CS412/413
Introduction to Compilers
Radu Rugina

Lecture 31: Subtyping
12 Apr 02

CS 412/413 Spring 2002 Introduction to Compilers 2

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)

CS 412/413 Spring 2002 Introduction to Compilers 3

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 n1, n2}?
  - ... or only when they have the same names?
    - Types with the same structure are different if they have different names

CS 412/413 Spring 2002 Introduction to Compilers 4

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);
  - struct foo # struct bar

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

CS 412/413 Spring 2002 Introduction to Compilers 5
Declared vs. Implicit Subtyping

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

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.

The Subtype Relation

For records:
```
S <: T
{int x; int y; int color; } <: { int x; int y; }
```

- Heap-allocated:
  - X Y <:
  - C

- Stack allocated:
  - X Y <:
  - C

Width Subtyping for Records

- **Example:**
  - \( \{ \text{int x; int y; int color;} \} \leq \{ \text{int x; int y;} \} \)

- **General rule:**
  - \( A \vdash \{ a_1: T_1, \ldots, a_n: T_n \} \leq \{ a_1: T_1, \ldots, a_n: T_n \} \)

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 \( \{ p: \text{ColoredPoint}; \text{int} z; \} <:\{ p: \text{Point}; \text{int} z; \} \)
- \( \text{class C1 (Point p; int z;)} \)
- \( \text{class C2 (ColoredPoint p; int z;)} \)
- \( \text{Point} \)
- \( \text{C1 c1; C2 c2 = new C2();} \)
- \( o1 = o2; \)
- \( o1.p = \text{new Point();} \)
- \( \text{ColoredPoint c2.p.c = 10;} \)
- **Mutable (assignable) fields must be type invariant!**
**Covariance**
- Immutable record fields may be type covariant (may allow subtyping)
- Suppose we allow variables to be declared final
  final int x
- Safe:
  \( \{ \text{final ColoredPoint } p; \text{ int } z \} <: \{ \text{final Point } p; \text{ int } z \} \)

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

\[
\begin{align*}
A & \vdash T_1 <: T_2 \quad (0 \leq i < n) \\
A & \vdash \{ a_1; T_1 \ldots a_n; T_n \} <: \{ a_1; T'_1 \ldots a_n; T'_n \} \\
n & \leq m
\end{align*}
\]

**Function Subtyping**
- Subtyping rules are the same as for records!
  ```
  interface List {
    List rest(int i);
  }
  class SimpleList implements List {
    SimpleList rest(int i);
  }
  ```
- Is this a valid program?
- Is the following subtyping relation correct?
  ```
  \( \text{rest: int} \rightarrow \text{SimpleList} <: \{ \text{rest: int} \rightarrow \text{List} \} \)
  ```

**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

**Function Subtyping**
- Mutable fields of a record must be invariant, immutable fields may be covariant
- Object is mostly a record where methods are immutable, non-final fields mutable
- Type of method fields is a function type: \( T_1 \times T_2 \times T_3 \rightarrow T_n \)
- Subtyping rules for function types will give us subtyping rules for methods

**Function Subtyping**
- class Shape {
  - int setCorner(Point p);
}  
  ```
  class ColoredRectangle extends Shape {
    int setCorner(ColoredPoint p);
  }
  ```
- Legal in language Eiffel. Safe?
- Question:
  ```
  ColoredPoint \rightarrow \text{int} <: \text{Point} \rightarrow \text{int} ?
  ```
Function Subtyping

- From definition of subtyping: \( F: T_1 \rightarrow T_2 \iff F': T'_1 \rightarrow T'_2 \)
  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 \( T_2 \iff T'_2 \)

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

General Rule

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

Contravariance/Covariance

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

\[
\begin{align*}
T'_1 & \ll T_1 \\
T_2 & \ll T'_2 \\
\frac{T_1 \rightarrow T_2}{T'_1 \rightarrow T'_2} \ll \end{align*}
\]

- Java is conservative!

  \{
  \text{rest: int} \rightarrow \text{SimpleList} \}
  \ll
  \{
  \text{rest: int} \rightarrow \text{List} \}

Java Arrays

- Java has array type constructor: for any type \( T \),
  \( T[\_] \) is an array of \( T \)'s
- 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:
  \[
  \text{Elephant} \ll \text{Animal} \\
  \text{Animal[\_] x;} \\
  \text{Elephant[\_] y;} \\
  \text{x(0) = new Rhinoceros(y)} // oops!
  \]

  \[
  \text{Covariant modification: unsound} \\
  \text{Java does run-time check!}
  \]

Unification

- Some rules more problematic: if

  \[
  \frac{A \mapsto E : \text{bool}}{A \mapsto S_1 : T} \\
  \frac{A \mapsto S_2 : T}{A \mapsto \text{if } (E) S_1 \text{ else } S_2 : T}
  \]

- Problem: if \( S_1 \) has principal type \( T_1 \), \( S_2 \) has principal type \( T_2 \).
  Old check: \( T_1 = T_2 \). New check: need principal type \( T \). How to unify \( T_1, T_2 \)?
  - Occurs in Java: \( ? : \text{operator} \)
**General Typing Derivation**

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

How to pick T?

**Unification**

- **Idea:** unified principal type is least common ancestor in type hierarchy (least upper bound)
- Partial order of types must be a lattice
  
  \[
  \text{if } (b) \text{ new } C_5 \text{ else new } C_3 : I_2
  \]
  
  \[
  \text{LUB}(C_3, C_5) = I_2
  \]

Logic: I_2 must be same as or a subtype of any type (e.g., I_1) that could be the type of both a value of type C3 and a value of type C5

What if no LUB?

**Summary**

- Type-checking for languages with subtyping
  - Subtyping rules often counter-intuitive
    - Types of mutable fields can't be changed (invariant), types of immutable fields can
    - Function return types covariant, argument types contravariant ()
    - Arrays must be type invariant (like mutable fields)
- Unification requires LUB