Type-Checking
Announcements

- Programming Project 2 due **tonight** at 11:59PM.
  - Office hours from 1:00PM – 3:00PM in Gates 160.
- Programming Project 3 out.
  - Scope checkpoint due **Saturday, July 23 at 11:59PM**.
  - This is a **hard deadline**, no late days allowed.
  - Final submission due **Wednesday, July 27 at 11:59PM**.
  - **Start early**; this assignment is significantly larger than the previous two assignments.
More Announcements

- Programming Assignment 1 graded and returned on paperless.stanford.edu.
  - Mean: 52.9 / 60
  - Stdev: 8
- Written Assignment 1 graded. Hard copies returned after class, electronic copies will be emailed later today.
  - Mean: 20.2 / 24
  - Stdev: 3
- Let us know ASAP if you haven't heard back from us by tomorrow morning.
Where We Are

- Source Code
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization
- Machine Code
Review from Last Time

class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = new string;

        x[5] = myInteger * y;
    }

    void doSomething() {
    }

    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
Review from Last Time

class MyClass implements MyInterface
    string myInteger;

void doSomething() {
    int[] x;
    x = new string;
    x[5] = myInteger * y;

    void doSomething() {
    }

    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = new string;
        x[5] = myInteger * y;
    }

    void doSomething() {
    }

    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }

    }

Wrong type

Variable not declared

Can't multiply strings

Can't redefine functions

Can't add void

No main function
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = new string;
        x[5] = myInteger * y;
    }

    void doSomething() {
    }

    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }

}
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = new string;
        x[5] = myInteger * y;
    }

    void doSomething() {
    }

    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}

Wrong type
Can't multiply strings
Can't add void
No main function
Review from Last Time

class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = new string;
        x[5] = myInteger * y;
    }

    void doSomething() {
    }

    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}

Can't multiply strings
Wrong type
Can't add void
What Remains to Check?

- Type errors.
- Today:
  - What are types?
  - What is type-checking?
  - A type system for Decaf.
What is a Type?

• This is the subject of some debate.
• To quote Alex Aiken:
  • “The notion varies from language to language.
  • The consensus:
    – A set of values.
    – A set of operations on those values”
• **Type errors** arise when operations are performed on values that do not support that operation.
Types of Type-Checking

- **Static type checking.**
  - Analyze the program during compile-time to prove the absence of type errors.
  - Never let bad things happen at runtime.

- **Dynamic type checking.**
  - Check operations at runtime before performing them.
  - More precise than static type checking, but usually less efficient.
  - (Why?)

- **No type checking.**
  - Throw caution to the wind!
Type Systems

• The rules governing permissible operations on types forms a type system.

• Strong type systems are systems that never allow for a type error.
  • Java, Python, JavaScript, LISP, Haskell, etc.

• Weak type systems can allow type errors at runtime.
  • C, C++
Type Wars

- **Endless** debate about what the “right” system is.
- Dynamic type systems make it easier to prototype; static type systems have fewer bugs.
- Strongly-typed languages are more robust, weakly-typed systems are often faster.
Type Wars

- **Endless** debate about what the “right” system is.
- Dynamic type systems make it easier to prototype; static type systems have fewer bugs.
- Strongly-typed languages are more robust, weakly-typed systems are often faster.
- *I'm staying out of this!*
Our Focus

- Decaf is typed \textit{statically} and \textit{weakly}:
  - Type-checking occurs at compile-time.
  - Runtime errors like dereferencing \texttt{null} or an invalid object are disallowed.
- Decaf uses \texttt{class-based} inheritance.
- Decaf distinguishes primitive types and classes.
Typing in Decaf
Static Typing in Decaf

- Static type checking in Decaf consists of two separate processes:
  - Inferring the type of each expression from the types of its components.
  - Confirming that the types of expressions in certain contexts matches what is expected.

- Logically two steps, but you will probably combine into one pass.
while (numBitsSet(x + 5) <= 10) {
    if (1.0 + 4.0) {
        /* ... */
    }

    while (5 == null) {
        /* ... */
    }
}
An Example

while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* ... */
    }

    while (5 == null) {
        /* ... */
    }

}
An Example

```java
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* ... */
    }

    while (5 == null) {
        /* ... */
    }
}
```
An Example

while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* ... */
    }

    while (5 == null) {
        /* ... */
    }

}
An Example

while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
An Example

while (numBitsSet(x + 5) <= 10) {
    if (1.0 + 4.0) {
        /* ... */
    }
    while (5 == null) {
        /* ... */
    }
}
Inferring Expression Types

• How do we determine the type of an expression?
• Think of process as logical inference.
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.

![Diagram of an expression with two IntConstant nodes](image)
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as logical inference.

![Expression Diagram]

```plaintext
int +
  |   |
  |___|
137   42
```

**IntConstant** 137 + **IntConstant** 42
Inferring Expression Types

• How do we determine the type of an expression?
• Think of process as **logical inference**.
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.
Inferring Expression Types

• How do we determine the type of an expression?
• Think of process as logical inference.
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.
Inferring Expression Types

• How do we determine the type of an expression?

• Think of process as *logical inference*. 

```
bool x Identifier = bool y Identifier = bool true BoolConstant
```
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as logical inference.
Sample Inference Rules

- “If $x$ is an identifier that refers to an object of type $t$, the expression $x$ has type $t$."
- “If $e$ is an integer constant, $e$ has type int.”
- “If the operands $e_1$ and $e_2$ of $e_1 + e_2$ are known to have types int and int, then $e_1 + e_2$ has type int.”
Type Checking as Proofs

• We can think of syntax analysis as proving claims about the types of expressions.

• We begin with a set of axioms, then apply our inference rules to determine the types of expressions.

• Many type systems can be thought of as proof systems.
Formalizing our Notation

- We will encode our axioms and inference rules using this syntax:

  Preconditions
  linik

- This is read “if **preconditions** are true, we can infer **postconditions**.”
Examples of Formal Notation

\[ A \rightarrow Bv \text{ is a production.} \]
\[ t \in \text{FIRST}(B) \]
\[ \underline{t \in \text{FIRST}(A)} \]

\[ A \rightarrow \varepsilon \text{ is a production.} \]
\[ \underline{\varepsilon \in \text{FIRST}(A)} \]

\[ A \rightarrow B_1B_2\ldots B_n tv \text{ is a production.} \]
\[ \varepsilon \in \text{FIRST}(B_i) \text{ for } 1 \leq i \leq n \]
\[ \underline{t \in \text{FIRST}(A)} \]

\[ A \rightarrow B_1 \ldots B_n \text{ is a production.} \]
\[ \varepsilon \in \text{FIRST}(B_i) \text{ for } 1 \leq i \leq n \]
\[ \underline{\varepsilon \in \text{FIRST}(A)} \]
Formal Notation for Type Systems

• We write

\[ \vdash e : T \]

if the expression \( e \) has type \( T \).

• The symbol \( \vdash \) means “we can infer...”
Our Starting Axioms
Our Starting Axioms

⊢ true : bool

⊢ false : bool
Some Simple Inference Rules
Some Simple Inference Rules

\[ i \text{ is an integer constant} \quad \vdash i : \text{int} \]

\[ s \text{ is a string constant} \quad \vdash s : \text{string} \]

\[ d \text{ is a double constant} \quad \vdash d : \text{double} \]
More Complex Inference Rules
More Complex Inference Rules

\[
\begin{align*}
\therefore e_1 &: \text{int} \\
\therefore e_2 &: \text{int} \\
\hline
\therefore e_1 + e_2 &: \text{int}
\end{align*}
\]

\[
\begin{align*}
\therefore e_1 &: \text{double} \\
\therefore e_2 &: \text{double} \\
\hline
\therefore e_1 + e_2 &: \text{double}
\end{align*}
\]
More Complex Inference Rules

If we can show that $e_1$ and $e_2$ have type int...

\[ \vdash e_1 : \text{int} \]
\[ \vdash e_2 : \text{int} \]

\[ \vdash e_1 + e_2 : \text{int} \]

\[ \vdash e_1 : \text{double} \]
\[ \vdash e_2 : \text{double} \]

\[ \vdash e_1 + e_2 : \text{double} \]
More Complex Inference Rules

If we can show that $e_1$ and $e_2$ have type int...

\[ \vdash e_1 : int \]
\[ \vdash e_2 : int \]

\[ \vdash e_1 + e_2 : int \]

... then we can show that $e_1 + e_2$ has type int as well.

\[ \vdash e_1 : double \]
\[ \vdash e_2 : double \]

\[ \vdash e_1 + e_2 : double \]
Even More Complex Inference Rules
Even More Complex Inference Rules

\[ \vdash e_1 : T \]
\[ \vdash e_2 : T \]
\[ T \text{ is a primitive type} \]
\[ \vdash e_1 = e_2 : \text{bool} \]

\[ \vdash e_1 : T \]
\[ \vdash e_2 : T \]
\[ T \text{ is a primitive type} \]
\[ \vdash e_1 \neq e_2 : \text{bool} \]
Why Specify Types this Way?

- Gives a **rigorous definition of types** independent of any particular implementation.
  - No need to say “you should have the same type rules as my reference compiler.”
- Gives **maximum flexibility in implementation**.
  - Can implement type-checking however you want, as long as you obey the rules.
- Allows **formal verification of program properties**.
  - Can do inductive proofs on the structure of the program.
- **This is what's used in the literature**.
  - Good practice if you want to study types.
A Problem
A Problem

\[ x \text{ is an identifier.} \]

\[ \vdash x : ?? \]
A Problem

$x$ is an identifier.

$\vdash x : ??$

How do we know the type of $x$ if we don’t know what it refers to?
An Incorrect Solution
An Incorrect Solution

\[ \begin{align*}
\text{x is an identifier.} \\
\text{x is in scope with type T.} \\
\hline
\neg \quad \text{x : T}
\end{align*} \]
An Incorrect Solution

x is an identifier.
x is in scope with type T.

\[ \vdash x : T \]

```c
int MyFunction(int x) {
    double x;
    if (x == 1.5) {
        /* ... */
    }
}
```
An Incorrect Solution

x is an identifier.

x is in scope with type T.

\[ \vdash x : T \]

```c
int MyFunction(int x) {
{
  double x;
}

  if (x == 1.5) {
    /* ... */
  }
}
```
An Incorrect Solution

\[ x \text{ is an identifier.} \]
\[ x \text{ is in scope with type T.} \]
\[ \vdash x : T \]

```c
int MyFunction(int x) {
{
    double x;
}
if (x == 1.5) {
    /* ... */
}
}
```
An Incorrect Solution

x is an identifier.
x is in scope with type T.

\[ \vdash x : T \]

```
int MyFunction(int x) {
{
    double x;
}

if (x == 1.5) {
    /* ... */
}
```
An Incorrect Solution

\[ \text{x is an identifier.} \]
\[ \text{x is in scope with type T.} \]

\[ \vdash x : T \]

```c
int MyFunction(int x) {
    {  
        double x;
    }
    if (x == 1.5) {
        /* ... */
    }
}
```
An Incorrect Solution

\textit{x} is an identifier.
x is in scope with type T.
\[ \vdash x : T \]

\begin{verbatim}
int MyFunction(int x) {
    {
        double x;
    }
    if (x == 1.5) {
        /* ... */
    }
}
\end{verbatim}
An Incorrect Solution

\[ x \text{ is an identifier.} \]
\[ x \text{ is in scope with type } T. \]
\[ \vdash x : T \]

```c
int MyFunction(int x) {
    {
        double x;
    }

    if (x == 1.5) {
        /* ... */
    }
}
```

<table>
<thead>
<tr>
<th>Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdash x : \text{double} )</td>
</tr>
<tr>
<td>( \vdash x : \text{int} )</td>
</tr>
</tbody>
</table>
An Incorrect Solution

\[ \vdash x : T \]

\[ \vdash d : \text{double} \]

\[
\text{int MyFunction(int x) }
\{
    \{
        \text{double } x;
    \}
    \text{if (x == 1.5) } \{
        \text{/* ... */}
    \}
\}
\]

Facts:

- \( \vdash x : \text{double} \)
- \( \vdash x : \text{int} \)

\( x \) is an identifier.
\( x \) is in scope with type \( T \).
\( d \) is a double constant.
An Incorrect Solution

\[ \begin{align*} x \text{ is an identifier.} \\
\text{x is in scope with type T.} \\
\end{align*} \]

\[ \vdash x : T \]

\[ \begin{align*} \text{int MyFunction(int x) {} } \\
\text{{} } \\
\text{{} } \\
\text{{} } \\
\text{if (x == 1.5) {} } \\
\text{{} } \\
\text{{} } \\
\text{}} \]

\[ \vdash d : \text{double} \]

\[ \begin{align*} d \text{ is a double constant} \\
\end{align*} \]

Facts

| \[ \vdash x : \text{double} \] |
| \[ \vdash x : \text{int} \] |
| \[ \vdash 1.5 : \text{double} \] |
An Incorrect Solution

\[ x \text{ is an identifier.} \]
\[ x \text{ is in scope with type } T. \]
\[ \vdash x : T \]

```c
int MyFunction(int x) {
{
    double x;
}

if (x == 1.5) {
    /* ... */
}
```
An Incorrect Solution

x is an identifier.
x is in scope with type T.

\[ \vdash x : T \]

```c
int MyFunction(int x) {
{
    double x;
}
    if (x == 1.5) {
        /* ... */
    }
}
```

<table>
<thead>
<tr>
<th>Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \vdash x : double ]</td>
</tr>
<tr>
<td>[ \vdash x : int ]</td>
</tr>
<tr>
<td>[ \vdash 1.5 : double ]</td>
</tr>
</tbody>
</table>
An Incorrect Solution

\[
\begin{align*}
\text{x is an identifier.} \\
\text{x is in scope with type T.} \\
\text{\( \vdash x : T \) }
\end{align*}
\]

\[
\begin{align*}
\text{int MyFunction(int x) {} } \\
\quad \{ \\
\quad \quad \text{double x;} \\
\quad \} \\
\quad \text{if (x \texttt{ == 1.5}) { } } \\
\quad \quad \text{/* ... */} \\
\quad \}
\end{align*}
\]

\[
\begin{align*}
\text{\( \vdash \texttt{x : double} \)} \\
\text{\( \vdash \texttt{x : int} \)} \\
\text{T is a primitive type} \\
\text{\( \vdash \texttt{e}_1 \texttt{ == e}_2 : \texttt{bool} \)}
\end{align*}
\]

Facts

\[
\begin{array}{|c|}
\hline
\text{\( \vdash x : \texttt{double} \)} \\
\text{\( \vdash x : \texttt{int} \)} \\
\text{\( \vdash 1.5 : \texttt{double} \)} \\
\hline
\end{array}
\]
An Incorrect Solution

```
int MyFunction(int x) {
    double x;
    if (x == 1.5) {
        /* ... */
    }
}
```

**Facts**

- \( \vdash x : T \)
- \( \vdash e_1 : T \)
- \( \vdash e_2 : T \)
- \( T \) is a primitive type
- \( \vdash e_1 == e_2 : \text{bool} \)

**Inference**

- \( x \) is an identifier.
- \( x \) is in scope with type \( T \).

\[
\vdash x : T
\]

**Type Inference**

<table>
<thead>
<tr>
<th>Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdash x : \text{double} )</td>
</tr>
<tr>
<td>( \vdash x : \text{int} )</td>
</tr>
<tr>
<td>( \vdash 1.5 : \text{double} )</td>
</tr>
<tr>
<td>( \vdash x == 1.5 : \text{bool} )</td>
</tr>
</tbody>
</table>
An Incorrect Solution

$x$ is an identifier.
$x$ is in scope with type $T$.

\[
\vdash x : T
\]

```c
int MyFunction(int x) {
    double x;
    
    if (x == 1.5) {
        /* ... */
    }
}
```

T is a primitive type

\[
\vdash e_1 == e_2 : bool
\]

Facts

- \( \vdash x : double \)
- \( \vdash x : int \)
- \( \vdash 1.5 : double \)
- \( \vdash x == 1.5 : bool \)
An Incorrect Solution

```
int MyFunction(int x) {
    double x;
    if (x == 1.5) {
        /* ... */
    }
}
```

Facts

- $x : T$
- $x : T$
- $e_1 : T$
- $e_2 : T$
- $T$ is a primitive type

Problem?

- $e_1 == e_2 : bool$
- $x == 1.5 : bool$
Strengthening our Inference Rules

- The facts we're proving have no context.
- We need to strengthen our inference rules to remember under what circumstances the results are valid.
Adding Scope

• We write

\[ S \vdash e : T \]

if \textbf{in scope} \( S \), the expression \( e \) has type \( T \).

• Types are now proven relative to the scope they are in.
Old Rules Revisited

\[ S \vdash \text{true} : \text{bool} \]

\( i \) is an integer constant

\[ S \vdash i : \text{int} \]

\[ S \vdash \text{false} : \text{bool} \]

\( s \) is a string constant

\[ S \vdash s : \text{string} \]

\( d \) is a double constant

\[ S \vdash d : \text{double} \]

\[ S \vdash e_1 : \text{double} \]
\[ S \vdash e_2 : \text{double} \]
\[ S \vdash e_1 + e_2 : \text{double} \]

\[ S \vdash e_1 : \text{int} \]
\[ S \vdash e_2 : \text{int} \]
\[ S \vdash e_1 + e_2 : \text{int} \]
A Correct Rule

\[ S \vdash x : T \]

\( x \) is an identifier.
\( x \) is a variable in scope \( S \) with type \( T \).
A Correct Rule

\[ \begin{align*}
\text{x is an identifier.} \\
\text{x is a variable in scope S with type T.} \\
\text{\hspace{1cm}} \\
\hspace{1cm} \text{\underline{S \vdash x : T}} \\
\end{align*} \]
Rules for Functions

\[ S \vdash f(e_1, ..., e_n) : ?? \]
Rules for Functions

$f$ is an identifier.

\[ S \vdash f(e_1, \ldots, e_n) : ?? \]
Rules for Functions

\[ S \vdash f(e_1, ..., e_n) : ?? \]

- \( f \) is an identifier.
- \( f \) is a non-member function in scope \( S \).

\[ S \vdash f(e_1, ..., e_n) : ?? \]
Rules for Functions

\[ f \text{ is an identifier.} \]
\[ f \text{ is a non-member function in scope } S. \]
\[ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \]

\[ S \vdash f(e_1, \ldots, e_n) : ?? \]
Rules for Functions

\[ S \vdash f(e_1, \ldots, e_n) : \text{??} \]

- \( f \) is an identifier.
- \( f \) is a non-member function in scope \( S \).
- \( f \) has type \((T_1, \ldots, T_n) \rightarrow U\)
- \( S \vdash e_i : T_i \) for \( 1 \leq i \leq n \)
- \[ S \vdash f(e_1, \ldots, e_n) : \text{??} \]
Rules for Functions

\[ f \text{ is an identifier.} \]
\[ f \text{ is a non-member function in scope } S. \]
\[ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \]
\[ S \vdash e_i : T_i \text{ for } 1 \leq i \leq n \]
\[ \underline{S \vdash f(e_1, \ldots, e_n) : U} \]
Rules for Functions

$S \vdash f(e_1, ..., e_n)$:

- $f$ is an identifier.
- $f$ is a non-member function in scope $S$.
- $f$ has type $(T_1, ..., T_n) \rightarrow U$

\[ S \vdash e_i : T_i \text{ for } 1 \leq i \leq n \]

\[ S \vdash f(e_1, ..., e_n) : U \]

Read rules like this
Rules for Arrays

\[ S \vdash e_1 : T[[]] \]
\[ S \vdash e_2 : \text{int} \]

\[ \frac{S \vdash e_2 : \text{int}}{S \vdash e_1[e_2] : T} \]
Rule for Assignment

\[
S \leftarrow e_1 : T \\
S \leftarrow e_2 : T \\
\hline
S \leftarrow e_1 = e_2 : T
\]
Rule for Assignment

\[
S \leftarrow e_1 : T \\
S \leftarrow e_2 : T \\
\]
\[
\underline{S \leftarrow e_1 = e_2 : T}
\]

Why isn't this rule a problem for this statement?

\[
5 = x ;
\]
Rule for Assignment

\[ S \vdash e_1 : T \]
\[ S \vdash e_2 : T \]
\[ \begin{array}{c}
S \vdash e_1 = e_2 : T
\end{array} \]

If Derived extends Base, will this rule work for this code?

Base    myBase;
Derived myDerived;

myBase = myDerived;
Typing with Classes

• How do we factor inheritance into our inference rules?
• We need to consider the shape of class hierarchies.
Single Inheritance

Instructor

Professor
  - AlexAiken

Lecturer
  - Keith

TA
  - Hrysoula
  - Riddhi

Animal

Man
  - Bear
  - Pig
Multiple Inheritance

- Instructor
  - Professor
    - AlexAiken
  - Lecturer
    - Keith
  - TA
    - Hrysoula
    - Riddhi

- Animal
  - Man
  - Bear
  - Pig
    - ManBearPig
Properties of Inheritance Structures

- Any class is convertible to itself. (**Reflexivity**)
- If A is convertible to B and B is convertible to C, then A is convertible to C. (**Transitivity**)
- If A is convertible to B and B is convertible to A, then A and B are the same type. (**Antisymmetry**)
- This defines a **partial order** over types.
Types and Partial Orders

• We say that $A \leq B$ if $A$ is convertible to $B$.
• We have that
  • $A \leq A$
  • $A \leq B$ and $B \leq C$ implies $A \leq C$
  • $A \leq B$ and $B \leq A$ implies $A = B$
Updated Rule for Assignment

\[ S \vdash e_1 = e_2 : ?? \]
Updated Rule for Assignment

\[ \begin{align*} 
S &\leftarrow e_1 : T_1 \\
S &\leftarrow e_2 : T_2 \\
\hline \\
S &\leftarrow e_1 = e_2 : ?? 
\end{align*} \]
Updated Rule for Assignment

\[ S \leftarrow e_1 : T_1 \]
\[ S \leftarrow e_2 : T_2 \]
\[ T_2 \leq T_1 \]

\[ \frac{S \leftarrow e_1 = e_2 : ??}{S \leftarrow e_1 = e_2} \]
Updated Rule for Assignment

\[ S \leftarrow e_1 : T_1 \]
\[ S \leftarrow e_2 : T_2 \]
\[ T_2 \leq T_1 \]

\[ \underline{S \leftarrow e_1 = e_2 : T_1} \]
Updated Rule for Assignment

\[ S \vdash e_1 : T_1 \]
\[ S \vdash e_2 : T_2 \]
\[ T_2 \leq T_1 \]
\[ S \vdash e_1 = e_2 : T_1 \]

Can we do better than this?
Updated Rule for Assignment

\[
S \vdash e_1 : T_1 \\
S \vdash e_2 : T_2 \\
T_2 \leq T_1
\]

\[
S \vdash e_1 = e_2 : T_2
\]
Updated Rule for Assignment

\[ S \vdash e_1 : T_1 \]
\[ S \vdash e_2 : T_2 \]
\[ T_2 \leq T_1 \]

\[ S \vdash e_1 = e_2 : T_2 \]

*Not required in your semantic analyzer, but easy extra credit!*
Updated Rule for Comparisons
Updated Rule for Comparisons

\[
\begin{align*}
S \vdash e_1 : T \\
S \vdash e_2 : T \\
T \text{ is a primitive type} \\
\hline
S \vdash e_1 == e_2 : \text{bool}
\end{align*}
\]
Updated Rule for Comparisons

\[
S \vdash e_1 : T \\
S \vdash e_2 : T \\
T \text{ is a primitive type}
\]

\[
S \vdash e_1 == e_2 : \text{bool}
\]

\[
S \vdash e_1 : T_1 \\
S \vdash e_2 : T_2 \\
T_1 \text{ and } T_2 \text{ are of class type.} \\
T_1 \leq T_2 \text{ or } T_2 \leq T_1
\]

\[
S \vdash e_1 == e_2 : \text{bool}
\]
Updated Rule for Comparisons

Can we unify these rules?

\[ S ⊢ e_1 : T \]
\[ S ⊢ e_2 : T \]
T is a primitive type

\[ S ⊢ e_1 == e_2 : \text{bool} \]

\[ S ⊢ e_1 : T_1 \]
\[ S ⊢ e_2 : T_2 \]
T_1 and T_2 are of class type.
\[ T_1 ≤ T_2 \text{ or } T_2 ≤ T_1 \]

\[ S ⊢ e_1 == e_2 : \text{bool} \]
The Shape of Types

- Engine
  - CarEngine
  - DieselEngine
- DieselCarEngine
- bool
- string
- int
- double
The Shape of Types

- Engine
  - CarEngine
  - DieselEngine
  - DieselCarEngine
- bool
- string
- int
- double

Array Types
Extending Convertibility

• If A is a primitive or array type, A is only convertible to itself.

• More formally, if A and B are types and A is a primitive or array type:
  • A ≤ B implies A = B
  • B ≤ A implies A = B
Updated Rule for Comparisons

\[
\begin{align*}
S &\vdash e_1 : T \\
S &\vdash e_2 : T \\
\text{T is a primitive type} &\\
\hline
S &\vdash e_1 == e_2 : \text{bool}
\end{align*}
\]

\[
\begin{align*}
S &\vdash e_1 : T_1 \\
S &\vdash e_2 : T_2 \\
\text{T}_1 \text{ and } T_2 \text{ are of class type. } &\\
\hline
T_1 &\leq T_2 \text{ or } T_2 \leq T_1 \\
S &\vdash e_1 == e_2 : \text{bool}
\end{align*}
\]
Updated Rule for Comparisons

\[
\begin{align*}
S &\vdash e_1 : T \\
S &\vdash e_2 : T \\
&\text{T is a primitive type}
\end{align*}
\]

\[
S \vdash e_1 == e_2 : \text{bool}
\]

\[
\begin{align*}
S &\vdash e_1 : T_1 \\
S &\vdash e_2 : T_2 \\
&\text{T}_1 \text{ and T}_2 \text{ are of class type.}
\end{align*}
\]

\[
T_1 \leq T_2 \text{ or } T_2 \leq T_1
\]

\[
S \vdash e_1 == e_2 : \text{bool}
\]
Updated Rule for Comparisons

\[
\begin{align*}
S \vdash e_1 &: T \\
S \vdash e_2 &: T \\
T \text{ is a primitive type}
\end{align*}
\]

\[
S \vdash e_1 = e_2 : \text{bool}
\]

\[
\begin{align*}
S \vdash e_1 &: T_1 \\
S \vdash e_2 &: T_2 \\
T_1 \text{ and } T_2 \text{ are of class type.}
\end{align*}
\]

\[
T_1 \leq T_2 \text{ or } T_2 \leq T_1
\]

\[
S \vdash e_1 = e_2 : \text{bool}
\]
Updated Rule for Function Calls

\[ f \text{ is an identifier.} \]
\[ f \text{ is a non-member function in scope } S. \]
\[ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \]
\[ S \leftarrow e_i : R_i \text{ for } 1 \leq i \leq n \]
\[ R_i \leq T_i \text{ for } 1 \leq i \leq n \]

\[ S \leftarrow f(e_1, \ldots, e_n) : U \]
A Tricky Case

⊢ null : ??
Back to the Drawing Board

- Engine
  - CarEngine
  - DieselEngine
    - DieselCarEngine

- bool
- string
- int
- double

Array Types
Back to the Drawing Board

Engine

CarEngine  DieselEngine  bool  string  int  double

DieselCarEngine

null Type

Array Types
Handling **null**

- Define a new type corresponding to the type of the literal **null**; call it “**null type.**”
- Define **null type** $\leq A$ for any class type $A$.
- The **null** type is not accessible to programmers; it's only used internally inside the compiler.
- Many programming languages have types like these.
A Tricky Case

\[ S \vdash \text{null} : ?? \]
A Tricky Case

\[ S \vdash \text{null} : \text{null type} \]
A Tricky Case

\[ S \leftarrow \text{null} : \text{null type} \]
Object-Oriented Considerations

\[ S \vdash \text{this} : T \]

T is a class type.

\[ S \vdash \text{new } T : T \]

S is in scope of class T.

\[ S \vdash \text{e} : \text{int} \]

\[ S \vdash \text{NewArray(e, T)} : T[\text{]} \]
Object-Oriented Considerations

\[
S \vdash this : T
\]

- \( T \) is a class type.
  \[
  S \vdash new T : T
  \]
  \[
  S \vdash e : int
  \]
  \[
  S \vdash NewArray(e, T) : T[
  \]

Why don't we need to check if \( T \) is \texttt{void}?
What's Left?

- We're missing a few language constructs:
  - Member functions.
  - Field accesses.
  - Miscellaneous operators.
- Good practice to fill these in on your own.
Typing is Nuanced

• The **ternary conditional operator** `? :` evaluates an expression, then produces one of two values.

• Works for primitive types:
  • `int x = random()? 137 : 42;`

• Works with inheritance:
  • `Base b = isB? new Base : new Derived;`

• What might the typing rules look like?
A Proposed Rule

\[ S \vdash \text{cond} \ ? \ e_1 : e_2 : ?? \]
A Proposed Rule

\[ S \vdash \text{cond} : \text{bool} \]

\[ S \vdash \text{cond} \ ? \ e_1 : e_2 : ?? \]
A Proposed Rule

\[
S \vdash \text{cond} : \text{bool} \\
S \vdash e_1 : T_1 \\
S \vdash e_2 : T_2 \\
\]

\[
S \vdash \text{cond} \, ? \, e_1 : e_2 : ??
\]
A Proposed Rule

\[
S \vdash cond : \texttt{bool} \\
S \vdash e_1 : T_1 \\
S \vdash e_2 : T_2 \\
T_1 \leq T_2 \text{ or } T_2 \leq T_1
\]

\[
S \vdash cond \ ? \ e_1 \ : \ e_2 \ : \ ??
\]
A Proposed Rule

\[
S \vdash \text{cond} : \text{bool} \\
S \vdash e_1 : T_1 \\
S \vdash e_2 : T_2 \\
T_1 \leq T_2 \text{ or } T_2 \leq T_1 \\
\hline \\
S \vdash \text{cond} ? e_1 : e_2 : \max(T_1, T_2)
\]
A Proposed Rule

\[ S \vdash \text{cond} : \text{bool} \]
\[ S \vdash e_1 : T_1 \]
\[ S \vdash e_2 : T_2 \]
\[ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \]

\[ S \vdash \text{cond} \ ? e_1 \ : e_2 : \text{max}(T_1, T_2) \]
A Proposed Rule

\[
S \vdash \text{cond} : \text{bool} \\
S \vdash e_1 : T_1 \\
S \vdash e_2 : T_2 \\
T_1 \leq T_2 \text{ or } T_2 \leq T_1
\]

\[
S \vdash \text{cond} \ ? \ e_1 \ : \ e_2 : \max(T_1, T_2)
\]

Is this really what we want?
A Small Problem

\[
S \vdash \text{cond} : \text{bool} \\
S \vdash e_1 : T_1 \\
S \vdash e_2 : T_2 \\
T_1 \leq T_2 \text{ or } T_2 \leq T_1 \\
S \vdash \text{cond} \ ? \ e_1 : e_2 : \text{max}(T_1, T_2)
\]
A Small Problem

\[
S \leftarrow \text{cond} : \text{bool}
\]

\[
S \leftarrow e_1 : T_1
\]

\[
S \leftarrow e_2 : T_2
\]

\[
T_1 \leq T_2 \text{ or } T_2 \leq T_1
\]

\[
S \leftarrow \text{cond} ? e_1 : e_2 : \text{max}(T_1, T_2)
\]

Base = random()?

    new Derived1 : new Derived2;
A Small Problem

$S \vdash \text{cond} : \text{bool}$
$S \vdash e_1 : T_1$
$S \vdash e_2 : T_2$

$T_1 \leq T_2 \text{ or } T_2 \leq T_1$

$S \vdash \text{cond} \ ? e_1 : e_2 : \text{max}(T_1, T_2)$

Base = random()?
new Derived1 : new Derived2;
Least Upper Bounds

• An upper bound of two types A and B is a type C such that $A \leq C$ and $B \leq C$.

• The least upper bound of two types A and B is a type C such that:
  • C is an upper bound of A and B.
  • If C' is an upper bound of A and B, $C \leq C'$.

• When the least upper bound of A and B exists, we denote it $A \cup B$.
  • (When might it not exist?)
A Better Rule

\[ S \vdash \text{cond} : \text{bool} \]
\[ S \vdash e_1 : T_1 \]
\[ S \vdash e_2 : T_2 \]
\[ T = T_1 \cup T_2 \]
\[ S \vdash \text{cond} ? e_1 : e_2 : T \]

Base = random()?

    new Derived1 : new Derived2;
… that still has problems

Base1 = random()?
    new Derived1 : new Derived2;

S ⊢ \text{cond} : \text{bool}
S ⊢ e_1 : T_1
S ⊢ e_2 : T_2
T = T_1 \cup T_2

\[ S ⊢ \text{cond} ? e_1 : e_2 : T \]
... that still has problems

Base1 = random()?
    new Derived1 : new Derived2;

$S \leftarrow \text{cond} : \text{bool}$
$S \leftarrow e_1 : T_1$
$S \leftarrow e_2 : T_2$
$T = T_1 \cup T_2$

$S \leftarrow \text{cond} ? e_1 : e_2 : T$
Multiple Inheritance is Messy

- Type hierarchy is no longer a tree.
- Two classes might not have a least upper bound.
- Occurs in Java due to interfaces.
- Not a problem in Decaf; there is no ternary conditional operator.
- How to fix?
Minimal Upper Bounds

- An **upper bound** of two types A and B is a type C such that $A \leq C$ and $B \leq C$.

- A **minimal upper bound** of two types A and B is a type C such that:
  - C is an upper bound of A and B.
  - If C' is an upper bound of C, then it is not true that $C' < C$.

- Minimal upper bounds are not necessarily unique.

- A least upper bound must be a minimal upper bound, but not the other way around.
A Correct Rule

\[ S \leftarrow \text{cond} : \text{bool} \]
\[ S \leftarrow e_1 : T_1 \]
\[ S \leftarrow e_2 : T_2 \]

T is a minimal upper bound of $T_1$ and $T_2$

\[ S \leftarrow \text{cond} \ ? \ e_1 : e_2 : T \]

Base1 = random()?
new Derived1 ; new Derived2;
A Correct Rule

\[ S \vdash \text{cond} : \text{bool} \]
\[ S \vdash e_1 : T_1 \]
\[ S \vdash e_2 : T_2 \]

T is a minimal upper bound of \(T_1\) and \(T_2\).

\[ S \vdash \text{cond} ? e_1 : e_2 : T \]

Can prove both that expression has type \text{Base1} and that expression has type \text{Base2}.

\[
\text{Base1} = \text{random()}? \quad \text{new Derived1} : \text{new Derived2};
\]
So What?

- Type-checking can be tricky.
- Strongly influenced by the choice of operators in the language.
- Strongly influenced by the legal type conversions in a language.
- In C++, the previous example doesn't compile.
- In Java, the previous example does compile, but the language spec is enormously complicated.
  - See §15.12.2.7 of the Java Language Specification.