Subtyping

March 26, 2001
CS 131: Programming Languages

Subtyping: Definition

• A subtyping relation is a preorder \( \preceq \) between types validating the subsumption rule:

\[
\Gamma \vdash e : t_1 \\
\Gamma \vdash e : t_2 \\
\frac{t_1 \preceq t_2}{\Gamma \vdash e : t_2}
\]

• If \( t_1 \preceq t_2 \) then we say that \( t_1 \) is a subtype of \( t_2 \).

NB: A preorder is a relation that is reflexive and transitive (but not necessarily antisymmetric)

Interpretations of Subtyping

• If \( t_1 \preceq t_2 \) then...
  1. The type \( t_1 \) is more precise (less general) description of a value than \( t_2 \).
  2. Every value of type \( t_1 \) also has type \( t_2 \).
  3. There is a standard way to convert values of type \( t_1 \) to values of type \( t_2 \).
  4. In any context where a value of type \( t_2 \) is expected, it is acceptable to provide a value of type \( t_1 \).

Examples

\[
\text{Integer} \preceq \text{Number} \preceq \text{Object}
\]

\[
\text{char} \preceq \text{int} \preceq \text{long} \preceq \text{float} \preceq \text{double}
\]

\[
\text{even} \preceq \text{nat} \quad \text{odd} \preceq \text{nat}
\]
Subtyping is not Inheritance!

- These concepts are conflated in C++, Java
  - Subclasses always generate subtypes
- But, these are really orthogonal concepts
  - Could have subtyping without inheritance
  - Could have inheritance without subtyping

Example Typing Derivation

- Assume
  \[ \text{int} \leq \text{real} \]
- Then

\[
\begin{array}{c}
3 : \text{int} \\
\text{int} \leq \text{real} \\
2.5 : \text{real} \\
\hline
3 : \text{real} \\
\hline
(3, 2.5) : \text{real*real}
\end{array}
\]

Language Design

- Is the choice of subtyping arbitrary?
  - Given the operational semantics, only certain choices for subtyping are sound.
    - Asking for trouble when this is ignored.
  - However, a language need not include all “natural” subtyping relationships.
    - Implementation costs
    - Methodological/simplicity arguments
  - Structural vs. By-Name subtyping

Inclusive Viewpoint

- Suppose we just throw in the subsumption rule into a NQSML-like type system.
  - With no change to operational semantics
  - No run-time data coercions.
- What definitions of \( \leq \) are sound?
- Informal methodology for deciding \( t_1 \leq t_2 \):
  - What can you do with values of type \( t_2 \)?
  - Question: would it be safe to apply these operations to an arbitrary value of type \( t_1 \)?
Pair Types

- Suppose $even \leq nat$.
  - Which of the following are ok?

  1. $even \times string \leq nat \times string$
  2. $nat \times string \leq even \times string$
  3. $even \times even \leq nat \times nat$

Tuple Types

- Suppose $even \leq nat$.
  - Which of the following are ok?

  1. $even \times even \times even \leq nat \times nat \times nat$
  2. $even \times string \times nat \leq even \times string$
  3. $even \times string \leq even \times string \times nat$
  4. $even \times even \times even \leq nat \times nat$

General Rule:

$t_1 \times t_2 \leq t_1' \times t_2'$

General Rule:

$t_1 \times \ldots \times t_n \leq t_1' \times \ldots \times t_n'$
**Function Types**

- Suppose `even` $\leq$ `nat`.
- Which of the following are ok?
  1. `even` $\rightarrow$ `even` $\leq$ `even` $\rightarrow$ `nat`
  2. `even` $\rightarrow$ `nat` $\leq$ `even` $\rightarrow$ `even`
  3. `even` $\rightarrow$ `even` $\leq$ `nat` $\rightarrow$ `even`
  4. `nat` $\rightarrow$ `even` $\leq$ `even` $\rightarrow$ `even`
  5. `even` $\rightarrow$ `even` $\leq$ `nat` $\rightarrow$ `nat`

**Reference Types**

- Suppose `even` $\leq$ `nat`.
  - Which of the following are ok?
    1. `even` `Ref` $\leq$ `nat` `Ref`
    2. `nat` `Ref` $\leq$ `even` `Ref`
Vector and Array Types

- Vector (immutable array)
  - Supports subscript operation
- Array
  - Supports subscript and update operations
- Which are ok?
  1. even vector \(\leq\) nat vector
  2. even array \(\leq\) nat array

Java Arrays

- The Java language is defined so that
  \(\text{Integer[]} \leq \text{Object[]}\)
- We've just argued that this is "unsafe"
- How does Java get around this problem?

Coercive Viewpoint

- \(t_1\) is a subtype of \(t_2\) when...
  - there is a standard way to convert values of type \(t_1\) to values of type \(t_2\).
  - Compiler will automatically insert run-time coercions where required
  - Coercions may involve actual work.
- Canonical example: \(\text{int} \leq \text{float}\)
  - Other coercions? \(\text{float} \rightarrow \text{int} \rightarrow \text{float}\)

Coherence

- Idea:
  - the way the compiler can insert implicit coercions shouldn't change the meaning of a program
  - Frequently an issue when subtyping is combined with overloading
    \[
    (6 / 7) \times 7.0
    \]
  - Even when there are fixed rules for inserting coercions, don't want surprising behavior
    \[
    (1 / 3) + 15
    \]
Information Loss

• Suppose \( \text{Integer} \leq \text{Numeric} \), and we want a function that takes a numeric object and adds it to itself.
• So far, the best we can do is write
  \[
  \text{double} : \text{Numeric} \rightarrow \text{Numeric}
  \]
• But this loses information. If \( n : \text{Integer} \), then
  \[
  \text{double}(n) : \text{Numeric}.
  \]

Can Polymorphism Help?

• If we could say
  \[
  \text{double} : \forall \alpha. \alpha \rightarrow \alpha
  \]
  then
  \[
  \text{double}[\text{Integer}](n) : \text{Integer}
  \]
  But we can't pass an arbitrary object to \( \text{double} \) because the code requires the argument have a method for addition.

Bounded Polymorphism

• Extension:
  – Polymorphic functions that take not an arbitrary type, but any subtype of a given type.
    \[
    \text{double} : \forall \alpha \in \text{Numeric}. \alpha \rightarrow \alpha
    \]
• Then
  \[
  \text{double}[\text{Integer}](n) : \text{Integer}.
  \]