CS412/413

Introduction to
Compilers and Translators
Andrew Myers
Cornell University

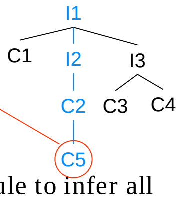
Lecture 20: Subtyping
13 March 00

Outline

- Last time: classes, interfaces in Java, Iota+ have a subtype relation
- Type-checking with subtypes
- Subtyping rules
  - Records, functions, methods
  - Why Java arrays are broken

Principal Type

- Idea: every expression has a *principal type* that is the most-specific type of the expression



- Can use subsumption rule to infer all supertypes if principal type is used

C5 <: C2 <: I2 <: I1

Type-checking interface

- Old method for checking types:

```
abstract class Node {
    abstract Type typeCheck(SymTab A);
      // Return the principal type of this
      // statement or expression
}
```

- No changes in interface needed to support subtyping, except interpretation of result of `typeCheck`

Type-checking rules

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

$$\frac{\begin{array}{c} id : T \in A \\ A \vdash E : T \end{array}}{A \vdash id = E\ \ : T}$$

What needs to change in implementation?

Type-checking code

```
class Assignment extends ASTNode {
   String id; Expr E;
   Type typeCheck(SymTab A) {
        Type Tp = E.typeCheck(A);
        Type T = A.lookup(id);
        if (Tp.subtypeOf(T)) return T;
        else throw new TypecheckError(E);
   }
}
```

$$\frac{\begin{array}{c} A \vdash E : T_p \\ id : T \in A \\ T_p <: T \end{array}}{A \vdash id = E;\ : T} \;=\; \overset{\textit{subsumption rule}}{\frac{\begin{array}{c} A \vdash E : T_p \\ T_p <: T \end{array}}{A \vdash E : T}} \;+\; \overset{\textit{original typing rule}}{\frac{\begin{array}{c} id : T \in A \\ A \vdash E : T \end{array}}{A \vdash id = E;\ : T}}$$

Unification

- Some rules more problematic: if
- Rule:

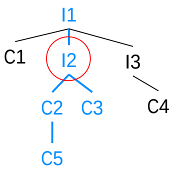
$$\frac{\begin{array}{c} A \vdash E : \mathbf{bool} \\ A \vdash S_1 : T \\ A \vdash S_2 : T \end{array}}{A \vdash \text{if } (\ E\ )\ S_1 \text{ else } S_2 : T}$$

- 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: ?: operator

Unification

- Idea: unified principal type is least common ancestor in type hierarchy (*least upper bound* in partial order)

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?

Type equivalence

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

Java: *name*

```
TYPE t1 = OBJECT
      x,y: INTEGER
END
TYPE t2 = OBJECT
      x,y: INTEGER
END
x: t1 := NEW t2
```

Modula-3: *structure*

Is this code legal?

Declared vs. implicit subtyping

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

Java

```
TYPE t1 = OBJECT
    x,y: INTEGER
END
TYPE t2 = OBJECT
    x,y,z: INTEGER
END
a: t1 = NEW t2
```

Modula-3

Java declarations

```
interface List {
  List rest( );
}
class SimpleList implements List {
  SimpleList rest( );
}
```

Is this a legal Java program?

Determining subtyping

- When is one type considered a subtype of another?
- Java, C++: if there is a extends/implements declaration *and* that declaration is legal
  - rules for checking legality are conservative
- Languages with structural type equivalence: explicit declaration not needed

Named vs. structural subtyping

- Java: all subtypes explicitly declared, name equivalence for types. Subtype relationships inferred by transitive extension.
- Languages with structural equivalence (e.g., Modula-3): subtypes inferred based on structure of types; `extends` declaration is optional
- Same checking done in each case!

Most permissive safe rules for implicit subtype

= most permissive safe rules for checking a subtype declaration

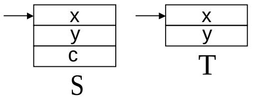
Testing subtype relation

- **Subtyping for records**

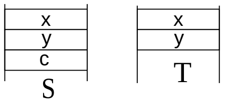
S <: T

{x: int, y: int, c: Color} <: { x: int, y: int } ?

- Impl #1:



- Impl #2



Subtype rule for records by reference

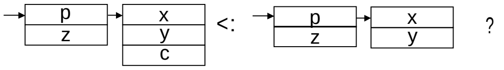
{x: int, y: int, c: Color} ≤ { x: int, y: int }

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

- Similar to our rule for checking modules: okay to extend record

Varying record field types

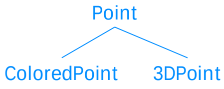
- What about allowing field types to vary?
- If ColoredPoint <: Point, allow { p:ColoredPoint, z:int } <: { p:Point, z:int } ?



Field Invariance

Try { p: ColoredPoint } <: { p: Point }

x: {p: Point}
y: {p: ColoredPoint}
x = y;
x.p = new 3DPoint( );

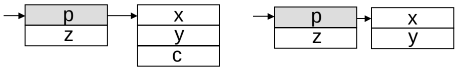


- Mutable (assignable) fields must be *invariant* in subtyping relation

Covariance

- Immutable record fields *may* change with subtyping (may be *covariant*)
- Suppose we allow variables to be declared *final* -- x : final int
- Safe:

{ p: final ColoredPoint, z: int } <: { p: final Point, z: int }



Immutable record subtyping

- Corresponding fields may be subtypes; exact match not required

$$\frac{m \geq n \qquad T_i <: T'_i \ (i \in 1..n)}{\{a_1 : T_1 \ldots a_m : T_m\} <: \{a_1 : T'_1 \ldots a_n : T'_n\}}$$

Record subtyping

	mutable fields	*immutable fields*
stack-allocated	subtyping = type equality C struct	covariant subtyping on fields [C++ class]
reference	can add new fields only Java class C++ class *	can add new fields, field types covariant [Java class C++ class *]

Subtyping on classes

- Subtyping rules are the same as for records!

```
interface List { List rest(int); }
class SimpleList implements List { SimpleList rest(int); }
```

⇒ declaration SimpleList implements List is safe if

$$\{ \text{rest: int} \rightarrow \text{SimpleList} \} <: \{ \text{rest: int} \rightarrow \text{List} \}$$

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 be identical, may remove exceptions

Checking conformance

- Mutable fields of a record must be invariant, immutable fields may be covariant
- Object is essentially a record where methods are immutable fields
- 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 tell us subtyping rules for methods

Function type subtyping

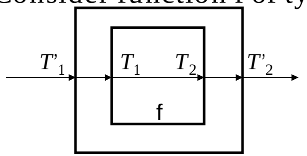
```
class Shape {
  int setLLCorner(Point p);
}
class ColoredRectangle extends Shape {
  int setLLCorner(ColoredPoint p);
}
```

- Legal in language Eiffel. Safe?
- *Question*:

  ColoredPoint → int <: Point → int ?

General rule

- *From definition of subtyping*:
  $T_1 \to T_2 <: T'_1 \to T'_2$ if a value of type $T_1 \to T_2$ can be used wherever $T'_1 \to T'_2$ is expected
- Consider function f of type $T_1 \to T_2$:



Contravariance/covariance

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

$$\frac{\begin{array}{c} T'_1 <: T_1 \\ T_2 <: T'_2 \end{array}}{T_1 \to T_2 \;<:\; T'_1 \to T'_2}$$

- 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

Elephant <: Animal
Animal [ ] x;
Elephant [ ] y;
x = y;
x[0] = new Rhinoceros(); // oops!

- Assignment as method:
  Animal[ ] : void setElem (Animal, int)
  Elephant[ ] : void setElem (Elephant, int)
- *covariant modification: unsound*
- Java does run-time check!

Summary

- Type-checking for languages with subtyping
- Often counter-intuitive
  - Mutable fields can't be changed (invariant), immutable fields can
  - Function return types covariant, argument types contravariant (!)
  - Arrays are invariant (like mutable fields)