Static Analysis and LLMs

CS 485/698: AI-Assisted SE

Today’s Agenda

- Team meeting (~10 minutes)
  - Show us your deployed app, now that P6 is finished
- Lecture on static analysis: linters, bug-finders, static types
- In-class activity: improve your team project’s static analysis posture

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!

Common static analyses

- **linters** or other style checkers
  - these are generally **syntactic**: just pattern matching

Common static analyses

- linters or other style checkers
  - these are generally syntactic: just pattern matching
- 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.

Common static analyses

- **linters** or other style checkers
  - these are generally **syntactic**: just pattern matching
- **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 **type systems**
  - may be required (e.g., in Java) or optional/***gradual*** (e.g., TypeScript)

Common static analyses

- linters or other style checkers
  - these are generally syntactic: just pattern matching
- heuristic bug-finding tools backed by dataflow analyses
  - built into modern IDEs, aim for low false positive rates
  - widely used in industry
    - ErrorProne at Goo
      places (including A
- static type systems
  - may be required (e.g., in Java) or optional/*gradual* (e.g., TypeScript)

Next, we're going to look at each of these categories in a little more detail

Linters

Definition: a ***linter*** is a static code-style checker

Linters

Definition: a ***linter*** is a static code-style checker

- Linters **find** style problems, while automated formatters **fix** style problems
  - The line between these two has increasingly blurred

Linters

Definition: a ***linter*** is a static code-style checker

- Linters **find** style problems, while automated formatters **fix** style problems
  - The line between these two has increasingly blurred
- Linters are checking ***syntax***, not semantics
  - syntax = “how it’s written”; semantics = “what it means”
  - Consequence: linters can only find shallow mistakes

Linters

Definition: a ***linter*** is a static code-style checker

- Linters **find** style problems, while automated formatters **fix** style problems
  - The line between these two has increasingly blurred
- Linters are checking ***syntax***, not semantics
  - syntax = “how it’s written”; semantics = “what it means”
  - Consequence: linters can only find shallow mistakes
- Best practice: always use the best linter(s) for your language
  - And always fix all of their complaints (no harm in doing so)

Aside: Undecidability of program properties

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

Aside: 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

Aside: 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**

Aside: 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?

Aside: 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

Aside: 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.

Aside: 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

Aside: 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!

Aside: 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** (*logical* loops - recursion counts!)

Heuristic Bug-finders

- ***Heuristic*** is a fancy word for “best effort”

Heuristic Bug-finders

- *Heuristic* is a fancy word for “best effort”
- Static analysis tools that heuristically find bugs **make no guarantees** about their output
  - warnings can be false alarms
  - real bugs might not lead to warnings

Heuristic Bug-finders

- *Heuristic* is a fancy word for "best effort"
- Static analysis tools that heuristically find bugs **make no guarantees** about their output
  - warnings can be false alarms
  - real bugs might not lead to warnings
- Instead, the goal of these tools is to identify **obvious bugs** with a small number of false alarms

Heuristic Bug-finders

Common use case for these tools: find **obvious security mistakes**

- ***Heuristic*** is a fancy word for "best effort"
- Static analysis tools that heuristically find bugs **make no guarantees** about their output
  - warnings can be false alarms
  - real bugs might not lead to warnings
- Instead, the goal of these tools is to identify **obvious bugs** with a small number of false alarms

Heuristic Bug-finders

Common use case for these tools: find **obvious security mistakes**

- *Heuristic* is a fancy word for "best effort"
- Static analysis tools that heuristically find bugs **make no guarantees** about their output
  - warnings can be false alarms
  - real bugs might not lead to warnings
- Instead, the goal of these tools is to identify **obvious bugs** with a small number of false alarms
- Some work via pattern matching, but most are backed by some form of ***dataflow 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

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
- Analysis designer abstracts what we want to learn (e.g., to help developers) down to a small set of ***abstract values***

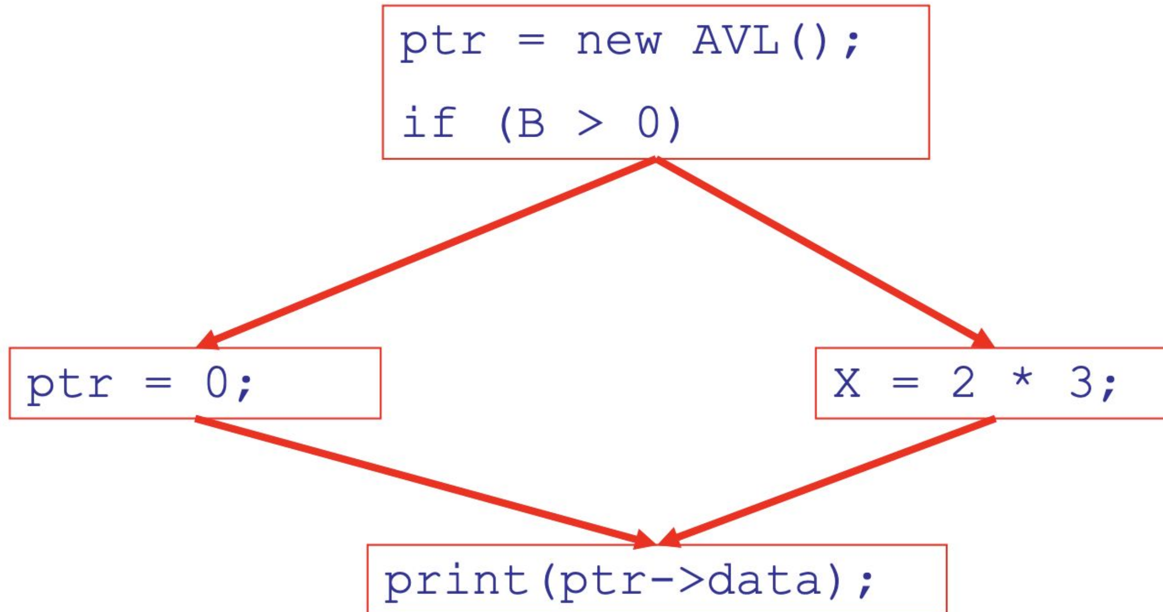
Dataflow analysis

- ***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
- Analysis designer abstracts what we want to learn (e.g., to help developers) down to a small set of ***abstract values***
- Analysis uses **rules** to compute a fixpoint of those abstract values
  - Dataflow analyses take programs as input and produce a set of "dataflow facts" (=abstract values at locations) as output
    - Heuristics in bug-finders convert facts to warnings

Dataflow analysis example: null-pointers

Dataflow analysis example: null-pointers

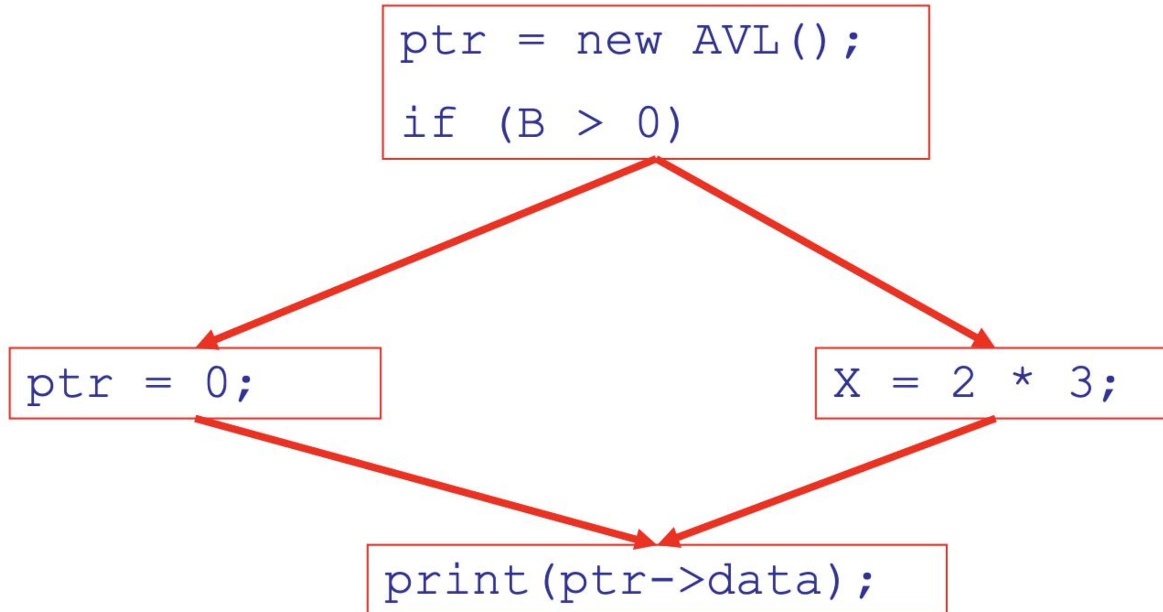
Question: is `ptr` ***always*** null when it is dereferenced?



Dataflow analysis e

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

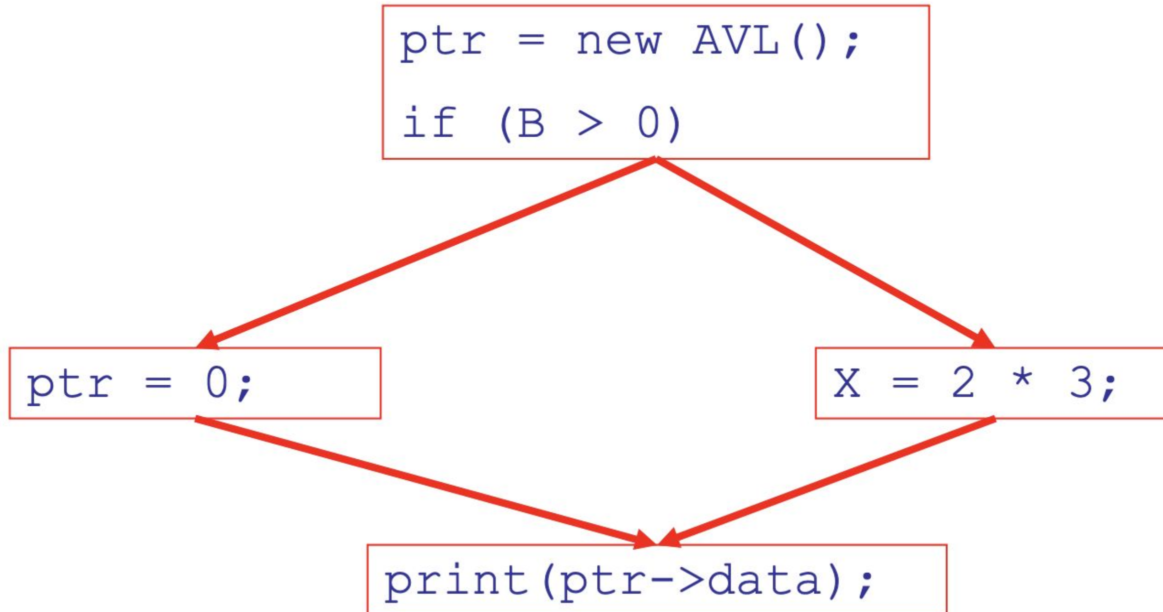
Question: is ptr ***always*** null when it is dereferenced?



Dataflow analysis e

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)

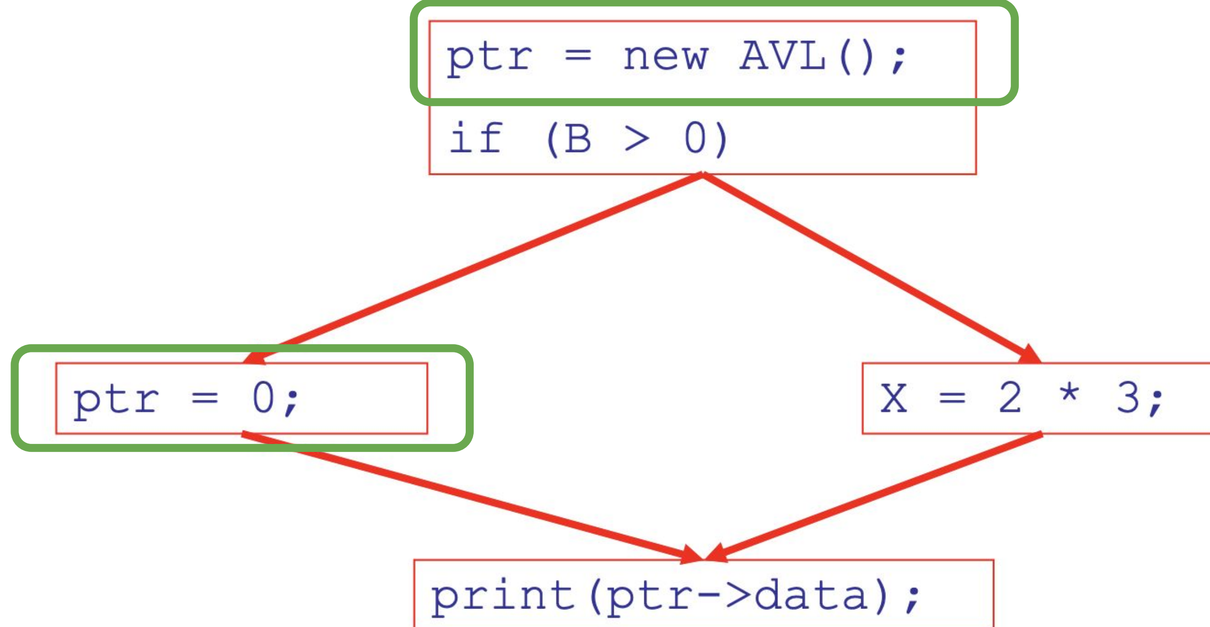
Question: is `ptr` ***always*** null when it is dereferenced?



Dataflow analysis e

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)

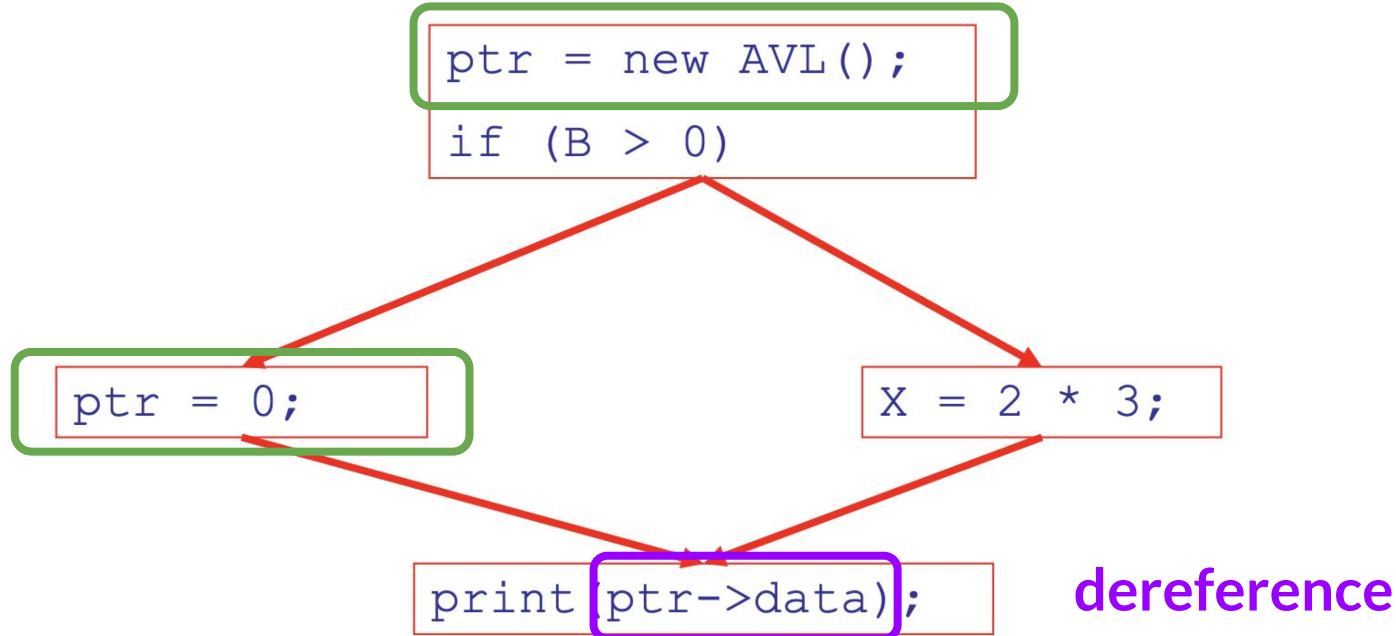
Question: is `ptr` ***always*** null when it is dereferenced?



Dataflow analysis e

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)

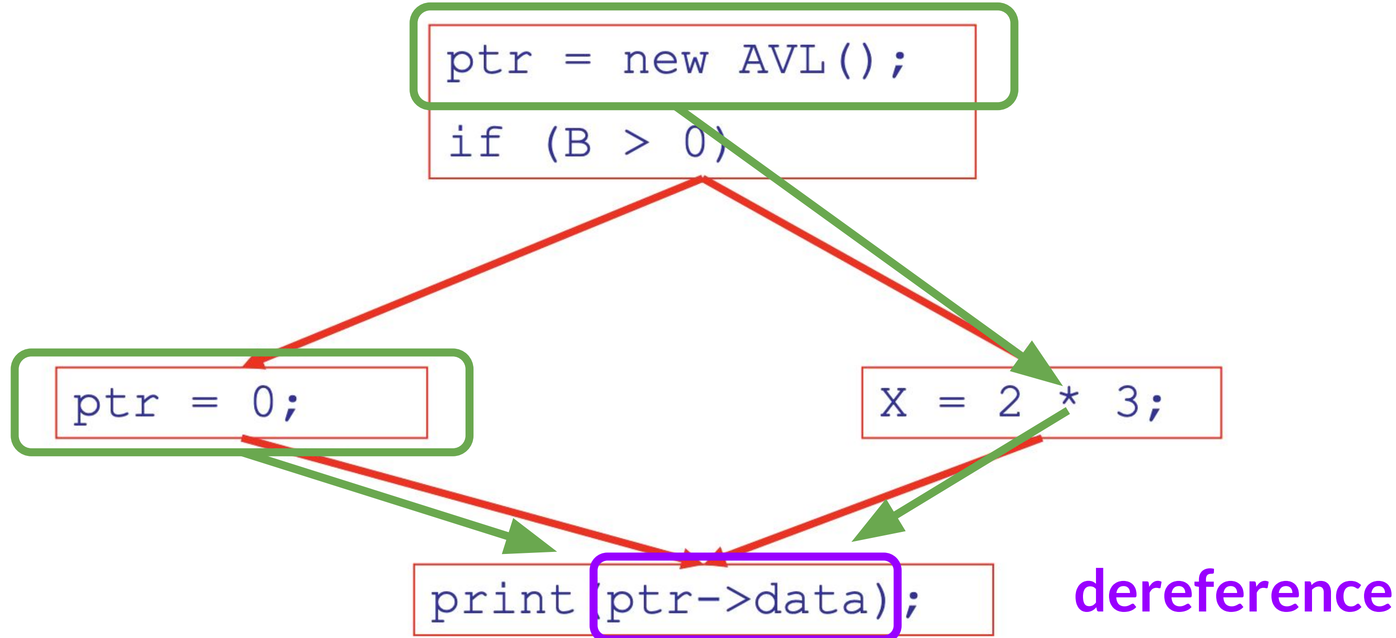
Question: is `ptr` ***always*** null when it is dereferenced?



Dataflow analysis e

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



Type systems

Type systems

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

Type systems

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

- goal of a type system: **prevent errors** at run time due to unexpected values

Type systems

Definition: a ***type system*** is a set of rules t[...] element a ***type***, which is an upper bound [...] that that element can take on at run time

Key idea: make it **impossible** to mix things that shouldn't be mixed

- goal of a type system: **prevent errors** at run time due to unexpected values

Type systems

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

- goal of a type system: **prevent errors** at run time due to unexpected values
- **type theory** is the discipline of math (yes!) that studies the formal properties of type systems

Type systems

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

- goal of a type system: **prevent errors** at run time due to unexpected values
- **type theory** is the discipline of math (yes!) that studies the formal properties of type systems
- most programming languages include some kind of type system
  - exceptions: assembly, Lisp, a few others

Type systems: static vs dynamic

- ***Static types*** are checked before the program runs, typically at compile time

Type systems: static vs dynamic

- ***Static types*** are checked before the program runs, typically at compile time
    - in contrast, ***dynamic types*** are checked by the runtime, during program execution (incurs significant overhead!)

Type systems: static vs dynamic

- ***Static types*** are checked before the program runs, typically at compile time
  - in contrast, ***dynamic types*** are checked by the runtime, during program execution (incurs significant overhead!)
- Static type systems are **just another static analysis**
  - "Abstract values" in dataflow analysis = types

Type systems: static vs dynamic

- ***Static types*** are checked before the program runs, typically at compile time
    - in contrast, ***dynamic types*** are checked by the runtime, during program execution (incurs significant overhead!)
- Static type systems are **just another static analysis**
    - "Abstract values" in dataflow analysis = types
- Most static type systems are ***sound***: "well typed programs don't go wrong"
    - What problem are they preventing?

Type systems: static vs dynamic

- ***Static types*** are checked before the
compile time
  - in contrast, ***dynamic types*** are c
program execution (incurs sign
- Static type systems are **just anothe**
  - "Abstract values" in dataflow analysis = types
- Most static type systems are ***sound***: "well typed programs don't go wrong"
  - What problem are they preventing?

If you already have a static type system, you can extend it with a ***pluggable type system*** to prevent more bugs

- e.g., nullness in Java

LLMs and Static Analysis

- Traditionally, the **biggest barriers** to wider adoption of static analysis were:
    - too many false alarms
    - too much effort to apply it to a codebase (e.g., writing types)

LLMs and Static Analysis

- Traditionally, the **biggest barriers** to wider adoption of static analysis were:
    - too many false alarms
    - too much effort to apply it to a codebase (e.g., writing types)
- LLMs change the calculus here, because unlike a human, an LLM has **infinite patience**:
    - doesn’t get bored wading through false alarms
    - doesn’t get annoyed about the need to write types

LLMs and Static Analysis

- Traditionally, the **biggest barriers** to wider adoption of static analysis were:
  - too many false alarms
  - too much effort to apply it to a codebase (e.g., writing types)
- LLMs change the calculus here, because unlike a human, an LLM has **infinite patience**:
  - doesn't get bored wading through false alarms
  - doesn't get annoyed about the need to write types
- So, when using a coding agent, my advice: **always use the best static analyses available**

In-class activity: statically analyze your project

- Each project team should have one **blue letter**
    - If your team has a "static analysis expert", give them the blue letter (swapping if necessary)
    - If you're on a team by yourself, make sure your letter is blue
- If you have a **red letter**, you should find the person with the corresponding blue letter (note there are duplicate red letters)

In-class activity: statically analyze your project

- Today’s goal: get an LLM to fully enroll the **blue team member**’s course project into a new static analysis and add it to CI
  - by the end of the class, every team project should have a new open PR that adds a new static analysis tool + resolves warnings
- Fun rule to make it more exciting: **only red team members can type**
  - blue team member’s role is advisory only
  - your choice of whose computer/LLM to use
- Suggestions for static analyzers on the following slides, by language
  - Frontend or backend is ok (but we suggest you pick one)
- Challenge problem: JavaScript -> TypeScript

JavaScript/TypeScript Static Analyzers

- Linters:
  - ESLint
  - Standard JS
  - JSHint
- Bug-finders:
  - Insider
  - SemGrep
  - SonarJS
- Static Types:
  - TypeScript
  - Closure
  - Flow
  - Hegel

Python Static Analyzers

- Linters:
  - ruff
  - Pylint
  - Flake8
- Bug-finders:
  - SemGrep
  - CodeQL
  - Bandit
  - Pysa
- Static Types:
  - Mypy
  - Pyright
  - Pyre

Java Static Analyzers

- Linters:
  - Checkstyle
  - SpotBugs
  - ErrorProne
- Bug-finders:
  - Infer
  - SemGrep
  - CodeQL
- (More) Static Types:
  - Checker Framework
  - NullAway

Go Static Analyzers

- Linters:
  - just run `go vet` (if you're not already)
  - golangci-lint
  - staticcheck
- Bug-finders:
  - gosec
  - govulncheck
  - errcheck
- Static Types:
  - I don't know of any extensions to the type system, so focus on linters and bug-finders.

Dart Static Analysis

- I'm not particularly familiar with Dart, so if you're the one group working in the language you're going to have to figure it out for yourself (sorry). Here are some links to get you started:
  - https://dart.dev/tools/analysis
  - https://dart.dev/tools/linter-rules/all

Wrapup and Reminders

- For grad students: don’t forget about A7
  - remember that you have to prepare to lead a discussion on your chosen paper
  - undergrads: it is not too late to sign up for an A7 topic, for extra credit
- Only 6 of you have filled out the course evaluation form so far
  - It doesn’t close until May 7, so you have time
  - *Especially* for a special topics course, I care a lot about your feedback! Please do actually do it.