Computer Science 131
Programming Languages

November 2, 2000

Subtyping

Preorders

- A *preorder* is a relation that is
  - Reflexive
  - Transitive
  - Needs not be antisymmetric

Subtyping: Definition

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

$$\frac{G \vdash \mathbf{e} : \mathbf{t_1} \qquad \mathbf{t_1} \preccurlyeq \mathbf{t_2}}{G \vdash \mathbf{e} : \mathbf{t_2}}$$

- If $\mathbf{t_1} \preccurlyeq \mathbf{t_2}$ then we say that $\mathbf{t_1}$ is a *subtype of* $\mathbf{t_2}$.

Interpretations of Subtyping

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

Examples

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

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

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

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 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: can $\mathbf{t_1} \preccurlyeq \mathbf{t_2}$
  - What can you do with values of type $\mathbf{t_2}$?
  - Question: would it be safe to apply these operations to an arbitrary value of type $\mathbf{t_1}$?

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

- In general,

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

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

- It follows that,

$$\frac{\mathtt{t}_1 \preccurlyeq \mathtt{t}_1' \quad \ldots \quad \mathtt{t}_n \preccurlyeq \mathtt{t}_n'}{\mathtt{t}_1 * \ldots * \mathtt{t}_{n+m} \preccurlyeq \mathtt{t}_1' * \ldots * \mathtt{t}_n'}$$

Function Types

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

1. **`even -> even`** $\preccurlyeq$ **`even -> nat`**
2. **`even -> nat`** $\preccurlyeq$ **`even -> even`**
3. **`even -> even`** $\preccurlyeq$ **`nat -> even`**
4. **`nat -> even`** $\preccurlyeq$ **`even -> even`**
5. **`even -> even`** $\preccurlyeq$ **`nat -> nat`**

Function Types

- In general,

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

Reference Types

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

1. **`even Ref`** $\preccurlyeq$ **`nat Ref`**
2. **`nat Ref`** $\preccurlyeq$ **`even Ref`**

Reference Types

- In general,

$$\frac{\mathtt{t_1 = t_2}}{\mathtt{t_1\ Ref} \preccurlyeq \mathtt{t_2\ Ref}}$$

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

- $\mathbf{t_1}$ a subtype of $\mathbf{t_2}$ when…
  - there is a standard way to convert values of type $\mathbf{t_1}$ to values of type $\mathbf{t_2}$.
  - Compiler will automatically insert run-time coercions where required
  - Coercions may involve actual work.
- Canonical example: **`int`** $\preccurlyeq$ **`float`**

Coercive Viewpoint

- Suppose we have a coercion function

```
c : int -> float
```

- What other natural coercions can we define?

```
int*int -> float*float
float*float -> int*int
int*float -> int*int*int
(float->int) -> (int->float)
(int->int) -> (float->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 : " a**.**a**->**a

  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 : " a≼**Numeric.** a **->** a

- Then

    double[Integer](n) : Integer.