Subtyping

March 26, 2001

CS 131: Programming Languages

Subtyping: Definition

- A subtyping relation is a preorder $\preccurlyeq$ between types validating the *subsumption* rule:

$$\boxed{\frac{\Gamma \vdash \mathtt{e} : \mathtt{t}_1 \qquad \mathtt{t}_1 \preccurlyeq \mathtt{t}_2}{\Gamma \vdash \mathtt{e} : \mathtt{t}_2}}$$

- If $\mathtt{t}_1 \preccurlyeq \mathtt{t}_2$ then we say that $\mathtt{t}_1$ is a *subtype of* $\mathtt{t}_2$.

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

Interpretations of Subtyping

- If $\mathtt{t}_1 \preccurlyeq \mathtt{t}_2$ then...
  1. The type $\mathtt{t}_1$ is more precise (less general) description of a value than $\mathtt{t}_2$.
  2. Every value of type $\mathtt{t}_1$ also has type $\mathtt{t}_2$.
  3. There is a standard way to convert values of type $\mathtt{t}_1$ to values of type $\mathtt{t}_2$.
  4. In any context where a value of type $\mathtt{t}_2$ is expected, it is acceptable to provide a value of type $\mathtt{t}_1$.

Examples

$$\mathtt{Integer} \preccurlyeq \mathtt{Number} \preccurlyeq \mathtt{Object}$$

$$\mathtt{char} \preccurlyeq \mathtt{int} \preccurlyeq \mathtt{long} \preccurlyeq \mathtt{float} \preccurlyeq \mathtt{double}$$

$$\mathtt{even} \preccurlyeq \mathtt{nat} \qquad \mathtt{odd} \preccurlyeq \mathtt{nat}$$

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

$$\texttt{int} \preccurlyeq \texttt{real}$$

- Then

$$\dfrac{\dfrac{\dfrac{}{\texttt{3 : int}} \quad \dfrac{}{\texttt{int} \preccurlyeq \texttt{real}}}{\texttt{3 : real}} \qquad \texttt{2.5 : real}}{\texttt{(3,2.5) : real*real}}$$

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 $\preccurlyeq$ are sound?
- Informal methodology for deciding $t_1 \preccurlyeq t_2$:
  - 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` $\preccurlyeq$ `nat`.
  - Which of the following are ok?

1. even*string $\preccurlyeq$ nat*string
2. nat*string $\preccurlyeq$ even*string
3. even*even $\preccurlyeq$ nat*nat

Pair Types

- General rule:

$$\frac{}{t_1 * t_2 \preccurlyeq t_1' * t_2'}$$

Tuple Types

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

1. even*even*even $\preccurlyeq$ nat*nat*nat
2. even*string*nat $\preccurlyeq$ even*string
3. even*string $\preccurlyeq$ even*string*nat
4. even*even*even $\preccurlyeq$ nat*nat

Tuple Types

- General Rule:

$$\frac{}{t_1 * \ldots * t_{n+m} \preccurlyeq t_1' * \ldots * t_n'}$$

Function Types

- Suppose `even` ≼ `nat`.
- Which of the following are ok?

1. `even -> even ≼ even -> nat`
2. `even -> nat ≼ even -> even`
3. `even -> even ≼ nat -> even`
4. `nat -> even ≼ even -> even`
5. `even -> even ≼ nat -> nat`

Function Types

- General rule:

$$\frac{}{t_1\text{->}t_2 \preccurlyeq t_1'\text{->}t_2'}$$

Reference Types

- Suppose `even` ≼ `nat`.
  - Which of the following are ok?

1. `even Ref ≼ nat Ref`
2. `nat Ref ≼ even Ref`

Reference Types

- General rule:

$$\frac{}{t_1\ \text{Ref} \preccurlyeq t_2\ \text{Ref}}$$

Vector and Array Types

- Vector (immutable array)
  - Supports subscript operation
- Array
  - Supports subscript and update operations
- Which are ok?

1. `even vector` $\preccurlyeq$ `nat vector`
2. `even array` $\preccurlyeq$ `nat array`

Java Arrays

- The Java language is defined so that
  `Integer[]` $\preccurlyeq$ `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: `int` $\preccurlyeq$ `float`
  - Other coercions? `float->int` to `int->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) * 7.0`
  - Even when there are fixed rules for inserting coercions, don't want surprising behavior
    `(1 / 3) + 15`

Information Loss

- Suppose `Integer` $\preccurlyeq$ `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

  `double : Numeric -> Numeric`
- But this loses information. If

  `n : Integer`

  then

  `double(n) : Numeric`.

Can Polymorphism Help?

- If we could say

  `double :` $\forall\alpha.\alpha\text{->}\alpha$

  then

  `double[Integer](n) : Integer`

  But we can't pass an arbitrary object to `double` because the code requires the argument have a method for addition.

Bounded Polymorphism

- Extension:
  - Polymorphic functions that take not an abitrary type, but any subtype of a given type.

    `double :` $\forall\alpha\preccurlyeq$`Numeric`$.\alpha\text{->}\alpha$
- Then

  `double[Integer](n) : Integer.`