CS412/413
Introduction to Compilers
Radu Rugina

Lecture 15: Subtyping
24 Feb 06

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 \( S \leq T \)
  – Similar for interfaces
  – Subtyping allows multiple implementations
  – Java: extends, implements

Subtypes in Java

interface \( I_1 \) extends \( I_2 \) \{ ... \}
implements \( I \) \{ ... \}
\( I_2 \)
\( I \)
\( I_1 \)
\( C \)
\( C \)
\( C_1 \)
\( C_2 \)

\( I_1 \leq I_2 \)
\( C \leq I \)
\( C_1 \leq C_2 \)

Subtype Hierarchy

• Introduction of subtype relation creates a hierarchy
  of types: subtype hierarchy

Type-checking

• Problem: what are the valid types for an object?
• Subsumption rule connects subtyping relation and
  ordinary typing judgements

\[
\frac{A \vdash E : S \quad S \leq T \quad \text{values}(S) \subseteq \text{values}(T)}{A \vdash E : T}
\]

• “If expression \( E \) has type \( S \), it also has type \( T \) for
  every \( T \) such that \( S \leq T \)”

Type-checking

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

\[
A \leftarrow id : T \quad A \leftarrow E : T
\]

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:

\[ A \leftarrow E : T' \]
\[ T' \leq T \]
\[ \text{id} : T \in A \]
\[ A \leftarrow \text{id} = E \]

Type-checking Code

```java
class Assignment extends ASTNode {
    Symbol id; ExprNode e;
    Type typeCheck(Symbol s) {
        Type Tp = s.typeCheck(s);
        Type T = a.lookup(id);
        if (Tp.typeOf(T)) return T;
        else throw new TypecheckError(E);
    }
}
```

Type Equivalence

• Types derived with constructors have names

• When are record types equivalent?

• When they have the same fields (i.e., same structure)?
  ```
  struct point { int x, y; } -? struct edge { int x1, y2; }
  ```

• ... or only when they have the same names?

  ```
  Types with the same name are different if they have different names
  ```

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)
  ```
  class C1 { Point p; } class C2 { ColoredPoint p; }
  ```

  ```
  Type Equivalence
  ```

```java
class C1 {
    int x, y;
}
class C2 {
    int x, y;
}
class C3 {
    int x, y;
}
```
Declared vs. Implicit Subtyping

Java

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

Modula-3

```modula-3
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

Width Subtyping for Records

- How to formally express subtyping in the presence of structural equivalence?
- Example:
  ```
  {int x; int y; int color; } ≤ { int x; int y; }
  ```
- General rule:
  ```
  n ≤ m
  A = (a0: T1, ..., an: Tn) ≤ (a0: T1', ..., an: Tn')
  ```

Object Fields

- Assume fields can be references to objects
- Subtype relations for individual fields
- How does it translate to subtyping for the whole record?
- If ColoredPoint ≤ Point, should we allow
  ```
  { ColoredPoint p; int z; } ≤ { Point p; int z; }
  ```

Field Invariance

- Try ( p: ColoredPoint; int x; ) ≤ ( p: Point; int x; )
- ```
  class C1 { Point p; int x; }
  class C2 { ColoredPoint p; int x; }
  C2 c2 = new C2();
  C1 c1 = c2;
  c1.p = new Point();
  c2.p.c = 10;
  ```
- Mutable (assignable) fields must be type invariant!

Immutable Record Subtyping

- Rule: corresponding immutable fields may be subtypes (also known as "depth subtyping")
  ```
  A = ( a0: T1, ..., an: Tn ) ≤ ( a0: T1', ..., an: Tn' )
  ```
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:
  interface List { List rest(int); }
  class SimpleList implements List
  { SimpleList rest(int); }
- Is the following subtyping relation safe?
  \( \text{rest: int->SimpleList} \leq \text{rest: int->List} \)
  \( \text{int->SimpleList} \leq \text{int->List} \) ?

Example 2

- Consider the program:
  class Shape { int setLLCorner(Point p); }
  class ColoredRectangle extends Shape
  { int setLLCorner(ColoredPoint p); }
- Legal in language Eiffel
- Is this safe?
  \( \text{ColoredPoint} \to \text{int} \leq \text{Point} \to \text{int} \) ?

Function Subtyping

- From definition of subtyping: \( F: T_1 \to T_2 \leq F': T'_1 \to T'_2 \) if a value of type \( T_1 \to T_2 \) can be used wherever \( T'_1 \to T'_2 \) is expected
- Requirement 1: whenever result of \( F' \) is used, result of \( F \) can also be used
  - Implies \( T_2 \leq T'_2 \)
- Requirement 2: any argument to \( F' \) must be a valid argument for \( F \)
  - Implies \( T'_1 \leq T_1 \)

General Rule

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

Contravariance/Covariance

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

\[
\frac{T'_1 \leq T_1 \quad T_2 \leq T'_2}{T_1 \Rightarrow T_2 \leq T'_1 \Rightarrow T'_2}
\]
- Java is conservative!

\( \text{rest: int->SimpleList} \leq \text{rest: int->List} \)
Java Arrays

- Java has an array type constructor; for any type T
  \[ \text{array}(T) \text{ is an array of } T\text{'s} \]
- Java has the following rule for array subtyping:
  \[
  T_1 \leq T_2 \\
  \text{array}(T_1) \leq \text{array}(T_2)
  \]

- Is this rule safe?

Java Array Subtype Problems

- Example:
  ```java
  Elephant << Animal, Whale << Animal
  Elephant[] y = new Elephant[10];
  Animal[] x = y;
  x[0] = new Whale();
  y[0].trunk = new Trunk(); // oops!
  
  // Covariant modification: unsound
  // Java does run-time check!
  // – Example above throws java.lang.ArrayStoreException
  ```

Unification

- Some rules more problematic
- Rule:
  \[
  A \vdash E_1 : \text{bool} \\
  A \vdash E_2 : T_2 \\
  A \vdash E_3 : T_3 \\
  A \vdash (E_1 ? E_2 : E_3) : ?
  \]

- Problem: How should we combine \( T_2 \) and \( T_3 \)?

Unification

- Idea: unified type is least common ancestor in type hierarchy
  (least upper bound)
- Partial order of types must be a lattice
  \[ b : \text{new C5()} : \text{new C3()} : I2 \]

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

- What if no LUB?