CMSC 330: Organization of Programming Languages

Subtyping
Subtyping

The Liskov Substitution Principle:

- Let \( P(x) \) be a property provable about objects \( x \) of type \( T \). Then \( P(y) \) should be true for objects \( y \) of type \( S \) where \( S \) is a subtype of \( T \).

In other words

- If \( S \) is a subtype of \( T \), then an \( S \) can be used anywhere a \( T \) is expected.

Commonly used in object-oriented programming

- Subclasses can be used where superclasses expected.
- This is a kind of polymorphism.
What is subtyping?

- Sometimes "every B is an A"
  - Example:
    - Every Circle or Square is a Shape

Subtyping expresses this

- "B is a subtype of A" means: "every object that satisfies the rules for a B also satisfies the rules for an A"

Goal: code written using A's specification operates correctly even if given a B

- Plus: clarify design, share tests, (sometimes) share code
Subtyping

- A type $S$ is a **subtype** of $T$, written $S <: T$, when any term of type $S$ can safely be used in a context where a term of type $T$ is expected.

- $S <: T$ means
  - $S$ is more informative than $T$.
  - the values of type $S$ are a subset of the values of type $T$. 
The Subsumption Rule

This rule tells us that, if $S <: T$, then every element $t$ of $S$ is also an element of $T$.

For example, if we define the subtype relation so that

\[
\text{If } G \vdash \{x:\text{Int}, y:\text{Int}\} <: \{x:\text{Int}\}
\]

then we can use the subsumption rule to derive

\[
\text{If } G \vdash \{x=0, y=1\} <: \{x:\text{Int}\}
\]

which is what we need to make our motivating example typecheck.
Subtyping: A Preorder

• The subtype relation is **formalized as a collection of inference rules** for deriving statements of the form $S <: T$, pronounced “S is a subtype of T” (or “T is a supertype of S”).

• The subtype relation should always be a **preorder**, meaning that it is reflexive and transitive.

  **Reflexivity:** $S <: S$ (S-REFL)

  **Transitivity:** $S <: U$ $U <: T$ $\Rightarrow$ $S <: T$ (S-TRANS)
Subtyping — Records: Width Subtyping

- **Width Subtyping:**

  \[
  \{ \forall i : T_i \; \text{for } i \in 1..n+k \} \quad <: \quad \{ \forall i : T_i \; \text{for } i \in 1..n \} \quad \text{S-RCDWIDTH}
  \]

- **A longer record** constitutes a more demanding—i.e., more informative—specification, and so describes a **smaller set** of values.

- **Examples:**
  - \{x:Int, y:Int\} <: \{x:Int\}
  - \{x:Int, y:Int, z:Bool\} <: \{x:Int\}
Quiz

{x: Int, y: Int} <: {y: Int}

A. True
B. False
Quiz

\{x: \text{Int}, y: \text{Int}\} \leq \{y: \text{Int}\}

A. True  
B. False
Subtyping — Records: Depth Subtyping

Depth Subtyping:

\[
\begin{array}{l}
\text{for each } i \quad S_i <: T_i \\
\{l_i:S_{i \in 1..n}\} <: \{l_i:T_{i \in 1..n}\}
\end{array}
\]

S-RCDDEPTH

- It is safe to allow the types of individual fields to vary, as long as the types of each corresponding field in the two records are in the subtype relation.

- Example:
  - \{x:{a:Int, b:Int}, y:{m:Int}\} <: \{x:{a:Int}, y:{}\}
Quiz

Which is the subtype of

{  x:{a:Int, b:Bool}  }

A. {a:Int,b:Bool}
B. {x:{a:Int}}
C. {x:{a:Int}, y:{b:Bool}}
D. {x:{a:Int, b:Bool,c:Int}, y:{d:Int}}
Which is the subtype of

\{x:\{a:\text{Int}, b:\text{Bool}\}\}\}

A. \{a:\text{Int},b:\text{Bool}\}
B. \{x:\{a:\text{Int}\}\}
C. \{x:\{a:\text{Int}\}, y:\{b:\text{Bool}\}\}
D. \{x:\{a:\text{Int}, b:\text{Bool},c:\text{Int}\}, y:\{d:\text{Int}\}\}
Subtyping Derivations

\[
\begin{align*}
\text{S-RCDWIDTH} & \quad \text{S-RCDWIDTH} \\
\{a:\text{Nat}, b:\text{Nat}\} & \lessdot \{a:\text{Nat}\} & \{m:\text{Nat}\} & \lessdot \{\} \\
\text{S-RCDDEPTH} & \quad \text{S-RCDDEPTH} \\
\{x:\{a:\text{Nat}, b:\text{Nat}\}, y:\{m:\text{Nat}\}\} & \lessdot \{x:\{a:\text{Nat}\}, y:\{}\}
\end{align*}
\]
Subtyping — Records: Permutation Subtyping

- Permutation Subtyping: the order of fields in a record does not make any difference to how we can safely use it

\[
\begin{align*}
\{k_j : S_j \mid j \in 1..n\} \text{ is a permutation of } \{l_i : T_i \mid i \in 1..n\} \\
\{k_j : S_j \mid j \in 1..n\} <: \{l_i : T_i \mid i \in 1..n\} & \quad \text{S-RCDPERM}
\end{align*}
\]

- Example:
  - \{c:Unit, b:Boolean, a:Integer\} <: \{a:Integer, b:Boolean, c:Unit\}
  - \{a:Natural, b:Boolean, c:Unit\} <: \{c:Unit, b:Boolean, a:Natural\}
Quiz

Which rules will we need to build a derivation of the following?

\{x: \text{Int}, y: \text{Int}, z: \text{Int}\} \ll \{y: \text{Int}\}

A. S-RCDDEPTH
B. S-RCDWIDTH
C. S-RCDPERM
D. S-TRANS
Quiz

Which rules will we need to build a derivation of the following?

\{x:\text{Int}, \ y:\text{Int}, \ z:\text{Int}\} \ <: \ \{y:\text{Int}\}

A. S-RCDDEPETH
B. S-RCDWIDTH
C. S-RCDPERM
D. S-TRANS
Subtyping — Functions

- Functions can be passed as arguments to other functions, we must also give a subtyping rule for function types

\[
T_1 <: S_1 \quad S_2 <: T_2 \\
\frac{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}{S-ARROW}
\]

- Notice that the sense of the subtype relation is reversed (contravariant) for the argument types in the left-hand premise, while it runs in the same direction (covariant) for the result types as for the function types themselves.
Subtyping — Functions

Intuition

- Let's say I have a Java function, f, which takes a Cat object and returns an Animal. What are the subtypes of this function? Well, if it takes a Cat then I can certainly replace this function with one that takes an Animal. Likewise, if it returns an Animal then I can certainly replace this function with one that returns a Cat (or Dog). Therefore, I conclude that...

\[(\text{Animal} \rightarrow \text{Cat}) <: (\text{Cat} \rightarrow \text{Animal})\]

\[(\text{Animal} \rightarrow \text{Dog}) <: (\text{Cat} \rightarrow \text{Animal})\]