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 $C_2$

$I_2$
$I_1$
$I$
$C$
$C_1$
$C_2$

$I_1 <: I_2$
$C <: I$
$C_1 <: C_2$

---

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
  \[
  \frac{A \vdash E : S}{S <: T} \quad S <: T \rightarrow \text{values}(S) \subseteq \text{values}(T)
  \]
  - "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:
  \[
  \frac{\text{id} : T \in A}{A \vdash \text{id} = E : T}
  \]
  \[
  \frac{A \vdash E : T}{A \vdash \text{id} = E : T}
  \]
  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:

\[
\begin{align*}
A \vdash E : T_p \\
T_p \ll T &\quad \Rightarrow A \vdash E : S \quad \frac{\text{id} : T \in A}{A \vdash \text{id} = E : T} \\
A \vdash E : T &\quad \quad \frac{S \ll T}{A \vdash E : T} \quad \frac{A \vdash \text{id} = E : T}{A \vdash \text{id} = E : T} \\
\end{align*}
\]

Type-checking Code

```java
class Assignment extends ASTNode {
  Variable var; ExprNode E;
  Type typeCheck() {
    Type Tp = E.typeCheck();
    Type T = var.getType();
    if (Tp.subtypeOf(T)) return T;
    else throw new TypecheckError(E); }
}
```

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)

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 x1, n2; } = 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

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

<table>
<thead>
<tr>
<th>Java: name</th>
<th>Modula-3: structure</th>
</tr>
</thead>
</table>
| Is this code legal? | TYPE t1 = OBJECT x,y: INTEGER END;
                   TYPE t2 = OBJECT x,y: INTEGER END;
                   VAR a: t1 := NEW(t2); |

| 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

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

Modula-3

```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:

\( S <: T \)

\( \{ \text{int } x; \text{int } y; \text{int color; } \} <: \{ \text{int } x; \text{int } y; \} \)?

- Heap-allocated:

  \[
  \begin{array}{c}
  x \\
  y \\
  c
  \end{array}
  \]

  \[
  \begin{array}{c}
  x \\
  y \\
  c
  \end{array}
  \]

- Stack allocated:

  \[
  \begin{array}{c}
  x \\
  y \\
  c
  \end{array}
  \]

  \[
  \begin{array}{c}
  x \\
  y \\
  c
  \end{array}
  \]

Width Subtyping for Records

- Example:

  \( \{ \text{int } x; \text{int } y; \text{int color; } \} <: \{ \text{int } x; \text{int } y; \} \)

- General rule:

  \[
  n \leq m
  \]

  \[
  A \rightarrow \{ a_1: T_1, \ldots, a_n: T_n \} <: \{ a_1: T_1, \ldots, a_n: T_n \}
  \]

Object Fields

- Assume fields can be objects
- Subtype relations for individual fields
- How does it translate to subtyping for the whole record?
- If \( \text{ColoredPoint} <: \text{Point} \), allow

  \[
  \begin{array}{c}
  \text{ColoredPoint } p; \text{int } z \} <: \{ \text{Point } p; \text{int } z; \}
  \]

Field Invariance

- Try \( \{ p: \text{ColoredPoint}; \text{int } z \} <: \{ p: \text{Point}; \text{int } z \} \)

  \[
  \begin{array}{c}
  \text{ColoredPoint } c1 \}
  \begin{array}{c}
  \text{Point } c2 \}
  \begin{array}{c}
  \text{Point } p1 = \text{new } Point();
  \begin{array}{c}
  \text{ColoredPoint } p2.c = 10;
  \end{array}
  \end{array}
  \end{array}
  \]

- Mutable (assignable) fields must be type invariant!
Immutable Record Subtyping

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

\[ A \vdash T_j \subset: T'_j \ (i = 1 \ldots n) \]
\[ A \vdash \{ a_1; T_1 \ldots a_n; T_n \} \subset: \{ a_i; T'_1 \ldots a_i; T'_n \} \]

\[ n \leq m \]
\[ A \vdash \{ a_1; T'_1 \ldots a_n; T'_n \} \subset: \{ a_i; T'_1 \ldots a_i; T'_n \} \]

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?

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

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?
  \[ \text{ColoredPoint} \rightarrow \text{int} \subset: \text{Point} \rightarrow \text{int} ? \]

Function Subtyping

- From definition of subtyping: \( F: T_1 \rightarrow T_2 \subset: F': T'_1 \rightarrow T'_2 \)
  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
  \[ \text{Implies} \ T_2 \subset: T'_2 \]

- Requirement 2: any argument to \( F \) must be a valid
  argument for \( F' \)
  \[ \text{Implies} \ T'_1 \subset: T_1 \]

General Rule

- Function subtyping: \( T_1 \rightarrow T_2 \subset: T'_1 \rightarrow T'_2 \)
- Consider function \( f \) of type \( T_1 \rightarrow T_2 \):

\[ T'_1 \]
\[ T_1 \]
\[ T_2 \]
\[ T'_2 \]
\[ f \]
**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!
  
  \{ rest: int \rightarrow SimpleList \} < : \{ rest: int \rightarrow 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**

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

- Covariant modification: unsound
- Java does run-time check!

---

**General Typing Derivation**

- Idea: unified type is least common ancestor in type hierarchy (least upper bound)
- Partial order of types must be a lattice

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

How to pick \( T \) ?

---

**Unification**

- Some rules more problematic: if
- Rule:

\[
A \vdash E : \text{bool} \\
A \vdash S_1 : T \\
A \vdash S_2 : T
\]

\[
A \vdash \text{if } (E) S_1 \text{ else } S_2 : T
\]

- 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: \( ? : \text{operator} \)

---

**Unification**

- Idea: unified type is least common ancestor in type hierarchy (least upper bound)
- Partial order of types must be a lattice

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

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

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?