CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 21: Subtyping
16 March 01

Review

- Multiple implementations supported by *subtyping*
- Subtyping characterized by new judgement: $S <: T$
- $A \vdash e : T$ judgement+ subsumption rule + $S <: T$ judgement = new type-checking
- extends, implements declarations declare a subtype relationship
- Question: when is $S <: T$?

Java declarations

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

Is this a legal Java program?

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?

Type equivalence

- *Name equivalence*:
  - Two types are equal if they are defined by the same type constructor expression (and bound to the same name)

  C/C++:

  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++:

  typedef struct foo t1[ ]; ...; typedef struct foo t2[ ];

  t1 = t2

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

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
- Java, etc: still need to check explicit declarations using same rules as for structural subtyping

Most permissive safe rules for implicit subtype
= most permissive safe rules for checking a subtype declaration

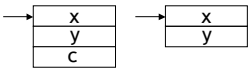
The subtype relation

For records:

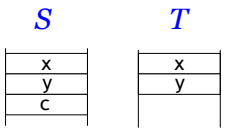
$$S <: T$$

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

- Impl #1:



- Impl #2



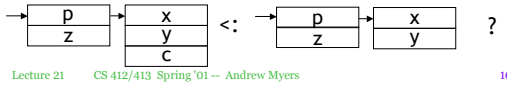
Width subtyping for records

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

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

Varying record field types

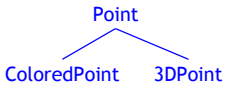
- What about allowing field types to vary?
- If ColoredPoint <: Point, allow
  { p:ColoredPoint, z:int } <: { p:Point, z:int } ?



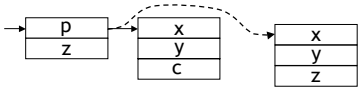
Field Invariance

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

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



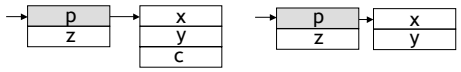
- Mutable (assignable) fields must be *invariant* under subtyping



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

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

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

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

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

int→SimpleList <: int→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 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 type subtyping

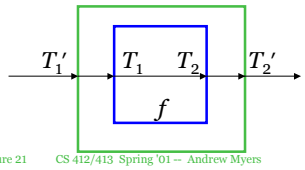
```
class Shape {
  int setLLCorner(Point p);
}
class ColoredRectangle extends Shape {
  int setLLCorner(ColoredPoint p);
}
```

- 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*

$$\frac{T_1' <: T_1 \quad T_2 <: T_2'}{T_1 \to T_2 \ <: \ T_1' \to T_2'}$$

- 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!

Unification

- Some rules more problematic: if
- Rule:

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

- 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

General typing derivation

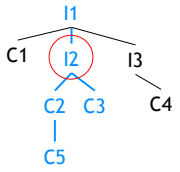
$$\frac{A \vdash E : \textbf{bool} \quad \dfrac{A \vdash S_1 : T_1 \quad T_1 <: T}{A \vdash S_1 : T} \quad \dfrac{A \vdash S_2 : T_2 \quad T_2 <: T}{A \vdash S_2 : T}}{A \vdash \text{if } (E)\ S_1 \text{ else } S_2 : T}$$

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 semilattice

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?

Summary

- Type-checking for languages with subtyping
- Subtyping rules 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)
- Unification requires LUB