Part III
Chapter 15: Subtyping

Subsumption
Subtype relation
Properties of subtyping and typing
Subtyping and other features
Intersection and union types
Subtyping
Motivation

With the *usual* typing *rule* for applications

\[
\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \\
\Gamma \vdash t_1 \ t_2 : T_{12} \quad \text{(T-App)}
\]

is the term

\[(\lambda r : \{x : \text{Nat}\}. \ r \ . \ x) \ \{x=0, y=1\}\]

right?

It is *not* well typed
Motivation

With the usual typing rule for applications

\[
\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \; t_2 : T_{12}} \hspace{1cm} \text{(T-APP)}
\]

the term

\[
(\lambda r:\{x:\text{Nat}\}. \; r.\; x) \; \{x=0, y=1\}
\]

is not well typed.

This is silly: what we’re doing is passing the function a better argument than it needs
More generally: some types *are better* than others, in the sense that *a value of one* can *always safely be used* where *a value of the other* is expected.

We can *formalize this intuition* by introducing:

1. a *subtyping relation* between types, written $S <: T$.
2. a rule of *subsumption* stating that, if $S <: T$, then any value of type $S$ can also be regarded as having type $T$, i.e.,

$$
\Gamma \vdash t : S \quad S <: T \quad \implies \quad \Gamma \vdash t : T
$$

*(T-Sub)*

*Principle of safe substitution*
Subtyping

Intuitions: \( S <: T \) means ...

“An element of \( S \) may safely be used wherever an element of \( T \) is expected” (Official)

- \( S \) is “better than” \( T \)
- \( S \) is a subset of \( T \)
- \( S \) is more informative / richer than \( T \)
We will define subtyping between record types so that, for example

\[ \{x: \text{Nat}, y: \text{Nat}\} \preceq \{x: \text{Nat}\} \]

by subsumption,

\[ \vdash \{x = 0, y = 1\} : \{x: \text{Nat}\} \]

and hence

\[ (\lambda r: \{x: \text{Nat}\}. \ r.x) \ \{x=0, y=1\} \]

is well typed.
The Subtype Relation: Records

“Width subtyping”: forgetting fields on the right

\[ \{ l_i : T_{i \in 1..n+k} \} \prec \{ l_i : T_{i \in 1..n} \} \quad (S\text{-RcdWidth}) \]

**Intuition:**

\( \{ x : Nat \} \) is the type of all records with at least a numeric \( x \) field
"Width subtyping" (forgetting fields on the right):

\[ \{ l_i : T_i^{i \in 1..n+k} \} <: \{ l_i : T_i^{i \in 1..n} \} \] (S-RcdWidth)

**Intuition:**

\( \{ x : Nat \} \) is the type of *all records* with *at least* a numeric \( x \) field.

**Note that** the record type with *more* fields is a *subtype* of the record type with *fewer* fields.

**Reason:** the type with more fields places *stronger constraints* on values, so it describes *fewer values*.
The Subtype Relation: Records

“Depth subtyping” within fields:

\[
\text{for each } i \quad S_i <: T_i \\
\{l_i:S_i \mid i \in 1..n\} <: \{l_i:T_i \mid i \in 1..n\} \quad \text{(S-RcdDepth)}
\]

The types of *individual fields* may change, *as long as* the type of each corresponding field in the two records are in the *subtype relation*
Examples

\{a: \text{Nat}, b: \text{Nat}\} \ll \{a: \text{Nat}\}

\{m: \text{Nat}\} \ll \{\}
Examples

We can also use S-RcdDepth to refine the type of just a single record field (instead of refining every field), by using S-REFL to obtain trivial subtyping derivations for other fields.

\[
\begin{align*}
\{a: \text{Nat}, b: \text{Nat}\} & \leq \{a: \text{Nat}\} & \text{S-RCDWIDTH} & \{m: \text{Nat}\} \leq \{m: \text{Nat}\} & \text{S-REFL} \\
\{x: \{a: \text{Nat}, b: \text{Nat}\}, y: \{m: \text{Nat}\}\} & \leq \{x: \{a: \text{Nat}\}, y: \{m: \text{Nat}\}\} & \text{S-RcdDepth}
\end{align*}
\]
The order of fields in a record doesn’t make any difference to how we can safely use it, since the only thing that we can do with records (projecting their fields) is insensitive to the order of fields.

S-RcdPerm tells us that

\{c:\text{Top}, b: \text{Bool}, a: \text{Nat}\} \prec \{a: \text{Nat}, b: \text{Bool}, c: \text{Top}\}

and

\{a: \text{Nat}, b: \text{Bool}, c: \text{Top}\} \prec \{c: \text{Top}, b: \text{Bool}, a: \text{Nat}\}
The Subtype Relation: Records

Permutation of fields:

\[ \{k_j:S_j \mid j \in 1\ldots n\} \text{ is a permutation of } \{l_i:T_i \mid i \in 1\ldots n\} \]

\[ \{k_j:S_j \mid j \in 1\ldots n\} \preceq \{l_i:T_i \mid i \in 1\ldots n\} \] (S-RcdPerm)

Using **S-RcdPerm** together with **S-RcdWidth** & **S-Trans** allows us to *drop arbitrary fields* within records
Variations

Real languages often choose *not to adopt all of these record subtyping rules*. For example, in Java,

- A subclass may not change the argument or result types of a method of its superclass (i.e., *no depth subtyping*).
- Each class has just one superclass ("*single inheritance*" of classes).

  *each class member (field or method) can be assigned a single index, adding new indices “on the right” as more members are added in subclasses (i.e., *no permutation for classes*).

- A class may implement multiple interfaces ("*multiple inheritance*" of interfaces).

  i.e., *permutation* is allowed for interfaces.
A high-order language, functions can be passed as arguments to other functions.

\[
\begin{align*}
&T_1 <: S_1 \quad S_2 <: T_2 \\
&S_1 \rightarrow S_2 <: T_1 \rightarrow T_2
\end{align*}
\]

\[(S\text{-ARROW})\]
The Subtype Relation: Arrow types

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

Note the order of $T_1$ and $S_1$ in the first premise.
The subtype relation is

- **contravariant** in the left-hand sides of arrows
- **covariant** in the right-hand sides of arrows
The Subtype Relation: Arrow types

\[
\begin{align*}
T_1 & \leq S_1 & S_2 & \leq T_2 \\
S_1 & \rightarrow S_2 & \leq T_1 & \rightarrow T_2
\end{align*}
\]

\((S\text{-ARROW})\)

**Intuition:** if we have a function \(f\) of type \(S_1 \rightarrow S_2\),

1. \(f\) accepts elements of type \(S_1\); clearly, \(f\) will also accept elements of any subtype \(T_1\) of \(S_1\)

2. the type of \(f\) also tells us that it returns elements of type \(S_2\); then these results can be viewed as belonging to any supertype \(T_2\) of \(S_2\)

i.e.,

any function \(f\) of *type* \(S_1 \rightarrow S_2\) can also be viewed as having *type* \(T_1 \rightarrow T_2\)
The Subtype Relation: Top

It is *convenient* to have a type that is a *supertype of every type*

We introduce a new *type constant* Top, plus *a rule* that makes Top a *maximum element* of the subtype relation

i.e.,

\[
S \subseteq \text{Top} \quad \text{(S-TOP)}
\]
It is *convenient* to have a type that is a *supertype of every type*

We introduce a new type constant Top, plus a *rule* that makes Top a *maximum element* of the subtype relation

\[
S <: \text{Top} \quad (S \text{-Top})
\]

Cf. *Object* in Java.
A subtyping is a binary relation between types that is closed under the following rules: 

\[ S <: S \quad \text{(S-REFL)} \]

\[
\begin{align*}
S &: U & U &: T \\
\hline
S &: T
\end{align*}
\quad \text{(S-TRANS)}
\]

\[ S <: \text{Top} \quad \text{(S-TOP)} \]
Subtype Relation

\[ S <: S \]  \hspace{2cm} (S-REFL)

\[ S <: U \quad U <: T \quad \frac{}{S <: T} \]  \hspace{2cm} (S-TRANS)

\[ \{l_i:T_i \mid i \in 1..n+k\} <: \{l_i:T_i \mid i \in 1..n\} \]  \hspace{2cm} (S-RCDWIDTH)

\[ \text{for each } i \quad S_i <: T_i \quad \frac{}{\{l_i:S_i \mid i \in 1..n\} <: \{l_i:T_i \mid i \in 1..n\}} \]  \hspace{2cm} (S-RCDDEPTH)

\[ \{k_j:S_j \mid j \in 1..n\} \text{ is a permutation of } \{l_i:T_i \mid i \in 1..n\} \quad \frac{}{\{k_j:S_j \mid j \in 1..n\} <: \{l_i:T_i \mid i \in 1..n\}} \]  \hspace{2cm} (S-RCDPERM)

\[ T_1 <: S_1 \quad S_2 <: T_2 \quad \frac{}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \]  \hspace{2cm} (S-ARROW)

\[ S <: \text{Top} \]  \hspace{2cm} (S-TOP)
HW for Chap15

• 15.2.2
• 15.3.2
• 15.3.6