CMSC 631 – Program Analysis and Understanding
Fall 2003

Subtyping

Subtyping

- Called *subclassing* in object-oriented programming

```
class A { ... }
class B extends A { ... }
A x = new A(); // valid
A x = new B(); // valid, since B is a subclass of A
B x = new A(); // invalid
```


Definition of Subtyping

- Liskov:
  - If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of o_1, the behavior of P is unchanged when o_2 is substituted for o_1 then S is a subtype of T.
- Informal statement
  - If anyone expecting a T can be given an S instead, then S is a subtype of T.

The Subtyping Relation

- Let's assume that we have
  - A set of primitive objects c (e.g., 0, 1, 0.1, 3.14, ...)
    - e ::= c | x | λx.e | e e
  - A set of primitive types C (e.g., int, float)
    - t ::= C | t → t
- We are also given a partial order ≤ on C
  - This is the subtyping relation
  - E.g., int ≤ float
  - Warning: this is a terrible example; implicitly allowing integer to floating point conversions leads to confusion

Type Checking Rules

$$\frac{}{A \vdash c : C(c)} \qquad \frac{x \in dom(A)}{A \vdash x : A(x)}$$

$$\frac{A, x{:}t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'} \qquad \frac{A \vdash e1 : t \to t' \quad A \vdash e2 : t}{A \vdash e1\ e2 : t}$$

$$\frac{A \vdash e : t \quad t \leq t}{A \vdash e : t}$$

Example

$$\dfrac{\dfrac{\vdash +. : f \to f \to f \qquad \dfrac{\vdash 3 : i \quad i \leq f}{\vdash 3 : f}}{\vdash +.\ 3 : float \to float} \qquad \vdash 4.0 : float}{\vdash +.\ 3\ 4.0 : float}$$

Subtyping and Type Constructors

- We're given $\leq$ on primitive types
  - How do we extend to type constructors?
    - A *type constructor* builds new types from existing types
    - E.g., $\rightarrow$, $\times$, ref
-
- Product types are straightforward

$$\frac{t1 \leq t1 \quad t2 \leq t2}{t1 \times t2 \leq t1 \times t2}$$

- Example: $int \times float \leq float \times float$

Subtyping and Function Types

- What about function types?

$$\frac{?}{t1 \rightarrow t2 \leq t1 \rightarrow t2}$$

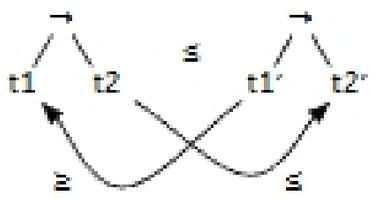
- Recall: S is a subtype of T if an S can be used anywhere a T is expected
  - When can we replace R[f x] with R[g x]?

Replacing R[f x] by R[g x]

- Assume $f : tf \rightarrow tf'$ and $g : tg \rightarrow tg'$
  - Also assume R[f x] type checks
  - When is $tg \rightarrow tg' \leq tf \rightarrow tf'$?
- Return type:
  - Every possible result of g x must be accepted
  - So $tg' \leq tf'$
- Argument type:
  - Every possible argument to f must also be accepted by g
  - So $tf \leq tg$

The Subtype Rule for Functions

$$\frac{t1 \leq t1 \quad t2 \leq t2}{t1 \rightarrow t2 \leq t1 \rightarrow t2}$$



- We say that $\rightarrow$ is
  - *Covariant* in its range (subtyping dir stays the same)
  - *Contravariant* in its domain (subtyping dir flips)

Subtyping and References

- The wrong rule for references

$$\frac{t \leq t'}{ref\ t \leq ref\ t}$$

Counterexample:

```
let x = ref 0 in
let y = x in
  y := 3.14;      // typechecks, since int ≤ float
  printInt (!x)   // oops! typechecks, since y : ref int
```


The Right Rule for References

- Reduce it to the result from functions
  - A reference is like an object with two methods
    - get : unit $\rightarrow$ t reads the value of the ref
    - set : t $\rightarrow$ unit writes the value of the ref
  - Notice that t occurs both co- and contravariantly
-
- The right rule:

$$\frac{t \leq t' \quad t' \leq t}{ref\ t \leq ref\ t} \quad \text{or} \quad \frac{t = t'}{ref\ t \leq ref\ t}$$

  - We say that ref is *nonvariant* or *invariant*

Subtyping Mistakes

- Well-known languages have gotten subtyping wrong
  - Eiffel function types are covariant in the domain
  - The Java array constructor is covariant, not invariant
    - S[] is a subtype of T[] if S is a subtype of T
- How do they get around the unsoundness?
  - Java adds run-time (dynamic) checks

Type Checking Considerations

- Our type system with subtyping was just our previous type system with the extra rule

$$\frac{A \vdash e : t \quad t \leq t}{A \vdash e : t} \text{ (Sub)}$$

- This rule seems to add non-determinism
  - We can apply it to any term, as often as we like

Type Checking Considerations (cont'd)

- Observation 1: Multiple sequential uses of (Sub) can be replaced with one single use
  - Proof: Transitivity of ≤
  -
- Observation 2: All uses of (Sub) can be pushed down the typing proof to occur just before function application
  - Proof: Omitted
  -
- Consequence: Can integrate (Sub) into other rules

A Type Checking Algorithm

$$\frac{}{A \vdash c : C(c)} \qquad \frac{x \quad dom(A)}{A \vdash x : A(x)}$$

$$\frac{A, x{:}t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'} \qquad \frac{A \vdash e1 : t \to t' \quad A \vdash e2 : t2 \quad t2 \leq t}{A \vdash e1\ e2 : t}$$

- These rules are deterministic
  - Easy to construct an algorithm from them
- This is sometimes called a *syntax-driven* system
  - At every step, rule choice determined by syntax

Subtype Polymorphism

- Subtyping (or subclassing from OOP) gives us one kind of *polymorphism*
  - A *polymorphic* type represents multiple types
  - For subtyping, we can think of type A as representing A and all of A's subtypes
  - This is called subtype polymorphism

Limitations of Subtyping

- Suppose $S \leq T$, and consider the four possible types identity function on S and T
  - $\lambda x.x : S \to S$ can't accept T's
  - $\lambda x.x : S \to T$ can't accept T's
  - $\lambda x.x : T \to S$ ill typed
  - $\lambda x.x : T \to T$ can accept S or T, but returns a T
- With parametric polymorphism, we can give this type $\alpha.\alpha \to \alpha$