Chap 16

Metatheory of Subtyping

Algorithmic Subtyping
Algorithmic Typing
Joins and Meets
Developing an algorithmic subtyping relation
Subtype Relation

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

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

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

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

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

\[ \{ k_j : S_j \mid j \in 1..n \} <: \{ l_i : T_i \mid i \in 1..n \} \]

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

\[ S <: \text{Top} \quad (S\text{-TOP}) \]
Issues in Subtyping

For a given subtyping statement, there are multiple rules that could be used 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.
What to do?

We'll turn the *declarative version* of subtyping into the *algorithmic version*.

The *problem* was that we don't have an algorithm to decide when $S <: T$ or $\Gamma \vdash t : T$.

Both sets of rules are not *syntax-directed*. 
Syntax-directed rules

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

\[
\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \; t_2 : T_{12}} \quad (T\text{-}APP)
\]
In the simply typed lambda-calculus (without subtyping), each rule can be “read from bottom to top” in a straightforward way.

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

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

Technically, the reason this works is that we can *divide the “positions”* of the typing relation into *input positions* (i.e., $\Gamma$ 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)

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

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

$$
\Gamma \vdash t : S \quad S \ll T \\
\hline
\Gamma \vdash t : T
$$

(T-Sub)

2. Worse yet, the new rule $T\text{-}\text{SUB}$ itself is not syntax directed: the *inputs* to the left-hand subgoal are exactly the same as the *inputs* to the main goal!

(Hence, if we translated the typing rules naively into a typechecking function, the case corresponding to $T\text{-}\text{SUB}$ 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

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

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 have to guess a value for \( U \)!
What to do?

1. **Observation**: We don’t need lots of 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.
Algorithmic Subtyping
What to do

How do we change the rules deriving $S <: T$ to be *syntax-directed*?

There are lots of ways to derive a given subtyping statement $S <: T$.

The general idea is to *change this system* so that there is *only one way* to derive it.
Step 1: simplify record subtyping

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

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

\[ S <: S \]  \hspace{1cm} (S-REFL)

\[
\begin{align*}
S &: U & U &: T \\
\hline \\
S &: T
\end{align*}
\]  \hspace{1cm} (S-TRANS)

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

\[
\begin{align*}
T_1 &: S_1 & S_2 &: T_2 \\
\hline \\
S_1 \rightarrow S_2 &: T_1 \rightarrow T_2
\end{align*}
\]  \hspace{1cm} (S-ARROW)

\[ S <: \text{Top} \]  \hspace{1cm} (S-TOP)
Step 2: Get rid of reflexivity

*Observation:* $S$-$\text{REFL}$ is unnecessary.

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

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

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

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

\[
S <: \text{Top} \\
\]
\[(S\text{-TOP})\]
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.
Even simpler subtype relation

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

\[ \{k_j : S_j \mid j \in 1..m\} \ll \{l_i : T_i \mid i \in 1..n\} \]

\[ T_1 \ll S_1 \quad S_2 \ll T_2 \]

\[ S_1 \rightarrow S_2 \ll T_1 \rightarrow T_2 \]

\[ S \ll \text{Top} \]

(S-RCD)

(S-ARROW)

(S-TOP)
“Algorithmic” subtype relation

\[
\begin{array}{c}
\implies S <: \text{Top} \\
\implies T_1 <: S_1 \\
\implies S_2 <: T_2 \\
\implies S_1 \rightarrow S_2 <: T_1 \rightarrow T_2
\end{array}
\]

\begin{array}{c}
\{l_i, i \in \{1, \ldots, n\}\} \subseteq \{k_j, j \in \{1, \ldots, m\}\} \\
\text{for each } k_j = l_i, \implies S_j <: T_i
\end{array}

\implies \{k_j : S_j, j \in \{1, \ldots, m\}\} <: \{l_i : T_i, i \in \{1, \ldots, n\}\}
Soundness and completeness

Theorem: \( S <: T \) iff \( \rightarrow S <: T \)

Terminology:

- The algorithmic presentation of subtyping is *sound* with respect to the original, if \( \rightarrow S <: T \) 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 \( \rightarrow S <: T \). (Everything true is validated by the algorithm.)
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 $u \in R$. 

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

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

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 \( S <: T \) (hence, by soundness of the algorithmic rules, \( S <: T \))
2. if \( \text{subtype}(S, T) = false \), then not \( S <: T \) (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**?
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 $subtype(S, T) = true$, then $\mapsto S <: T$ (hence, by soundness of the algorithmic rules, $S <: T$)
2. if $subtype(S, T) = false$, then $not \mapsto S <: T$ (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!
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 $u \in R$.

Example:

\[
U = \{1, 2, 3\} \\
R = \{(1, 2), (2, 3)\}
\]

Note that, we are saying nothing about computability.
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 \( u \in R \).

Example:

\[ U = \{1, 2, 3\} \]
\[ R = \{(1, 2), (2, 3)\} \]

The function \( p \) whose graph is

\[
\{ ((1, 2), true), ((2, 3), true),
((1, 1), false), ((1, 3), false),
((2, 1), false), ((2, 2), false),
((3, 1), false), ((3, 2), false), ((3, 3), false) \}
\]

is a decision function for \( R \).
**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 $u \in R$.

Example:

\[
U = \{1, 2, 3\} \\
R = \{(1, 2), (2, 3)\}
\]

The function $p'$ whose graph is

\[
\{(((1, 2), \text{true}), ((2, 3), \text{true})\}
\]

is not a decision function for $R$. 
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 $u \in R$.

Example:

$U = \{1, 2, 3\}$
$R = \{(1, 2), (2, 3)\}$

The function $p''$ whose graph is

\{\((1, 2), \text{true}\), \((2, 3), \text{true}\), \((1, 3), \text{false}\)\}

is also not a decision function for $R$. 
We want a decision procedure to be a procedure.

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

\[ U = \{1, 2, 3\} \]
\[ R = \{(1, 2), (2, 3)\} \]

The function
\[ p(x, y) = \text{if } x = 2 \text{ and } y = 3 \text{ then } \text{true} \]
\[ \text{else if } x = 1 \text{ and } y = 2 \text{ then } \text{true} \]
\[ \text{else } \text{false} \]

whose graph is
\[ \{(1, 2), \text{true}\}, \{(2, 3), \text{true}\}, \]
\[ \{(1, 1), \text{false}\}, \{(1, 3), \text{false}\}, \]
\[ \{(2, 1), \text{false}\}, \{(2, 2), \text{false}\}, \]
\[ \{(3, 1), \text{false}\}, \{(3, 2), \text{false}\}, \{(3, 3), \text{false}\}\] is a decision procedure for \( R \).
Example

\[ U = \{1, 2, 3\} \]
\[ R = \{(1, 2), (2, 3)\} \]

The recursively defined *partial function* \( p(x, y) \):

\[ p(x, y) = \begin{cases} 
\text{true} & \text{if } x = 2 \text{ and } y = 3 \\
\text{true} & \text{if } x = 1 \text{ and } y = 2 \\
\text{false} & \text{if } x = 1 \text{ and } y = 3 \\
\text{else } p(x, y) 
\end{cases} \]
Example

\[ U = \{1, 2, 3\} \]
\[ R = \{(1, 2), (2, 3)\} \]

The recursively defined partial function

\[
p(x, y) = \begin{cases} 
  \text{true} & \text{if } x = 2 \text{ and } y = 3 \\
  \text{true} & \text{if } x = 1 \text{ and } y = 2 \\
  \text{false} & \text{if } x = 1 \text{ and } y = 3 \\
  p(x, y) & \text{otherwise}
\end{cases}
\]

whose graph is

\[ \{((1, 2), \text{true}), ((2, 3), \text{true}), ((1, 3), \text{false})\} \]

is not a decision procedure for \( R \).
Subtyping Algorithm

This *recursively defined total function* is a decision procedure for the subtype relation:

\[
\text{subtype}(S, T) = \\
\text{if } T = \text{Top}, \text{ then } true \\
\text{else if } S = S_1 \rightarrow S_2 \text{ and } T = T_1 \rightarrow T_2 \\
\text{ then } \text{subtype}(T_1, S_1) \land \text{subtype}(S_2, T_2) \\
\text{else if } S = \{k_j: S_j^{j \in 1..m}\} \text{ and } T = \{l_i: T_i^{i \in 1..n}\} \\
\text{ then } \{l_i^{i \in 1..n}\} \subseteq \{k_j^{j \in 1..m}\} \\
\quad \land \text{ for all } i \in 1..n \text{ there is some } j \in 1..m \text{ with } k_j = l_i \\
\text{ and } \text{subtype}(S_j, T_i) \\
\text{else false.}
\]
Subtyping Algorithm

This *recursively defined total function* is a decision procedure for the subtype relation:

$$\text{subtype}(S, T) =$$

- if $T = \text{Top}$, then $true$
- 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 $S = \{k_j : S_j^{j\in1..m}\}$ and $T = \{l_i : T_i^{i\in1..n}\}$
  - then $\{l_i^{i\in1..n}\} \subseteq \{k_j^{j\in1..m}\}$
    - and for all $i \in 1..n$ there is some $j \in 1..m$ with $k_j = l_i$
    - and $\text{subtype}(S_j, T_i)$
  - else $false$.

To show this, we need to prove:

1. that it returns $true$ whenever $S <: T$, and
2. that it returns either $true$ or $false$ on all inputs.
Algorithmic Typing
Algorithmic typing

How do we implement a type checker for the lambda-calculus with subtyping?

Given a context $\Gamma$ and a term $t$, how do we determine its type $T$, such that $\Gamma \vdash t : T$?
For the typing relation, we have *just one problematic rule* to deal with: subsumption rule

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

Q: where is this rule really needed?
Issue

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

\[
\begin{array}{c}
\Gamma \vdash t : S \\
S \prec T \\
\hline
\Gamma \vdash t : T
\end{array}
\]  

(T-Sub)

Q: 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

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

(Q-SUB)

Q: 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??
Issue

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

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

Q: 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.
Plan

1. Investigate *how subsumption is used* in typing derivations by *looking at examples* of how it can be “pushed through” other rules

2. Use the intuitions gained from these examples to design a new, algorithmic typing relation that
   – *Omits subsumption*
   – Compensates for its absence by *enriching the application rule*

3. *Show that* the algorithmic typing relation is essentially equivalent to the original, declarative one
Example (T-ABS)

\[
\begin{array}{c}
\Gamma, x : S_1 \vdash s_2 : S_2 \\
\hline
\Gamma \vdash \lambda x : S_1 . s_2 : S_1 \rightarrow T_2
\end{array}
\]

\[
\begin{array}{c}
\vdash \\
S_2 <: T_2 \\
\hline
\Gamma, x : S_1 \vdash s_2 : T_2
\end{array}
\] (T-Sub)

\[
\begin{array}{c}
\Gamma \vdash \lambda x : S_1 . s_2 : S_1 \rightarrow T_2
\end{array}
\] (T-Abs)
Example (T-ABS)

\[
\begin{align*}
\Gamma, x: S_1 \vdash s_2 : S_2 \\
\hline
S_2 <: T_2 \\
\hline
\Gamma, x: S_1 \vdash s_2 : T_2 \\
\hline
\Gamma \vdash \lambda x: S_1 . s_2 : S_1 \rightarrow T_2
\end{align*}
\]

becomes

\[
\begin{align*}
\Gamma, x: S_1 \vdash s_2 : S_2 \\
\hline
\Gamma \vdash \lambda x: S_1 . s_2 : S_1 \rightarrow S_2 \\
\hline
\Gamma \vdash \lambda x: S_1 . s_2 : S_1 \rightarrow T_2
\end{align*}
\]
Intuitions

These examples show that we do not need T-SUB to “enable” T-ABS: given any typing derivation, we can construct a derivation *with the same conclusion* in which T-SUB is never used immediately before T-ABS.

What about T-APP?

We’ve already observed that T-SUB is required for typechecking some *applications*. So we expect to find that we *cannot* play the same game with T-APP as we’ve done with T-ABS.

Let’s see why.
Example (T–Sub with T-APP on the left)

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

\[
\begin{align*}
T_{11} &\prec S_{11} \\
S_{12} &\prec T_{12} \\
\hline
\hline
\Gamma \vdash s_1 &: T_{11} \\
\hline
\Gamma \vdash s_2 &: T_{11} \\
\hline
\Gamma \vdash s_1 \circ s_2 &: T_{12} \\
\hline
\end{align*}
\]

becomes

\[
\begin{align*}
\Gamma \vdash s_1 &: S_{11} \rightarrow S_{12} \\
\hline
\hline
\Gamma \vdash s_2 &: T_{11} \\
\hline
\Gamma \vdash s_2 &: S_{11} \\
\hline
\Gamma \vdash s_1 \circ s_2 &: S_{12} \\
\hline
\Gamma \vdash s_1 \circ s_2 &: T_{12} \\
\hline
\end{align*}
\]
Example (T–Sub with T-APP on the right)

\[
\begin{align*}
\Gamma \vdash s_1 : T_{11} \rightarrow T_{12} & \quad \frac{\Gamma \vdash s_2 : T_2 \quad T_2 \triangleleft T_{11}}{\Gamma \vdash s_2 : T_{11}} \quad (T\text{-Sub}) \\
\frac{\Gamma \vdash s_1 \ s_2 : T_{12}}{} & \\
\end{align*}
\]

becomes

\[
\begin{align*}
\Gamma \vdash s_1 : T_{11} \rightarrow T_{12} & \quad \frac{T_2 \triangleleft T_{11} \quad T_{12} \triangleleft T_{12}}{} \quad (S\text{-REFL}) \\
\quad \frac{T_{11} \rightarrow T_{12} \triangleleft T_2 \rightarrow T_{12}}{\Gamma \vdash T_{11} \rightarrow T_{12} \triangleleft T_2 \rightarrow T_{12}} \quad (S\text{-ARROW}) \\
\frac{\Gamma \vdash s_1 : T_2 \rightarrow T_{12}}{} \quad \frac{\Gamma \vdash s_2 : T_2}{\Gamma \vdash s_1 \ s_2 : T_{12}} \quad (T\text{-APP}) \\
\end{align*}
\]
Observations

So we’ve seen that uses of subsumption can be “pushed” from one of immediately before T-APP’s premises to the other, but \textit{cannot be completely eliminated}.
Example (nested uses of T-Sub)

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

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

(T-SUB)
Example (nested uses of T-Sub)

\[
\begin{align*}
\Gamma \vdash s : S & \quad S <: U \\
\hline
\Gamma \vdash s : U & \quad U <: T \\
\hline
\Gamma \vdash s : T & \quad (T\text{-}SUB)
\end{align*}
\]

becomes

\[
\begin{align*}
\vdash & \quad (S\text{-}TRANS) \\
\vdash & \quad (T\text{-}SUB)
\end{align*}
\]
Summary

What we’ve learned:

– Uses of the T-Sub rule can be “pushed down” through typing derivations until they encounter either
  1. a use of T-App or
  2. the root of the derivation tree.
– In both cases, multiple uses of T-Sub can be coalesced into a single one.
Summary

What we’ve learned:

– Uses of the T-Sub rule can be “pushed down” through typing derivations until they encounter either
  1. a use of T-App or
  2. the root of the derivation tree.
– In both cases, multiple uses of T-Sub can be collapsed into a single one.

This suggests a notion of “normal form” for typing derivations, in which there is

– exactly one use of T-Sub before each use of T-App
– one use of T-Sub at the very end of the derivation
– no uses of T T-Sub anywhere else.
Algorithmic Typing

The next step is to “build in” the use of subsumption in application rules, by changing the $T$-App rule to incorporate a subtyping premise.

$$\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_2 \quad \vdash T_2 <: T_{11}$$

$$\Gamma \vdash t_1 \ t_2 : T_{12}$$

Given any typing derivation, we can now

1. **normalize** it, to move all uses of subsumption to either just before applications (in the right-hand premise) or at the very end

2. **replace** uses of $T$-App with $T$-SUB in the right-hand premise by uses of the extended rule above

This yields a derivation in which there is just one use of subsumption, at the very end!
Minimal Types

But... if subsumption is only used at the very end of derivations, then it is actually *not needed* in order to show that *any term is typable*!

It is just used to give *more* types to terms that have already been shown to have a type.

In other words, if we dropped subsumption completely (after refining the application rule), we would still be able to give types to exactly the same set of terms — we just would not be able to give as many types to some of them.

If we drop subsumption, then the remaining rules will assign a *unique, minimal* type to each typable term.

For purposes of building a typechecking algorithm, this is enough.
Final Algorithmic Typing Rules

\[
\begin{align*}
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} & \quad \text{(TA-VAR)} \\
\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} & \quad \text{(TA-ABS)} \\
\frac{\Gamma \vdash t_1 : T_1 \quad T_1 = T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 \ t_2 : T_{12}} & \quad \text{(TA-APP)} \\
\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_1=t_1 \ldots l_n=t_n\} : \{l_1:T_1 \ldots l_n:T_n\}} & \quad \text{(TA-RCD)} \\
\frac{\Gamma \vdash t_1 : R_1 \quad R_1 = \{l_1:T_1 \ldots l_n:T_n\}}{\Gamma \vdash t_1.l_i : T_i} & \quad \text{(TA-PROJ)}
\end{align*}
\]
Completeness of the algorithmic rules

Theorem [Minimal Typing]: If \( \Gamma \vdash t : T \), then \( \Gamma \mapsto t : S \) for some \( S <: T \).
Completeness of the algorithmic rules

Theorem [Minimal Typing]: If $\Gamma \vdash t : T$, then $\Gamma \mapsto t : S$ for some $S <: T$.

Proof: Induction on typing derivation.

(N.b.: All the messing around with transforming derivations was just to build intuitions and decide what algorithmic rules to write down and what property to prove: the proof itself is a straightforward induction on typing derivations.)
Meets and Joins
Adding Booleans

Suppose we want to add *booleans* and *conditionals* to the language we have been discussing.

For the declarative presentation of the system, we just add in the appropriate syntactic forms, evaluation rules, and typing rules.

\[
\begin{align*}
\Gamma \vdash \text{true} : \text{Bool} \\
\Gamma \vdash \text{false} : \text{Bool} \\
\Gamma \vdash t_1 : \text{Bool} & \quad \Gamma \vdash t_2 : T & \quad \Gamma \vdash t_3 : T \\
\hline
\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T
\end{align*}
\]
A Problem with Conditional Expressions

For the algorithmic presentation of the system, however, we encounter a little difficulty.

What is the minimal type of

\[
\text{if } \text{true} \text{ then } \{x = \text{true}, y = \text{false}\} \text{ else } \{x = \text{true}, z = \text{true}\}
\]
The Algorithmic Conditional Rule

More generally, we can use subsumption to give an expression

\[ \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \]

any type that is a possible type of both \( t_2 \) and \( t_3 \).

So the \textit{minimal} type of the conditional is the \textit{least common supertype} (or \textit{join}) of the minimal type of \( t_2 \) and the minimal type of \( t_3 \).

\[
\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \lor T_3} \quad \text{(T-IF)}
\]
The Algorithmic Conditional Rule

More generally, we can use subsumption to give an expression

\[ \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \]

any type that is a possible type of both \( t_2 \) and \( t_3 \).

So the \textit{minimal} type of the conditional is the \textit{least common supertype} (or \textit{join}) of the minimal type of \( t_2 \) and the minimal type of \( t_3 \).

\[ \Gamma \proves t_1 : \text{Bool} \quad \Gamma \proves t_2 : T_2 \quad \Gamma \proves t_3 : T_3 \]

\[ \Gamma \proves \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \lor T_3 \quad (\text{T-IF}) \]

Q: Does such a type exist for every \( T_2 \) and \( T_3 \)??
Existence of Joins

**Theorem:** For every pair of types $S$ and $T$, there is a type $J$ such that

1. $S <: J$
2. $T <: J$
3. If $K$ is a type such that $S <: K$ and $T <: K$, then $J <: K$.

i.e., $J$ is the smallest type that is a supertype of both $S$ and $T$.

How to prove it?
Examples

What are the joins of the following pairs of types?

1. \( \{ x: \text{Bool}, y: \text{Bool} \} \) and \( \{ y: \text{Bool}, z: \text{Bool} \} \)?
2. \( \{ x: \text{Bool} \} \) and \( \{ y: \text{Bool} \} \)?
3. \( \{ x: \{ a: \text{Bool}, b: \text{Bool} \} \} \) and \( \{ x: \{ b: \text{Bool}, c: \text{Bool} \}, y: \text{Bool} \} \)?
4. \( \{ \} \) and \( \text{Bool} \)?
5. \( \{ x: \{ \} \} \) and \( \{ x: \text{Bool} \} \)?
6. \( \text{Top} \rightarrow \{ x: \text{Bool} \} \) and \( \text{Top} \rightarrow \{ y: \text{Bool} \} \)?
7. \( \{ x: \text{Bool} \} \rightarrow \text{Top} \) and \( \{ y: \text{Bool} \} \rightarrow \text{Top} \)?
Meets

To calculate joins of arrow types, we also need to be able to calculate meets (greatest lower bounds)!

Unlike joins, meets do not necessarily exist.
E.g., \( \text{Bool} \rightarrow \text{Bool} \) and \{\} have no common subtypes, so they certainly don’t have a greatest one!

However...
Existence of Meets

**Theorem:** For every pair of types $S$ and $T$, if there is any type $N$ such that $N <: S$ and $N <: T$, then there is a type $M$ such that

1. $M <: S$
2. $M <: T$
3. If $O$ is a type such that $O <: S$ and $O <: T$, then $O <: M$.

i.e., $M$ (when it exists) is the largest type that is a subtype of both $S$ and $T$. 
Existence of Meets

**Theorem:** For every pair of types $S$ and $T$, if there is any type $N$ such that $N <: S$ and $N <: T$, then there is a type $M$ such that

1. $M <: S$
2. $M <: T$
3. If $O$ is a type such that $O <: S$ and $O <: T$, then $O <: M$.

i.e., $M$ (when it exists) is the largest type that is a subtype of both $S$ and $T$.

Jargon: In the simply typed lambda calculus with subtyping, records, and booleans...

- The subtype relation *has joins*
- The subtype relation *has bounded meets*
Examples

What are the meets of the following pairs of types?

1. \{x: \text{Bool}, y: \text{Bool}\} and \{y: \text{Bool}, z: \text{Bool}\}?
2. \{x: \text{Bool}\} and \{y: \text{Bool}\}?
3. \{x: \{a: \text{Bool}, b: \text{Bool}\}\} and \{x: \{b: \text{Bool}, c: \text{Bool}\}, y: \text{Bool}\}?
4. \{\} and \text{Bool}?
5. \{x: \{\}\} and \{x: \text{Bool}\}?
6. \text{Top} \rightarrow \{x: \text{Bool}\} and \text{Top} \rightarrow \{y: \text{Bool}\}?
7. \{x: \text{Bool}\} \rightarrow \text{Top} and \{y: \text{Bool}\} \rightarrow \text{Top}?
Calculating Joins

\[
S \lor T = \begin{cases}
\text{Bool} & \text{if } S = T = \text{Bool} \\
M_1 \rightarrow J_2 & \text{if } S = S_1 \rightarrow S_2 \quad T = T_1 \rightarrow T_2 \\
\{j_I : J_I \mid I \in \{1, \ldots, q\}\} & \text{if } S = \{k_j : S_j \mid j \in \{1, \ldots, m\}\} \quad T = \{l_i : T_i \mid i \in \{1, \ldots, n\}\} \\
\text{Top} & \text{otherwise}
\end{cases}
\]

\[
S_I \lor T_I = J_I \quad \text{for each } j_I = k_j = l_i
\]
Calculating Meets

\[ S \land T = \begin{cases} 
S & \text{if } T = \text{Top} \\
T & \text{if } S = \text{Top} \\
\text{Bool} & \text{if } S = T = \text{Bool} \\
J_1 \rightarrow M_2 & \text{if } S = S_1 \rightarrow S_2 \quad T = T_1 \rightarrow T_2 \\
\quad \quad S_1 \lor T_1 = J_1 \quad S_2 \land T_2 = M_2 \\
\{m_i : M_i \}_{i \in 1..q} & \text{if } S = \{k_j : S_j \}_{j \in 1..m} \\
\quad \quad T = \{1_i : T_i \}_{i \in 1..n} \\
\quad \quad \{m_i \}_{i \in 1..q} = \{k_j \}_{j \in 1..m} \cup \{1_i \}_{i \in 1..n} \\
\quad \quad S_j \land T_i = M_i & \text{for each } m_i = k_j = 1_i \\
\quad \quad M_i = S_j & \text{if } m_i = k_j \text{ occurs only in } S \\
\quad \quad M_i = T_i & \text{if } m_i = 1_i \text{ occurs only in } T \\
\text{fail} & \text{otherwise}
\end{cases} \]
Homework

• Read and digest chapter 16 & 17

• HW: 16.1.2; 16.2.6