Semantic Analysis: Type Checking

April 22, 2013
Where We Are

Source Code

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization

Machine Code
Last Time...

Goals of a Semantic Analyzer

• Compiler must do more than recognize whether a sentence belongs to the language...

• Find all possible remaining errors that would make program invalid
  • undefined variables, types
  • type errors that can be caught **statically**

• **Terminology**
  • Static checks – done by the compiler
  • Dynamic checks – done at run time
Why Separate Semantic Analysis?

• Parsing cannot catch some errors
• Why?

• Some language constructs are not context-free
  – Example: All used variables must have been declared (scoping)
  – Example: A method must be invoked with arguments of proper type (typing)
What Does Semantic Analysis Do?

• Checks of many kinds:
  1. All identifiers are declared
  2. Types
  3. Inheritance relationships
  4. Classes defined only once
  5. Methods in a class defined only once
  6. Reserved identifiers are not misused
    And others . . .

• The requirements depend on the language
Typical Semantic Errors

- **Multiple declarations:** a variable should be declared (in the same scope) at most once
- **Undeclared variable:** a variable should not be used before being declared
- **Type mismatch:** type of the left-hand side of an assignment should match the type of the right-hand side
- **Wrong arguments:** methods should be called with the right number and types of arguments
A Sample Semantic Analyzer

- Works in two phases
  - traverses the AST created by the parser

1. For each scope in the program
   - process the declarations
     - add new entries to the symbol table and
     - report any variables that are multiply declared
   - process the statements
     - find uses of undeclared variables, and
     - update the "ID" nodes of the AST to point to the appropriate symbol-table entry.

2. Process all of the statements in the program again
   - use the symbol-table information to determine the type of each expression, and to find type errors.
Scoping: General Rules

• The scope rules of a language:
  – Determine which declaration of a named object corresponds to each use of the object
  – Scoping rules map uses of objects to their declarations

• C++ and Java use static scoping:
  – Mapping from uses to declarations at compile time
  – C++ uses the "most closely nested" rule
    • a use of variable \( x \) matches the declaration in the most closely enclosing scope
    • such that the declaration precedes the use
Dynamic Scoping Revisited

A use of a variable that has no corresponding declaration in the same function corresponds to the declaration in the most-recently-called still active function

• int i = 1;
• void func() {
  cout << i << endl;
}
• int main () {
  int i = 2;
  func();
  return 0;
}
Dynamic Scoping Revisited

A use of a variable that has no corresponding declaration in the same function corresponds to the declaration in the most-recently-called still active function

```cpp
• int i = 1;
• void func() {
•     cout << i << endl;
• }
• int main () {
•     int i = 2;
•     func();
•     return 0;
• }
```

If C++ used dynamic scoping, this would print out 2, not 1
Dynamic Scoping Revisited

- Each function has an environment of definitions
- If a name that occurs in a function is not found in its environment, its caller’s environment is searched
- And if not found there, the search continues back through the chain of callers

- Now, with that cleared up...
  - Assuming that dynamic scoping is used, what is output by the following program?

```c
void main() { int x = 0; f1(); g(); f2(); }
void f1() { int x = 10; g(); }
void f2() { int x = 20; f1(); g(); }
void g() { print(x); }
```
Symbol Tables

• purpose:
  – keep track of names declared in the program
  – names of
    • variables, classes, fields, methods
• symbol table entry:
  – associates a name with a set of attributes, e.g.:
    • kind of name (variable, class, field, method, etc)
    • type (int, float, etc)
    • nesting level
    • memory location (i.e., where will it be found at runtime)
class MyClass implements MyInterface {
    string myInteger;

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

        x[5] = myInteger * y;
    }

    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);
    }
}

Interface not declared
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;
    }

    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);
  }

}
Review from Last Time

class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = \textcolor{red}{\text{new string}};
        x[5] = \textcolor{red}{\text{myInteger \times y}};
    }

    \textcolor{red}{\text{Wrong type}}

    \textcolor{red}{\text{Can't multiply strings}}

    void doSomething() {
    }

    \textcolor{red}{\text{Can't add void}}

    int fibonacci(int n) {
        return doSomething() + \textcolor{red}{\text{fibonacci(n - 1)}};
    }

    \textcolor{red}{\text{Can't multiply strings}}

    \textcolor{red}{\text{Wrong type}}
}

Monday, April 22, 13
What Remains to Check?

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

• What is a type?
  – The notion varies from language to language

• Consensus
  – A set of values
  – A set of operations allowed on those values
Why Do We Need Type Systems?

• Consider the assembly language fragment
  • addi $r1, $r2, $r3

• What are the types of $r1, $r2, $r3?
Types and Operations

• Certain operations are legal for values of each type

  – It doesn’t make sense to add a function pointer and an integer in C

  – It does make sense to add two integers

  – But both have the same assembly language implementation!
Type Systems

• A language’s type system specifies which operations are valid for which types

• The goal of type checking is to ensure that operations are used with the correct types
  – Enforces intended interpretation of values

• Type systems provide a concise formalization of the semantic checking rules
What Can Types do For Us?

• Can detect certain kinds of errors
• Memory errors:
  – Reading from an invalid pointer, etc.
• Violation of abstraction boundaries
Type Checking Overview

• Three kinds of languages:
  
  – **Statically typed**: All or almost all checking of types is done as part of compilation (C, Java)

  – **Dynamically typed**: Almost all checking of types is done as part of program execution (Scheme)
    • Variable types depend on the path

  – **Untyped**: No type checking (machine code)
The Type Wars

• Competing views on static vs. dynamic typing
• Static typing proponents say:
  – Static checking catches many programming errors at compile time
  – Avoids overhead of run-time type checks
• Dynamic typing proponents say:
  – Static type systems are restrictive
  – Rapid prototyping easier in a dynamic type system
• In practice, most code is written in statically typed languages with an “escape” mechanism
  – Unsafe casts in C, Java
  – The best or worst of both worlds?
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 Checking and Type Inference

- **Type Checking** is the process of verifying fully typed programs
  - Given an operation and an operand of some type, determine whether the operation is allowed
- **Type Inference** is the process of filling in missing type information
  - Given the type of operands, determine
    - the meaning of the operation
    - the type of the operation
  - OR, without variable declarations, infer type from the way the variable is used

- The two are different, but are often used interchangeably
Issues in Typing

• Does the language have a type system?
  – Untyped languages (e.g. assembly) have no type system at all

• When is typing performed?
  – Static typing: At compile time
  – Dynamic typing: At runtime

• How strictly are the rules enforced?
  – Strongly typed: No exceptions
  – Weakly typed: With well-defined exceptions

• Type equivalence & subtyping
  – When are two types equivalent?
    • What does "equivalent" mean anyway?
  – When can one type replace another?
Components of a Type System

• Built-in types
• Rules for constructing new types
  – Where do we store type information?
• Rules for determining if two types are equivalent
• Rules for inferring the types of expressions
Component: Built-in Types

- Integer
  - usual operations: standard arithmetic
- Floating point
  - usual operations: standard arithmetic
- Character
  - character set generally ordered lexicographically
  - usual operations: (lexicographic) comparisons
- Boolean
  - usual operations: not, and, or, xor
Component: Type Constructors

- Pointers
  - addresses
  - operations: arithmetic, dereferencing, referencing
  - issue: equivalency

- Function types
  - A function such as "int add(real, int)" has type real×int→int
Component: Type Equivalence

• Name equivalence
  – Types are equiv only when they have the same name

• Structural equivalence
  – Types are equiv when they have the same structure

• Example
  – C uses structural equivalence for structs and name equivalence for arrays/pointers
Component: Type Equivalence

• Type coercion
  – If x is float, is x=3 acceptable?
    • Disallow
    • Allow and implicitly convert 3 to float
    • "Allow" but require programmer to explicitly convert 3 to float
  – What should be allowed?
    • float to int ?
    • int to float ?
    • What if multiple coercions are possible?
      – Consider 3 + "4" ...
Our Focus

- Decaf is typed **statically** and **weakly**:
  - Type-checking occurs at compile-time.
  - Runtime errors like dereferencing `null` or an invalid object are allowed.
- Decaf uses **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.
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) {
        /* ... */
    }
}


An Example

while (numBitsSet(x + 5) <= 10) {
  if (1.0 + 4.0) {
    /* … */
  }
  while (5 == null) {
    /* … */
  }
}

Well-typed expression with wrong type.
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) {
        /* ... */
    }
}
```

Expression with type error
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**.

```
+  
/ 
int 137 + 42
```

```
IntConstant
```

```
IntConstant
```
Inferring Expression Types

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

```
+  
|  
+---+---+
|   |   |
| int| int|
|    |    |
| IntConstant| IntConstant|
|    |    |
| 137 | 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**.
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.
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 \( \text{int} \)."

• "If the operands \( e_1 \) and \( e_2 \) of \( e_1 + e_2 \) are known to have types \( \text{int} \) and \( \text{int} \), then \( e_1 + e_2 \) has type \( \text{int} \)."
Formalizing our Notation

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

\[
\begin{array}{c}
\text{Preconditions} \\
\hline
\text{Postconditions}
\end{array}
\]

• This is read “if *preconditions* are true, we can infer *postconditions.*”
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

\[ \vdash \text{true} : \text{bool} \quad \vdash \text{false} : \text{bool} \]
Some Simple Inference Rules

\[ i \text{ is an integer constant} \quad \text{and} \quad s \text{ is a string constant} \]

\[ \vdash i : \text{int} \quad \text{and} \quad \vdash s : \text{string} \]

\[ d \text{ is a double constant} \]

\[ \vdash d : \text{double} \]
More Complex Inference Rules

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

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

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

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

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

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

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

| ⊢ e_1 : int   | ⊢ e_1 : double |
| ⊢ e_2 : int   | ⊢ e_2 : double |
| ⊢ e_1 + e_2 : int | ⊢ e_1 + e_2 : double |
Even More Complex Inference Rules

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

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

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

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

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

\[ \vdash x : \text{??} \]

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

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

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

\[\vdash x : T\]

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

Facts
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

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

\[ \vdash x : T \]

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

Facts

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

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

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

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

Facts

\[ \vdash x : \text{double} \]
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) {
        /* ... */
    }
}
```

<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

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

```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>

Monday, April 22, 13
An Incorrect Solution

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

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

\[
\begin{align*}
\text{d is a double constant} & \\
\therefore & \quad d : \text{double}
\end{align*}
\]

\[
\begin{array}{|c|}
\hline
\text{Facts} \\
\hline
\therefore x : \text{double} \\
\therefore x : \text{int} \\
\hline
\end{array}
\]
An Incorrect Solution

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

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

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

\begin{tabular}{|l|}
\hline
\textbf{Facts} \\
\hline
\hline
\textbf{\vdash x : double} \\
\hline
\textbf{\vdash x : int} \\
\hline
\textbf{\vdash 1.5 : double} \\
\hline
\end{tabular}
An Incorrect Solution

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

\[ \vdash x : T \]

```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>
<tr>
<td>( \vdash 1.5 : \text{double} )</td>
</tr>
</tbody>
</table>
An Incorrect Solution

\[ \text{If } x \text{ is an identifier and } x \text{ is in scope with type } T, \text{ then } \vdash x : T \]

```java
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>
<tr>
<td>( \vdash 1.5 : \text{double} )</td>
</tr>
</tbody>
</table>
An Incorrect Solution

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

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

\[
\begin{align*}
\text{int MyFunction(int x) { } } \\
\{ \\
\text{double x; } \\
\} \\
\text{if (x == 1.5) { } } \\
\text{/* ... */} \\
\}
\]
An Incorrect Solution

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

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

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

**Facts**

\[
\begin{array}{|l|}
\hline
\vdash x : double \\
\vdash x : int \\
\vdash 1.5 : double \\
\vdash x == 1.5 : bool \\
\hline
\end{array}
\]

Monday, April 22, 13
An Incorrect Solution

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

\[
\text{\hline} \\
\text{\hline} \\
\text{\hline} \\
\therefore \, e_1 = e_2 : \text{bool}
\]

\[
\begin{align*}
\therefore \, e_1 : T \\
\therefore \, e_2 : T \\
\text{T is a primitive type} \\
\therefore \, x : \text{T}
\end{align*}
\]

\[
\begin{align*}
\therefore \, e_1 = e_2 : \text{bool}
\end{align*}
\]

\[
\begin{align*}
\therefore \, x : \text{double} \\
\therefore \, x : \text{int} \\
\therefore \, 1.5 : \text{double} \\
\therefore \, x == 1.5 : \text{bool}
\end{align*}
\]
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, in scope \( S \), expression \( e \) has type \( T \).

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

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

\( i \) is an integer constant
\[ S \vdash i : \text{int} \]

\( s \) 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

\[ S \vdash x : T \]

\( x \) is an identifier.
\( x \) is a variable in scope \( S \) with type \( T \).
Rules for Functions

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

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

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

\[ f \text{ is an identifier.} \]
\[ f \text{ is a non-member function in scope } S. \]

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

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

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

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

- \( f \) 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 \( 1 \leq i \leq n \)
- \( S \vdash f(e_1, \ldots, e_n) : ?? \)
Rules for Functions

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

- \( f \) 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 \( 1 \leq i \leq n \)

\[ S \vdash f(e_1, \ldots, e_n) : U \]
Rules for Arrays

\[
\begin{align*}
S \vdash e_1 &: T[] \\
S \vdash e_2 &: \text{int} \\
\hline
S \vdash e_1[e_2] &: T
\end{align*}
\]
Rule for Assignment

$$\begin{align*}
S &\vdash e_1 : T \\
S &\vdash e_2 : T \\
\hline \\
S &\vdash e_1 = e_2 : T
\end{align*}$$
Rule for Assignment

\[
S \vdash e_1 : T \\
S \vdash e_2 : T \\
\hline
S \vdash e_1 = e_2 : T
\]

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

```java
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

- Animal
  - Man
  - Bear
  - Pig
Multiple Inheritance

- Animal
  - Man
  - Bear
  - Pig

- ManBearPig
Properties of Inheritance Structures

- Any type 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

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

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

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

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

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

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

\[ S \leftarrow e_1 = e_2 : T_1 \]

Can we do better than this?
Updated Rule for Assignment

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

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

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

\[
S \vdash e_1 : T_1 \\
S \vdash e_2 : T_2 \\
T_1 \text{ and } T_2 \text{ are of class type.} \\
T_1 \leq T_2 \text{ or } T_2 \leq T_1 \\
S \vdash e_1 == e_2 : \text{bool}
\]
Updated Rule for Comparisons

Can we unify these rules?

$S \vdash e_1 : T$
$S \vdash e_2 : T$
$T$ is a primitive type

$S \vdash e_1 = e_2 : \text{bool}$

$S \vdash e_1 : T_1$
$S \vdash e_2 : T_2$
$T_1$ and $T_2$ are of class type.
$T_1 \leq T_2$ or $T_2 \leq T_1$

$S \vdash e_1 = e_2 : \text{bool}$
The Shape of Types

Diagram:

- Engine
  - CarEngine
  - DieselEngine
  - DieselCarEngine
The Shape of Types
The Shape of Types

- Engine
  - CarEngine
  - DieselEngine
  - DieselCarEngine

- Types: 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:
  • $A \leq B$ implies $A = B$
  • $B \leq A$ implies $A = B$
Updated Rule for Comparisons

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

\[ S \vdash e_1 : T_1 \]
\[ S \vdash e_2 : T_2 \]
T₁ and T₂ 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

\[ S \vdash e_1 : T \]
\[ S \vdash e_2 : T \]

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

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

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

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

\[ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \]
\[ S \vdash e_1 = e_2 : \text{bool} \]
Updated Rule for Comparisons

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

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

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

S ⊢ null : ??
Back to the Drawing Board
Back to the Drawing Board
Handling **null**

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

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

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

S ⊢ null : null type
Object-Oriented Considerations

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

- \( S \vdash \text{this : } T \)

- \( T \) is a class type.

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

\[ S \vdash \text{NewArray}(e, T) : T[\] \]

- \( S \vdash \text{new } T : T \)

- \( S \) is in scope of class \( T \).
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

\[
\begin{align*}
S & \vdash \text{cond} : \text{bool} \\
S & \vdash e_1 : T_1 \\
S & \vdash e_2 : T_2 \\
T_1 & \leq T_2 \text{ or } T_2 & \leq T_1 \\
\hline \\
S & \vdash \text{cond} \ ? \ e_1 : e_2 : ??
\end{align*}
\]
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

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

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

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

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

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

\[
S \vdash \text{cond ? e}_1 : e_2 : \max(T_1, T_2)
\]

Is this really what we want?
A Small Problem

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

\[ S \vdash \text{cond} \ ? e_1 : e_2 : \max(T_1, T_2) \]
Base = random()?
    new Derived1 : new Derived2;

A Small Problem

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

\[
\begin{align*}
S &\vdash \text{cond : bool} \\
S &\vdash e_1 : T_1 \\
S &\vdash e_2 : T_2 \\
T_1 \leq T_2 \text{ or } T_2 \leq T_1 \\
S &\vdash \text{cond ? } e_1 : e_2 : \max(T_1, T_2)
\end{align*}
\]

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, then $C \leq C'$.
- When the least upper bound of A and B exists, we denote it $A \lor B$.
  - (When might it not exist?)
A Better Rule

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

Base = random()?
new Derived1 : new Derived2;
... that still has problems

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

\[
S \vdash \text{cond} : \text{bool} \\
S \vdash e_1 : T_1 \\
S \vdash e_2 : T_2 \\
T = T_1 \lor T_2 \\
S \vdash \text{cond} ? e_1 : e_2 : T
\]
... that still has problems

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

$T = T_1 \lor T_2$

$S \vdash cond \ ? e_1 : e_2 : T$

Base = random()?
    new Derived1 : new Derived2;
Multiple Inheritance is Messy

- Type hierarchy is no longer a tree.
- Two classes might not have a least upper bound.
- Occurs C++ because of multiple inheritance and 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

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

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

\[
S \vdash \text{cond ? e}_1 : e_2 : T
\]

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

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

\[ T \text{ is a minimal upper bound of } T_1 \text{ and } T_2 \]

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

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

Base1 = random()?
new Derived1 : 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.
Next Time

• More type checking...