Syntax-directed rules

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

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

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 \( T_{11} \rightarrow T_{12} \)
3. finding (recursively) a type for \( t_2 \)
4. checking that it is the same as \( T_{11} \)

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 \( T\text{-App} \). 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.

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

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 two rules that can be used to give a type to terms of a given shape (the old one plus \( T\text{-Sub} \))

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

2. Worse yet, the new rule \( T\text{-Sub} \) 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 \( T\text{-Sub} \) would cause divergence.)
Non-syntax-directedness of subtyping

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

   \[ \frac{S < U \quad U < T}{S < T} \]  

   (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.

Developing an algorithmic subtyping relation

Subtype relation

\[ S <: S \quad \text{(S-Refl)} \]
\[ S <: U \quad U <: T \quad S <: T \quad \text{(S-Trans)} \]
\[ \{l_i : T_i \mid i \in 1..n+k\} <: \{l_i : T_i \mid i \in 1..n\} \quad \text{(S-RcdWidth)} \]

for each \( i \) \( S_i <: T_i \)
\[ \{k_j : S_j \mid j \in 1..n\} \text{ is a permutation of } \{l_i : T_i \mid i \in 1..n\} \quad \text{(S-RcdPerm)} \]
\[ T_1 <: S_1 \quad S_2 <: T_2 \]
\[ S_1 => S_2 <: T_1 => T_2 \quad \text{(S-Arrow)} \]
\[ S <: \text{Top} \quad \text{(S-Top)} \]

Issues

For a given subtyping statement, there are multiple rules that could be used last in a derivation.

1. The conclusions of S-RcdWidth, S-RcdDepth, and S-RcdPerm overlap with each other.
2. S-Refl and 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

\[
\{l_i : S_i \mid i \in 1..n\} \subseteq \{k_j : T_j \mid j \in 1..m\} \quad k_j = l_i \implies S_j <: T_i
\]

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

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.

Step 3: Get rid of transitivity

Observation: S-Trans is unnecessary.

**Lemma:** If $S <: T$ can be derived, then it can be derived without using S-Trans.

Simpler subtype relation

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

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

\[
\{l_i : S_i \mid i \in 1..n\} \subseteq \{k_j : T_j \mid j \in 1..m\} \quad k_j = l_i \implies S_j <: T_i
\]

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

\[
T_1 <: S_1 \quad S_2 <: T_2
\]

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

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

Even simpler subtype relation

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

\[
\{l_i : S_i \mid i \in 1..n\} \subseteq \{k_j : T_j \mid j \in 1..m\} \quad k_j = l_i \implies S_j <: T_i
\]

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

\[
T_1 <: S_1 \quad S_2 <: T_2
\]

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

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

“Algorithmic” subtype relation

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

\[
\therefore T_1 <: S_1 \quad \therefore S_2 <: T_2
\]

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

\[
\therefore \{l_i : S_i \mid i \in 1..n\} \subseteq \{k_j : T_j \mid j \in 1..m\} \quad \text{for each } k_j = l_i, \therefore S_j <: T_i
\]

\[
\therefore \{k_j : S_j \mid j \in 1..m\} <: \{l_i : T_i \mid i \in 1..n\} \quad \text{(SA-Rcd)}
\]
Soundness and completeness

**Theorem:** \( S \triangleleft T \) iff \( S \triangleleft T \).

**Proof:** (Homework)

Terminology:

- The algorithmic presentation of subtyping is *sound* with respect to the original if \( \triangleright S \triangleleft T \) implies \( S \triangleleft T \). (Everything validated by the algorithm is actually true.)
- The algorithmic presentation of subtyping is *complete* with respect to the original if \( S \triangleleft T \) implies \( \triangleright S \triangleleft T \). (Everything true is validated by the algorithm.)

Subtyping Algorithm (pseudo-code)

The algorithmic rules can be translated directly into code:

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

\( S \subseteq T \) if
\( S \triangleleft T \) for all \( u \) and \( j \)
\( \text{subtype}(u, j) \text{ iff } u \triangleleft T \)
\( S = T \) if \( \text{subtype}(S, T) \text{ and } T = \text{subtype}(T, S) \)
\( S \subseteq T \text{ and } \text{subtype}(S, T) \text{ for all } i \in [1..n] \) there is some \( j \in [1..m] \) with \( k_j = 1 \) and \( \text{subtype}(S_j, T_i) \)

else \text{false}.

Decision Procedures

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

Q: What’s missing?
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!

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

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

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.

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

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

(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-App on the right)

\[
\begin{align*}
\Gamma \vdash s_1 : T_{11} \rightarrow T_{12} & \quad \Gamma \vdash s_2 : T_{11} \\
\Gamma \vdash s_1 \ s_2 : T_{12} &
\end{align*}
\]

Example (T-Sub)

\[
\begin{align*}
\Gamma \vdash s : S & \quad S \subset U \\
\Gamma \vdash s : U & \quad U \subset T \\
\Gamma \vdash s : T &
\end{align*}
\]

Example (T-App)

\[
\begin{align*}
\Gamma \vdash s_1 : T_{11} \rightarrow T_{12} & \quad \Gamma \vdash s_2 : T_{11} \\
\Gamma \vdash s_1 \ s_2 : T_{12} &
\end{align*}
\]

Example (T-Sub)

\[
\begin{align*}
\Gamma \vdash s : S & \quad S \subset U \\
\Gamma \vdash s : U & \quad U \subset T \\
\Gamma \vdash s : T &
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash s_1 : T_{11} \rightarrow T_{12} & \quad \Gamma \vdash s_2 : T_{11} \\
\Gamma \vdash s_1 \ s_2 : T_{12} &
\end{align*}
\]