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

Class C1: TYPE \( t1 = \) OBJECT \n\( x,y: INTEGER \) END

Type \( t2 = \) OBJECT \n\( x,y: INTEGER \) END

\( x: t1 := \) NEW \( t2 \)

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: name

```
class C1 {
    int x, y;
}
class C2 extends C1 {
    int z;
}
```

Class C1: TYPE \( t1 = \) OBJECT \n\( x,y: INTEGER \) END

Type \( t2 = \) OBJECT \n\( x,y,z: INTEGER \) END

\( a: t1 := \) NEW \( t2 \)

Java

Modula-3

```
class C1 {
    int x, y;
}
class C2 extends C1 {
    int z;
}
```

Class C1: TYPE \( t1 = \) OBJECT \n\( x,y: INTEGER \) END

Type \( t2 = \) OBJECT \n\( x,y,z: INTEGER \) END

\( a: t1 := \) NEW \( t2 \)

Java

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:

\[ S <: T \]

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

- Impl #1:

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

- Impl #2

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

---

Width subtyping for records

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

\[ n \leq m \]

\[ A \vdash \{ a_1: T_1, \ldots, a_m: T_m \} <: \{ a_1: T_1, \ldots, a_m: T_m \} \]

---

Varying record field types

- What about allowing field types to vary?
- If ColoredPoint <: Point, allow

\[ \{ p: \text{ColoredPoint}, z: \text{int} \} <: \{ p: \text{Point}, z: \text{int} \} ? \]

---

Field Invariance

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

- 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

\[ \text{final} -- x : \text{final int} \]

- Safe:

\[ \{ p: \text{final ColoredPoint}, z: \text{int} \} <: \{ p: \text{final Point}, z: \text{int} \} \]
Immutable record subtyping

- Corresponding fields may be subtypes; exact match not required

\[
A \vdash T_i \ 	ext{type equality} \quad (i \in 1..n) \\
A \vdash \{ a_1: T_1 \ldots a_n: T_n \} \ 	ext{covariant subtyping on fields} \\
\quad (\text{can add new fields, field types covariant}) \\
A \vdash \{ a_1: T_1 \ldots a_n: T_n \} \quad \text{can add new fields only} \\
\quad \text{Java class} \quad \text{C++ class *} \\
\quad \text{stack-allocated} \quad \text{reference} \\
\end{align}
\]

\[
A \vdash \{ a_1: T_1 \ldots a_n: T_m \} \\
\quad \text{mutable fields} \quad \text{immutable fields} \\
\end{align}
\]

Subtyping on classes

- Subtyping rules are the same as for records!

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

\[
\Rightarrow \text{declaration SimpleList implements List is safe if} \\
\quad \{ \text{rest: int} \rightarrow \text{SimpleList} \} \ 	ext{covariant subtyping on fields} \\
\quad \text{can add new fields, field types covariant} \\
\quad \text{Java class} \quad \text{C++ class *} \\
\quad \text{stack-allocated} \quad \text{reference} \\
\end{align}
\]

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 \((T_1 \times T_2 \rightarrow T_m)\)
- 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:
  \[
  \text{ColoredPoint} \rightarrow \text{int} \ 	ext{covariant subtyping on fields} \\
  \quad \text{can add new fields, field types covariant} \\
  \quad \text{Java class} \quad \text{C++ class *} \\
  \quad \text{stack-allocated} \quad \text{reference} \\
  \end{align}
\]
General rule

- From definition of subtyping:
  \[ T_1 \rightarrow T_2 \prec T'_1 \rightarrow T'_2 \] 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 \( T_1 \rightarrow T_2 \):

\[
\begin{array}{c}
T_1' \\
T_1 \\
T_2 \\
T_2'
\end{array}
\]

Contravariance/covariance

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

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

Java is conservative!

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

Java arrays

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

\[
\begin{align*}
T_1 : & T_2 \\
T_1[\ ] : & T_2[\ ]
\end{align*}
\]

Is this rule safe?

Java array subtype problems

- Elephant <: Animal
- Animal[ ] \( x; \)
- Elephant[ ] \( y; \)
- \( x = y; \)
- \( x[0] = \text{new Rhinoceros}(); // oops! \)
- 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:

\[
\begin{align*}
A \vdash E : \text{bool} \\
A \vdash S_1 : T \\
A \vdash S_2 : T \\
A \vdash \text{if } (E) S_1 \text{ else } S_2 : T
\end{align*}
\]

- 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

\[
\begin{align*}
A \vdash S_1 : T_1 <: T_3 \\
A \vdash S_2 : T_2 <: T_3 \\
A \vdash \text{if } (E) S_1 \text{ else } S_2 : T
\end{align*}
\]

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 semilattice

\[
\text{if } (b \text{ new } C5) \text{ else new } C3 : I2
\]

\[
\text{LUB}(C3, C5) = I2
\]

**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