Subtyping: Definition

- A subtyping relation is a preorder \( \leq \) between types used by the subsumption rule:

\[
\begin{align*}
\text{tenv} & \vdash \text{exp} : \text{type}_1 \\
\text{type}_1 & \leq \text{type}_2 \\
\text{tenv} & \vdash \text{exp} : \text{type}_2
\end{align*}
\]

- If \( \text{type}_1 \leq \text{type}_2 \) then we say that \( \text{type}_1 \) is a subtype of \( \text{type}_2 \).

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

Interpretations of Subtyping

If \( \text{type}_1 \leq \text{type}_2 \) then...

1. The type \( \text{type}_1 \) is more precise (less general) description of a value than \( \text{type}_2 \).
2. Either every value of type \( \text{type}_1 \) also has type \( \text{type}_2 \), or, there is a standard way to convert values of type \( \text{type}_1 \) to values of type \( \text{type}_2 \).
3. In any context where a value of type \( \text{type}_2 \) is expected, it is acceptable to provide a value of type \( \text{type}_1 \).

Examples

- Integer \( \leq \) Number \( \leq \) Object
- char \( \leq \) int \( \leq \) long \( \leq \) float \( \leq \) double
- even \( \leq \) nat \quad odd \( \leq \) nat
Subtyping is not Inheritance!

• These concepts are conflated in C++, Java
  - Subclasses always generate subtypes

• But, these are really orthogonal concepts
  - Can have subtyping without inheritance
    • e.g., primitive types in C
  - Can have inheritance without subtyping
    • e.g., C++ private inheritance

Example Typing Derivation

• Assume
  \[ \text{int} \preceq \text{real} \]

• Then

\[
\begin{align*}
3 : \text{int} & \quad \text{int} \preceq \text{real} \\
3 : \text{real} & \quad 2.5 : \text{real} \\
(3, 2.5) : \text{real} \times \text{real}
\end{align*}
\]

Language Design

• Is the choice of subtyping arbitrary?
  - Given the dynamic semantics, only certain choices for subtyping avoid run-time type errors.
    • 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 the type system we saw last class.
  - With no change to operational semantics
  - No run-time data coercions.

• What definitions of \(\preceq\) are sound?
• Informal methodology for deciding \(\tau_1 \preceq \tau_2\):
  - What can you do with values of type \(\tau_1\)?
  - Question: would it be safe to apply these operations to an arbitrary value of type \(\tau_1\)?
**Pair Types**

- Suppose \( \text{even} \subseteq \text{nat} \).
  - Which of the following are ok?

  1. \( \text{even}*\text{string} \subseteq \text{nat}*\text{string} \)
  2. \( \text{nat}*\text{string} \subseteq \text{even}*\text{string} \)
  3. \( \text{even}*\text{even} \subseteq \text{nat}*\text{nat} \)

\( \text{type}_1*\text{type}_2 \subseteq \text{type'}_1*\text{type'}_2 \)

**Tuple Types**

- Suppose \( \text{even} \subseteq \text{nat} \).
  - Which of the following are ok?

  1. \( \text{even}*\text{even}*\text{even} \subseteq \text{nat}*\text{nat}*\text{nat} \)
  2. \( \text{even}*\text{string}*\text{nat} \subseteq \text{even}*\text{string} \)
  3. \( \text{even}*\text{string} \subseteq \text{even}*\text{string}*\text{nat} \)
  4. \( \text{even}*\text{even}*\text{even} \subseteq \text{nat}*\text{nat} \)

**Function Types**

- Suppose \( \text{even} \subseteq \text{nat} \).
  - Which of the following are ok?

  1. \( \text{even} \rightarrow \text{even} \subseteq \text{even} \rightarrow \text{nat} \)
  2. \( \text{even} \rightarrow \text{nat} \subseteq \text{even} \rightarrow \text{even} \)
  3. \( \text{even} \rightarrow \text{even} \subseteq \text{nat} \rightarrow \text{even} \)
  4. \( \text{nat} \rightarrow \text{even} \subseteq \text{even} \rightarrow \text{even} \)
  5. \( \text{even} \rightarrow \text{even} \subseteq \text{nat} \rightarrow \text{nat} \)

\( \text{type}_1 \rightarrow \text{type}_2 \subseteq \text{type'}_1 \rightarrow \text{type'}_2 \)

**Reference Types**

- Suppose \( \text{even} \subseteq \text{nat} \).
  - Which of the following are ok?

  1. \( \text{even ref} \subseteq \text{nat ref} \)
  2. \( \text{nat ref} \subseteq \text{even ref} \)

\( \text{type}_1 \text{ ref} \subseteq \text{type}_2 \text{ ref} \)
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
  \texttt{Integer[]} \(\leq\) \texttt{Object[]}
- We’ve just argued that this is “unsafe”
- How does Java get around this problem?

Coercive Viewpoint

- \textit{type}\(_1\) is a subtype of \textit{type}\(_2\) when...
  - there is a standard way to convert values of \textit{type}\(_1\) to values of \textit{type}\(_2\).
  - Compiler will automatically insert run-time coercions where required
  - Coercions may involve actual work.
- Canonical example: \texttt{int} \(\leq\) \texttt{float}
- Other coercions?
  - \texttt{float}\(\rightarrow\text{int}\) \(\rightarrow\) \texttt{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 \equiv 5.33333 \text{ (???)}