Subtyping

Adam Chlipala
MIT

6.820
September 30, 2013
Related Reading

Chapter 15 of Pierce, “Subtyping”
Subtyping in Java: Primitive Types

```java
void foo(int n) {
    float f = n;
    // ...and so on.
}
```

`int ≤ float`
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

Q: How do we decide if A is a subtype of B?
A: Graph reachability! (Easy, right?)

We do need to think harder when the graph can be infinite. (E.g., what about generics?)
Subtyping as a Formal Judgment

Reflexivity: \( \tau \leq \tau \)

Transitivity: \( \tau \leq \tau' \quad \tau' \leq \tau'' \Rightarrow \tau \leq \tau'' \)

Primitive rule: \( \text{int} \leq \text{float} \)

Inheritance: \( \text{class } A \text{ extends } B \Rightarrow A \leq B \)

Interfaces: \( \text{class } A \text{ implements } B \Rightarrow A \leq B \)

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 $\tau$.

$[\text{int}] = \mathbb{Z}$
$[\text{float}] = \mathbb{R}$

What's a natural way to formulate subtyping here?

$\tau_1 \leq \tau_2 \iff [\tau_1] \subseteq [\tau_2]$

What about cases like:

```c
struct s1 { int a; int b; }
struct s2 { float b; }
```

Is either of these a subtype of the other?
A More Helpful Guiding Principle

\[ \tau_1 \leq \tau_2 \]

if

Anywhere it is legal to use a \( \tau_2 \),

it is also legal to use a \( \tau_1 \).
Sanity-Checking the Principle

Primitive rule: \( \texttt{int} \leq \texttt{float} \)

✔ Any integer N can be treated as N.0, with no loss of meaning.

Primitive rule: \( \texttt{float} \leq \texttt{int} \)

✗ E.g., “%” operator defined for \texttt{int} but not \texttt{float}.

Primitive rule: \( \texttt{int} \leq \texttt{int} \rightarrow \texttt{int} \)

✗ Can't call an \texttt{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? Ask ourselves: What operations does it support?

1. Pull out a $\tau_1$.
2. Pull out a $\tau_2$.

$$\tau_1 \leq \tau'_1$$
$$\tau_2 \leq \tau'_2$$

$$\overline{\tau_1 \times \tau_2 \leq \tau'_1 \times \tau'_2}$$

Jargon: The pair type constructor is covariant.
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 \( \tau_i \).

What operations must we support?

1. For any \( i \), pull out a \( \tau_i \) from \( a_i \).

**Depth subtyping:**

\[
\forall i. \, \tau_i \leq \tau'_i \implies \{ a_i : \tau \} \leq \{ a_i : \tau'_i \} \quad \text{Same field names, possibly with different types}
\]

**Width subtyping:**

\[
\forall j. \, \exists i. \, a_i = a'_j \wedge \tau_i = \tau'_j \implies \{ a_i : \tau \} \leq \{ a'_j : \tau'_j \} \quad \text{Field names may be different}
\]
Record Type Examples

Depth:  
\[ \forall i. \tau_i \leq \tau'_i \]
\[ \{a_i : \tau_i\} \leq \{a_i : \tau'_i\} \]

Width:  
\[ \forall j. \exists i. a_i = a'_j \land \tau_i = \tau'_j \]
\[ \{a_i : \tau_i\} \leq \{a'_j : \tau'_j\} \]

\{A: int, B: float\} \leq \{A: float, B: float\}  
\[ \leq \]
\[ \leq \]  
Yes!

\{A: float, B: float\} ? \leq \{A: int, B: float\}  
\[ \leq \]
\[ \times \]  
No!

\{A: int, B: float\} \leq \{A: float\}  
\[ \leq \]  
Yes!
Function Types

Consider types $\tau_1 \rightarrow \tau_2$.

What operations must we support?
1. Call with a $\tau_1$ to receive a $\tau_2$ as output.

Optimistic covariant rule:

$$\frac{\tau_1 \leq \tau'_1 \quad \tau_2 \leq \tau'_2}{\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2}$$

Counterexample: $\text{int} \rightarrow \text{int} \leq \text{float} \rightarrow \text{int}$

$$(\lambda x : \text{int}. x \% 2) : \text{int} \rightarrow \text{int}$$

Breaks when we call it with 1.23!
Function Types

Consider types $\tau_1 \rightarrow \tau_2$.

What operations must we support?
1. Call with a $\tau_1$ to receive a $\tau_2$ as output.

Swap order for function domains!

The function arrow is **contravariant** in the domain and **covariant** in the range!

Example: float $\rightarrow$ int $\leq$ int $\rightarrow$ int
Assume $f$: float $\rightarrow$ int
Build $(\lambda x. f(\text{intToFloat}(x))):$ int $\rightarrow$ int
Arrays

Consider types $\tau[\cdot]$.

What operations must we support?
1. *Read* a $\tau$ from some index.
2. *Write* a $\tau$ to some index.

**Covariant** rule:

\[
\frac{\tau_1 \leq \tau_2}{\tau_1[\cdot] \leq \tau_2[\cdot]}
\]

Counterexample:

```java
int[] x = new int[1];
float[] y = x; // Use subtyping here.
y[0] = 1.23;
int z = x[0]; // Not an int!
```
Arrays

Consider types $\tau[\ ]$.

What operations must we support?
1. Read a $\tau$ from some index.
2. Write a $\tau$ to some index.

Contravariant rule: $\frac{\tau_2 \leq \tau_1}{\tau_1[\ ] \leq \tau_2[\ ]}$

Counterexample:

```java
float[] x = new float[1];
int[] y = x; // Use subtyping here.
x[0] = 1.23;
int z = y[0]; // Not an int!
```
Arrays

Consider types \( \tau [ ] \).

What operations must we support?
1. Read a \( \tau \) from some index.
2. Write a \( \tau \) to some index.

**Correct** rule: None at all!
Only reflexivity applies to array types.

In other words, the array type constructor is **invariant**.

Java and many other “practical” languages use the covariant rule for convenience.
Run-time type errors (exceptions) are possible!
Subtyping Variance and Generics/Polymorphism

\[ \text{List}<\tau_1> \leq \text{List}<\tau_2> \]

List and most other “data structures” will be **covariant**.
There are reasonable uses for **contravariant** and **invariant** generics, including mixing these modes across multiple generic parameters.
Languages like OCaml and Scala allow generics to be annotated with variance.

[Haskell doesn't have subtyping and avoids the whole mess.]
Algebraic Structure of Subtyping

New universal \textbf{super}type ("top")

\begin{array}{c}
\text{float} \\
\text{int} \\
\text{nat}
\end{array}

\begin{array}{c}
\text{float} \times \text{float} \\
\text{int} \times \text{float} \\
\text{float} \times \text{int}
\end{array}

\begin{array}{c}
\text{int} \rightarrow \text{int}
\end{array}

\text{New universal \textbf{sub}type ("bottom")}

\begin{array}{c}
\tau \leq \top \\
\bot \leq \tau
\end{array}

With this addition, subtyping forms a \textbf{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:
\[
\tau \cup \top = \top \\
\tau \cap \bot = \bot \\
(\tau_1 \times \tau_2) \cup (\tau'_1 \times \tau'_2) = (\tau_1 \cup \tau'_1) \times (\tau_2 \cup \tau'_2)
\]
Review of Formal Type Systems

\[
\begin{align*}
\Gamma |- e : \tau \\
\end{align*}
\]

Variable types
Expression

\[
\begin{align*}
\Gamma |- e_1 : \tau' \rightarrow \tau \quad \Gamma |- e_2 : \tau' \\
\hline
\Gamma |- e_1 e_2 : \tau \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : \tau' |- e : \tau \\
\hline
\Gamma |- \lambda x. e : \tau' \rightarrow \tau \\
\end{align*}
\]

\[
\begin{align*}
\Gamma |- e : \tau' \quad \tau' \leq \tau \\
\hline
\Gamma |- e : \tau \\
\end{align*}
\]

How could we incorporate subtyping?
Review of Formal Type Soundness

<table>
<thead>
<tr>
<th>Variable types</th>
<th>Its type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td></td>
</tr>
</tbody>
</table>

**Type soundness:**
Whenever \( |- e : \tau \) and \( e \Rightarrow^* e' \), either \( e' \) is a value or \( \exists e''. e' \Rightarrow e'' \).

**Type progress:**
Whenever \( |- e : \tau \), either \( e \) is a value or \( \exists e'. e \Rightarrow e' \).

**Type preservation:**
Whenever \( |- e : \tau \) and \( e \Rightarrow e' \), \( |- e' : \tau \)

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