**CS42/413**

Introduction to Compilers  
Radu Rugina

Lecture 15: Subtyping  
27 Feb 04

---

### Subtypes in Java

<table>
<thead>
<tr>
<th>Interface I₁ extends I₂ ⟨...⟩</th>
<th>Class C implements I ⟨...⟩</th>
<th>Class C₂ extends C₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>I₁</td>
<td>I</td>
<td>C₂</td>
</tr>
<tr>
<td>I</td>
<td>C</td>
<td>C₁</td>
</tr>
<tr>
<td>I₁,₂</td>
<td>C₁ &lt;: I</td>
<td>C₁,₂</td>
</tr>
</tbody>
</table>

---

### Subtype Hierarchy

- Introduction of subtype relation creates a hierarchy of types: subtype hierarchy
- Type or subtype hierarchy

---

### Type-checking

- **Problem:** what are the valid types for an object?
- **Subsumption rule** connects subtyping relation and ordinary typing judgements
  
  \[
  \begin{align*}
  A & \vdash E : S \\
  S & <: T \\
  A & \vdash E : T \\
  \end{align*}
  \]

  \[
  S <: T \implies \text{values}(S) \subseteq \text{values}(T)
  \]

- "If expression E has type S, it also has type T for every T such that S <: T"

---

### Review

- **Objects:** fields, methods, public/private qualifiers
- **Object types:** field types + method signatures
  - Interfaces = pure types  
  - Objects = types and implementation
- **Object inheritance**
  - Induces a subtyping relationship S <: T  
  - Similar for interfaces  
  - Subtyping allows multiple implementations  
  - Java: extends, implements

---

### Type-checking

- Rules for checking code must allow a subtype where a supertype was expected  
- Old rule for assignment:

  \[
  \begin{align*}
  \text{id : T} & \in A \\
  A & \vdash E : T \\
  A & \vdash \text{id = E : T}
  \end{align*}
  \]

  What needs to change here?
Type-checking Overview

- Rules for checking code must allow a subtype where a supertype was expected
- New rule for assignment:

\[
A \rightarrow E : T_p \\
T_p < : T \\
de : T \in A \\
A \rightarrow id = E : T
\]

\[
A \rightarrow E : T_p \\
T_p < : T \\
\text{id} : T \in A \\
A \rightarrow E : T \\
A \rightarrow id = E : T
\]

Type-checking Code

class Assignment extends ASTNode {
    Variable var; ExprNode E;
    Type typeCheck(Symtab s) {
        Type Tp = E.typeCheck(s);
        Type T = s.lookup(var);
        if (Tp.subtypeOf(T)) return T;
        else throw new TypecheckError(E); }
    }

A \rightarrow E : T_p \\
T_p < : T \\
\text{id} : T \in A \\
A \rightarrow id = E : T

Issues

- When are two object(record) types identical?
  - Do struct foo (int x,y;) and struct bar (int x,y;) have the same type?

- We know inheritance (i.e. adding methods and fields) induces subtyping relation

- Issues in the presence of subtyping:
  1. Types of records with object fields
     class C1 { Point p; }
     class C2 { ColoredPoint p; }
  2. Is it safe to allow fields to be written?
  3. Types of functions (methods)
     Point foo(Point p)
     ColoredPoint bar(ColoredPoint p)

Type Equivalence

- Types derived with constructors have names
- When are record types equivalent?
  - When they have the same fields (i.e. same \text{structure})?
    struct point (int x1, y1;) = struct edge (int n1, n2;)
  - ... or only when they have the same \text{names}?
    - Types with the same \text{structure} are different if they have different names

Type Equivalence

- Name equivalence: types are equal if they are defined by the same type constructor expression and bound to the same name
  - C/C++ example:
    struct foo (int x;)
    struct bar (int x;)

- Structural equivalence: two types are equal if their constructor expressions are equivalent
  - C/C++ example:
    typedef struct foo t1[ ];
    typedef struct foo t2[ ];

Is this code legal?
Declared vs. Implicit Subtyping

**Java**
```java
class C1 {
    int x, y;
}
class C2 extends C1 {
    int z;
    C1 a = new C2();
}
```

**Modula-3**
```plaintext
TYPE t1 = OBJECT
   x,y: INTEGER
END;
TYPE t2 = OBJECT
   x,y,z: INTEGER
END;
VAR a: t1 := NEW(t2);
```

Named vs. Structural Subtyping

- **Name equivalence of types (e.g., Java):** direct subtypes explicitly declared; subtype relationships inferred by transitivity
- **Structural equivalence of types (e.g., Modula-3):** subtypes inferred based on structure of types; extends declaration is optional
- Java: still need to check explicit interface declarations similarly to structural subtyping

The Subtype Relation

For records:

\[ S <: T \]
\[ \{ \text{int } x; \text{int } y; \text{int color;} \} <: \{ \text{int } x; \text{int } y; \} ? \]

- Heap-allocated:
  \[
  \begin{array}{c}
  x \\
  y \\
  c
  \end{array}
  <=:
  \[
  \begin{array}{c}
  x \\
  y
  \end{array}
  \]

- Stack allocated:
  \[
  \begin{array}{c}
  x \\
  y \\
  c
  \end{array}
  <=:
  \[
  \begin{array}{c}
  x \\
  y
  \end{array}
  \]

Width Subtyping for Records

- How to formally express subtyping in the presence of structural equivalence?
- Example:
  \[ \{ \text{int } x; \text{int } y; \text{int color;} \} <: \{ \text{int } x; \text{int } y; \} \]
- General rule:
  \[
  n \leq m \\
  A \vdash \{ a_1: T_1; \ldots; a_m: T_m \} <: \{ a_1: T_1; \ldots; a_m: T_m \}
  \]

Object Fields

- Assume fields can be objects
- Subtype relations for individual fields
- How does it translate to subtyping for the whole record?
- If \( \text{ColoredPoint} <: \text{Point} \), allow
  \[
  \{ \text{ColoredPoint } p; \text{int } z; \} <: \{ \text{Point } p; \text{int } z; \} ?
  \]

Field Invariance

- Try \( \{ p: \text{ColoredPoint}; \text{int } z; \} <: \{ p: \text{Point}; \text{int } z; \} \)
- Assume fields can be objects
- Subtype relations for individual fields
- How does it translate to subtyping for the whole record?
- If \( \text{ColoredPoint} <: \text{Point} \), allow
  \[
  \{ \text{ColoredPoint } p; \text{int } z; \} <: \{ \text{Point } p; \text{int } z; \} ?
  \]

 mutable (assignable) fields must be type invariant!
Immutable Record Subtyping

- **Rule:** corresponding immutable fields may be subtypes; exact match not required

\[
\begin{align*}
A \vdash T_j <: T'_j \quad (0 \leq j < n) \\
A \vdash \{a_1: T_1, \ldots, a_n: T_n\} <: \{a_1: T'_1, \ldots, a_n: T'_n\}
\end{align*}
\]

- \( n \leq m \)

\[
A \vdash \{a_1: T'_1, \ldots, a_n: T'_n\} <: \{a_1: T_1, \ldots, a_n: T_n\}
\]

Signature Conformance

- Subclass method signatures must conform to those of superclass
  - Argument types
  - Return type
  - Exceptions
  - How much conformance is really needed?

- **Java rule:** arguments and returns must have identical types, may remove exceptions

Example 1

- Consider the program:
  ```java
  interface List { List rest(int); }
  class SimpleList implements List
  { SimpleList rest(int); }
  ```

- Is this a valid program?
- Is the following subtyping relation correct?

  ```java
  { rest: int→SimpleList } <: { rest: int→List }
  ```

```
int→SimpleList <: int→List ?
```

Example 2

- Consider the program:
  ```java
  class Shape { int setLLCorner(Point p); }
  class ColoredRectangle extends Shape
  { int setLLCorner(ColoredPoint p); }
  ```

- Legal in language Eiffel
- Is this safe?

```
ColoredPoint → int  <:  Point → int ?
```

Function Subtyping

- From definition of subtyping: \( F: T_1 \rightarrow T_2 <: F': T'_1 \rightarrow T'_2 \)
  - if a value of type \( T_1 \rightarrow T_2 \) can be used wherever \( T'_1 \rightarrow T'_2 \)
    is expected

- **Requirement 1:** whenever result of \( F' \) is used, result of \( F \)
  can also be used
  - Implies \( T_2 <: T'_2 \)

- **Requirement 2:** any argument to \( F' \) must be a valid argument for \( F \)
  - Implies \( T'_1 <: T_1 \)

General Rule

- Function subtyping: \( T_1 \rightarrow T_2 <: T'_1 \rightarrow T'_2 \)
- Consider function \( f \) of type \( T_1 \rightarrow T_2 \):

```
T_1'  
\quad T'_1 
\quad T_1 
\quad T_2 
\quad T_2' 
\quad f
```
Contravariance/Covariance

- Function argument types may be contravariant
- Function result types may be covariant

\[
\frac{T_1' : T_1}{T_2' : T_2} \quad \frac{T_1 \to T_2}{T_1' \to T_2'}
\]

- Java is conservative!
  \{ rest: int \to SimpleList \} \ll \{ rest: int \to List \}

Java Arrays

- Java has array type constructor: for any type T, T[ ] is an array of Ts
- Java also has subtype rule:

\[
\frac{T_1 \ll T_2}{T_1[\ ] \ll T_2[\ ]}
\]

- Is this rule safe?

Java Array Subtype Problems

- Example:
  
  \begin{align*}
  \text{Elephant} & \ll \text{Animal, Whale} \ll \text{Animal} \\
  \text{Elephant[\ ]} & \dd y = \text{new Elephant[10]}; \\
  \text{Animal[\ ]} & \dd x = y; \\
  y[0].\text{trunk} & = \text{new Trunk(); } // \text{oops!}
  \end{align*}

- Covariant modification: unsound
- Java does run-time check!

Unification

- Some rules more problematic: if
- Rule:

\[
\begin{align*}
A \leftarrow & E : \text{bool} \\
A \leftarrow & S_1 : T \\
A \leftarrow & S_2 : T \\
A \leftarrow & \text{if ( } E \text{ ) } S_1 \text{ else } S_2 : T
\end{align*}
\]

- Problem: if \( S_1 \) has type \( T_1 \), \( S_2 \) has type \( T_2 \). Old check: \( T_1 = T_2 \). New check: need type \( T \). How to unify \( T_1, T_2 \)?
- Occurs in Java: \(?\) operator

General Typing Derivation

\[
\begin{align*}
A \leftarrow & S_1 : T_1 < : T \\
A \leftarrow & S_2 : T_2 < : T \\
A \leftarrow & \text{if ( } E \text{ ) } S_1 \text{ else } S_2 : T
\end{align*}
\]

How to pick \( T \)?

Unification

- Idea: unified type is least common ancestor in type hierarchy (least upper bound)
- Partial order of types must be a lattice

\[
\text{if ( b ) new C5() else new C3() : I2}
\]

\[
\text{LUB(C3, C5) = I2}
\]

Logic: I2 must be same as or a subtype of any type (e.g, I1) that could be the type of both a value of type C3 and a value of type C5

What if no LUB?