis for aircraft autopilots? False negatives cause crashes, so choose soundness.

# Soundness vs completeness

- Which is more important: **soundness** or **completeness**?
- Answer: **it depends**!
  - Are you writing a bug-finding analysis for websites that show pictures of cats? False positives waste time, so choose completeness.
    - “I don’t know” = don’t issue a warning
  - Are you writing a bug-finding analysis for aircraft autopilots? False negatives cause crashes, so choose soundness.
    - “I don’t know” = do issue a warning

# Soundness vs completeness

- In practice, most static analyses are **neither** sound nor complete

# Soundness vs completeness

- In practice, most static analyses are **neither** sound nor complete
  - e.g., FindBugs from today's reading has both false positives and false negatives

# Soundness vs completeness

- In practice, most static analyses are **neither** sound nor complete
  - e.g., FindBugs from today's reading has both false positives and false negatives
  - most common exception: most **type systems** are sound

# Soundness vs completeness

- In practice, most static analyses are **neither** sound nor complete
  - e.g., FindBugs from today's reading has both false positives and false negatives
  - most common exception: most **type systems** are sound
    - remember: type systems are just another static analysis

# Soundness vs completeness

- In practice, most static analyses are **neither** sound nor complete
  - e.g., FindBugs from today's reading has both false positives and false negatives
  - most common exception: most **type systems** are sound
    - remember: type systems are just another static analysis
  - few complete analyses exist in practice

# Soundness vs completeness

- In practice, most static analyses are **neither** sound nor complete
  - e.g., FindBugs from today's reading has both false positives and false negatives
  - most common exception: most **type systems** are sound
    - remember: type systems are just another static analysis
  - few complete analyses exist in practice
    - theory is underdeveloped, but another area of active research!

# Limitations of static analysis

# Limitations of static analysis

- static analysis **abstracts away** information to remain decidable

# Limitations of static analysis

- static analysis **abstracts away** information to remain decidable
  - **potential problem**: what if the information that was abstracted away is important?

# Limitations of static analysis

- static analysis **abstracts away** information to remain decidable
  - **potential problem**: what if the information that was abstracted away is important?
  - can we ever have a "**perfect**" abstraction?

# Limitations of static analysis

- static analysis **abstracts away** information to remain decidable
  - **potential problem**: what if the information that was abstracted away is important?
  - can we ever have a "**perfect**" abstraction?
    - of course not (Rice's theorem again)

# Limitations of static analysis

- static analysis **abstracts away** information to remain decidable
  - **potential problem**: what if the information that was abstracted away is important?
  - can we ever have a "**perfect**" abstraction?
    - of course not (Rice's theorem again)
    - but, in practice, we can get very close

# Limitations of static analysis

- static analysis is **best** when the rules it enforces are:

# Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply

# Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply
- **implication**: if you find yourself struggling to follow a well-defined (but complicated for a human) rule set while writing code, it might be time to reach for a static analysis

# Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply
- **implication**: if you find yourself struggling to follow a well-defined (but complicated for a human) rule set while writing code, it might be time to reach for a static analysis
  - this sort of situation comes up often:

# Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply
- **implication**: if you find yourself struggling to follow a well-defined (but complicated for a human) rule set while writing code, it might be time to reach for a static analysis
  - this sort of situation comes up often:
    - x86/64 calling convention

# Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply
- **implication**: if you find yourself struggling to follow a well-defined (but complicated for a human) rule set while writing code, it might be time to reach for a static analysis
  - this sort of situation comes up often:
    - x86/64 calling convention
    - complex API protocols (“call A then B then C then …”)

# Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply
- **implication**: if you find yourself struggling to follow a well-defined (but complicated for a human) rule set while writing code, it might be time to reach for a static analysis
  - this sort of situation comes up often:
    - x86/64 calling convention
    - complex API protocols (“call A then B then C then …”)
    - security rules, etc.

# Static analysis in practice

You’re likely to encounter:

# Static analysis in practice

You’re likely to encounter:

- static **type systems** (sound)

# Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)

# Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)
- "***heuristic***" bug-finding tools backed by dataflow analyses

# Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)
- "***heuristic***" bug-finding tools backed by dataflow analyses

> ***heuristic*** is a fancy word for "best effort"

# Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)
- "***heuristic***" bug-finding tools backed by dataflow analyses
  - built into modern IDEs

# Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)
- "***heuristic***" bug-finding tools backed by dataflow analyses
  - built into modern IDEs
  - aim for low false positive rates

# Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)
- "***heuristic***" bug-finding tools backed by dataflow analyses
  - built into modern IDEs
  - aim for low false positive rates
  - widely used in industry:
    - ErrorProne at Google, Infer at Meta, SpotBugs at many places (including Amazon), Coverity, Fortify, etc.

# Static analysis in practice

Less common, but useful to know about:

# Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems

# Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java

# Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)

# Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- ***formal verification***

# Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- ***formal verification***
  - you write a specification

# Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- ***formal verification***
  - you write a specification
  - tool verifies that code matches that specification

# Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- ***formal verification***
  - you write a specification
  - tool verifies that code matches that specification
  - very high effort, but enables sound reasoning about complex properties (= worth it for very high value systems)

# Static analysis in practice: soundiness

- all “**sound**” static analyses have a ***trusted computing base*** (***TCB***)

# Static analysis in practice: soundiness

- all “**sound**” static analyses have a ***trusted computing base*** (***TCB***)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound

# Static analysis in practice: soundiness

- all “**sound**” static analyses have a ***trusted computing base*** (***TCB***)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between “sound” analyses

# Static analysis in practice: soundiness

- all "**sound**" static analyses have a ***trusted computing base*** (***TCB***)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between "sound" analyses
  - e.g., TCB for many of my pluggable type systems includes the entire Java compiler (limits soundness a lot!)

# Static analysis in practice: soundiness

- all "**sound**" static analyses have a ***trusted computing base*** (***TCB***)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between "sound" analyses
  - e.g., TCB for many of my pluggable type systems includes the entire Java compiler (limits soundness a lot!)
  - TCB for some formal verifiers is **very small** (a few kLoC)
    - but these tools (e.g., Coq) are **much harder to use**

# Static analysis in practice: soundiness

- all "**sound**" static analyses have a ***trusted computing base*** (***TCB***)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between "sound" analyses
  - e.g., TCB for many of my pluggable type systems includes the entire Java compiler (limits soundness a lot!)
  - TCB for some formal verifiers is **very small** (a few kLoC)
    - but these tools (e.g., Coq) are **much harder to use**
- soundness theorems also usually make some **assumptions** about the code being analyzed (e.g., no calls to native code, no reflection)

# Static analysis: summary

- static analysis is very good at enforcing **simple rules**
  - **much** better than humans at this
- all interesting semantic properties of programs are **undecidable**, so all static analyses must **approximate**
  - goal in analysis design is to **abstract away unimportant details**, but keep important details
  - **dataflow analysis** is one technique for static analysis
  - trade-offs between false positives, false negatives, analysis time
- soundness & completeness are **possible, but rare**
  - all soundness guarantees come with caveats about the TCB