Semantic Analysis
- Type Checking 2

April 24th
Reminder

- Project 2 Due Sunday (April 28th 11:59pm)
- Project 3 (Semantic Analyzer) (April 29th - May 16th)
BRACE YOURSELF

MIDTERM IS COMING
Midterm

• Monday May 6th here (5pm-6:20pm)
• Covers materials up to type checking (today included)
• Review session in class next Monday
• Practice exam will be released next week
• 20% + extra credit
Where We Are

Source Code

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

Machine Code
Review from Last Time

• 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.
Review from Last Time

```java
while (numBitsSet(x + 5) <= 10) {
    if (1.0 + 4.0) {
        /* ... */
    }

    while (5 == null) {
        /* ... */
    }
}
```
Review from Last Time

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

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

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

Review from Last Time

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

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

while (5 == null) {
    /* … */
}
}
Review from Last Time

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

Well-typed expression with wrong type.
Review from Last Time

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

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

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

}
Review from Last Time

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

Expression with type error
Review from Last Time

We write

\[
    S \vdash e : T
\]

if in scope \( S \), the expression \( e \) has type \( T \).
Review from Last Time

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

\[ S \vdash f(e_1, \ldots, e_n) : U \]
Review from Last Time

$f$ 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 $1 \leq i \leq n$

$\overline{S \vdash f(e_1, \ldots, e_n) : U}$

Read rules like this
Review from Last Time

- We say that $A \leq B$ if $A$ is convertible to $B$.
- The **least upper bound** of $A$ and $B$ is the class $C$ where
  - $A \leq C$
  - $B \leq C$
  - $C \leq C'$ for all other upper bounds.
- The least upper bound is denoted $A \lor B$ when it exists.
- A **minimal upper bound** of $A$ and $B$ is
  - an upper bound of $A$ and $B$
  - that is not larger than any other upper bound.
Review from Last Time

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

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

\[ R_i \leq T_i \text{ for } 1 \leq i \leq n \]

\[ S \vdash f(e_1, \ldots, e_n) : U \]
Review from Last Time

\[ S \vdash \text{null} : \text{null type} \]
Overview for Today

- Type-checking statements.
- Practical type-checking considerations.
- Type-checking practical language constructs:
  - Function overloading.
  - Specializing overrides.
• **Method overloading:** “language feature that allows creating several methods with the same name which differ from each other in the type of the input and the output of the function. It is simply defined as the ability of one function to perform different tasks.”

• **Method overriding:** “in object oriented programming, is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has same name, same parameters or signature, and same return type as the method in the parent class”
Overloading Example

void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);

Function();
Function(137);
Function(42.0);
Function(new Base);
Function(new Derived);
const int cNumAnimals = 3;

int main()
{
    Mammal* pAnimal[cNumAnimals] = { new Dog, new Cat, new Mammal };

    for (int i = 0 ; i < cNumAnimals ; ++i)
    {
        std::cout << pAnimal[i]->SendLoudNoise() << std::endl;
        delete pAnimal[i];
    }

    return 0;
}

class Mammal {
    public:
        Mammal() {};
        virtual ~Mammal() {};

        virtual std::string SendLoudNoise()
        {
            std::string str("I am a generic mammal");
            return str;
        }
};

class Dog : public Mammal {
    public:
        Dog() {};
        virtual ~Dog() {};

        std::string SendLoudNoise()
        {
            std::string str("Woof woof!");
            return str;
        }
};

class Cat : public Mammal {
    public:
        Cat() {};
        virtual ~Cat() {};

        std::string SendLoudNoise()
        {
            std::string str("--- Twitch my tail ---");
            return str;
        }
};
Using our Type Proofs

- We can now prove the types of various expressions.
- How do we check...
  - ... that if statements have well-formed conditional expressions?
  - ... that return statements actually return the right type of value?
- Use another proof system!
Proofs of Structural Soundness

• Idea: extend our proof system to statements to confirm that they are well-formed.

• We say that

\[ S \vdash WF(stmt) \]

if the statement \( stmt \) is well-formed in scope \( S \).

• The type system is satisfied if for every function \( f \) with body \( B \) in scope \( S \), we can show \( S \vdash WF(B) \).
A Simple Well-Formedness Rule

\[
S \vdash expr : T \\
\hline
S \vdash WF(expr;) 
\]
A Simple Well-Formedness Rule

If we can assign a valid type to an expression in scope $S$...
A Simple Well-Formedness Rule

If we can assign a valid type to an expression in scope $S$...

...then it is a valid statement in scope $S$. 

$S \vdash expr : T$ 

$S \vdash WF(expr ;)$
A More Complex Rule
A More Complex Rule

\[
\begin{align*}
S \vdash WF(\text{stmt}_1) \\
S \vdash WF(\text{stmt}_2) \\
\hline
S \vdash WF(\text{stmt}_1 \ \text{stmt}_2)
\end{align*}
\]
Rules for break
Rules for break

S is in a for or while loop.

S ⊢ WF(break;)

A Rule for Loops
A Rule for Loops

\[ S \vdash expr : bool \]

S' is the scope inside the loop.

\[ S' \vdash WF(stmt) \]

\[ S \vdash WF(while (expr) stmt) \]
Rules for return
Rules for `return`

S is in a function returning T
S ⊢ `expr : T'`
T' ≤ T

S ⊢ `WF(return expr;)`

S is in a function returning `void`

S ⊢ `WF(return;)`
Checking Well-Formedness

- Recursively walk the AST.
- For each statement:
  - Typecheck any subexpressions it contains.
    - Report errors if no type can be assigned.
    - Report errors if the wrong type is assigned.
  - Typecheck child statements.
- Check the overall correctness.
Practical Concerns
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* … */
}
```

Facts
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

Facts
Something is Very Wrong Here

```c
int x, y, z;
if ( ((x == y) > 5 && x + y < z) || x == z ) {
    /* ... */
}
```

**Facts**

- `x` is an identifier.
- `x` is a variable in scope `S` with type `T`.

---

`S ⊢ x : T`
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

Facts

- \( S \vdash x : \text{int} \)
  - \( x \) is an identifier.
  - \( x \) is a variable in scope \( S \) with type \( T \).
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}

**Facts**

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

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

\[ S \vdash x : T \]
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

**Facts**

- $S \vdash x : \text{int}$
- $S \vdash y : \text{int}$
- $S \vdash z : \text{int}$

$x$ is an identifier.
$x$ is a variable in scope $S$ with type $T$. 

$S \vdash x : T$
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
   /* ... */
}

\[
S \vdash x : \text{int} \\
S \vdash y : \text{int} \\
S \vdash z : \text{int}
\]

- \( x \) is an identifier.
- \( x \) is a variable in scope \( S \) with type \( T \).
- \( S \vdash x : T \)
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

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

Facts

<table>
<thead>
<tr>
<th>S ⊢ x : int</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ⊢ y : int</td>
</tr>
<tr>
<td>S ⊢ z : int</td>
</tr>
</tbody>
</table>
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}

<table>
<thead>
<tr>
<th>Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ⊢ x: int</td>
</tr>
<tr>
<td>S ⊢ y: int</td>
</tr>
<tr>
<td>S ⊢ z: int</td>
</tr>
<tr>
<td>S ⊢ x == y: bool</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
S &\vdash e_1 : T_1 \\
S &\vdash e_2 : T_2 \\
T_1 &\leq T_2 \text{ or } T_2 \leq T_1 \\
\hline
S &\vdash e_1 == e_2 : bool
\end{align*}
\]
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

Facts

<table>
<thead>
<tr>
<th>S ⊢ e₁ : T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ⊢ e₂ : T₂</td>
</tr>
<tr>
<td>T₁ ≤ T₂ or T₂ ≤ T₁</td>
</tr>
</tbody>
</table>

____________________

S ⊢ e₁ == e₂ : bool
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

\[ S \vdash x : \text{int} \]
\[ S \vdash y : \text{int} \]
\[ S \vdash z : \text{int} \]
\[ S \vdash x == y : \text{bool} \]

\( i \) is an integer constant

\[ S \vdash i : \text{int} \]
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

### Facts

| S ⊢ x : int |
| S ⊢ y : int |
| S ⊢ z : int |
| S ⊢ x == y : bool |
| S ⊢ 5 : int |

\[ i \text{ is an integer constant} \]

\[ S ⊢ i : \text{int} \]
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

- **Facts**
  - $S \vdash x : \text{int}$
  - $S \vdash y : \text{int}$
  - $S \vdash z : \text{int}$
  - $S \vdash x == y : \text{bool}$
  - $S \vdash 5 : \text{int}$

- $i$ is an integer constant

- $S \vdash i : \text{int}$
```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* … */
}
```

---

**Facts**

| S ⊢ x : int |
| S ⊢ y : int |
| S ⊢ z : int |
| S ⊢ x == y : bool |
| S ⊢ 5 : int |
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

### Facts

<table>
<thead>
<tr>
<th>Statement</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(x == y)</code></td>
<td><code>bool</code></td>
</tr>
<tr>
<td><code>(x + y)</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>5</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>x + y</code></td>
<td><code>int</code></td>
</tr>
</tbody>
</table>

### Proof

- `S |- e_1 : int`
- `S |- e_2 : int`

**Conclusion:**

`S |- e_1 + e_2 : int`
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}

S ⊢ x : int
S ⊢ y : int
S ⊢ z : int
S ⊢ x == y : bool
S ⊢ 5 : int
S ⊢ x + y : int
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
int x, y, z;
if ( ((x == y) > 5 && x + y < z) || x == z ) {
    /* ... */
}

### Facts

<table>
<thead>
<tr>
<th>$S \vdash x : \text{int}$</th>
<th>$S \vdash y : \text{int}$</th>
<th>$S \vdash z : \text{int}$</th>
<th>$S \vdash x == y : \text{bool}$</th>
<th>$S \vdash 5 : \text{int}$</th>
<th>$S \vdash x + y : \text{int}$</th>
<th>$S \vdash x + y &lt; z : \text{bool}$</th>
</tr>
</thead>
</table>

$S \vdash e_1 : \text{int}$
$S \vdash e_2 : \text{int}$

$S \vdash e_1 < e_2 : \text{bool}$
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

**Facts**

<table>
<thead>
<tr>
<th>S ⊢ x : int</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ⊢ y : int</td>
</tr>
<tr>
<td>S ⊢ z : int</td>
</tr>
<tr>
<td>S ⊢ x == y : bool</td>
</tr>
<tr>
<td>S ⊢ 5 : int</td>
</tr>
<tr>
<td>S ⊢ x + y : int</td>
</tr>
<tr>
<td>S ⊢ x + y &lt; z : bool</td>
</tr>
</tbody>
</table>
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}

\[
S ⊢ e_1 : T_1 \\
S ⊢ e_2 : T_2 \\
T_1 \leq T_2 \text{ or } T_2 \leq T_1 \\
\hline
S ⊢ e_1 == e_2 : bool
\]

<table>
<thead>
<tr>
<th>Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>------------------------------</td>
</tr>
<tr>
<td>( S ⊢ x : \text{int} )</td>
</tr>
<tr>
<td>( S ⊢ y : \text{int} )</td>
</tr>
<tr>
<td>( S ⊢ z : \text{int} )</td>
</tr>
<tr>
<td>( S ⊢ x == y : \text{bool} )</td>
</tr>
<tr>
<td>( S ⊢ 5 : \text{int} )</td>
</tr>
<tr>
<td>( S ⊢ x + y : \text{int} )</td>
</tr>
<tr>
<td>( S ⊢ x + y &lt; z : \text{bool} )</td>
</tr>
<tr>
<td>( S ⊢ x == z : \text{bool} )</td>
</tr>
</tbody>
</table>
Something is Very Wrong Here

```c
int x, y, z;
if ((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

### Facts

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

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

<table>
<thead>
<tr>
<th>S \vdash x : int</th>
</tr>
</thead>
<tbody>
<tr>
<td>S \vdash y : int</td>
</tr>
<tr>
<td>S \vdash z : int</td>
</tr>
<tr>
<td>S \vdash x == y : bool</td>
</tr>
<tr>
<td>S \vdash 5 : int</td>
</tr>
<tr>
<td>S \vdash x + y : int</td>
</tr>
<tr>
<td>S \vdash x + y &lt; z : bool</td>
</tr>
<tr>
<td>S \vdash x == z : bool</td>
</tr>
</tbody>
</table>
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

Facts

<table>
<thead>
<tr>
<th>S ⊢ e₁ : int</th>
<th>S ⊢ e₂ : int</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ⊢ e₁ &gt; e₂ : bool</td>
<td></td>
</tr>
</tbody>
</table>

| S ⊢ x : int |
| S ⊢ y : int |
| S ⊢ z : int |
| S ⊢ x == y : bool |
| S ⊢ 5 : int |
| S ⊢ x + y : int |
| S ⊢ x + y < z : bool |
| S ⊢ x == z : bool |
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

**Facts**

<table>
<thead>
<tr>
<th><code>S ⊢ e_1 : int</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>S ⊢ e_2 : int</code></td>
</tr>
<tr>
<td><code>S ⊢ e_1 &gt; e_2 : bool</code></td>
</tr>
</tbody>
</table>
int x, y, z;
if ((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}

> Error: Cannot compare int and bool
Something is Very Wrong Here

```c
int x, y, z;
if ((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

> Error: Cannot compare int and bool

### Facts

<table>
<thead>
<tr>
<th><code>S ⊢ e_1 : int</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>S ⊢ e_2 : int</code></td>
</tr>
<tr>
<td><code>S ⊢ e_1 &gt; e_2 : bool</code></td>
</tr>
</tbody>
</table>

| `S ⊢ x : int` |
| `S ⊢ y : int` |
| `S ⊢ z : int` |
| `S ⊢ x == y : bool` |
| `S ⊢ 5 : int` |
| `S ⊢ x + y : int` |
| `S ⊢ x + y < z : bool` |
| `S ⊢ x == z : bool` |
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

> Error: Cannot compare int and bool

### Facts

- `S ⊢ x : int`
- `S ⊢ y : int`
- `S ⊢ z : int`
- `S ⊢ x == y : bool`
- `S ⊢ 5 : int`
- `S ⊢ x + y : int`
- `S ⊢ x + y < z : bool`
- `S ⊢ x == z : bool`
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

Facts

| ⊢ x : int |
| ⊢ y : int |
| ⊢ z : int |
| ⊢ x == y : bool |
| ⊢ 5 : int |
| ⊢ x + y : int |
| ⊢ x + y < z : bool |
| ⊢ x == z : bool |

> Error: Cannot compare int and bool
Error: Cannot compare ??? and bool
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

> Error: Cannot compare int and bool

Error: Cannot compare ??? and bool
int x, y, z;
if ((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}

S ⊢ x : int
S ⊢ y : int
S ⊢ z : int
S ⊢ x == y : bool
S ⊢ 5 : int
S ⊢ x + y : int
S ⊢ x + y < z : bool
S ⊢ x == z : bool

> Error: Cannot compare int and bool
Error: Cannot compare ??? and bool
Something is Very Wrong Here

```c
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* ... */
}
```

### Facts

- $S \vdash x : \text{int}$
- $S \vdash y : \text{int}$
- $S \vdash z : \text{int}$
- $S \vdash x == y : \text{bool}$
- $S \vdash 5 : \text{int}$
- $S \vdash x + y : \text{int}$
- $S \vdash x + y < z : \text{bool}$
- $S \vdash x == z : \text{bool}$

> Error: Cannot compare int and bool
> Error: Cannot compare ??? and bool
> Error: Cannot compare ??? and bool
Cascading Errors

- A **static type error** occurs when we cannot prove that an expression has a given type.

- Type errors can easily cascade:
  - Can't prove a type for \( e_1 \), so can't prove a type for \( e_1 + e_2 \), so can't prove a type for \( (e_1 + e_2) + e_3 \), etc.

- How do we resolve this?
The Shape of Types

Vegetable

Carrot

Artichoke
The Shape of Types

Vegetable

Carrot  Artichoke  bool  string  int  double

Array Types
The Shape of Types

Vegetable

Carrot

Artichoke

bool

string

int

double

null Type

Array Types
The Shape of Types

Vegetable

Carrot  Artichoke

bool  string  int  double

null Type

Error Type

Array Types
The Error Type

- Introduce a new type representing an error into the type system.
- The **error type** is less than all other types and is denoted \( \bot \).
  - It is sometimes called the **bottom type**.
- By definition, \( \bot \leq A \) for any type \( A \).
- On discovery of a type error, pretend that we can prove the expression has type \( \bot \).
- Update our inference rules to support \( \bot \).
Updated Rules for Addition

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

\[ S \vdash e_1 + e_2 : \text{double} \]
Updated Rules for Addition

\[
\begin{align*}
S & \vdash e_1 : T_1 \\
S & \vdash e_2 : T_2
\end{align*}
\]

\[
S \vdash e_1 + e_2 : \text{double}
\]
Updated Rules for Addition

\[ S \vdash e_1 : T_1 \]
\[ S \vdash e_2 : T_2 \]
\[ T_1 \leq \text{double} \]
\[ T_2 \leq \text{double} \]

\[ S \vdash e_1 + e_2 : \text{double} \]
Updated Rules for Addition

\[
\begin{align*}
S \vdash e_1 : T_1 \\
S \vdash e_2 : T_2 \\
T_1 &\leq \text{double} \\
T_2 &\leq \text{double}
\end{align*}
\]

What does this mean?
Updated Rules for Addition

\[
\begin{align*}
S & \vdash e_1 : T_1 \\
S & \vdash e_2 : T_2 \\
T_1 & \leq \text{double} \\
T_2 & \leq \text{double}
\end{align*}
\]

\[
\begin{align*}
S & \vdash e_1 + e_2 : \text{double}
\end{align*}
\]

\[
\begin{align*}
S & \vdash e_1 : T_1 \\
S & \vdash e_2 : T_2 \\
T_1 & \leq \text{int} \\
T_2 & \leq \text{int}
\end{align*}
\]

\[
\begin{align*}
S & \vdash e_1 + e_2 : \text{int}
\end{align*}
\]
Updated Rules for Addition

Prevents errors from propagating.
Updated Rules for Addition

\[
\begin{align*}
S & \vdash e_1 : T_1 \\
S & \vdash e_2 : T_2 \\
T_1 & \leq \text{double} \\
T_2 & \leq \text{double}
\end{align*}
\]

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

\[
\begin{align*}
S & \vdash e_1 : T_1 \\
S & \vdash e_2 : T_2 \\
T_1 & \leq \text{int} \\
T_2 & \leq \text{int}
\end{align*}
\]

\[S \vdash e_1 + e_2 : \text{int}\]
Updated Rules for Addition

\[
\begin{align*}
S &\vdash e_1 : T_1 \\
S &\vdash e_2 : T_2 \\
T_1 &\leq \text{double} \\
T_2 &\leq \text{double}
\end{align*}
\]

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

\[
\begin{align*}
S &\vdash e_1 : T_1 \\
S &\vdash e_2 : T_2 \\
T_1 &\leq \text{int} \\
T_2 &\leq \text{int}
\end{align*}
\]

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

What happens if both operands have error type?
Error-Recovery in Practice

- In your semantic analyzer, you will need to do some sort of error recovery.
- We provide an error type `Type::errorType`.
- But what about other cases?
  - Calling a nonexistent function.
  - Declaring a variable of a bad type.
  - Treating a non-array as an array.
- There are no right answers to these questions; just better and worse choices.
Implementing Convertibility

- How do we implement the $\leq$ operator we've described so far?

- Lots of cases:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Class Type</th>
<th>Primitive Type</th>
<th>Array Type</th>
<th>Null Type</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Type</td>
<td>If same or inherits from</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Primitive Type</td>
<td>No</td>
<td>If same type</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Array Type</td>
<td>No</td>
<td>No</td>
<td>If underlying types match</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Null Type</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Error Type</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## Methods You Might Want...

- **virtual bool Type::IsIdenticalTo(Type* other);**
  - Returns whether two types represent the same actual type.

- **virtual bool Type::IsConvertibleTo(Type* other);**
  - Returns whether one type is convertible to some other type.
Function Overloading
Function Overloading

- Two functions are said to be overloads of one another if they have the same name but a different set of arguments.
- At compile-time, determine which function is meant by inspecting the types of the arguments.
- Report an error if no one function is the best function.
Overloading Example

```cpp
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);

Function();
Function(137);
Function(42.0);
Function(new Base);
Function(new Derived);
```
Overloading Example

```c
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```c
Function();
Function(137);
Function(42.0);
Function(new Base);
Function(new Derived);
```
Implementing Overloading

void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
Implementing Overloading

```c
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```c
Function(137);
```
Implementing Overloading

```c
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```c
Function(137);
```
Implementing Overloading

```cpp
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```cpp
Function(137);
```
Implementing Overloading

```c
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

Function(137);
Implementing Overloading

```c
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```
Function(137);
```
Implementing Overloading

void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);

Function(137);
Implementing Overloading

void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);

Function(137);
Implementing Overloading

void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);

Function(137);
Simple Overloading

• We begin with a set of overloaded functions.
• After filtering out functions that cannot match, we have a **candidate set** (C++ terminology) or set of **potentially applicable methods** (Java-speak).
• If no functions are left, report an error.
• If exactly one function left, choose it.
• (We'll deal with two or more in a second)
Overloading with Inheritance

```c
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```
Overloading with Inheritance

```c
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```c
Function(new Derived);
```
Overloading with Inheritance

```c
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```c
Function(new Derived);
```
Overloading with Inheritance

```c
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```
Function(new Derived);
```

![Diagram showing function overloading with inheritance](Diagram)
Overloading with Inheritance

void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);

How do we compare these?

Function(new Derived);
Finding the Best Match

• Choose one function over another if it's strictly more specific.

• Given two candidate functions A and B with argument types $A_1, A_2, ..., A_n$ and $B_1, B_2, ..., B_n$, we say that $A <: B$ if $A_i \leq B_i$ for all $i$, $1 \leq i \leq n$.
  
  • This relation is also a partial order.

• A candidate function A is the best match if for any candidate function B, $A <: B$.
  
  • It's at least as good any other match.

• If there is a best match, we choose that function. Otherwise, the call is ambiguous.
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);

Function(new Derived);
Overloading with Inheritance

```c
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```c
Function(new Derived);
```
Overloading with Inheritance

```c
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```
Function(new Derived);
(int x)
(double x)
(Base b)
(Derived d)
```

Function(new Derived);
Ambiguous Calls

void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Base b1, Derived d2);
Ambiguous Calls

```c
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Base b1, Derived d2);
```

```c
Function(new Derived, new Derived);
```
Ambiguous Calls

```c
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Base b1, Derived d2);
```

Function(new Derived, new Derived);
Ambiguous Calls

void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Base b1, Derived d2);

Function(new Derived, new Derived);
Ambiguous Calls

void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Base b1, Derived d2);

Neither of these is a better match than the other.

Function(new Derived, new Derived);
In the Real World

- Often much more complex than this.
- Example: **variadic functions**.
  - Functions that can take multiple arguments.
- Supported by C, C++, and Java.
Overloading with Variadic Functions

void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
Overloading with Variadic Functions

```c
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```

```c
Function(new Derived, new Derived);
```
Overloading with Variadic Functions

```c
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```

```c
Function(new Derived, new Derived);
```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);

Function(new Derived, new Derived);
Overloading with Variadic Functions

- Option one: **Consider the call ambiguous**.
  - There are indeed multiple valid function calls, and that's that!
- Option two: **Prefer the non-variadic function**.
  - A function specifically designed to handle a set of arguments is probably a better match than one designed to handle arbitrarily many parameters.
  - Used in both C++ and (with minor modifications) Java.
Hierarchical Function Overloads

- Idea: Have a hierarchy of candidate functions.
- Conceptually similar to a scope chain:
  - Start with the lowest hierarchy level and look for an overload.
  - If a match is found, choose it.
  - If multiple functions match, report an ambiguous call.
  - If no match is found, go to the next level in the chain.
- Similar techniques used in other places:
  - Template / generic functions.
  - Implicit conversions
Overloading with Variadic Functions

```c
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```

Function(new Derived, new Derived);
Overloading with Variadic Functions

```c
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```

```
Function(new Derived, new Derived);
```
Overloading with Variadic Functions

```c
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```

Function(NEW Derived, NEW Derived);
Overloading with Variadic Functions

```cpp
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```

```
Function(new Derived, new Derived);
```

Overloading with Variadic Functions

```c
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```

```c
Function(new Derived, new Derived);
```
Overloading with Variadic Functions

void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);

void Function(new Derived, new Derived);
Covariance and Contravariance
covariant: converting from a specialized type (Cats) to a more general type (Animals): Every cat is an animal.

covariant: converting from a specialized type (Cats) to a more general type (Animals): Every cat is an animal.

covariant: converting from a specialized type (Cats) to a more general type (Animals): Every cat is an animal.

covariant: converting from a specialized type (Cats) to a more general type (Animals): Every cat is an animal.

covariant: converting from a specialized type (Cats) to a more general type (Animals): Every cat is an animal.

covariant: converting from a specialized type (Cats) to a more general type (Animals): Every cat is an animal.
A Rule for Member Functions

\[ S \vdash e_0.f(e_1, \ldots, e_n) : ? \]
A Rule for Member Functions

\[ S \vdash e_0.f(e_1, \ldots, e_n) : ? \]

\( f \) is an identifier.
A Rule for Member Functions

$f$ is an identifier.

\[ S \vdash e_0 : M \]

\[ S \vdash e_0.f(e_1, ..., e_n) : ? \]
A Rule for Member Functions

$f$ is an identifier.
$S \vdash e_0 : M$

$f$ is a member function in class $M$.

$S \vdash e_0.f(e_1, ..., e_n) : ?$
A Rule for Member Functions

\[ f \text{ is an identifier.} \]
\[ S \vdash e_0 : M \]
\[ f \text{ is a member function in class } M. \]
\[ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \]

\[ S \vdash e_0.f(e_1, \ldots, e_n) : ? \]
A Rule for Member Functions

\[ f \text{ is an identifier.} \]
\[ S \vdash e_0 : M \]
\[ f \text{ is a member function in class } M. \]
\[ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \]
\[ S \vdash e_i : R_i \text{ for } 1 \leq i \leq n \]
\[ R_i \leq T_i \text{ for } 1 \leq i \leq n \]
\[ \overline{S \vdash e_0.f(e_1, \ldots, e_n) : ?} \]
A Rule for Member Functions

$f$ is an identifier.

\[ S \vdash e_0 : M \]

$f$ is a member function in class $M$.

$f$ has type $(T_1, \ldots, T_n) \rightarrow U$

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

\[ R_i \leq T_i \text{ for } 1 \leq i \leq n \]

\[ S \vdash e_0.f(e_1, \ldots, e_n) : U \]
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* … */
    }
}

class Ego extends Id {
    void bePractical() {
        /* … */
    }
}

int main() {
    (new Ego).me().bePractical();
}
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* ... */
    }
}

class Ego extends Id {
    void bePractical() {
        /* ... */
    }
}

int main() {
    (new Ego).me().bePractical();
}
class Id {
  Id me() {
    return this;
  }
  void beSelfish() {
    /* ... */
  }
}

class Ego extends Id {
  void bePractical() {
    /* ... */
  }
}

int main() {
  (new Ego).me().bePractical();
}
Legality and Safety

class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* ... */
    }
}

class Ego extends Id {
    void bePractical() {
        /* ... */
    }
}

int main() {
    (new Ego).me().bePractical();
}
Legality and Safety

class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* ... */
    }
}

class Ego extends Id {
    void bePractical() {
        /* ... */
    }
}

int main() {
    (new Ego).me().bePractical();
}

\( S \vdash e_0 : M \)

\( f \) is a member function in class \( M \).
\( f \) has type \( (T_1, \ldots, T_n) \rightarrow U \)
\( S \vdash e_i : R_i \) for \( 1 \leq i \leq n \)
\( R_i \leq T_i \) for \( 1 \leq i \leq n \)
\( S \vdash e_0.f(e_1, \ldots, e_n) : U \)
Legality and Safety

class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* ... */
    }
}

class Ego extends Id {
    void bePractical() {
        /* ... */
    }
}

int main() {
    (new Ego).me().bePractical();
}

\[
\begin{align*}
f & \text{ is an identifier.} \\
S \vdash e_0 : M \\
f & \text{ is a member function in class M.} \\
f & \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\
S & \vdash e_i : R_i \text{ for } 1 \leq i \leq n \\
R_i & \leq T_i \text{ for } 1 \leq i \leq n \\
S & \vdash e_0.f(e_1, \ldots, e_n) : U
\end{align*}
\]
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* ... */
    }
}

class Ego extends Id {
    void bePractical() {
        /* ... */
    }
}

int main() {
    (new Ego).me().bePractical();
}

\[ S \vdash e_0 : M \]
\[ f \text{ is an identifier.} \]
\[ S \vdash e_i : R_i \text{ for } 1 \leq i \leq n \]
\[ R_i \leq T_i \text{ for } 1 \leq i \leq n \]
\[ S \vdash e_0.f(e_1, ..., e_n) : U \]
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* … */
    }
}

class Ego extends Id {
    void bePractical() {
        /* … */
    }
}

int main() {
    (new Ego).me().bePractical();
}

\[
S \vdash e_0 : M
\]

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

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

\[
R_i \leq T_i \quad \text{for } 1 \leq i \leq n
\]

\[
S \vdash e_0.f(e_1, ..., e_n) : U
\]
class Id {
    Id me() {
        return this;
    }

    void beSelfish() {
        /* ... */
    }
}

class Ego extends Id {
    void bePractical() {
        /* ... */
    }
}

int main() {
    (new Ego).me().bePractical();
}

\[
S \vdash e_0 : M
\]
\[
f \text{ is a member function in class } M. \\
\text{ }f \text{ has type } (T_1, \ldots, T_n) \rightarrow U
\]
\[
S \vdash e_i : R_i \text{ for } 1 \leq i \leq n
\]
\[
R_i \leq T_i \text{ for } 1 \leq i \leq n
\]
\[
S \vdash e_0.f(e_1, \ldots, e_n) : U
\]

bePractical

is not in Id!
Limitations of Static Type Systems

- Static type systems are often **incomplete**.
  - There are valid programs that are rejected.
- Tension between the **static** and **dynamic** types of objects.
  - Static type is the type declared in the program source.
  - Dynamic type is the actual type of the object at runtime.
Soundness and Completeness

• Static type systems sometimes reject valid programs because they cannot prove the absence of a type error.

• A type system like this is called **incomplete**.

• Instead, try to prove for every expression that

\[ \text{DynamicType}(E) \leq \text{StaticType}(E) \]

• A type system like this is called **sound**.
An Impossibility Result

- Unfortunately, for most programming languages, it is provably impossible to have a sound and complete static type checker.
- Intuition: Could build a program that makes a type error iff a certain Turing machine accepts a given string.
- Type-checking equivalent to solving the halting problem!
Building a Good Static Checker

- It is difficult to build a good static type checker.
  - Easy to have unsound rules.
  - Impossible to accept all valid programs.
- Goal: make the language as complete as possible with sound type-checking rules.
Relaxing our Restrictions

class Base {
    Base clone() {
        return new Base;
    }
}

class Derived extends Base {
    Base clone() {
        return new Derived;
    }
}
class Base {
    Base clone() {
        return new Base;
    }
}

class Derived extends Base {
    Base clone() {
        return new Derived;
    }
}
class Base {
    Base clone() {
        return new Base;
    }
}

class Derived extends Base {
    Derived clone() {
        return new Derived;
    }
}
class Base {
    Base clone() {
        return new Base;
    }
}

class Derived extends Base {
    Derived clone() {
        return new Derived;
    }
}

Is this safe?
The Intuition

Base b = new Base;
Derived d = new Derived;
Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
The Intuition

Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
The Intuition

Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
The Intuition

Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
The Intuition

Base b = new Base;
Derived d = new Derived;
Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
Derived d3 = d.clone();
The Intuition

Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
Derived d3 = d.clone();

Base reallyD = new Derived;
Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
Derived d3 = d.clone();

Base reallyD = new Derived;
Base b4 = reallyD.clone();
The Intuition

Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
Derived d3 = d.clone();

Base reallyD = new Derived;
Base b4 = reallyD.clone();
Derived d4 = reallyD.clone();
The Intuition

Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
Derived d3 = d.clone();

Base reallyD = new Derived;
Base b4 = reallyD.clone();
Derived d4 = reallyD.clone();
$f$ is an identifier.
\[ S \vdash e_0 : M \]

$f$ is a member function in class $M$.
$f$ has type $(T_1, \ldots, T_n) \rightarrow U$
\[ S \vdash e_i : R_i \text{ for } 1 \leq i \leq n \]
\[ R_i \leq T_i \text{ for } 1 \leq i \leq n \]

\[ S \vdash e_0.f(e_1, \ldots, e_n) : U \]
Is this Safe?

- $f$ is an identifier.
  
  $S \vdash e_0 : M$

- $f$ is a member function in class M. $f$ has type $(T_1, \ldots, T_n) \rightarrow U$
  
  $S \vdash e_i : R_i$ for $1 \leq i \leq n$

  $R_i \leq T_i$ for $1 \leq i \leq n$

  
  $S \vdash e_0.f(e_1, \ldots, e_n) : U$

This refers to the static type of the function.
Is this Safe?

\[ \text{f is an identifier.} \]
\[ S \vdash e_0 : M \]

\[ \text{f is a member function in class M.} \]
\[ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \]
\[ S \vdash e_i : R_i \text{ for } 1 \leq i \leq n \]
\[ R_i \leq T_i \text{ for } 1 \leq i \leq n \]

\[ S \vdash e_0 . f(e_1, \ldots, e_n) : U \]

This refers to the static type of the function.

\[ f \text{ has dynamic type } (T_1, T_2, \ldots, T_n) \rightarrow V \]

and we know that

\[ V \leq U \]
Is this Safe?

- **f** is an identifier.
  
  \[ S \vdash e_0 : M \]

- **f** is a member function in class **M**.
  
  \[ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \]

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

  \[ R_i \leq T_i \text{ for } 1 \leq i \leq n \]

  \[ S \vdash e_0.f(e_1, \ldots, e_n) : U \]

  This refers to the **static type** of the function.

- **f** has **dynamic type**

  \[ (T_1, T_2, \ldots, T_n) \rightarrow V \]

  and we know that

  \[ V \leq U \]

  So the rule is sound!
Covariant Return Types

- Two functions $A$ and $B$ are covariant in their return types if the return type of $A$ is convertible to the return type of $B$.
- Many programming languages support covariant return types.
  - C++ and Java, for example.
- Not supported in Decaf.
  - But easy extra credit!
Relaxing our Restrictions (Again)

class Base {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Base B) {
        /* ... */
    }
}
Relaxing our Restrictions (Again)

class Base {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Base B) {
        /* ... */
    }
}
Relaxing our Restrictions (Again)

class Base {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Derived B) {
        /* ... */
    }
}
class Base {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Derived D) {
        /* ... */
    }
}
Relaxing our Restrictions (Again)

class Base {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Derived D) {
        /* ... */
    }
}
Is this Safe?

- $f$ is an identifier.
  - $S \vdash e_0 : M$

- $f$ is a member function in class $M$.
  - $f$ has type $(T_1, \ldots, T_n) \rightarrow U$
  - $S \vdash e_i : R_i$ for $1 \leq i \leq n$
  - $R_i \leq T_i$ for $1 \leq i \leq n$

- $S \vdash e_0.f(e_1, \ldots, e_n) : U$
Is this Safe?

- \(f\) is an identifier.
- \(S \vdash e_0 : M\)

- \(f\) is a member function in class \(M\).
- \(f\) has type \((T_1, ..., T_n) \rightarrow U\)
- \(S \vdash e_i : R_i\) for \(1 \leq i \leq n\)
- \(R_i \leq T_i\) for \(1 \leq i \leq n\)

\[ S \vdash e_0.f(e_1, ..., e_n) : U \]

This refers to the static type of the function.
Is this Safe?

\[ f \text{ is an identifier.} \]
\[ S \vdash e_0 : M \]

\[ f \text{ is a member function in class M.} \]
\[ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \]
\[ S \vdash e_i : R_i \text{ for } 1 \leq i \leq n \]
\[ R_i \leq T_i \text{ for } 1 \leq i \leq n \]

\[ S \vdash e_0.f(e_1, \ldots, e_n) : U \]

This refers to the static type of the function.

\[ f \text{ has dynamic type } (V_1, V_2, \ldots, V_n) \rightarrow U \]

and we know that
\[ V_i \leq T_i \text{ for } 1 \leq i \leq n \]
$S \vdash e_0 : M$

$f$ is an identifier.

$S \vdash e_i : R_i$ for $1 \leq i \leq n$

$R_i \leq T_i$ for $1 \leq i \leq n$

$S \vdash e_0.f(e_1, \ldots, e_n) : U$

$R_i \leq T_i$ for $1 \leq i \leq n$

$V_i \leq T_i$ for $1 \leq i \leq n$

This refers to the static type of the function.

$f$ has dynamic type $(V_1, V_2, \ldots, V_n) \rightarrow U$

and we know that

$V_i \leq T_i$ for $1 \leq i \leq n$
Is this Safe?

This refers to the static type of the function.

\( f \) has dynamic type \((V_1, V_2, \ldots, V_n) \rightarrow U\)

and we know that

\( V_i \leq T_i \) for \(1 \leq i \leq n\)

This doesn't mean that

\( R_i \leq V_i \) for \(1 \leq i \leq n\)
A Concrete Example
A Concrete Example

class Fine {
    void nothingFancy(Fine f) {
        /* ... do nothing ... */
    }
}

A Concrete Example

class Fine {
    void nothingFancy(Fine f) {
        /* ... do nothing ... */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken;
    f.nothingFancy(f);
}
A Concrete Example

class Fine {
    void nothingFancy(Fine f) {
        /* … do nothing … */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken;
    f.nothingFancy(new Fine);
}
A Concrete Example

class Fine {
    void nothingFancy(Fine f) {
        /* ... do nothing ... */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken;
    f.nothingFancy(new Fine);
}
A Concrete Example

class Fine {
    void nothingFancy(Fine f) {
        /* ... do nothing ... */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken;
    f.nothingFancy(new Fine);
}
A Concrete Example

class Fine {
    void nothingFancy(Fine f) {
        /* ... do nothing ... */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken;
    f.nothingFancy(new Fine);
}
Covariant Arguments are Unsafe

• Allowing subclasses to restrict their parameter types is **fundamentally unsafe**.

• Calls through base class can send objects of the wrong type down to base classes.

• This is why Java's `Object.equals` takes another `Object`.

• Some languages got this wrong.
  • Eiffel allows functions to be covariant in their arguments; can cause runtime errors.
Contravariant Arguments

class Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Base B) {
        /* ... */
    }
}
Contravariant Arguments

class Super {
}
class Base extends Super {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Base B) {
        /* ... */
    }
}
Contravariant Arguments

class Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Super B) {
        /* ... */
    }
}
Contravariant Arguments

class Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Super B) {
        /* ... */
    }
}
Contravariant Arguments

class Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Super B) {
        /* ... */
    }
}
Is this Safe?

$f$ is an identifier.
S $\vdash e_0 : M$

$f$ is a member function in class $M$.
$f$ has type $(T_1, \ldots, T_n) \rightarrow U$
S $\vdash e_i : R_i$ for $1 \leq i \leq n$
S $\vdash e_i : R_i$ for $1 \leq i \leq n$

$S \vdash e_0.f(e_1, \ldots, e_n) : U$
Is this Safe?

\[ S \vdash e_0 : M \]

\( f \) is an identifier.

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

\( f \) is a member function in class M.

\( f \) has type \((T_1, \ldots, T_n) \to U\)

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

\[ R_i \leq T_i \text{ for } 1 \leq i \leq n \]

\[ S \vdash e_0.f(e_1, \ldots, e_n) : U \]

This refers to the **static** type of the function.
Is this Safe?

\( f \) is an identifier.
\[
S \vdash e_0 : M
\]

\( f \) is a member function in class \( M \).
\( f \) has type \((T_1, \ldots, T_n) \rightarrow U\)
\[
S \vdash e_i : R_i \quad \text{for } 1 \leq i \leq n
\]
\[
R_i \leq T_i \quad \text{for } 1 \leq i \leq n
\]
\[
S \vdash e_0.f(e_1, \ldots, e_n) : U
\]

This refers to the static type of the function.

\( f \) has dynamic type \((V_1, V_2, \ldots, V_n) \rightarrow U\)
and we know that
\[
T_i \leq V_i \quad \text{for } 1 \leq i \leq n
\]
Is this Safe?

- If $f$ is an identifier:
  - $S ⊢ e_0 : M$
- If $f$ is a member function in class $M$:
  - $f$ has type $(T_1, ..., T_n) → U$
  - $S ⊢ e_i : R_i$ for $1 ≤ i ≤ n$
  - $R_i ≤ T_i$ for $1 ≤ i ≤ n$
  - $S ⊢ e_0.f(e_1, ..., e_n) : U$
  - $R_i ≤ T_i$ for $1 ≤ i ≤ n$
  - $T_i ≤ V_i$ for $1 ≤ i ≤ n$

This refers to the static type of the function.

- $f$ has dynamic type $(V_1, V_2, ..., V_n) → U$
- and we know that $T_i ≤ V_i$ for $1 ≤ i ≤ n$
Is this Safe?

- $f$ is an identifier.
  - $S \vdash e_0 : M$

- $f$ is a member function in class $M$.
  - $f$ has type $(T_1, \ldots, T_n) \rightarrow U$
  - $S \vdash e_i : R_i$ for $1 \leq i \leq n$
  - $R_i \leq T_i$ for $1 \leq i \leq n$
  - $S \vdash e_0.f(e_1, \ldots, e_n) : U$

- $R_i \leq T_i$ for $1 \leq i \leq n$
- $T_i \leq V_i$ for $1 \leq i \leq n$
- So
  - $R_i \leq V_i$ for $1 \leq i \leq n$

This refers to the static type of the function.

$f$ has dynamic type $(V_1, V_2, \ldots, V_n) \rightarrow U$
and we know that

$T_i \leq V_i$ for $1 \leq i \leq n$
Contravariant Arguments are Safe

- Intuition: When called through base class, will accept anything the base class already would.
- Most languages do not support contravariant arguments.
- Why?
  - Increases the complexity of the compiler and the language specification.
  - Increases the complexity of checking method overrides.
Contravariant Overrides

class Super {}  
class Duper extends Super {}  
class Base extends Super {  
  bool equalTo(Base B) {  
    /* ... */  
  }  
}  

class Derived extends Base {  
  bool equalTo(Super B) {  
    /* ... */  
  }  
  bool equalTo(Duper B) {  
    /* ... */  
  }  
}
Contravariant Overrides

class Super {}
class Duper extends Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Super B) {
        /* ... */
    }
    bool equalTo(Duper B) {
        /* ... */
    }
}
So What?

- Need to be very careful when introducing language features into a statically-typed language.
- Easy to design language features; hard to design language features that are type-safe.
- Type proof system can sometimes help detect these errors in the abstract.
Summary

- We can extend our type proofs to handle well-formedness proofs.
- The **error type** is convertible to all other types and helps prevent cascading errors.
- Overloading is resolved at compile-time and determines which of many functions to call.
- Overloading ranks functions against one another to determine the best match.
- Functions can safely be **covariant** in their return types and **contravariant** in their argument types.