Metatheory of Subtyping
Syntax-directed rules

In the simply typed lambda-calculus (without subtyping), each rule can be “read from bottom to top” in a straightforward way.

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

If we are given some \( \Gamma \) and some \( t \) of the form \( t_1 \ t_2 \), we can try to find a type for \( t \) by

1. finding (recursively) a type for \( t_1 \)
2. checking that it has the form \( T_{11} \to T_{12} \)
3. finding (recursively) a type for \( t_2 \)
4. checking that it is the same as \( T_{11} \)
Technically, the reason this works is that we can divide the “positions” of the typing relation into *input positions* ($\Gamma$ and $t$) and *output positions* ($T$).

- For the input positions, all metavariables appearing in the premises also appear in the conclusion (so we can calculate inputs to the “subgoals” from the subexpressions of inputs to the main goal)

- For the output positions, all metavariables appearing in the conclusions also appear in the premises (so we can calculate outputs from the main goal from the outputs of the subgoals)

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

The second important point about the simply typed lambda-calculus is that the set of typing rules is syntax-directed, in the sense that, for every “input” \( \Gamma \) and \( t \), there is one rule that can be used to derive typing statements involving \( t \).

E.g., if \( t \) is an application, then we must proceed by trying to use \( \text{T-App} \). If we succeed, then we have found a type (indeed, the unique type) for \( t \). If it fails, then we know that \( t \) is not typable.

\[ \longrightarrow \text{no backtracking!} \]
Non-syntax-directedness of typing

When we extend the system with subtyping, both aspects of syntax-directedness get broken.

1. The set of typing rules now includes \textit{two} rules that can be used to give a type to terms of a given shape (the old one plus \textit{T-Sub})

\[
\Gamma \vdash t : S \quad S <: T \\
\hline \\
\Gamma \vdash t : T
\]

(\textit{T-Sub})

2. Worse yet, the new rule \textit{T-Sub} itself is not syntax directed: the inputs to the left-hand subgoal are exactly the same as the inputs to the main goal! (If we translated the typing rules naively into a typechecking function, the case corresponding to \textit{T-Sub} would cause divergence.)
Moreover, the subtyping relation is not syntax directed either.

1. There are *lots* of ways to derive a given subtyping statement.
2. The transitivity rule

\[
\begin{align*}
S &<: U \quad U <: T \\
\hline \\
S &<: T
\end{align*}
\]

(S-TRANS)

is badly non-syntax-directed: the premises contain a metavariable (in an “input position”) that does not appear at all in the conclusion.

To implement this rule naively, we’d have to *guess* a value for \(U\)!
What to do?

1. Observation: We don’t need 1000 ways to prove a given typing or subtyping statement — one is enough.
   - Think more carefully about the typing and subtyping systems to see where we can get rid of excess flexibility
2. Use the resulting intuitions to formulate new “algorithmic” (i.e., syntax-directed) typing and subtyping relations
3. Prove that the algorithmic relations are “the same as” the original ones in an appropriate sense.
What to do?

1. Observation: We don’t need 1000 ways to prove a given typing or subtyping statement — one is enough.
   → Think more carefully about the typing and subtyping systems to see where we can get rid of excess flexibility.

2. Use the resulting intuitions to formulate new “algorithmic” (i.e., syntax-directed) typing and subtyping relations.

3. Prove that the algorithmic relations are “the same as” the original ones in an appropriate sense.
Developing an algorithmic subtyping relation
Subtype relation

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

\[ S <: U \quad U <: T \quad \text{S} \quad U <: T \quad \text{(S-TRANS)} \]

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

\[ \text{for each } i \quad S_i <: T_i \quad \text{(S-RCDDEPTH)} \]

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

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

\[ \{ k_j : S_j \ j \in 1..n \} <: \{ l_i : T_i \ i \in 1..n \} \quad \text{(S-RCDPERM)} \]

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

\[ S <: \text{Top} \quad \text{(S-TOP)} \]
For a given subtyping statement, there are multiple rules that could be used last in a derivation.


2. \textit{S-Refl} and \textit{S-Trans} overlap with every other rule.
Step 1: simplify record subtyping

Idea: combine all three record subtyping rules into one “macro rule” that captures all of their effects

\[
\{l_i \mid i \in 1..n\} \subseteq \{k_j \mid j \in 1..m\} \quad \text{if } k_j = l_i \implies S_j <: T_i \text{ (S-RCD)}
\]
Simpler subtype relation

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

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

\[
\begin{align*}
\{l_i: i \in 1..n\} & \subseteq \{k_j: j \in 1..m\} & \quad k_j = l_i \text{ implies } S_j &: T_i \\
\{k_j: S_j: j \in 1..m\} & <: \{l_i: T_i: i \in 1..n\} \\
\text{(S-RCDD)}
\end{align*}
\]

\[
\begin{align*}
T_1 &: S_1 & S_2 &: T_2 \\
\Rightarrow & \quad S_1 \rightarrow S_2 &: T_1 \rightarrow T_2 \\
\text{(S-ARROW)}
\end{align*}
\]

\[
S &: \text{Top} \\
\text{(S-TOP)}
\]
Step 2: Get rid of reflexivity

Observation: $S$-REFL is unnecessary.

Lemma: $S <: S$ can be derived for every type $S$ without using $S$-REFL.
Even simpler subtype relation

\[
\begin{align*}
S &: U & U &: T & S &: T \\
\{l_i : i \in 1..n\} & \subseteq \{k_j : j \in 1..m\} & k_j = l_i \text{ implies } S_j &: T_i \\
\{k_j : S_j : j \in 1..m\} & \ll \{l_i : T_i : i \in 1..n\} \\
T_1 &: S_1 & S_2 &: T_2 & S_1 \rightarrow S_2 &: T_1 \rightarrow T_2 \\
S &: \text{Top} \\
\end{align*}
\]
Step 3: Get rid of transitivity

Observation: \( S\text{-TRANS} \) is unnecessary.

**Lemma:** If \( S <: T \) can be derived, then it can be derived without using \( S\text{-TRANS} \).
“Algorithmic” subtype relation

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

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

\[ \{ l_i \mid i \in 1..n \} \subseteq \{ k_j \mid j \in 1..m \} \quad \text{for each } k_j = l_i, \quad \vdash S_j <: T_i \quad \text{(SA-RCD)} \]

\[ \vdash \{ k_j : S_j \mid j \in 1..m \} <: \{ l_i : T_i \mid i \in 1..n \} \]
Soundness and completeness

Theorem: $S <: T$ iff $\vdash S <: T$.

Proof: (Homework)

Terminology:

- The algorithmic presentation of subtyping is sound with respect to the original if $\vdash S <: T$ implies $S <: T$. (Everything validated by the algorithm is actually true.)

- The algorithmic presentation of subtyping is complete with respect to the original if $S <: T$ implies $\vdash S <: T$. (Everything true is validated by the algorithm.)
Subtyping Algorithm (pseudo-code)

The algorithmic rules can be translated directly into code:

\[
\text{subtype}(S, T) =
\]

\[
\text{if } T = \text{Top}, \text{ then } true
\]

\[
\text{else if } S = S_1 \rightarrow S_2 \text{ and } T = T_1 \rightarrow T_2
\]

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

\[
\text{else if } S = \{ k_j : S_j \mid j \in 1..m \} \text{ and } T = \{ l_i : T_i \mid i \in 1..n \}
\]

\[
\text{then } \{ l_i : i \in 1..n \} \subseteq \{ k_j : j \in 1..m \}
\]

\[
\land \text{ for all } i \in 1..n \text{ there is some } j \in 1..m \text{ with } k_j = l_i
\]

\[
\text{and subtype}(S_j, T_i)
\]

\[
\text{else false.}
\]
Recall: A decision procedure for a relation $R \subseteq U$ is a total function $p$ from $U$ to $\{true, false\}$ such that $p(u) = true$ iff $u \in R$. 

Q: What's missing?
A: How do we know that subtype is a total function? Prove it!
Recall: A decision procedure for a relation $R \subseteq U$ is a total function $p$ from $U$ to $\{true, false\}$ such that $p(u) = true$ iff $u \in R$.

Is our subtype function a decision procedure?
Recall: A decision procedure for a relation $R \subseteq U$ is a total function $p$ from $U$ to $\{true, false\}$ such that $p(u) = true$ iff $u \in R$.

Is our subtype function a decision procedure?

Since subtype is just an implementation of the algorithmic subtyping rules, we have

1. if $subtype(S, T) = true$, then $\vdash S <: T$
   (hence, by soundness of the algorithmic rules, $S <: T$)

2. if $subtype(S, T) = false$, then not $\vdash S <: T$
   (hence, by completeness of the algorithmic rules, not $S <: T$)
Recall: A decision procedure for a relation $R \subseteq U$ is a total function $p$ from $U$ to $\{true, false\}$ such that $p(u) = true$ iff $u \in R$.

Is our subtype function a decision procedure?

Since subtype is just an implementation of the algorithmic subtyping rules, we have:

1. if $subtype(S, T) = true$, then $\triangleright S <: T$
   (hence, by soundness of the algorithmic rules, $S <: T$)
2. if $subtype(S, T) = false$, then not $\triangleright S <: T$
   (hence, by completeness of the algorithmic rules, not $S <: T$)

Q: What’s missing?
Decision Procedures

Recall: A *decision procedure* for a relation $R \subseteq U$ is a total function $p$ from $U$ to \{true, false\} such that $p(u) = true$ iff $u \in R$.

Is our *subtype* function a decision procedure?

Since *subtype* is just an implementation of the algorithmic subtyping rules, we have

1. if $\text{subtype}(S, T) = true$, then $\vdash S <: T$
   (hence, by soundness of the algorithmic rules, $S <: T$)
2. if $\text{subtype}(S, T) = false$, then not $\vdash S <: T$
   (hence, by completeness of the algorithmic rules, not $S <: T$)

Q: What’s missing?

A: How do we know that *subtype* is a *total* function?
Recall: A decision procedure for a relation $R \subseteq U$ is a total function $p$ from $U$ to \{true, false\} such that $p(u) = true$ iff $u \in R$.

Is our subtype function a decision procedure?

Since subtype is just an implementation of the algorithmic subtyping rules, we have

1. if $\text{subtype}(S, T) = true$, then $\vdash S <: T$
   (hence, by soundness of the algorithmic rules, $S <: T$)

2. if $\text{subtype}(S, T) = false$, then not $\vdash S <: T$
   (hence, by completeness of the algorithmic rules, not $S <: T$)

Q: What’s missing?

A: How do we know that subtype is a total function?

Prove it!
Metatheory of Typing
For the typing relation, we have just one problematic rule to deal with: subsumption.

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

Where is this rule really needed?

For applications. E.g., the term \((\lambda r: \{x: \text{Nat}\}. r.x) \{x=0, y=1\}\) is not typable without using subsumption.

Where else? Nowhere else! Uses of subsumption to help typecheck applications are the only interesting ones.
For the typing relation, we have just one problematic rule to deal with: subsumption.

\[
\Gamma \vdash t : S \quad S <: T \\
\Gamma \vdash t : T
\]

(\text{T-SUB})

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.
Issue

For the typing relation, we have just one problematic rule to deal with: subsumption.

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

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.

Where else??
For the typing relation, we have just one problematic rule to deal with: subsumption.

\[ \Gamma \vdash t : S \quad S <: T \quad \Rightarrow \quad \Gamma \vdash t : T \]  

(T-SUB)

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.

Where else??

*Nowhere else!* Uses of subsumption to help typecheck applications are the only interesting ones.
Example (T-Abs)

\[ \Gamma, x : S_1 \vdash s_2 : S_2 \quad S_2 <: T_2 \]

\[
\begin{array}{c}
\Gamma \vdash \lambda x : S_1 . s_2 : S_1 \to T_2 \\
\end{array}
\]

\[ (T-\text{Abs}) \]

\[ (T-\text{Sub}) \]
Example (T-Abs)

\[
\begin{align*}
\Gamma, x : S_1 &\vdash s_2 : S_2 \\
\hline
\Gamma, x : S_1 &\vdash s_2 : T_2 \quad (T-Sub) \\
\hline
\Gamma &\vdash \lambda x : S_1 . s_2 : S_1 \rightarrow T_2 \quad (T-Abs)
\end{align*}
\]

becomes

\[
\begin{align*}
\Gamma, x : S_1 &\vdash s_2 : S_2 \\
\hline
\Gamma &\vdash \lambda x : S_1 . s_2 : S_1 \rightarrow S_2 \quad (T-Abs) \\
\hline
\Gamma &\vdash \lambda x : S_1 . s_2 : S_1 \rightarrow T_2 \quad (T-Sub)
\end{align*}
\]

\[
\begin{align*}
S_1 &\vdash S_1 \quad (S-Refl) \\
\hline
S_1 \rightarrow S_2 &\vdash S_1 \rightarrow T_2 \quad (S-Arrow)
\end{align*}
\]

\[
\begin{align*}
S_2 &\vdash T_2 \\
\hline
\Gamma &\vdash \lambda x : S_1 . s_2 : S_1 \rightarrow T_2
\end{align*}
\]
Example (\textsc{T-App} on the left)

\[
\begin{align*}
\Gamma \vdash s_1 &: S_{11} \rightarrow S_{12} \quad &\vdash T_{11} &: S_{11} \quad &\vdash S_{12} &: T_{12} \\
\Gamma \vdash s_2 &: T_{11} \quad &\vdash S_{11} &: S_{12} \rightarrow T_{11} \rightarrow T_{12} \\
\Gamma \vdash s_1 \ s_2 &: T_{12}
\end{align*}
\]
Example (T-App on the left)

\[ \vdash s_1 : S_{11} \rightarrow S_{12} \quad S_{11} \rightarrow S_{12} \leq : T_{11} \rightarrow T_{12} \]  
\[ \vdash s_2 : T_{11} \rightarrow T_{12} \]  
\[ \vdash s_1 \cdot s_2 : T_{12} \]

becomes

\[ \vdash s_2 : T_{11} \quad T_{11} \leq : S_{11} \]  
\[ \vdash s_1 : S_{11} \rightarrow S_{12} \quad \vdash s_2 : S_{11} \]  
\[ \vdash s_1 \cdot s_2 : S_{12} \]  
\[ \vdash s_1 \cdot s_2 : T_{12} \]
Example \((T\text{-}\text{APP on the right})\)
Example \((T\text{-App} \text{ on the right})\)

\[
\begin{align*}
\Gamma \vdash s_1 &: T_{11} \rightarrow T_{12} \\
\Gamma \vdash s_2 &: T_{11} \quad T_{2} \ll T_{11} &\quad \text{(T-Sub)} \\
\Gamma \vdash s_2 &: T_{11} &\quad \text{(T-App)} \\
\Gamma \vdash s_1 \; s_2 &: T_{12}
\end{align*}
\]

becomes

\[
\begin{align*}
\Gamma \vdash s_1 &: T_{11} \rightarrow T_{12} \\
\Gamma \vdash s_2 &: T_{11} \quad T_{2} \ll T_{11} &\quad \text{(S-Ref)} \\
T_{11} \rightarrow T_{12} \ll T_{2} \rightarrow T_{12} &\quad \text{(S-Arrow)} \\
\Gamma \vdash s_1 &: T_{11} \rightarrow T_{12} \quad \Gamma \vdash s_2 &: T_{2} &\quad \text{(T-App)} \\
\Gamma \vdash s_1 \; s_2 &: T_{12}
\end{align*}
\]
Example (T-SUB)

\[
\begin{array}{c}
\vdash s : S \\
\hline
\vdash s : U \\
\hline
\vdash s : T
\end{array}
\]

\[
\begin{array}{c}
\vdash s : S \quad S <: U \\
\hline
\vdash s : U \\
\hline
\vdash s : T
\end{array}
\]

\text{(T-SUB)}
Example (T-SUB)

\[
\begin{array}{c}
\vdash s : S \\
S <: U \\
\hline
\vdash s : U
\end{array}
\]

\[
\begin{array}{c}
\vdash s : U \\
U <: T \\
\hline
\vdash s : T
\end{array}
\]

becomes

\[
\begin{array}{c}
\vdash s : S \\
S <: U \\
\hline
\vdash s : U
\end{array}
\]

\[
\begin{array}{c}
\vdash s : U \\
U <: T \\
\hline
\vdash s : T
\end{array}
\]

\[
\begin{array}{c}
\vdash s : S \\
S <: T \\
\hline
\vdash s : T
\end{array}
\]