CS412/413

Introduction to Compilers

Radu Rugina

Lecture 15: Subtyping

27 Feb 04

Review

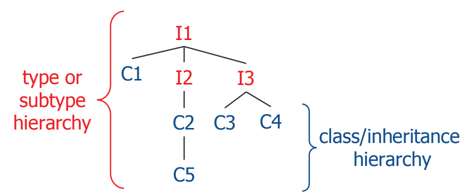
- Objects: fields, methods, public/private qualifiers
- Object types: field types + method signatures
  - Interfaces = pure types
  - Objects = types and implementation
- Object inheritance
  - Induces a subtyping relationship S <: T
  - Similar for interfaces
  - Subtyping allows multiple implementations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

Subtypes in Java

interface I_1 extends I_2 { … }	class C implements I { … }	class C_1 extends C_2
I_2	I	C_2
I_1	C	C_1
I_1 <: I_2	C <: I	C_1 <: 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

$$\frac{A \vdash E : S \qquad S <: T}{A \vdash E : T}$$

S <: T → values(S) ⊆ values(T)

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

Type-checking

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

$$\frac{id : T \in A \qquad A \vdash E : T}{A \vdash id = E \; : T}$$

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:

$$\frac{A \vdash E : T_p \quad T_p <: T \quad id : T \in A}{A \vdash id{=}E : T} \;=\; \frac{A \vdash E : S \quad S <: T}{A \vdash E : T} \;+\; \frac{id : T \in A \quad A \vdash E : T}{A \vdash id = E \;: T}$$

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); }}
```

$$\frac{A \vdash E : T_p \qquad T_p <: T \qquad id : T \in A}{A \vdash id{=}E : T}$$

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 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

Type Equivalence

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

Java: name

```
TYPE t1 = OBJECT
     x,y: INTEGER
END
TYPE t2 = OBJECT
     x,y: INTEGER
END;
VAR a: t1 := NEW(t2);
```

Modula-3: structure

Is this code legal?

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

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

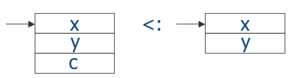
The Subtype Relation

For records:

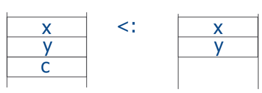
S <: T

{int x; int y; int color; } <: { int x; int y; } ?

- Heap-allocated:



- Stack allocated



Width Subtyping for Records

- How to formally express subtyping in the presence of structural equivalence?
- Example:

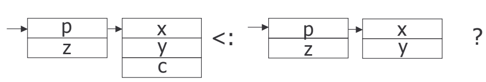
  {int x; int y; int color; } <: { int x; int y; }

- General rule:

$$\frac{n \le m}{A \vdash \{a_1: T_1, \ldots, a_m: T_m\} <: \{a_1: T_1, \ldots, a_n: T_n\}}$$

Object Fields

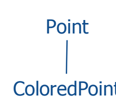
- Assume fields can be objects
- Subtype relations for individual fields
- How does it translate to subtyping for the whole record?
- If ColoredPoint <: Point, allow
  { ColoredPoint p; int z; } <: { Point p; int z; } ?



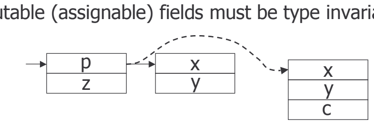
Field Invariance

- Try { p: ColoredPoint; int z; } <: { p: Point; int z; }

```
class C1 { Point p; int z; }
class C2 { ColoredPoint p; int z; }
C2 o2 = new C2();
C1 o1 = o2;
o1.p = new Point( );
o2.p.c = 10;
```



- Mutable (assignable) fields must be type invariant!



Immutable Record Subtyping

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

$$\frac{A \vdash T_i <: T_i' \ ^{(i \in 1..n)}}{A \vdash \{a_1: T_1 \dots a_n{:}T_n\} <: \{a_1: T_1' \dots a_n: T_n'\}}$$

$$\frac{n \le m}{A \vdash \{a_1: T_1, \dots, a_m: T_m\} <: \{a_1: T_1, \dots, a_n: T_n\}}$$

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:
  interface List { List rest(int); }
  class SimpleList implements List
  { SimpleList rest(int); }
- Is this a valid program?
- Is the following subtyping relation correct?

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

int→SimpleList <: 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?

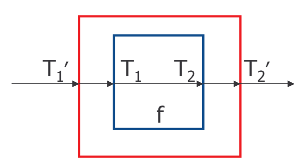
ColoredPoint → int <: Point → int ?

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 $T_1 \rightarrow T_2$:



Contravariance/Covariance

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

$$\frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \rightarrow T_2 \;<:\; T_1' \rightarrow T_2'}$$

- Java is conservative!

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

Java Arrays

- Java has array type constructor: for any type T, T [ ] is an array of T's
- Java also has subtype rule:

$$\frac{T_1 <: T_2}{T_1[] <: T_2[]}$$

- Is this rule safe?

Java Array Subtype Problems

- Example:

```
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!

Unification

- Some rules more problematic: if
- Rule:

$$\frac{A \vdash E : bool \qquad A \vdash S_1 : T \qquad A \vdash S_2 : T}{A \vdash \text{if (} E \text{) } S_1 \text{ else } S_2 : T}$$

- 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

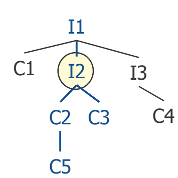
$$\frac{A \vdash E : bool \qquad \dfrac{A\vdash S_1 : T_1 \quad T_1 <: T}{A \vdash S_1 : T} \qquad \dfrac{A\vdash S_2 : T_2 \quad T_2 <: T}{A \vdash S_2 : T}}{A \vdash \text{if (} E \text{) } S_1 \text{ else } S_2 : T}$$

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

if (b) new C5() else new C3() : I2



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?