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

\[ C_5 <: C_2 <: I_2 <: I_1 \]

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

Type-checking interface
- Old method for checking types:

```java
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 code

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

What needs to change in implementation?
Unification

- Some rules more problematic:
- Rule: $A \vdash E : \text{bool}
  A \vdash S_1 : T
  A \vdash S_2 : T
  \frac{A \vdash \text{if } (E) S_1 \text{ else } S_2 : T}{A \vdash \text{if } (E) S_1 \text{ else } S_2 : T}$
- 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

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  

class C2 extends C1 {  
  int z;  
}  
C1 a = new C2()  
Modula-3: structure
```

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  

class C2 extends C1 {  
  int z;  
}  
C1 a = new C2()  
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
  \[ S <: T \]
  \[ \{x: \text{int}, y: \text{int}, c: \text{Color}\} <: \{x: \text{int}, y: \text{int}\} ? \]
  
  Impl #1:
  \[
  \begin{array}{ccc}
  x & y & c \\
  S & T \\
  \end{array}
  \]

  Impl #2
  \[
  \begin{array}{ccc}
  x & y & c \\
  S & T \\
  \end{array}
  \]

Varying record field types

- What about allowing field types to vary?
- If \( \text{ColoredPoint} <: \text{Point} \), allow
  \[ \{p: \text{ColoredPoint}, z: \text{int}\} <: \{p: \text{Point}, z: \text{int}\} ? \]

Field Invariance

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

x: \{p: \text{Point}\}

y: \{p: \text{ColoredPoint}\}

x = y;

\(x.p = \text{new 3DPoint}()\);

- 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 -- \(x: \text{final int}\)
- Safe:
  \[ \{p: \text{final ColoredPoint}, z: \text{int}\} <: \{p: \text{final Point}, z: \text{int}\} \]

Subtype rule for records by reference

\[ \{x: \text{int}, y: \text{int}, c: \text{Color}\} \leq \{x: \text{int}, y: \text{int}\} \]

\[ m \geq n \]

\(\{a_1: T_1, \ldots, a_m: T_m\} \leq \{a_1: T_1, \ldots, a_n: T_n\}\)

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

- Corresponding fields may be subtypes; exact match not required

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

Record subtyping

\[
\begin{array}{|c|c|}
\hline
\text{mutable fields} & \text{immutable fields} \\
\hline
\text{subtyping} & \text{covariant subtyping} \\
\text{type equality} & \text{on fields} \\
\text{C struct} & \text{[C++ class]} \\
\hline
\text{can add new fields only} & \text{can add new fields, field types covariant} \\
\text{Java class} & \text{Java class} \\
\text{C++ class *} & \text{C++ class *} \\
\hline
\end{array}
\]

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 \((T_1 \times T_2 \times T_3 \rightarrow T_4)\)
- 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: \(\text{ColoredPoint} \rightarrow \text{int} \ <: \text{Point} \rightarrow \text{int}\)?
General rule

- From definition of subtyping:
  \[ T_1 \rightarrow T_2 <: T'_1 \rightarrow T'_2 \] if a value of type
  \( T_1 \rightarrow T_2 \) can be used wherever
  \( T'_1 \rightarrow T'_2 \) is expected
- Consider function \( f \) of type \( T_1 \rightarrow T_2 \):

  \[
  \begin{array}{c}
  T_1 \\
  T_1 \rightarrow T_2 \\
  f \\
  T_2 \\
  T'_1 \\
  T'_1 \rightarrow T'_2
  \end{array}
  \]

Contravariance/covariance

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

\[
T'_1 <: T_1 \\
T'_2 <: T_2 \\
T_1 \rightarrow T_2 <: T'_1 \rightarrow T'_2
\]

- Java is conservative!

\[
\{ \text{rest: int} \rightarrow \text{SimpleList} \} <: \{ \text{rest: int} \rightarrow \text{List} \}
\]

Java arrays

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

\[
T_1 <: T_2 \\
T_1[\ ] <: T_2[\ ]
\]

- 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)