Types and Programming Languages

Lecture 7. Subtyping

Xiaojuan Cai

cxj@sjtu.edu.cn

BASICS Lab, Shanghai Jiao Tong University

Fall, 2016
Motivation

Then

$$\begin{align*}
\text{T-App} & \quad \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1 \\
\therefore & \quad \Gamma \vdash t_1 t_2 : T_2
\end{align*}$$

$$(\lambda r : \{x : \text{Nat}\}. \ r.x) \ {\{x = 0, y = 1\}} \not\rightarrow$$
Motivation

\[ \text{T-App} \quad \frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2} \]

Then

\[ (\lambda r : \{x : \text{Nat}\}. r.x) \{x = 0, y = 1\} \not\rightarrow \]

How to make this well-behaved term reducible?
Outline

Subtype relation

Subtyping and other features

Metatheory of subtyping
Subtyping

- Subtyping is found in object-oriented languages and is an essential feature of the object-oriented style.
Subtyping is found in object-oriented languages and is an essential feature of the object-oriented style.

Subtype $S <: T$: type $S$ are more informative than $T$. 
Subtyping

- Subtyping is found in object-oriented languages and is an essential feature of the object-oriented style.
- Subtype $S <: T$ : type $S$ are more informative than $T$.
- Principle of safe substitution. If $S <: T$, then any term of type $S$ can safely be used in a context where a term of type $T$ is expected.
Subtyping is found in object-oriented languages and is an essential feature of the object-oriented style.

Subtype $S <: T$: type $S$ are more informative than $T$.

Principle of safe substitution. If $S <: T$, then any term of type $S$ can safely be used in a context where a term of type $T$ is expected.

We can simply consider $ <: $ as $ \subseteq $. 
Subtype rules

\[
T\text{-}SUB \quad \frac{\Gamma \vdash t : S \quad S \ll T}{\Gamma \vdash t : T}
\]
Subtype rules

\[
T\text{-}\text{SUB} \quad \frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}
\]

**Quiz.** Write rules to make \(<:\) reflexive and transitive.
Subtype rules

\[
\text{T-SUB} \quad \frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}
\]

**Quiz.** Write rules to make \(<::\) reflexive and transitive.

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

\[
\text{S-TRAN} \quad \frac{S <:: T \quad T <:: U}{S <:: U}
\]
Subtype rules for records

\{l_i : T_i \text{ for } i \in 1..n\}

Quiz. What rules should we add to generate the following subtype relations?

- \{x : \text{Nat}, y : \text{nat}\} <: \{x : \text{Nat}\}
- \{x : \{a : \text{Nat}, b : \text{nat}\}, y : \text{Nat}\} <: \{x : \{a : \text{Nat}\}, y : \text{Nat}\}
- \{x : \text{Nat}, y : \text{nat}\} <: \{y : \text{Nat}\}
Subtype rules for records

\{l_i : T_i \}_{i=1..n}\}

Quiz. What rules should we add to generate the following subtype relations?

- \{x : \text{Nat}, y : \text{nat}\} \ll \{x : \text{Nat}\}
- \{x : \{a : \text{Nat}, b : \text{nat}\}, y : \text{Nat}\} \ll \{x : \{a : \text{Nat}\}, y : \text{Nat}\}
- \{x : \text{Nat}, y : \text{nat}\} \ll \{y : \text{Nat}\}

\text{S-RcdWidth:} \quad \{l_i : T_i \}_{i=1..n+k}\}\ll \{l_i : T_i \}_{i=1..n}\}

\text{S-RcdDepth:} \quad \text{foreach } i \in 1..n \quad S_i \ll T_i
\{l_i : S_i \}_{i=1..n}\}\ll \{l_i : T_i \}_{i=1..n}\}

\text{S-RcdPerm:} \quad \{k_j : S_j \}_{j=1..n}\}\ll \{k_j : S_j \}_{j=1..n}\}
\{l_i : T_i \}_{i=1..n}\}
Quiz. Come back to the motivation example:

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

Please draw its type derivation tree.
Arrow type

Give the subtyping rule for arrow type $T_1 \rightarrow T_2$?
Arrow type

Give the subtyping rule for arrow type $T_1 \rightarrow T_2$?

Examples.

```plaintext
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
    p : {x:real,y:real}) =
  let val p2 : {x:real,y:real} = f p
  val dx : real = p2.x - p.x
  val dy : real = p2.y - p.y
  in Math.sqrt(dx*dx + dy*dy)
  end

fun flip p = {x = ~p.x, y=~p.y}
val d = distMoved(flip, {x=3.0, y=4.0})
```
Arrow type

Give the subtyping rule for arrow type $T_1 \rightarrow T_2$.

Examples.

```ocaml
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
    let val p2 : {x:real,y:real} = f p
    val dx : real = p2.x - p.x
    val dy : real = p2.y - p.y
    in Math.sqrt(dx*dx + dy*dy)
    end

fun flipGreen p = {x = ~p.x, y=~p.y, color="green"}
val d = distMoved(flipGreen, {x=3.0, y=4.0})
```
Arrow type

Give the subtyping rule for arrow type \( T_1 \rightarrow T_2 \).

Examples.

```haskell
fun distMoved (f : \{x:real,y:real\}=>\{x:real,y:real\},
               p : \{x:real,y:real\}) =
    let val p2 : \{x:real,y:real\} = f p
        dx : real = p2.x - p.x
        dy : real = p2.y - p.y
    in Math.sqrt(dx*dx + dy*dy)
end

fun flipIfGreen p =
    if p.color = "green"
        then \{x = ~p.x, y=~p.y\}
        else \{x = p.x, y=p.y\}
val d = distMoved(flipIfGreen, \{x=3.0, y=4.0\})
```
Arrow type

Give the subtyping rule for arrow type $T_1 \rightarrow T_2$.

Examples.

```javascript
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
  let val p2 : {x:real,y:real} = f p
      dx : real = p2.x - p.x
      dy : real = p2.y - p.y
  in Math.sqrt(dx*dx + dy*dy)
end

fun flipX_Y0 p = {x = ~p.x, y=0.0}
val d = distMoved(flipX_Y0, {x=3.0, y=4.0})
```
Arrow type

Give the subtyping rule for arrow type $T_1 \rightarrow T_2$.

Examples.

```haskell
fun distMoved (f : {x:real,y:real}->{x:real,y:real}, p : {x:real,y:real}) =
    let val p2 : {x:real,y:real} = f p
    val dx : real = p2.x - p.x
    val dy : real = p2.y - p.y
    in Math.sqrt(dx*dx + dy*dy)
end

fun flipX_Y0 p = {x = ~p.x, y=0.0}
val d = distMoved(flipX_Y0, {x=3.0, y=4.0})
```

$$S\text{:\textbf{-ARROW}} \frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$
Top and Bot

A final rule: Any type has a supertype top.

\[ S \subseteq \text{top} \]
**Top and Bot**

A final rule: Any type has a supertype top.

\[ S <: \text{top} \]

The maximal type \( \text{Top} \top \) is included in most presentations:

- It corresponds to the type Object.
- It is convenient technical device for combining subtypes and parametric polymorphism.
Top and Bot

A final rule: Any type has a supertype top.

\[ S <: \top \]

The maximal type \( \text{Top} \top \) is included in most presentations:
- It corresponds to the type \( \text{Object} \).
- It is convenient technical device for combining subtypes and parametric polymorphism.

What about the minimal type \( \text{Bot} \bot \)?

\[ \text{S-Bot} \bot <: \text{T} \]

- \( \text{Bot} \) is empty.
- \( \text{Bot} \) is useful: We can give \( \text{error} : \bot \) and make this term well-typed:
  \[ \lambda x : \text{T}. \text{if} \ x \ \text{then} \ t \ \text{else} \ \text{error} \]
Properties of subtyping

**Theorem 15.3.5 [Preservation]**: If \( \Gamma \vdash t : T \) and \( t \rightarrow t' \), then \( \Gamma \vdash t' : T \).

**Theorem 15.3.7 [Progress]**: If \( t \) is a closed, well-typed term, then either \( t \) is a value or else there is some \( t' \) with \( t \rightarrow t' \).
Properties of subtyping

**Theorem 15.3.5 [Preservation]**: If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

**Theorem 15.3.7 [Progress]**: If $t$ is a closed, well-typed term, then either $t$ is a value or else there is some $t$ with $t \rightarrow t$.

**Quiz.** Try to predict where difficulties might arise to prove type safety of subtyping.
Lemma 15.3.2 [Inversion of the subtype relation]:

- If $S <: T_1 \to T_2$, then $S$ has the form $S_1 \to S_2$ and $T_1 <: S_1$ and $S_2 <: T_2$;
- If $S <: \{l_i : T_i^{i\in1..n}\}$, then $S$ has the form $\{k_j : S_j^{j\in1..m}\}$ and $\{l_i^{i\in1..n}\} \subseteq \{k_j^{j\in1..m}\}$ and for every common label $l_i = k_j$ $S_j <: T_i$.
Lemma 15.3.2 [Inversion of the subtype relation]:

- If $S <: T_1 \rightarrow T_2$, then $S$ has the form $S_1 \rightarrow S_2$ and $T_1 <: S_1$ and $S_2 <: T_2$;
- If $S <: \{ l_i : T_i \}^{i \in 1..n}$, then $S$ has the form $\{ k_j : S_j \}^{j \in 1..m}$ and $\{ l_i \}^{i \in 1..n} \subseteq \{ k_j \}^{j \in 1..m}$ and for every common label $l_i = k_j$ $S_j <: T_i$.

Lemma 15.3.3

- If $\Gamma \vdash \lambda x : S_1. s_2 : T_1 \rightarrow T_2$, then $T_1 <: S_1$ and $\Gamma, x : S_1 \vdash s_2 : T_2$.
- If $\Gamma \vdash \{ k_a = s_a \}^{a \in 1..m} : \{ l_i : T_i \}^{i \in 1..n}$, then $\{ l_i \}^{i \in 1..n} \subseteq \{ k_a \}^{a \in 1..m}$ and $\Gamma \vdash s_a : T_i$ for each common label $k_a = l_i$. 
Lemma 15.3.2 [Inversion of the subtype relation]:

- If \( S <: T_1 \rightarrow T_2 \), then \( S \) has the form \( S_1 \rightarrow S_2 \) and \( T_1 <: S_1 \) and \( S_2 <: T_2 \);
- If \( S <: \{ l_i : T_i^{i \in 1..n} \} \), then \( S \) has the form \( \{ k_j : S_j^{j \in 1..m} \} \) and \( \{ l_i^{i \in 1..n} \} \subseteq \{ k_j^{j \in 1..m} \} \) and for every common label \( l_i = k_j \) \( S_j <: T_i \).

Lemma 15.3.3

- If \( \Gamma \vdash \lambda x : S_1. s_2 : T_1 \rightarrow T_2 \), then \( T_1 <: S_1 \) and \( \Gamma, x : S_1 \vdash s_2 : T_2. \)
- If \( \Gamma \vdash \{ k_a = s_a^{a \in 1..m} \} : \{ l_i : T_i^{i \in 1..n} \} \), then \( \{ l_i^{i \in 1..n} \} \subseteq \{ k_a^{a \in 1..m} \} \) and \( \Gamma \vdash s_a : T_i \) for each common label \( k_a = l_i \).

Lemma 15.3.4 [Substitution] If \( \Gamma, x : S \vdash t : T \) and \( \Gamma \vdash s : S \), then \( \Gamma \vdash [x \mapsto s]t : T \).
Outline

Subtype relation

Subtyping and other features

Metatheory of subtyping
Ascription and cast

Review

- The ascription operator \( t \) as \( T \) allows the programmer to record some subterm of a complex expression has some particular type.

\[
\text{T-ASCRIBE} \quad \frac{\Gamma \vdash t : T}{\Gamma \vdash t \text{ as } T : T}
\]

- Up-cast: a term is ascribed a supertype of the type.
- Down-cast: a term is ascribed a subtype of the type.

A surprising rule for down-cast

\[
\text{T-DOWNCAST} \quad \frac{\Gamma \vdash t : S}{\Gamma \vdash t \text{ as } T : T}
\]
Ascription and cast

Review

- The ascription operator \( t \text{ as } T \) allows the programmer to record some subterm of a complex expression has some particular type.

\[
\Gamma \vdash t : T \\
\Gamma \vdash t \text{ as } T : T
\]

- **Up-cast**: a term is ascribed a supertype of the type.

- **Down-cast**: a term is ascribed a subtype of the type.
Ascription and cast

Review

- The ascription operator $t \ as \ T$ allows the programmer to record some subterm of a complex expression has some particular type.

\[
\text{T-ASCRIBE} \quad \frac{\Gamma \vdash t : T}{\Gamma \vdash t \ as \ T : T}
\]

- *Up-cast*: a term is ascribed a supertype of the type.
- *Down-cast*: a term is ascribed a subtype of the type.

A surprising rule for down-cast

\[
\text{T-DOWNCAST} \quad \frac{\Gamma \vdash t : S}{\Gamma \vdash t \ as \ T : T}
\]
Downcast

\[
\frac{\Gamma \vdash t : S}{\Gamma \vdash t \text{ as } T : T}
\]

- \( f = \lambda(x : \text{Top}).((x \text{ as } \{a : \text{Nat}\}).a) \)

"I know that \( f \) will always be applied to record arguments with numeric \( a \) fields; I want you to trust me on this one."

Our motto is "trust, but verify."

We need dynamic type test:

\[ E\text{-Downcast} \vdash \]

Quiz. With \( T\text{-Downcast}, E\text{-Downcast} \), which property will be lost, preservation or progress? Progress will be lost.
Downcast

\[
\begin{align*}
\text{T-Downcast} & \quad \Gamma \vdash t : S \\
\hline
\Gamma \vdash t \text{ as } T : T
\end{align*}
\]

- \( f = \lambda(x : \text{Top}).((x \text{ as } \{a : \text{Nat}\}).a) \)

- "I know that \( f \) will always be applied to record arguments with numeric \( a \) fields; I want you to trust me on this one."

Our motto is "trust, but verify". We need dynamic type test: E-Downcast
Downcast

\[
\frac{\Gamma \vdash t : S}{\Gamma \vdash t \text{ as } T : T}
\]

- \( f = \lambda (x : \text{Top}).((x \text{ as } \{a : \text{Nat}\}).a) \)
- "I know that \( f \) will always be applied to record arguments with numeric \( a \) fields; I want you to trust me on this one."
- Our motto is "trust, but verify".
- We need dynamic type test:

\[
\frac{\vdash v_1 : T}{v_1 \text{ as } T \rightarrow v_1}
\]
Downcast

\[
\text{T-Downcast} \quad \frac{\Gamma \vdash t : S}{\Gamma \vdash t \text{ as } T : T}
\]

- \( f = \lambda(x : \text{Top}).((x \text{ as } \{a : \text{Nat}\}).a) \)
- "I know that \( f \) will always be applied to record arguments with numeric \( a \) fields; I want you to trust me on this one."
- Our motto is "trust, but verify".
- We need dynamic type test:

\[
\text{E-Downcast} \quad \frac{}{\vdash v_1 : T}
\quad \frac{}{v_1 \text{ as } T \rightarrow v_1}
\]

Quiz. With T-Downcast, E-Downcast, which property will lost, preservation or progress?
Downcast

\[
\text{T-Downcast} \quad \frac{\Gamma \vdash t : S}{\Gamma \vdash t \text{ as } T : T}
\]

- \( f = \lambda(x : \text{Top}).((x \text{ as } \{a : \text{Nat}\}).a) \)
- "I know that \( f \) will always be applied to record arguments with numeric a fields; I want you to trust me on this one."
- Our motto is "trust, but verify".
- We need dynamic type test:

\[
\text{E-Downcast} \quad \frac{\vdash \nu_1 : T}{\nu_1 \text{ as } T \rightarrow \nu_1}
\]

Quiz. With T-Downcast, E-Downcast, which property will lost, preservation or progress? Progress will lost.
Subtyping for variants

**Quiz.** Try to give a subtyping rule like $S$-$\text{RCDWIDTH}$ for variant type.
Quiz. Try to give a subtyping rule like $S$-\textsc{Rcd\textwidth} for variant type.

\begin{align*}
S\text{-}\textsc{Var\textwidth} & \quad \frac{\langle l_i : T_i \mid i \in 1..n \rangle <: \langle l_i : T_i \mid i \in 1..n+k \rangle}{\langle l_i : \langle T_i \mid i \in 1..n+k \rangle \rangle <: \langle l_i : \langle T_i \mid i \in 1..n \rangle \rangle} \\
S\text{-}\textsc{Var\textwidth} \quad \text{foreach } i \in 1..n & \quad S_i <: T_i \\
\langle l_i : S_i \mid i \in 1..n \rangle <: \langle l_i : \langle T_i \mid i \in 1..n \rangle \rangle & \\
S\text{-}\textsc{Var\textwidth} \quad \langle k_j : S_j \mid j \in 1..n \rangle & \quad \text{is a permutation of } \langle l_i : T_i \mid i \in 1..n \rangle \\
\langle k_j : S_j \mid j \in 1..n \rangle <: \langle l_i : \langle T_i \mid i \in 1..n \rangle \rangle & \\
\end{align*}
Subtyping for variants

**Quiz.** Try to give a subtyping rule like \( S-RCD\text{WIDTH} \) for variant type.

\[
\begin{align*}
S-\text{VAR}\text{WIDTH} & : \langle l_i : T_i \mid i \in 1..n \rangle \ll : \langle l_i : T_i \mid i \in 1..n+k \rangle \\
S-\text{VAR}\text{DEPTH} & : \text{foreach } i \in 1..n \; S_i \ll : T_i \\
& : \langle l_i : S_i \mid i \in 1..n \rangle \ll : \langle l_i : T_i \mid i \in 1..n \rangle \\
S-\text{VAR}\text{PERM} & : \langle k_j : S_j \mid j \in 1..n \rangle \text{ is a permutation of } \langle l_i : T_i \mid i \in 1..n \rangle \\
& : \langle k_j : S_j \mid j \in 1..n \rangle \ll : \langle l_i : T_i \mid i \in 1..n \rangle \\
\end{align*}
\]

**Lists**

\[
\begin{align*}
S-\text{LIST} & : S_1 \ll : S_2 \\
& : \text{list } S_1 \ll : \text{list } S_2
\end{align*}
\]
Quiz.

\[
\begin{align*}
S_{\text{REF}} \quad ? \quad \frac{\text{Ref } S_1 \llt \text{Ref } T_1}{\text{Ref } S_1 \llt \text{Ref } T_1}
\end{align*}
\]
Quiz.

\[
\text{S-Ref} \quad ? \quad \text{Ref } S_1 <: \text{Ref } T_1
\]

Not all type constructors are **covariant** or **contravariant**.

\[
\text{S-Ref} \quad S_1 <: T_1 \quad T_1 <: S_1 \quad \text{Ref } S_1 <: \text{Ref } T_1
\]

Array is similar to Ref type. So we have

\[
\text{S-Array} \quad S_1 <: T_1 \quad T_1 <: S_1 \quad \text{Array } S_1 <: \text{Array } T_1
\]
Quiz.

\[
\text{S-Ref} \quad ? \quad \text{Ref } S_1 \ll : \text{Ref } T_1
\]

Not all type constructors are \textit{covariant} or \textit{contravariant}.

\[
\text{S-Ref} \quad S_1 \ll : T_1 \quad T_1 \ll : S_1 \quad \text{Ref } S_1 \ll : \text{Ref } T_1
\]

Array is similar to \texttt{Ref} type. So we have

\[
\text{S-Array} \quad S_1 \ll : T_1 \quad T_1 \ll : S_1 \quad \text{Array } S_1 \ll : \text{Array } T_1
\]

Java actually permits covariant subtyping of arrays

\[
\text{S-Array} \quad S_1 \ll : T_1 \quad \text{Array } S_1 \ll : \text{Array } T_1
\]
Refined references

More refined reference type [by Reynolds] is by introducing two new type constructors, Source and Sink.

- Source T is for reading values of type T.
- Sink T is for writing values of type T.

\[
\begin{align*}
\Gamma, \Sigma \vdash t_1 : \text{Source } T_1 & \quad \Gamma, \Sigma \vdash !t_1 : T_1 \\
\Gamma, \Sigma \vdash t_1 : \text{Sink } T_1, \quad \Gamma, \Sigma \vdash t_2 : T_1 & \quad \Gamma, \Sigma \vdash t_1 := t_2 : \text{unit} \\
S_1 <: T_1 & \quad T_1 <: S_1 \\
\text{Source } S_1 <: \text{Source } T_1 & \quad \text{Sink } S_1 <: \text{Sink } T_1 \\
\text{Ref } T_1 <: \text{Source } T_1 & \quad \text{Ref } T_1 <: \text{Sink } T_1
\end{align*}
\]

In Pict, the input channel is Source type and the output channel is Sink type.
Outline

Subtype relation

Subtyping and other features

Metatheory of subtyping
Towards implementation

- Subtype relation is not immediately suitable for implementation. It is not syntax directed.

Coming soon: Replacing the subtyping and typing relation by two new relations: algorithmic subtyping and algorithmic typing, whose inference rules are syntax directed.
Towards implementation

- Subtype relation is not immediately suitable for implementation. It is not syntax directed.
- Subtype cannot just be "read from bottom to top" to yield a typechecker:
  - $T\text{-Abs}$ applies only to abstractions, $T\text{-Var}$ applies only to variables, etc.
Towards implementation

- Subtype relation is not immediately suitable for implementation. It is not syntax directed.
- Subtype cannot just be "read from bottom to top" to yield a typechecker:
  - T-Abs applies only to abstractions, T-Var applies only to variables, etc. However, T-Sub can be applied to any term \( t \).

\[
\begin{align*}
\text{T-Sub: } & \quad \Gamma \vdash t : S \quad S <: T \\
\quad \Rightarrow & \quad \Gamma \vdash t : T
\end{align*}
\]
Towards implementation

- Subtype relation is not immediately suitable for implementation. It is not syntax directed.

- Subtype cannot just be "read from bottom to top" to yield a typechecker:
  - $T$-Abs applies only to abstractions, $T$-Var applies only to variables, etc. However, $T$-Sub can be applied to any term $t$.

  \[
  \frac{
    \text{T-SUB} \quad \Gamma \vdash t : S \quad S \ll T
  }{
    \Gamma \vdash t : T
  }
  \]

- $S$-Trans requires to guess a $T$.

  \[
  \frac{
    \text{S-TRANS} \quad S \ll T \quad T \ll U
  }{
    S \ll U
  }
  \]

Coming soon: Replacing the subtyping and typing relation by two new relations: algorithmic subtyping and algorithmic typing, whose inference rules are syntax directed.
Towards implementation

- Subtype relation is not immediately suitable for implementation. It is not syntax directed.
- Subtype cannot just be ”read from bottom to top” to yield a typechecker:
  - T-ABS applies only to abstractions, T-VAR applies only to variables, etc. However, T-SUB can be applied to any term t.

\[
\begin{align*}
T\text{-SUB} & \quad \Gamma \vdash t : S \quad S \ll T \\
\Gamma \vdash t : T
\end{align*}
\]

- S-TRANS requires to guess a T.

\[
\begin{align*}
S\text{-TRANS} & \quad S \ll T \quad T \ll U \\
S \ll U
\end{align*}
\]

- Coming soon: Replacing the subtyping and typing relation by two new relations: algorithmic subtyping and algorithmic typing, whose inference rules are syntax directed.
Subtype relation

The first calculus we study is simply typed $\lambda$-calculus with subtyping and records. (No booleans, nats).

$\textbf{S-Ref}l$ \quad $S <: S$

$\textbf{S-Trans} \quad S <: T$ \quad $T <: U$ \quad $S <: U$

$\textbf{S-Top} \quad S <: \text{Top}$

$\textbf{S-Arrow} \quad T_1 <: S_1$ \quad $S_2 <: T_2$

$\quad S_1 \to S_2 <: T_1 \to T_2$

$\textbf{S-Rcd} \quad \{l_i^{1..n}\} \subseteq \{k_j^{1..m}\}$ and $k_j = l_i$ implies $S_j <: T_i$

$\quad \{k_j : S_j^{1..m}\} <: \{l_i : T_i^{1..n}\}$

We compact $\textbf{S-RcdWidth}$, $\textbf{S-RcdDepth}$, $\textbf{S-RcdPerm}$ into one rule $\textbf{S-Rcd}$. 

Algorithmic subtyping

**Lemma 16.1.2.**

- $S <: S$ can be derived for every type $S$ without using $S$-$\text{RefL}$.
- If $S <: T$ is derivable, then it can be derived without using $S$-$\text{Trans}$. 
Algorithmic subtyping

Lemma 16.1.2.

- $S <: S$ can be derived for every type $S$ without using $S$-Refl.
- If $S <: T$ is derivable, then it can be derived without using $S$-Trans.

Algorithmic subtyping

\[
\begin{align*}
SA-\text{TOP} & \quad \models S <: \text{Top} \\
SA-\text{ARROW} & \quad \models T_1 <: S_1 \quad \models S_2 <: T_2 \\
& \quad \models S_1 \rightarrow S_2 <: T_1 \rightarrow T_2 \\
SA-\text{RCD} & \quad \{ l_i \in 1..n \} \subseteq \{ k_j \in 1..m \} \text{ and } k_j = l_i \text{ implies } \models S_j <: T_i \\
& \quad \models \{ k_j : S_j \in 1..m \} <: \{ l_i : T_i \in 1..n \}
\end{align*}
\]
Correctness of SA-rules

Proposition 16.1.5 [Soundness and Completeness].
S <: T iff \( \models S <: T \)
Correctness of SA-rules

**Proposition 16.1.5 [Soundness and Completeness].**

\[ S <: T \text{ iff } \models S <: T \]

An algorithm for subtyping:

\[
\text{subtype}(S, T) = \begin{cases} 
\text{true} & \text{if } T = \text{top} \\
\text{false} & \text{otherwise} \\
\end{cases}
\]

else if \( S = S_1 \rightarrow S_2 \) and \( T = T_1 \rightarrow T_2 \)

then \( \text{subtype}(T_1, S_1) \land \text{subtype}(S_2, T_2) \)

else if \( S = \{k_j : S_j \in 1..m\} \) and \( T = \{l_i : T_i \in 1..n\} \)

then \( \{l_i^{1..n}\} \subseteq \{k_j^{1..m}\} \land \text{for all } i \text{ there is some } j \in 1..m \text{ with } k_j = l_i \land \text{subtype}(S_j, T_i) \)
Correctness of SA-rules

**Proposition 16.1.5 [Soundness and Completeness].**
\( S <: T \) iff \( \models S <: T \)

An algorithm for subtyping:

\[
\text{subtype}(S, T) = \begin{cases} 
\text{true} & \text{if } T = \text{top} \\
\text{false} & \text{else if } S = S_1 \to S_2 \text{ and } T = T_1 \to T_2 \\
\text{subtype}(T_1, S_1) \land \text{subtype}(S_2, T_2) & \text{else if } S = \{ k_j : S_j^j \in 1..m \} \text{ and } T = \{ l_i : T_i^i \in 1..n \} \\
\{ l_i^i \in 1..n \} \subseteq \{ k_j^j \in 1..m \} & \land \text{for all } i \text{ there is some } j \in 1..m \text{ with } k_j = l_i \\
\text{and subtype}(S_j, T_i) & \text{else if } S = \{ k_j : S_j \in 1..m \} \text{ and } T = \{ l_i : T_i \in 1..n \} \\
\end{cases}
\]

**Proposition 16.1.16 [Termination].** If \( \models S <: T \) is derivable, then \( \text{subtype}(S, T) \) will return \text{true}. If not, then it will return \text{false}. 
Algorithmic typing

- The only rule we need to consider is $T$-Sub:

$$
T\text{-Sub} \quad \frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}
$$

- Can we remove it as we remove $S$-TRANS in subtyping relation?

- The term $$(\lambda r : \{ x : \text{Nat} \}. r.x)\{x = 0, y = 1\}$$ is not typable without $T$-Sub.

- However, we can always transform the derivation to the one where $T$-Sub only appears before $T$-App or at the very end.
The only rule we need to consider is $T\text{-}\text{Sub}$:

$$
T\text{-}\text{Sub} \\
\Gamma \vdash t : S \quad S <: T \\
\hline
\Gamma \vdash t : T
$$

Can we remove it as we remove $S\text{-}\text{Trans}$ in subtyping relation?

No. The term $(\lambda r : \{x : \text{Nat}\}. \; r.x) \; \{x = 0, y = 1\}$ is not typable without $T\text{-}\text{Sub}$. 
Algorithmic typing

- The only rule we need to consider is $T$-$Sub$:

\[
T$-$Sub \quad \begin{array}{c}
\Gamma \vdash t : S \\
S <: T
\end{array} \quad \Gamma \vdash t : T
\]

- Can we remove it as we remove $S$-$Trans$ in subtyping relation?

- No. The term $(\lambda r : \{x : \text{Nat}\}. r.x) \{x = 0, y = 1\}$ is not typable without $T$-$Sub$.

- However, we can always transform the derivation to the one where $T$-$Sub$ only appears before $T$-$App$ or at the very end.
Algorithmic typing

Instead of $T_{\text{SUB}}$, we add

$\begin{array}{c}
T_{\text{APP}} \\
\Gamma \models t_1 : T_1 \rightarrow T_2 \quad \Gamma \models t_2 : S_1 \quad \models S_1 <: T_1 \\
\hline
\Gamma \models t_1 \ t_2 : T_2
\end{array}$

Theorem 16.2.4 [Soundness]. If $\Gamma \models t : T$, then $\Gamma \vdash t : T$.

Theorem 16.2.5 [Completeness]. If $\Gamma \vdash t : T$, then $\Gamma \models t : S_1$ for some $S_1 <: T_1$. 
Algorithmic typing

Instead of $T_{-SUB}$, we add

\[
\frac{\Gamma \models t_1 : T_1 \rightarrow T_2 \quad \Gamma \models t_2 : S_1 \models S_1 <: T_1}{\Gamma \models t_1 t_2 : T_2}
\]

**Theorem 16.2.4 [Soundness].** If $\Gamma \models t : T$, then $\Gamma \vdash t : T$.

**Theorem 16.2.5 [Completeness].** If $\Gamma \vdash t : T$, then $\Gamma \models t : S$ for some $S <: T$. 
Conclusion

- Subtype relation can be founded in many OO languages.
- We do not have any evaluation rules for subtyping, instead, we use subsumption rule:
  \[ T-\text{SUB} \quad \frac{\Gamma \vdash t : S \quad S \subset: T}{\Gamma \vdash t : T} \]
- Some features, such as downcasts, may break type safety.
Homework

- 15.2.1, 15.2.2, 15.2.3, 15.2.4, 15.2.5, 16.2.3