Pluggable Types

Martin Kellogg

Course Announcements

- PA4c1 is due today

Course Announcements

- PA4c1 is due today
- I’m out of town for the rest of the week at ICSE

Course Announcements

- PA4c1 is due today
- I’m out of town for the rest of the week at ICSE
  - I will not hold office hours on Wednesday

Course Announcements

- PA4c1 is due today
- I’m out of town for the rest of the week at ICSE
  - I will not hold office hours on Wednesday
  - My PhD student Erfan Arvan will lecture on Wednesday
    - please be nice to him!

Course Announcements

- PA4c1 is due today
- I'm out of town for the rest of the week at ICSE
  - I will not hold office hours on Wednesday
  - My PhD student Erfan Arvan will lecture on Wednesday
    - please be nice to him!
  - If you want to see a test case before next Monday, you need to ask right after class
    - my flight is at 3pm, so I'm going to go to the airport ~12:30

Agenda

- Motivation
- Limitations of type systems
  - …or lack thereof (Curry-Howard Correspondence)
- Pluggable Types
  - Formalism
  - Practical example (Nullability in Java)
  - Discussion of state-of-the-art (if time)

Review: What is a type system, anyway?

Review: What is a type system, anyway?

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

Review: What is a type system, anyway?

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

Review: What is a type system, anyway?

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
  - a type can also encode the set of ***valid operations*** on values of that type

Review: What is a type system, anyway?

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
  - a type can also encode the set of ***valid operations*** on values of that type
- many programming languages (including Cool) check types at compile time (***static types***)

Review: What is a type system, anyway?

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
  - a type can also encode the set of ***valid operations*** on values of that type
- many programming languages (including Cool) check types at compile time (***static types***)
  - some languages instead check types at run time (***dynamic types***)

Motivation

- Traditional static type systems like Cool's are already **widely-used** by software engineers

Motivation

- Traditional static type systems like Cool's are already **widely-used** by software engineers
  - in practice, type systems are one of the few common techniques for improving correctness
    - along with testing and code review

Motivation

- Traditional static type systems like Cool's are already **widely-used** by software engineers
  - in practice, type systems are one of the few common techniques for improving correctness
    - along with testing and code review
- However, traditional type systems prevent **some but not all** run-time errors

Motivation

- Traditional static type systems like Cool's are already **widely-used** by software engineers
  - in practice, type systems are one of the few common techniques for improving correctness
    - along with testing and code review
- However, traditional type systems prevent **some but not all** run-time errors
  - prevented: unsuccessful vtable lookups, missing fields, mixing up incompatible values (e.g., integers and strings), etc.

Motivation

- Traditional static type systems like Cool's are already **widely-used** by software engineers
  - in practice, type systems are one of the few common techniques for improving correctness
    - along with testing and code review
- However, traditional type systems prevent **some but not all** run-time errors
  - prevented: unsuccessful vtable lookups, missing fields, mixing up incompatible values (e.g., integers and strings), etc.
  - not prevented: out-of-bounds array accesses, null-pointer dereferences, memory leaks, mismatched units, etc.

Motivation: Verification

Motivation: Verification

- Many techniques for **verification** of computer programs have been proposed
  - e.g., Floyd-Hoare logic, guarded commands, abstract interpretation, etc. (ask in office hours)

Motivation: Verification

- Many techniques for **verification** of computer programs have been proposed
  - e.g., Floyd-Hoare logic, guarded commands, abstract interpretation, etc. (ask in office hours)
- Unlike static type systems, these techniques **have not been widely-adopted** outside of niche industries :(

Motivation: Verification

- Many techniques for **verification** of computer programs have been proposed
  - e.g., Floyd-Hoare logic, guarded commands, abstract interpretation, etc. (ask in office hours)
- Unlike static type systems, these techniques **have not been widely-adopted** outside of niche industries :(
  - and computer programs remain full of bugs that a verifier could have prevented…

Motivation: Verification

- Many techniques for **verification** of computer programs have been proposed
  - e.g., Floyd-Hoare logic, guarded commands, abstract interpretation, etc. (ask in office hours)
- Unlike static type systems, these techniques **have not been widely-adopted** outside of niche industries :(
  - and computer programs remain full of bugs that a verifier could have prevented…
- One of the biggest obstacles to the adoption of verification technologies in practice is the need to write **specifications**
  - e.g., pre- and post-conditions, loop invariants, etc.

Pluggable Types as Verification

- **Observation**: programmers have already shown that they are willing to write down static types.

Pluggable Types as Verification

- **Observation**: programmers have already shown that they are willing to write down static types.
  - **Insight:** What if we allowed them to write down the specifications that we need for verification as types?

Pluggable Types as Verification

- **Observation**: programmers have already shown that they are willing to write down static types.
  - **Insight:** What if we allowed them to write down the specifications that we need for verification as types?
  - Pluggable type systems allow us to do just that

Pluggable Types as Verification

- **Observation**: programmers have already shown that they are willing to write down static types.
  - **Insight:** What if we allowed them to write down the specifications that we need for verification as types?
  - Pluggable type systems allow us to do just that
- More formally, a ***pluggable type system*** extends a host type system with a set of ***type qualifiers*** that model a property of interest (usually “the program has no bugs of kind X”, for some X)

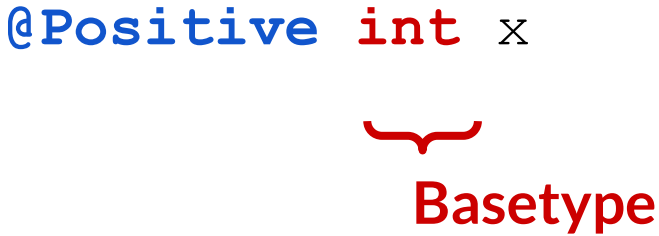
Pluggable Types as Verification

- **Observation**: programmers have already shown that they are willing to write down static types.
  - **Insight:** What if we allowed them to write down the specifications that we need for verification as types?
  - Pluggable type systems allow us to do just that
- More formally, a ***pluggable type system*** extends a host type system with a set of ***type qualifiers*** that model a property of interest (usually “the program has no bugs of kind X”, for some X)
  - each type qualifier **refines** the type in question
    - that is, each qualifier is “more specific” than the base type (i.e., a subtype)

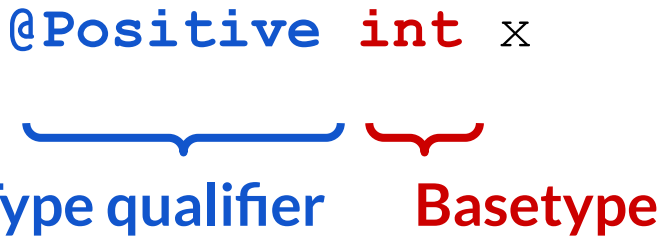
Pluggable Types: Terminology

```
@Positive int x
```

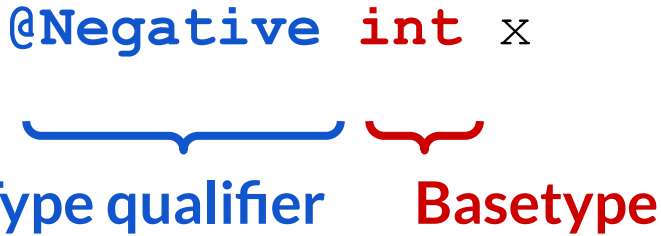
Pluggable Types: Terminology



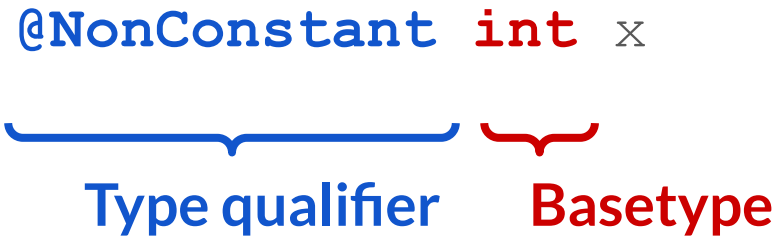
Pluggable Types: Terminology



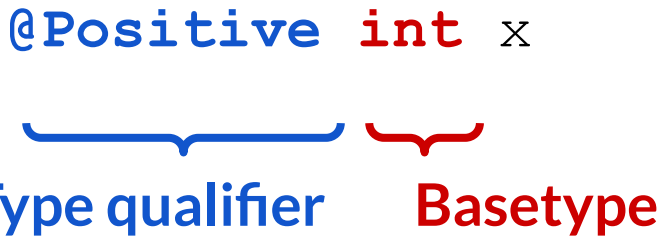
Pluggable Types: Terminology



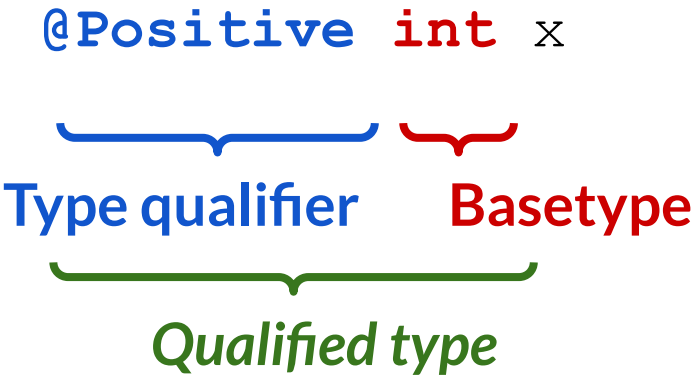
Pluggable Types: Terminology



Pluggable Types: Terminology



Pluggable Types: Terminology



Pluggable Types: One Slide Summary

- Developers already use static type systems, so they're familiar with the general idea of types => **relatively easy to use** (compared to other verification techniques)

Pluggable Types: One Slide Summary

- Developers already use static type systems, so they're familiar with the general idea of types => **relatively easy to use** (compared to other verification techniques)
- Type qualifiers **encode** property of interest
  - effectively a "second" type system

Pluggable Types: One Slide Summary

- Developers already use static type systems, so they're familiar with the general idea of types => **relatively easy to use** (compared to other verification techniques)
- Type qualifiers **encode** property of interest
  - effectively a "second" type system
- Qualified types are like a **Cartesian product** of a type from the pluggable type system and a type from the base type system

Pluggable Types: One Slide Summary

- Developers already use static type systems, so they're familiar with the general idea of types => **relatively easy to use** (compared to other verification techniques)
- Type qualifiers **encode** property of interest
  - effectively a "second" type system
- Qualified types are like a **Cartesian product** of a type from the pluggable type system and a type from the base type system
- Typechecking is naturally **modular** = fast
  - but this comes at a cost: programmers need to write types

Pluggable Types: Formalism

Pluggable Types: Formalism

- Qualified types are like a **Cartesian product** of a type from the pluggable type system and a type from the base type system
  - What does this actually mean?

Pluggable Types: Formalism

- Qualified types are like a **Cartesian product** of a type from the pluggable type system and a type from the base type system
  - What does this actually mean?
- Analogy: each pluggable type system is like a **dimension** in a traditional coordinate system:

Pluggable Types: Formalism

- Qualified types are like a **Cartesian product** of a type from the pluggable type system and a type from the base type system
  - What does this actually mean?
- Analogy: each pluggable type system is like a **dimension** in a traditional coordinate system:
  - the base type system is the first dimension

Pluggable Types: Formalism

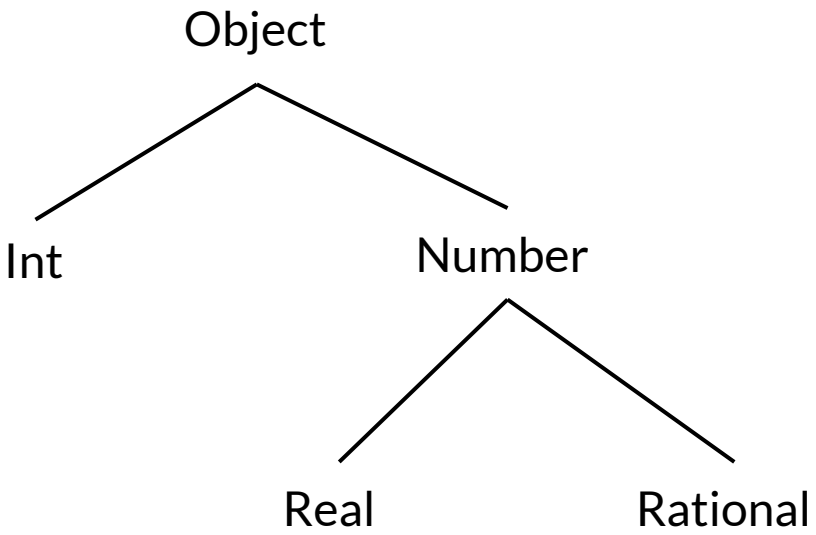
- Qualified types are like a **Cartesian product** of a type from the pluggable type system and a type from the base type system
  - What does this actually mean?
- Analogy: each pluggable type system is like a **dimension** in a traditional coordinate system:
  - the base type system is the first dimension
  - each additional qualifier adds one more dimension

Pluggable Types: Formalism

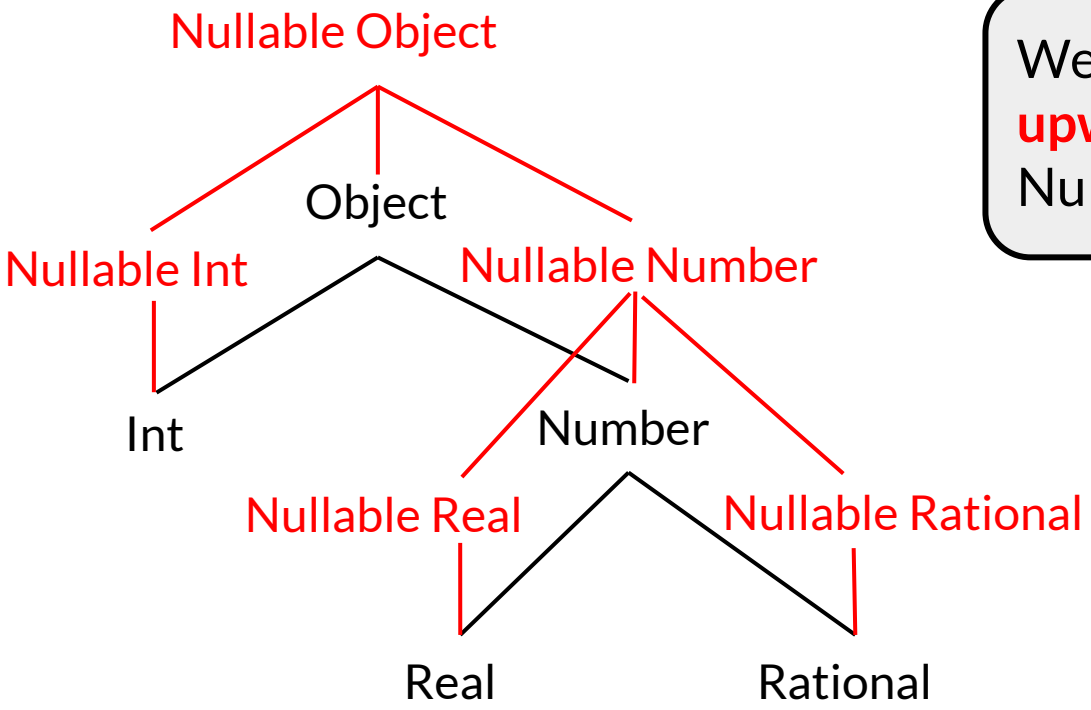
- Qualified types are like a **Cartesian product** of a type from the pluggable type system and a type from the base type system
  - What does this actually mean?
- Analogy: each pluggable type system is like a **dimension** in a traditional coordinate system:
  - the base type system is the first dimension
  - each additional qualifier adds one more dimension
  - let’s see an example

Pluggable Types: Formalism: Products

Consider this simple type hierarchy.



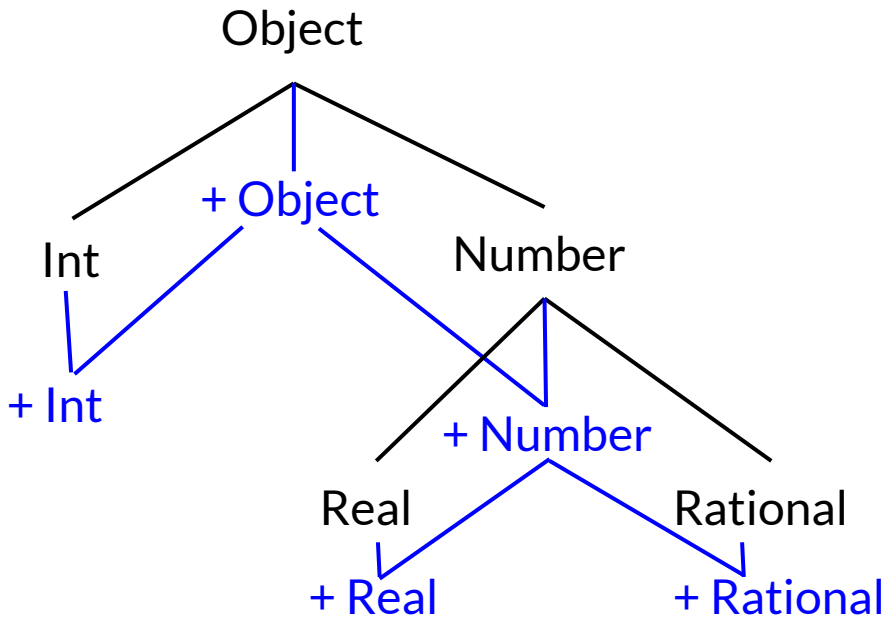
Pluggable Types: Formalism: Products



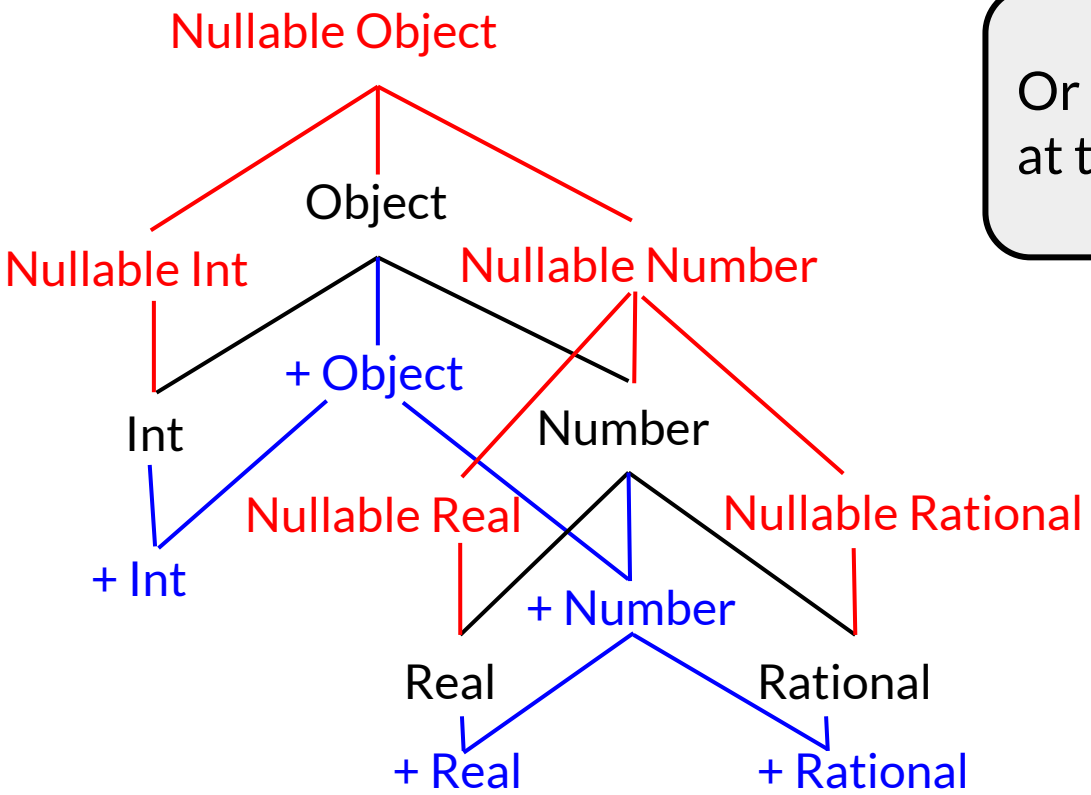
We can extend it **upwards** with a Nullable qualifier…

Pluggable Types: Formalism: Products

Or we can extend it **downwards** with a positive qualifier…



Pluggable Types: Formalism: Products



Or we can do **both** at the same time!

Pluggable Types: Product Types

- More formally, in a pluggable type system, every type is a ***product type*** of a type qualifier and a base type.

Pluggable Types: Product Types

- More formally, in a pluggable type system, every type is a ***product type*** of a type qualifier and a base type.
  - A product type is a **logical "and" of two types** (more on this in a moment)

Pluggable Types: Product Types

- More formally, in a pluggable type system, every type is a ***product type*** of a type qualifier and a base type.
  - A product type is a **logical "and" of two types** (more on this in a moment)
- Instead of writing **x : T**, we write **x : Q T**

Pluggable Types: Product Types

- More formally, in a pluggable type system, every type is a ***product type*** of a type qualifier and a base type.
  - A product type is a **logical "and" of two types** (more on this in a moment)
- Instead of writing **x : T**, we write **x : Q T**
  - **T** is the base type, **Q** is the type qualifier from the pluggable type system of interest

Pluggable Types: Product Types

- More formally, in a pluggable type system, every type is a ***product type*** of a type qualifier and a base type.
  - A product type is a **logical "and" of two types** (more on this in a moment)
- Instead of writing **x : T**, we write **x : Q T**
  - **T** is the base type, **Q** is the type qualifier from the pluggable type system of interest
- We can extend this formalism arbitrarily, e.g., by writing **x : Q_1 Q_2 T**
  - (As long as Q_1 and Q_2 are from different type systems)

Pluggable Types: Product Types

- More formally, in a plugg
  type of a type qualifier an
  - A product type is a **lo**
    moment)
- Instead of writing **x : T**, we write **x : Q T**
  - **T** is the base type, **Q** is the type qualifier from the pluggable type system of interest
- We can extend this formalism arbitrarily, e.g., by writing **x : Q_1 Q_2 T**
  - (As long as Q_1 and Q_2 are from different type systems)

Note that there is not really anything special about **T** in this formalism - we can view the base type system as just another pluggable type that's "in use by default"

Aside: Curry-Howard Correspondence

- I hinted that product types are “like” a logical AND gate

Aside: Curry-Howard Correspondence

- I hinted that product types are “like” a logical AND gate
- This relationship is one of the cornerstones of the ***Curry-Howard correspondence*** between mathematical proofs and computer programs

Aside: Curry-Howard Correspondence

- I hinted that product types are “like” a logical AND gate
- This relationship is one of the cornerstones of the ***Curry-Howard correspondence*** between mathematical proofs and computer programs
  - There is an **isomorphism** between the proof systems and the models of computation: these two families of formalisms can be considered as identical
    - (an ***isomorphism*** is a one-to-one correspondence between the mathematical structures of two objects)

Aside: Curry-Howard Correspondence

- I hinted that product types are “like” a logical AND gate
- This relationship is one of the cornerstones of the ***Curry-Howard correspondence*** between mathematical proofs and computer programs
  - There is an **isomorphism** between the proof systems and the models of computation: these two families of formalisms can be considered as identical
    - (an ***isomorphism*** is a one-to-one correspondence between the mathematical structures of two objects)
  - In other words: a proof is a program, and the formula it proves is the type for the program

Aside: Curry-Howard Correspondence

- The Curry-Howard Correspondence has **deep implications**:

Aside: Curry-Howard Correspondence

- The Curry-Howard Correspondence has **deep implications**:
  - It is possible to “run” proofs, as we would computer programs

Aside: Curry-Howard Correspondence

- The Curry-Howard Correspondence has **deep implications**:
  - It is possible to “run” proofs, as we would computer programs
    - This implies that it’s possible to “co-develop” a program and the proof of its correctness

Aside: Curry-Howard Correspondence

- The Curry-Howard Correspondence has **deep implications**:
  - It is possible to “run” proofs, as we would computer programs
    - This implies that it’s possible to “co-develop” a program and the proof of its correctness
    - ***Interactive theorem provers*** like Rocq or Lean operationalize this idea

Aside: Curry-Howard Correspondence

- The Curry-Howard Correspondence has **deep implications**:
  - It is possible to “run” proofs, as we would computer programs
    - This implies that it’s possible to “co-develop” a program and the proof of its correctness
    - ***Interactive theorem provers*** like Rocq or Lean operationalize this idea
  - A type system is just a way to **write down a theorem** about the program

Aside: Curry-Howard Correspondence

- The Curry-Howard Correspondence has **deep implications**:
  - It is possible to "run" proofs, as we would computer programs
    - This implies that it's possible to "co-develop" a program and the proof of its correctness
    - ***Interactive theorem provers*** like Rocq or Lean operationalize this idea
  - A type system is just a way to **write down a theorem** about the program
    - Different type systems let us write down **different theorems**

Aside: Curry-Howard Correspondence

- The Curry-Howard Correspondence has **deep implications**:
  - It is possible to "run" proofs, as we would computer programs
    - This implies that it's possible to "
      the proof of its correctness
    - ***Interactive theorem provers*** like Rocq or Lean operationalize this idea
  - A type system is just a way to **write down a theorem** about the program
    - Different type systems let us write down **different theorems**

For more on this topic, ask in office hours and/or take CS 735 with me in Sp26.

Adding Pluggable Types to Cool

Adding Pluggable Types to Cool

- The modifications to most of the type rules to support a pluggable type system are straightforward: just add a qualifier term to each type:
  - **T -> Q T**

Adding Pluggable Types to Cool

- The modifications to most of the type rules to support a pluggable type system are straightforward: just add a qualifier term to each type:
  - **T -> Q T**
  - Effectively, this requires the programmer to write a type qualifier everywhere they write a type

Adding Pluggable Types to Cool

- The modifications to most of the type rules to support a pluggable type system are straightforward: just add a qualifier term to each type:
  - **T -> Q T**
  - Effectively, this requires the programmer to write a type qualifier everywhere they write a type
- The **subtyping** and **least upper bound** rules are bit trickier (but not much).

Adding Pluggable Types to Cool

- The modifications to most of the type rules to support a pluggable type system are straightforward: just add a qualifier term to each type:
  - **T -> Q T**
  - Effectively, this requires the programmer to write a type qualifier everywhere they write a type
- The **subtyping** and **least upper bound** rules are bit trickier (but not much).
  - We need to account for all possible cases

Adding Pluggable Types to Cool: Subtyping

Adding Pluggable Types to Cool: Subtyping

- Subtyping can generally be split into two parts:

Adding Pluggable Types to Cool: Subtyping

- Subtyping can generally be split into two parts:
  - the base types part, which just defers to the old subtyping rule

Adding Pluggable Types to Cool: Subtyping

- Subtyping can generally be split into two parts:
  - the base types part, which just defers to the old subtyping rule
  - the type qualifier part

Adding Pluggable Types to Cool: Subtyping

- Subtyping can generally be split into two parts:
  - the base types part, which just defers to the old subtyping rule
  - the type qualifier part
    - for this part, assume that the type qualifiers are arranged into a **lattice** (sound familiar?)

Adding Pluggable Types to Cool: Subtyping

- Subtyping can generally be split into two parts:
  - the base types part, which just defers to the old subtyping rule
  - the type qualifier part
    - for this part, assume that the type qualifiers are arranged into a **lattice** (sound familiar?)

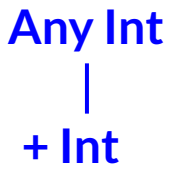
Nullable T

|

Nonnull T

Adding Pluggable Types to Cool: Subtyping

- Subtyping can generally be split into two parts:
  - the base types part, which just defers to the old subtyping rule
  - the type qualifier part
    - for this part, assume that the type qualifiers are arranged into a **lattice** (sound familiar?)



Adding Pluggable Types to Cool: Subtyping

- Subtyping can generally be split into two parts:
  - the base types part, which just defers to the old subtyping rule
  - the type qualifier part
    - for this part, assume that the type qualifiers are arranged into a **lattice** (sound familiar?)
    - then, we can just read subtyping information from the lattice

Nullable T
|
Nonnull T

Any Int
|
+ Int

Adding Pluggable Types to Cool: Subtyping

- Subtyping can generally be split into two parts:
  - the base types part, which just defers to the old subtyping rule
  - the type qualifier part

the type

to a **lattice**

typing

ce

If this looks similar to **abstract interpretation**, that's because it is (Cousot proved it in POPL 1997). Pluggable types can be viewed as an alternative syntax for abstract interpretation.

Nullable T
|
Nonnull T

Any Int
|
+ Int

Adding Pluggable Types to Cool: LUB

Adding Pluggable Types to Cool: LUB

- Least upper bounds can also be split between the base types and the qualifiers:
  - LUB(Q_1 T_1, Q_2 T_2) = LUB(Q_1, Q_2) LUB(T_1, T_2)

Adding Pluggable Types to Cool: LUB

- Least upper bounds can also be split between the base types and the qualifiers:
  - LUB(Q_1 T_1, Q_2 T_2) = LUB(Q_1, Q_2) LUB(T_1, T_2)
- Some pluggable type systems might have **interactions** between qualifiers and base types.

Adding Pluggable Types to Cool: LUB

- Least upper bounds can also be split between the base types and the qualifiers:
  - LUB(Q_1 T_1, Q_2 T_2) = LUB(Q_1, Q_2) LUB(T_1, T_2)
- Some pluggable type systems might have **interactions** between qualifiers and base types.
  - e.g., it might not make sense to have a “positive string”

Adding Pluggable Types to Cool: LUB

- Least upper bounds can also be split between the base types and the qualifiers:
  - LUB(Q_1 T_1, Q_2 T_2) = LUB(Q_1, Q_2) LUB(T_1, T_2)
- Some pluggable type systems might have **interactions** between qualifiers and base types.
  - e.g., it might not make sense to have a “positive string”
  - in practice, this is fine - just treat like bottom

Pluggable Types: Defaulting

Pluggable Types: Defaulting

- While it's plausible to require programmers to write a type qualifier at each type use, in practice that's **inconvenient**

Pluggable Types: Defaulting

- While it's plausible to require programmers to write a type qualifier at each type use, in practice that's **inconvenient**
  - instead, we'd like to pick **sensible defaults** for unqualified types

Pluggable Types: Defaulting

- While it's plausible to require programmers to write a type qualifier at each type use, in practice that's **inconvenient**
  - instead, we'd like to pick **sensible defaults** for unqualified types
- For example, consider a nullability pluggable type system with two qualifiers: **Nullable** and **NonNull**.

Pluggable Types: Defaulting

- While it's plausible to require programmers to write a type qualifier at each type use, in practice that's **inconvenient**
  - instead, we'd like to pick **sensible defaults** for unqualified types
- For example, consider a nullability pluggable type system with two qualifiers: **Nullable** and **NonNull**.
  - if we see an unqualified type **T**, should we treat it as:
    - **Nullable T**?
    - or **NonNull T**?

Pluggable Types: Defaulting

- While it's plausible to require programmers to write a type qualifier at each type use, in practice that's **inconvenient**
  - instead, we'd like to pick **sensible defaults** for unqualified types
- For example, consider a nullability pluggable type system with two qualifiers: **Nullable** and **NonNull**.
  - if we see an unqualified type **T**, should we treat it as:
    - **Nullable T**?
    - or **NonNull T**?
  - interestingly, **either choice is correct**! (Why?)

Pluggable Types: Defaulting: CLIMB-to-top

- Practical pluggable type systems use a specific defaulting scheme called ***CLIMB-to-top***

Pluggable Types: Defaulting: CLIMB-to-top

- Practical pluggable type systems use a specific defaulting scheme called ***CLIMB-to-top***
  - this approach uses a simple **dataflow analysis** to propagate types within a method body

Pluggable Types: Defaulting: CLIMB-to-top

- Practical pluggable type systems use a specific defaulting scheme called ***CLIMB-to-top***
  - this approach uses a simple **dataflow analysis** to propagate types within a method body
    - also **saves user effort**!

Pluggable Types: Defaulting: CLIMB-to-top

- Practical pluggable type systems use a specific defaulting scheme called ***CLIMB-to-top***
  - this approach uses a simple **dataflow analysis** to propagate types within a method body
    - also **saves user effort**!
  - it defaults any location where dataflow discovers a qualifier to that qualifier, regardless of the “usual” defaulting rules

Pluggable Types: Defaulting: CLIMB-to-top

- Practical pluggable type systems use a specific defaulting scheme called ***CLIMB-to-top***
  - this approach uses a simple **dataflow analysis** to propagate types within a method body
    - also **saves user effort**!
  - it defaults any location where dataflow discovers a qualifier to that qualifier, regardless of the “usual” defaulting rules
    - e.g., in `Object x = null`, `x` will be **Nullable** even if the usual default is **NonNull**

Pluggable Types: Defaulting: CLIMB-to-top

- Practical pluggable type systems use a specific defaulting scheme called ***CLIMB-to-top***
  - this approach uses a simple **dataflow analysis** to propagate types within a method body
    - also **saves user effort**!
  - it defaults any location where dataflow discovers a qualifier to that qualifier, regardless of the “usual” defaulting rules
    - e.g., in `Object x = null`, `x` will be **Nullable** even if the usual default is **NonNull**
  - there are a number of other subtleties which I’ll elide for time

Pluggable Types: Custom Type Rules

- Many pluggable type systems exist to enforce some kind of **custom type rule** (which might replace an existing rule)

Pluggable Types: Custom Type Rules

- Many pluggable type systems exist to enforce some kind of **custom type rule** (which might replace an existing rule)
  - for example, a nullability type system might have a custom rule that forbids dereferencing a type with a nullable qualifier:

$$\frac{\Gamma \vdash e_1 : \text{NonNull } T \qquad \Gamma \vdash e_2 : Q\ T}{\Gamma \vdash e_1.e_2 : Q\ T}\ \textbf{[Deref]}$$

Pluggable Types: Custom Type Rules

- Many pluggable type systems exist to enforce some kind of **custom type rule** (which might replace an existing rule)
  - for example, a nullability type system might have a custom rule that forbids dereferencing a type with a nullable qualifier:

$$\frac{\Gamma \vdash e_1 : \text{NonNull } T \qquad \Gamma \vdash e_2 : Q\ T}{\Gamma \vdash e_1.e_2 : Q\ T}\ \textbf{[Deref]}$$

- These rules can be arbitrary, but generally the type system designer chooses them to make some **soundness proof** work

Pluggable Types: Examples

Pluggable Types: Examples

- Null dereferences (@NonNull)
  - >200 errors in Google Collections, javac, ...

Pluggable Types: Examples

- Null dereferences (@NonNull)
  - >200 errors in Google Collections, javac, ...
- Equality tests (@Interned)
  - >200 problems in Xerces, Lucene, ...

Pluggable Types: Examples

- Null dereferences (@NonNull)
  - >200 errors in Google Collections, javac, ...
- Equality tests (@Interned)
  - >200 problems in Xerces, Lucene, ...
- Concurrency / locking (@GuardedBy)
  - >500 errors in BitcoinJ, Derby, Guava, Tomcat, ...

Pluggable Types: Examples

- Null dereferences (@NonNull)
  - >200 errors in Google Collections, javac, ...
- Equality tests (@Interned)
  - >200 problems in Xerces, Lucene, ...
- Concurrency / locking (@GuardedBy)
  - >500 errors in BitcoinJ, Derby, Guava, Tomcat, ...
- Array bounds (@IndexFor, @NonNegative, etc.)
  - defects in Google Guava, JFreeChart, …

Pluggable Types: Examples

- Null dereferences (@NonNull)
  - >200 errors in Google Collections, javac, ...
- Equality tests (@Interned)
  - >200 problems in Xerces, Lucene, ...
- Concurrency / locking (@GuardedBy)
  - >500 errors in BitcoinJ, Derby, Guava, Tomcat, ...
- Array bounds (@IndexFor, @NonNegative, etc.)
  - defects in Google Guava, JFreeChart, …
- Resource leaks (@MustCall)
  - defects in Zookeeper, Hadoop, HBase, …

Pluggable Types: Examples

- Null dereferences (@NonNull)
  - >200 errors in Google Collections, javac, ...
- Equality tests (@Interned)
  - >200 problems in Xer
- Concurrency / locking (@G
  - >500 errors in Bitcoin
- Array bounds (@IndexFor, @NonNegative, etc.)
  - defects in Google Guava, JFreeChart, …
- Resource leaks (@MustCall)
  - defects in Zookeeper, Hadoop, HBase, …

These last two were part of my PhD work!

Trivia Break: History of Mathematics

This Ancient Greek mathematician, physicist, engineer, astronomer, and inventor from the ancient city of Syracuse in Sicily is regarded as the greatest mathematician of ancient history. He anticipated modern calculus and analysis by applying the concept of the infinitely small and the method of exhaustion to derive and rigorously prove many geometrical theorems. These include the area of a circle, the surface area and volume of a sphere, the area of an ellipse, the area under a parabola, the volume of a segment of a paraboloid of revolution, the volume of a segment of a hyperboloid of revolution, and the area of a spiral.

Pluggable Types: Deeper Dive Into Nullability

Pluggable Types: Deeper Dive Into Nullability

- Let’s consider one specific example in detail: nullability in Java
  - Tony Hoare’s “**billion-dollar mistake**” etc

Pluggable Types: Deeper Dive Into Nullability

- Let's consider one specific example in detail: nullability in Java
  - Tony Hoare's "**billion-dollar mistake**" etc
- While my examples in this section will all concern nullability, many of the **principles** at play are applicable to any pluggable typechecker

Pluggable Types: Deeper Dive Into Nullability

- Let's consider one specific example in detail: nullability in Java
  - Tony Hoare's "**billion-dollar mistake**" etc
- While my examples in this section will all concern nullability, many of the **principles** at play are applicable to any pluggable typechecker
  - I'll try to point these out as we come to them, but as a hint about exam questions - asking you about how one of these principles might apply to a **different** type system would be fair game

Nullability: Whose Bug Is This?

- Where is the defect in the following code?

Nullability: Whose Bug Is This?

- Where is the defect in the following code?
  - Whose fault: library implementer or client?

Nullability: Whose Bug Is This?

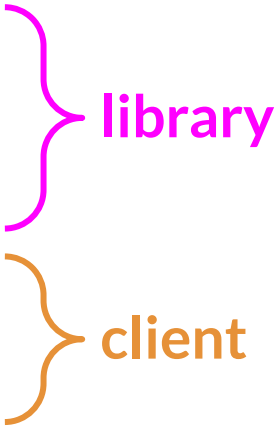
- Where is the defect in the following code?
  - Whose fault: library implementer or client?

```
String op(Data in) {
  return "transform: " + in.getF();
}
```

Nullability: Whose Bug Is This?

- Where is the defect in the following code?
  - Whose fault: library implementer or client?

```
String op(Data in) {
  return "transform: " + in.getF();
}

String s = op(null);
```



Nullability: Whose Bug Is This?

- Where is the defect in the following code?
  - Whose fault: library implementer or client?

```
String op(Data in) {
  return "transform: " + in.getF();
}

String s = op(null);
```

Cannot decide without a **specification**!

Nullability: Whose Bug Is This?

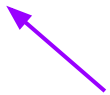
- **Specification option 1**:
  - Non-null parameter

```
String op(@NonNull Data in) {
  return "transform: " + in.getF();
}

String s = op(null);
```

Nullability: Whose Bug Is This?

- **Specification option 1**:
  - Non-null parameter

```
String op(@NonNull Data in) {
  return "transform: " + in.getF();
}

String s = op(null);
```



with this specification, defect is here!

Nullability: Whose Bug Is This?

- Specification option 2:
  - Nullable parameter

```
String op(@Nullable Data in) {
  return "transform: " + in.getF();
}

String s = op(null);
```

Nullability: Whose Bug Is This?

- Specification option 2:
  - Nullable parameter

```
String op(@Nullable Data in) {
  return "transform: " + in.getF();
}

String s = op(null);
```

with this specification, defect is here, instead!

Nullability: Basic Type System

Nullability: Basic Type System

- Two **type qualifiers** (= abstract values):

Nullability: Basic Type System

- Two **type qualifiers** (= abstract values):
  - **@Nullable** = might be null

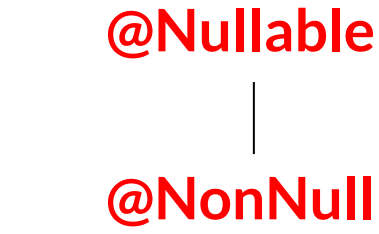
Nullability: Basic Type System

- Two **type qualifiers** (= abstract values):
  - **@Nullable** = might be null
  - **@NonNull** = definitely not null

Nullability: Basic Type System

- Two **type qualifiers** (= abstract values):
  - **@Nullable** = might be null
  - **@NonNull** = definitely not null
- Default is **@NonNull**

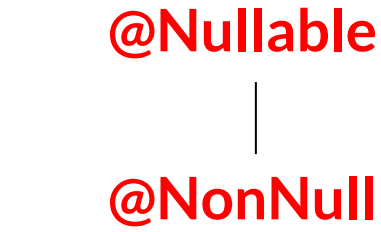
Lattice:



Nullability: Basic Type System

- Two **type qualifiers** (= abstract values):
  - **@Nullable** = might be null
  - **@NonNull** = definitely not null
- Default is **@NonNull**
  - requires fewer annotations

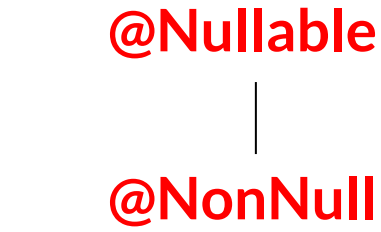
Lattice:



Nullability: Basic Type System

- Two **type qualifiers** (= abstract values):
  - **@Nullable** = might be null
  - **@NonNull** = definitely not null
- Default is **@NonNull**
  - requires fewer annotations
  - makes the dangerous case explicit

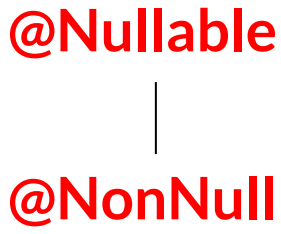
Lattice:



Nullability: Basic Type System

- Two **type qualifiers** (= abstract values):
  - **@Nullable** = might be null
  - **@NonNull** = definitely not null
- Default is **@NonNull**
  - requires fewer annotations
  - makes the dangerous case explicit
  - note: opposite of Java's implicit default!

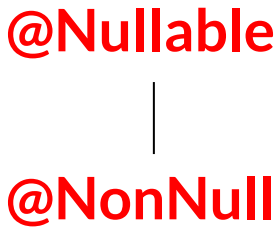
Lattice:



Nullability: Basic Type System

- Two **type qualifiers** (= abstract values):
  - **@Nullable** = might be null
  - **@NonNull** = definitely not null
- Default is **@NonNull**
  - requires fewer annotations
  - makes the dangerous case explicit
  - note: opposite of Java's implicit default!
- Nearly no annotations required in method bodies

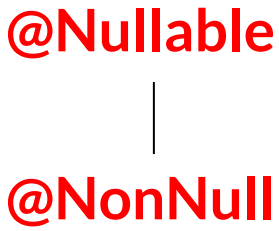
Lattice:



Nullability: Basic Type System

- Two **type qualifiers** (= abstract values):
  - **@Nullable** = might be null
  - **@NonNull** = definitely not null
- Default is **@NonNull**
  - requires fewer annotations
  - makes the dangerous case explicit
  - note: opposite of Java's implicit default!
- Nearly no annotations required in method bodies
  - dataflow analysis handles most cases (CLIMB-to-top)

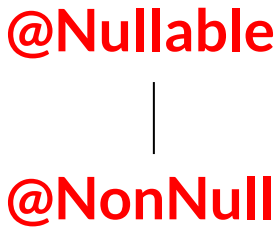
Lattice:



Nullability: Basic Type System

- Two **type qualifiers** (= abstract values):
  - **@Nullable** = might be null
  - **@NonNull** = definitely not null
- Default is **@NonNull**
  - requires fewer annotations
  - makes the dangerous case explicit
  - note: opposite of Java's implicit default!
- Nearly no annotations required in method bodies
  - dataflow analysis handles most cases (CLIMB-to-top)
  - …except for e.g., generics: `List<@Nullable String>`

Lattice:



Nullability: Flow-Sensitive Type Refinement

Nullability: Flow-Sensitive Type Refinement

- **Dataflow analysis** performs refinement within method bodies:

```
if (myField != null) {
    myField.hashCode();
}
```

Nullability: Flow-Sensitive Type Refinement

- **Dataflow analysis** performs refinement within method bodies:

```
if (myField != null) {
    myField.hashCode();
}
```

- No need to e.g., declare a new local variable

Nullability: Flow-Sensitive Type Refinement

- **Dataflow analysis** performs refinement within method bodies:

```
if (myField != null) {
    myField.hashCode();
}
```

- No need to e.g., declare a new local variable
- This is the same "*refinement*" that we saw when we talked about abstract interpretation earlier

Nullability: Side-Effects

```
if (myField != null) {
    method1();
    myField.hashCode();
}
```

- What about the code above? Should it typecheck?

Nullability: Side-Effects

```
if (myField != null) {
    method1();
    myField.hashCode();
}
```

- What about the code above? Should it typecheck?
  - No! `method1()` could side-effect `myField` and set it to null

Nullability: Side-Effects

```
if (myField != null) {
    method1();
    myField.hashCode();
}
```

- What about the code above? Should it typecheck?
  - No! `method1()` could side-effect `myField` and set it to null
  - Three ways to tell the typechecker this is ok:
    - **@SideEffectFree** void method1() { ... }
    - **@MonotonicNonNull** myField;
    - **@EnsuresNonNull("myField")** method1() {...}

Nullability: Side-Effect Annotations

Nullability: Side-Effect Annotations

- Three annotations for methods:

Nullability: Side-Effect Annotations

- Three annotations for methods:
  - **@SideEffectFree**
    - Does not modify externally-visible state

Nullability: Side-Effect Annotations

- Three annotations for methods:
  - **@SideEffectFree**
    - Does not modify externally-visible state
  - **@Deterministic**
    - If called with == args again, gives == result

Nullability: Side-Effect Annotations

- Three annotations for methods:
  - **@SideEffectFree**
    - Does not modify externally-visible state
  - **@Deterministic**
    - If called with == args again, gives == result
  - **@Pure**
    - Both side-effect-free and deterministic

Nullability: Side-Effect Annotations

- Three annotations for methods:
  - **@SideEffectFree**
    - Does not modify externally-visible state
  - **@Deterministic**
    - If called with == args again, gives == result
  - **@Pure**
    - Both side-effect-free and deterministic
- These are ***method annotations***, not type qualifiers

Nullability: Side-Effect Annotations

- Three annotations for methods:
  - **@SideEffectFree**
    - Does not modify externally-visible state
  - **@Deterministic**
    - If called with == args again, gives == result
  - **@Pure**
    - Both side-effect-free and deterministic
- These are ***method annotations***, not type qualifiers
  - Checker Framework trusts rather than checks them by default

Nullability: Side-Effect Annotations

- Three annotations for methods:
  - **@SideEffectFree**
    - Does not modify externally-visible state
  - **@Deterministic**
    - If called with == args again, gives == result
  - **@Pure**
    - Both side-effect-free and deterministic
- These are ***method annotations***, not type qualifiers
  - Checker Framework trusts rather than checks them by default
    - checking these with few false positives is an open problem

Nullability: Side-Effect Annotations

- Three annotations for methods:
  - **@SideEffectFree**
    - Does not modify externally-visible state
  - **@Deterministic**
    - If called with == args again, gives == result
  - **@Pure**
    - Both side-effect-fr
- These are ***method annotat***
  - Checker Framework tr t
    - checking these with n

These can be used with any Checker Framework checker, not just the nullability checker

Nullability: Why @Deterministic?

Nullability: Why @Deterministic?

- Consider the following code:

```
if (method2() != null) {
    method2().hashCode();
}
```

Nullability: Why @Deterministic?

- Consider the following code:

```
if (method2() != null) {
    method2().hashCode();
}
```

- This code is safe if and only if `method2()` returns the same value **every time** we call it

Nullability: Why @Deterministic?

- Consider the following code:

```
if (method2() != null) {
    method2().hashCode();
}
```

- This code is safe if and only if `method2()` returns the same value **every time** we call it
  - **@Deterministic** exactly expresses this property

Nullability: Why @Deterministic?

- Consider the following code:

```
if (method2() != null) {
    method2().hashCode();
}
```

- This code is safe if and only if `method2()` returns the same value **every time** we call it
  - **@Deterministic** exactly expresses this property
  - Why might a method not be deterministic?

Nullability: Lazy Initialization

Nullability: Lazy Initialization

- Another way to prove that a value is persistent across side-effects is the **@MonotonicNonNull** type annotation, written on a field type

Nullability: Lazy Initialization

- Another way to prove that a value is persistent across side-effects is the **@MonotonicNonNull** type annotation, written on a field type
  - Might be null or non-null

Nullability: Lazy Initialization

- Another way to prove that a value is persistent across side-effects is the **@MonotonicNonNull** type annotation, written on a field type
  - Might be null or non-null
  - May only be (re-)assigned a non-null value

Nullability: Lazy Initialization

- Another way to prove that a value is persistent across side-effects is the **@MonotonicNonNull** type annotation, written on a field type
  - Might be null or non-null
  - May only be (re-)assigned a non-null value
- Purpose: avoid re-checking

Nullability: Lazy Initialization

- Another way to prove that a value is persistent across side-effects is the **@MonotonicNonNull** type annotation, written on a field type
  - Might be null or non-null
  - May only be (re-)assigned a non-null value
- Purpose: avoid re-checking
- Once non-null, **always** non-null
  - Example: Singleton pattern

Nullability: Lazy Initialization

- Another way to prove that a value is persistent across side-effects is the **@MonotonicNonNull** type annotation, written on a field type
  - Might be null or non-null
  - May only be (re-)assigned a non-null value
- Purpose: avoid re-checking
- Once non-null, **always** non-null
  - Example: Singleton pattern
- Fields with this type don't need to be "unrefined" at possibly-side-effecting method calls

Nullability: Method Pre- and Post-conditions

Nullability: Method Pre- and Post-conditions

- More method annotations that express facts that a method requires or guarantees about program elements other than parameters/return values (e.g., in-scope fields)

Nullability: Method Pre- and Post-conditions

- More method annotations that express facts that a method requires or guarantees about program elements other than parameters/return values (e.g., in-scope fields)
  - **@RequiresNonNull** (***pre-condition***)

Nullability: Method Pre- and Post-conditions

- More method annotations that express facts that a method requires or guarantees about program elements other than parameters/return values (e.g., in-scope fields)
  - **@RequiresNonNull** (***pre-condition***)
  - **@EnsuresNonNull** (***post-condition***)

Nullability: Method Pre- and Post-conditions

- More method annotations that express facts that a method requires or guarantees about program elements other than parameters/return values (e.g., in-scope fields)
  - **@RequiresNonNull** (***pre-condition***)
  - **@EnsuresNonNull** (***post-condition***)
  - **@EnsuresNonNullIf** (***conditional post-condition***)

Nullability: Method Pre- and Post-conditions

- More method annotations that express facts that a method requires or guarantees about program elements other than parameters/return values (e.g., in-scope fields)
  - **@RequiresNonNull** (***pre-condition***)
  - **@EnsuresNonNull** (***post-condition***)
  - **@EnsuresNonNullIf** (***conditional post-condition***)
- Example:

```
@EnsuresNonNullIf(expression="#1", result=true)
public boolean equals(@Nullable Object obj) { ... }
```

Nullability: Qualifier Polymorphism

Nullability: Qualifier Polymorphism

```
/** Interns a String, and handles null. */
@PolyNull String intern(@PolyNull String a) {
  if (a == null) {
    return null;
  }
  return a.intern();
}
```

Nullability: Qualifier Polymorphism

```
/** Interns a String, and handles null. */
@PolyNull String intern(@PolyNull String a) {
  if (a == null) {
    return null;
  }
  return a.intern();
}
```

- Like defining two overloaded methods:
  - `@NonNull String intern(@NonNull String a) {...}`
  - `@Nullable String intern(@Nullable String a) {...}`

Nullability: Initialization

- It’s possible for an **@NonNull** field to contain null at run time! How?

Nullability: Initialization

- It's possible for an **@NonNull** field to contain null at run time! How?
- Consider the following code:

```
@NonNull String name;
MyClass() {  // constructor
    ... this.name.hashCode() ...
}
```

Nullability: Initialization

- It's possible for an **@NonNull** field to contain null at run time! How?
- Consider the following code:

```
@NonNull String name;
MyClass() {  // constructor
    ... this.name.hashCode() ...
}
```

- Problem: Java **initializes** all non-primitive fields to null during object initialization, before running the constructor

Nullability: Initialization

- It's possible for an **@NonNull** field to contain null at run time! How?
- Consider the following code:

```
@NonNull String name;
MyClass() {  // constructor
    ... this.name.hashCode() ...
}
```

- Problem: Java **initializes** all non-primitive fields to null during object initialization, before running the constructor
  - Does Cool do this? How do you know?

Nullability: Initialization

- It's possible for an **@NonNull** field to contain null at run time! How?
- Consider the following code:

```
@NonNull String name;
MyClass() {  // constructor
    ... this.name.hashCode() ...
}
```

- Problem: Java **initializes** all non-primitive fields to null during object initialization, before running the constructor
  - Does Cool do this? How do you know?
    - (new operational semantics rule)

Nullability: Initialization

- So, **another type system** tracks the initialization status of each object:

Nullability: Initialization

- So, **another type system** tracks the initialization status of each object:
  - **@Initialized** = constructor has completed

Nullability: Initialization

- So, **another type system** tracks the initialization status of each object:
  - **@Initialized** = constructor has completed
  - **@UnderInitialization(Frame.class)**
    - `Frame`'s constructor is currently executing

Nullability: Initialization

- So, **another type system** tracks the initialization status of each object:
  - **@Initialized** = constructor has completed
  - **@UnderInitialization(Frame.class)**
    - `Frame`'s constructor is currently executing
  - **@UnknownInitialization(Frame.class)**
    - Might be initialized or under initialization

Nullability: Initialization

- So, **another type system** tracks the initialization status of each object:
  - **@Initialized** = constructor has completed
  - **@UnderInitialization(Frame.class)**
    - `Frame`'s constructor is currently executing
  - **@UnknownInitialization(Frame.class)**
    - Might be initialized or under initialization
- Type rule for initialization: cannot dereference fields (even nonnull fields!) until the object is **@Initialized**

Nullability: Initialization

- So, **another type system** tracks the initialization status of each object:
  - **@Initialized** = constructor has completed
  - **@UnderInitialization(Frame.class)**
    - `Frame`'s constructor is currently executing
  - **@UnknownInitialization(Frame.class)**
    - Might be initialized or under initialization
- Type rule for initialization: cannot dereference fields (even nonnull fields!) until the object is **@Initialized**
  - What's the type hierarchy/lattice for these?

Nullability: Auxiliary Type Systems

- Nullability checker in the CF also has a type system for tracking which values are known to be keys in which maps

Nullability: Auxiliary Type Systems

- Nullability checker in the CF also has a type system for tracking which values are known to be keys in which maps
  - `Map.get()` can return null in general, but map operations are so common that we need to handle the common case

Nullability: Auxiliary Type Systems

- Nullability checker in the CF also has a type system for tracking which values are known to be keys in which maps
  - `Map.get()` can return null in general, but map operations are so common that we need to handle the common case
  - **@KeyFor("m")** means "this value is definitely a valid key in map `m`, so calling `Map.get()` on this value will return something nonnull"

Nullability: Auxiliary Type Systems

- Nullability checker in the CF also has a type system for tracking which values are known to be keys in which maps
  - `Map.get()` can return null in general, but map operations are so common that we need to handle the common case
  - **@KeyFor("m")** means "this value is definitely a valid key in map `m`, so calling `Map.get()` on this value will return something nonnull"
- These sort of auxiliary type systems are a **common pattern** in practical typecheckers

Nullability: Auxiliary Type Systems

- Nullability checker in the CF also has a type system for tracking which values are known to be keys in which maps
  - `Map.get()` can return null in general, but map operations are so common that we need to handle the common case
  - **@KeyFor("m")** means "this value is definitely a valid key in map `m`, so calling `Map.get()` on this value will return something nonnull"
- These sort of auxiliary type systems are a **common pattern** in practical typecheckers
  - My first project in grad school had 7!

Nullability: Warning Suppression

Nullability: Warning Suppression

- Practical checkers still have **false positives**

Nullability: Warning Suppression

- Practical checkers still have **false positives**
  - Users need ways to suppress them easily, or they won't use the tools

Nullability: Warning Suppression

- Practical checkers still have **false positives**
  - Users need ways to suppress them easily, or they won't use the tools
- Two main ways:

Nullability: Warning Suppression

- Practical checkers still have **false positives**
  - Users need ways to suppress them easily, or they won't use the tools
- Two main ways:
  - **Casts**. A cast in "regular Java" (without a pluggable typechecker) is already used to **suppress a warning from the regular type system**
    - Can also write a type qualifier in the casted-to type

Nullability: Warning Suppression

- Practical checkers still have **false positives**
  - Users need ways to suppress them easily, or they won't use the tools
- Two main ways:
  - **Casts**. A cast in "regular Java" (without a pluggable typechecker) is already used to **suppress a warning from the regular type system**
    - Can also write a type qualifier in the casted-to type
  - Explicit warning suppressions via `@SuppressWarnings`
    - More flexible and easier to read

Nullability: Effectiveness

Nullability: Effectiveness

	Null pointer errors		False warnings	Annotations written
	Found	Missed		
Checker Framework	9	0	4	35
FindBugs	0	9	1	0
Jlint	0	9	8	0
PMD	0	9	0	0
Eclipse, in 2017	0	9	8	0
Intellij (@NotNull default), in 2017	0	9	1	0
	3	6	1	925 + 8

Checking the Lookup program for file system searching (4kLOC)

Pluggable Types: State-of-the-Art

Pluggable Types: State-of-the-Art

- Extant type systems are **pretty good**

Pluggable Types: State-of-the-Art

- Extant type systems are **pretty good**
  - Nullness Checker I’ve just shown is in wide use

Pluggable Types: State-of-the-Art

- Extant type systems are **pretty good**
  - Nullness Checker I’ve just shown is in wide use
  - Uber’s NullAway is a big competitor
    - “No false negatives in practice”

Pluggable Types: State-of-the-Art

- Extant type systems are **pretty good**
  - Nullness Checker I’ve just shown is in wide use
  - Uber’s NullAway is a big competitor
    - “No false negatives in practice”
  - Type systems for other problems (e.g., resource leaks) becoming more popular

Pluggable Types: State-of-the-Art

- Extant type systems are **pretty good**
  - Nullness Checker I've just shown is in wide use
  - Uber's NullAway is a big competitor
    - "No false negatives in practice"
  - Type systems for other problems (e.g., resource leaks) becoming more popular
- Big problem: **getting people to use them**

Pluggable Types: State-of-the-Art

- Extant type systems are **pretty good**
  - Nullness Checker I've just shown is in wide use
  - Uber's NullAway is a big competitor
    - "No false negatives in practice"
  - Type systems for other problems (e.g., resource leaks) becoming more popular
- Big problem: **getting people to use them**
  - Annotation burden for legacy code is the most serious issue

Pluggable Types: State-of-the-Art

- Extant type systems are **pretty good**
  - Nullness Checker I've just shown is in wide use
  - Uber's NullAway is a big competitor
    - "No false negatives in practice"
  - Type systems for other problems (e.g., resource leaks) becoming more popular
- Big problem: **getting people to use them**
  - Annotation burden for legacy code is the most serious issue
  - Now I'm going to talk about some ongoing research work

Solution: Type Inference

- Traditional type inference: **constraint solving**

Solution: Type Inference

- Traditional type inference: **constraint solving**
  - **Problem**: need a new constraint system **for each type system**

Solution: Type Inference

- Traditional type inference: **constraint solving**
  - **Problem**: need a new constraint system **for each type system**
- Recent work has proposed 3 alternative type inference techniques:
  - Checker Framework Whole-Program Inference (ASE 2023)
  - NullAway Annotator (FSE 2023)
  - NullGTN (arxiv 2024)

CF Whole-Program Inference (ASE 2023)

CF Whole-Program Inference (ASE 2023)

- Most pluggable typecheckers already implement **local type inference** within method bodies

Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, Michael D. Ernst. *Pluggable Type Inference for Free.* In ASE 2023.

CF Whole-Program Inference (ASE 2023)

- Most pluggable typecheckers already implement **local type inference** within method bodies
  - Reduces user effort: no annotations on **local variables**

Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, Michael D. Ernst. *Pluggable Type Inference for Free*. In ASE 2023.

CF Whole-Program Inference (ASE 2023)

- Most pluggable typecheckers already implement **local type inference** within method bodies
  - Reduces user effort: no annotations on **local variables**
  - Implemented as intra-procedural **dataflow analysis**

Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, Michael D. Ernst. *Pluggable Type Inference for Free.* In ASE 2023.

CF Whole-Program Inference (ASE 2023)

- Most pluggable typecheckers already implement **local type inference** within method bodies
  - Reduces user effort: no annotations on **local variables**
  - Implemented as intra-procedural **dataflow analysis**
  - Typically implemented at the **framework level**

CF Whole-Program Inference (ASE 2023)

- Most pluggable typecheckers already implement **local type inference** within method bodies
  - Reduces user effort: no annotations on **local variables**
  - Implemented as intra-procedural **dataflow analysis**
  - Typically implemented at the **framework level**
- **Key idea**: iteratively run local inference and propagate results

Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, Michael D. Ernst. *Pluggable Type Inference for Free*. In ASE 2023.

CF Whole-Program Inference (ASE 2023)

- Most pluggable typecheckers already implement **local type inference** within method bodies
  - Reduces user effort: no annotations on **local variables**
  - Implemented as intra-procedural **dataflow analysis**
  - Typically implemented at the **framework level**
- **Key idea**: iteratively run local inference and propagate results
  - Advantage: works with any typechecker built on a framework "for free" (no per-typechecker code required)

Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, Michael D. Ernst. *Pluggable Type Inference for Free*. In ASE 2023.

CF WPI: Example

```
class DataHandler {
  Data data = null;

  void lazyInit() {
    if (data == null)
      data = new Data(...);
  }
  String serialize() {
    lazyInit();
    return data.toString(); // safe
  }
}
```

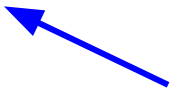
CF WPI: Example

```
class DataHandler {
  +@Nullable Data data = null;

  void lazyInit() {
    if (data == null)
      data = new Data(...);
  }
  String serialize() {
    lazyInit();
    return data.toString(); // safe
  }
}
```

local dataflow proves that null can flow into this field

CF WPI: Example

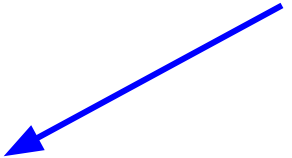
```
class DataHandler {
  +@Nullable Data data = null;
  +@EnsuresNonNull(expr={"this.data"})
  void lazyInit() {
    if (data == null)
      data = new Data(...);
  }
  String serialize() {
    lazyInit();
    return data.toString(); // safe
  }
}
```



CF also supports postcondition annotations (WPI advantage!)

CF WPI: Example

```
class DataHandler {
  +@Nullable Data data = null;
  +@EnsuresNonNull(expr={"this.data"})
  void lazyInit() {
    if (data == null)
      data = new Data(...);
  }
  String serialize() {
    lazyInit();
    return data.toString(); // safe
  }
}
```



CF also supports postcondition annotations (WPI advantage!), enabling verification of this code

NullAway Annotator (FSE 2023)

- **Key idea**: use warnings from the checker as a **fitness function** for annotations

Nima Karimipour, Justin Pham, Lazaro Clapp, and Manu Sridharan. *Practical Inference of Nullability Types*. In FSEC/FSE 2023.

NullAway Annotator (FSE 2023)

- **Key idea**: use warnings from the checker as a **fitness function** for annotations
- Iterative, bounded-depth search for annotation set that **minimizes checker warnings**
  - With some optimizations to reduce the search space

Nima Karimipour, Justin Pham, Lazaro Clapp, and Manu Sridharan. *Practical Inference of Nullability Types*. In FSEC/FSE 2023.

NullAway Annotator (FSE 2023)

- **Key idea**: use warnings from the checker as a **fitness function** for annotations
- Iterative, bounded-depth search for annotation set that **minimizes checker warnings**
  - With some optimizations to reduce the search space
- Only implementation is for NullAway (FSE ‘19, developed at Uber)

Nima Karimipour, Justin Pham, Lazaro Clapp, and Manu Sridharan. *Practical Inference of Nullability Types*. In FSEC/FSE 2023.

NullAway Annotator (FSE 2023)

- **Key idea**: use warnings from the checker as a **fitness function** for annotations
- Iterative, bounded-depth search for annotation set that **minimizes checker warnings**
  - With some optimizations to reduce the search space
- Only implementation is for NullAway (FSE '19, developed at Uber)
  - Only infers `@Nullable` annotations

Nima Karimipour, Justin Pham, Lazaro Clapp, and Manu Sridharan. *Practical Inference of Nullability Types*. In FSEC/FSE 2023.

(same example from a few slides ago…)

NullAway Annotator: Example

```
class DataHandler {
  Data data = null;

  void lazyInit() {
    if (data == null)
      data = new Data(...);
  }
  String serialize() {
    lazyInit();
    return data.toString(); // safe
  }
}
```


Nima Karimipour, Justin Pham, Lazaro Clapp, and Manu Sridharan. *Practical Inference of Nullability Types*. In FSEC/ESE 2023

(same example from a few slides ago…)

NullAway Annotator: Example

```
class DataHandler {
  Data data = null;

  void lazyInit() {
    if (data == null)
      data = new Data(...);
  }
  String serialize() {
    lazyInit();
    return data.toString(); // safe
  }
}
```

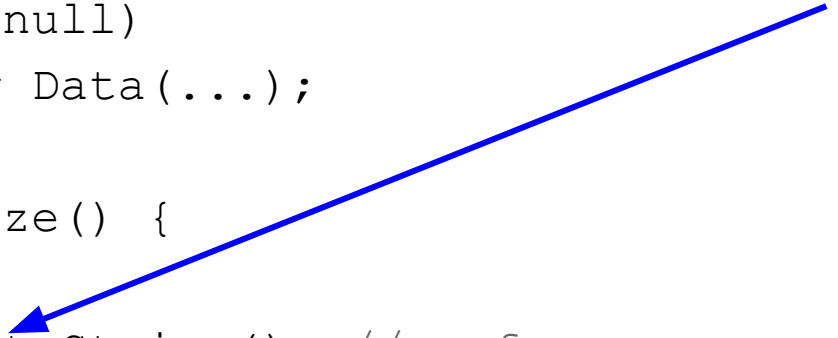
Annotator considers adding an @Nullable annotation, but does not: doing so would not decrease the error count

(same example from a few slides ago…)

NullAway Annotator: Example

```
class DataHandler {
  Data data = null;

  void lazyInit() {
    if (data == null)
      data = new Data(...);
  }
  String serialize() {
    lazyInit();
    return data.toString(); // safe
  }
}
```

Annotator considers adding an @Nullable annotation, but does not: doing so would not decrease the error count (new error here)



NullAway Annotator: Another Example

```
class MyActivity {
  Name name;
  Address addr;
  MyActivity() {
    name = null; addr = null; }
  @Initializer
  void onStart() { name = defaultName(); }
  void doFirst() { name.showFirst(); }
  void doLast() {
    name.showLast();
    if (addr != null) addr.show();
} }
```

NullAway Annotator: Another Example

```
class MyActivity {
  Name name;
  Address addr;
  MyActivity() {
    name = null; addr = null; }
  @Initializer
  void onStart() { name = defaultName(); }
  void doFirst() { name.showFirst(); }
  void doLast() {
    name.showLast();
    if (addr != null) addr.show();
} }
```

Annotator does not add @Nullable to name, because it would add two new errors (correct answer!)

NullAway Annotator: Another Example

```
class MyActivity {
  Name name;
  +@Nullable Address addr;
  MyActivity() {
    name = null; addr = null; }
  @Initializer
  void onStart() { name = defaultName(); }
  void doFirst() { name.showFirst(); }
  void doLast() {
    name.showLast();
    if (addr != null) addr.show();
} }
```

However, it does add @Nullable to addr (always null-checked before use)

NullGTN (arxiv 2024/ongoing work)

- Graph-based **deep learning** model

Kazi Amanul Islam Siddiqui and Martin Kellogg. *Inferring Pluggable Types with Machine Learning*. https://arxiv.org/abs/2406.15676

NullGTN (arxiv 2024/ongoing work)

- Graph-based **deep learning** model
  - Inspired by recent success of similar ML-based techniques for inferring Python and TypeScript type annotations

Kazi Amanul Islam Siddiqui and Martin Kellogg. *Inferring Pluggable Types with Machine Learning*. https://arxiv.org/abs/2406.15676

NullGTN (arxiv 2024/ongoing work)

- Graph-based **deep learning** model
  - Inspired by recent success of similar ML-based techniques for inferring Python and TypeScript type annotations
- **Key idea**: place annotations like a human would

Kazi Amanul Islam Siddiqui and Martin Kellogg. *Inferring Pluggable Types with Machine Learning*. https://arxiv.org/abs/2406.15676

NullGTN (arxiv 2024/ongoing work)

- Graph-based **deep learning** model
  - Inspired by recent success of similar ML-based techniques for inferring Python and TypeScript type annotations
- **Key idea**: place annotations like a human would
- Trained on ~32k classes with at least one `@Nullable` annotation from GitHub
  - Data from **many sources**: checkers, documentation, etc.

Kazi Amanul Islam Siddiqui and Martin Kellogg. *Inferring Pluggable Types with Machine Learning*. https://arxiv.org/abs/2406.15676

NullGTN (arxiv 2024/ongoing work)

- Graph-based **deep learning** model
  - Inspired by recent success of similar ML-based techniques for inferring Python and TypeScript type annotations
- **Key idea**: place annotations like a human would
- Trained on ~32k classes with at least one `@Nullable` annotation from GitHub
  - Data from **many sources**: checkers, documentation, etc.
- Only infers `@Nullable`

Kazi Amanul Islam Siddiqui and Martin Kellogg. *Inferring Pluggable Types with Machine Learning*. https://arxiv.org/abs/2406.15676

NullGTN: Example

```
public int sumLengths(
  int [] v1,
  int [] v2) {
    int result = 0;
    result += (v1 == null) ? 0 : v1.length;
    result += (v2 == null) ? 0 : v2.length;
    return result;
}
```

NullGTN: Example

```
public int sumLengths(
  int +@Nullable [] v1,
  int +@Nullable [] v2) {
    int result = 0;
    result += (v1 == null) ? 0 : v1.length;
    result += (v2 == null) ? 0 : v2.length;
    return result;
}
```

NullGTN can infer @Nullable here even if there are no call sites

Key Differences

Key Differences

- CF WPI ***implementation*** is the only one that supports multiple different pluggable type systems
  - Others claim they should generalize, but it's not evaluated

Key Difference

Implication: comparison has to be focused on nullability type systems, for now

- CF WPI ***implementation*** is the only one that supports multiple different pluggable type systems
  - Others claim they should generalize, but it's not evaluated

Key Differences

- CF WPI ***implementation*** is the only one that supports multiple different pluggable type systems
  - Others claim they should generalize, but it's not evaluated
- Only NullGTN can possibly annotate **entrypoint parameters**
  - (Assuming no test cases)
  - In WPI's evaluation, this was the largest cause of missed human-written annotations (11%)

Pluggable Types in Practice

- checkerframework.org has a few dozen checkers
  - also practical to build your own!

Pluggable Types in Practice

- checkerframework.org has a few dozen checkers
  - also practical to build your own!
- Specialized checkers are now being built at tech companies
  - e.g., NullAway at Uber

Pluggable Types in Practice

- checkerframework.org has a few dozen checkers
  - also practical to build your own!
- Specialized checkers are now being built at tech companies
  - e.g., NullAway at Uber
- If we can get type inference working, maybe these will be in widespread use in a few more years :)