Subtyping
[redacted version]

Adam Chlipala
MIT

6.820
September 30, 2013
void foo(int n) {
    float f = n;
    // ...and so on.
}
Subtyping in Java: Interfaces

interface List {
    List append(List);
}

class Nil implements List {
    Nil() {
    }
    List append(ls) {
        return ls;
    }
}

class Cons implements List {
    private int data;
    private List tail;
    Cons(int d, List t) {
        data = d;
        tail = t;
    }
    List append(ls) {
        return Cons(data, tail.append(ls));
    }
}
Subtyping in Java: Inheritance

class Cons implements List {
    /* ... */
}

class LoggingCons extends Cons {
    private int numAppends;

    LoggingCons(int d, List t) {
        super(d, t);
        numAppends = 0;
    }

    List append(ls) {
        ++numAppends;
        return super.append(ls);
    }

    int howManyAppends() { return numAppends; }
}
Subtyping as Graph Search

- Arrow from A to B to indicate that B is a “direct” subtype of A

Types:
- float
- int
- List
- Nil
- Cons
- LoggingCons
Subtyping as a Formal Judgment

Reflexivity: \[ \tau \leq \tau \]

Primitive rule: \[ \text{int} \leq \text{float} \]

Inheritance: \[ \text{class A extends B} \]
\[ A \leq B \]

Interfaces: \[ \text{class A implements B} \]
\[ A \leq B \]

This style of subtyping is called **nominal**, because the edges between user-defined types are all declared **explicitly**, via the **names** of those types.
What is Subtyping, Really?

Assume we have some operator $[.]$, such that $[\tau]$ is a mathematical set that represents $\tau$.

\[
\begin{align*}
\text{[int]} &= \mathbb{Z} \\
\text{[float]} &= \mathbb{R}
\end{align*}
\]

What's a natural way to formulate subtyping here?
A More Helpful Guiding Principle
Sanity-Checking the Principle

Primitive rule: $\text{int} \leq \text{float}$

Primitive rule: $\text{float} \leq \text{int}$

Primitive rule: $\text{int} \leq \text{int} \rightarrow \text{int}$
A **structural** subtyping system includes rules that analyze the *structure* of types, rather than just using graph edges declared by the user explicitly.
Pair Types

Consider types $\tau_1 \times \tau_2$, consisting of (immutable) pairs of a $\tau_1$ and a $\tau_2$.

What is a good subtyping rule for this feature?
Record Types

Consider types like \{ a_1 : \tau_1, \ldots, a_N : \tau_N \}, consisting of, for each \( i \), a field \( a_i \) of type \( \tau_i \).
Function Types

Consider types $\tau_1 \rightarrow \tau_2$. 
Arrays

Consider types $\tau [ ]$. 
Subtyping Variance and Generics/Polymorphism

\[ \text{List}<\tau_1> \leq \text{List}<\tau_2> \]
Algebraic Structure of Subtyping

New universal supertype ("top")

\[ \tau \leq \top \]
\[ \bot \leq \tau \]

With this addition, subtyping forms a complete lattice!
What is a Complete Lattice?

- A set $S$
- A partial order $\subseteq$ over $S$
- A binary least upper bound operator $\cup$
- A binary greater lower bound operator $\cap$

We can use $\leq$ as a partial order over types!
The unique greatest element is $\top$.
The unique least element is $\bot$.
$\cup$ moves up the graph to the lowest common supertype.
$\cap$ moves down the graph to the highest common subtype.

Example algebraic laws we can derive:
$\tau \cup \top = \top$
$\tau \cap \bot = \bot$
$\left(\tau_1 \times \tau_2\right) \cup \left(\tau'_1 \times \tau'_2\right) = \left(\tau_1 \cup \tau'_1\right) \times \left(\tau_2 \cup \tau'_2\right)$
How could we incorporate subtyping?
Review of Formal Type Soundness

Type soundness:
Whenever |- e : τ and e \:\Rightarrow^* \:\text{e}',
either e' is a value or \:\exists e''. e' \:\Rightarrow \:\text{e}''.

Standard lemmas to establish soundness:

Type progress:
Whenever |- e : τ, either e is a value or \:\exists e'. e \:\Rightarrow \:\text{e}'.

Type preservation:
Whenever |- e : τ and e \:\Rightarrow \:\text{e}', |- e' : τ

This notion of soundness is the perfect formal test for the subtyping rules we've been developing!
(And we can use the same proof approach as before.)
Coq demo with L08-Subtyping.v