Recap on Subtyping
Subsumption

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.

Which can be formalized as 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 $T$

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

*Principle of safe substitution*
Subtype Relation

$$S <: S$$  \hspace{1cm} (S\text{-Ref})

$$S <: U \quad U <: T \quad \frac{}{S <: T}$$  \hspace{1cm} (S\text{-Trans})

$$\{l_i: T_i \mid i \in 1..n+k\} <: \{l_i: T_i \mid i \in 1..n\}$$  \hspace{1cm} (S\text{-RcdWidth})

for each i $$S_i <: T_i$$

$$\{l_i: S_i \mid i \in 1..n\} <: \{l_i: T_i \mid i \in 1..n\}$$  \hspace{1cm} (S\text{-RcdDepth})

$$\{k_j: S_j \mid j \in 1..n\}$$ is a permutation of $$\{l_i: T_i \mid i \in 1..n\}$$

$$\{k_j: S_j \mid j \in 1..n\} <: \{l_i: T_i \mid i \in 1..n\}$$  \hspace{1cm} (S\text{-RcdPerm})

$$T_1 <: S_1 \quad S_2 <: T_2 \quad \frac{T_1 \rightarrow S_2 :: S_1 \rightarrow T_2}{S <: \text{Top}}$$  \hspace{1cm} (S\text{-Top})
Syntax
\[ t ::= \]
\[ x \quad \text{variable} \]
\[ \lambda x : T . t \quad \text{abstraction} \]
\[ t \: t \quad \text{application} \]
\[ v ::= \]
\[ \lambda x : T . t \quad \text{abstraction value} \]
\[ T ::= \]
\[ \text{Top} \quad \text{maximum type} \]
\[ T \rightarrow T \quad \text{type of functions} \]
\[ \Gamma ::= \]
\[ \emptyset \quad \text{empty context} \]
\[ \Gamma , x : T \quad \text{term variable binding} \]

Subtyping
\[
\text{S} <: \text{S} \quad \text{(S-REFL)}
\]
\[
\frac{\text{S} <: \text{U} \quad \text{U} <: \text{T}}{\text{S} <: \text{T}} \quad \text{(S-TRANS)}
\]
\[
\frac{\text{T}_1 <: \text{S}_1 \quad \text{S}_2 <: \text{T}_2}{\text{S}_1 \rightarrow \text{S}_2 <: \text{T}_1 \rightarrow \text{T}_2} \quad \text{(S-ARROW)}
\]

Typing
\[
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}
\]
\[
\frac{\Gamma , x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2} \quad \text{(T-ABS)}
\]
\[
\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \: t_2 : T_{12}} \quad \text{(T-APP)}
\]
\[
\frac{\Gamma \vdash t : S \quad \text{S} <: \text{T}}{\Gamma \vdash t : T} \quad \text{(T-SUB)}
\]

Evaluation
\[
\frac{t_1 \rightarrow t'_1}{t_1 \: t_2 \rightarrow t'_1 \: t_2} \quad \text{(E-APP1)}
\]
\[
\frac{t_2 \rightarrow t'_2}{v_1 \: t_2 \rightarrow v_1 \: t'_2} \quad \text{(E-APP2)}
\]
\[
(\lambda x : T_{11} . t_{12}) \: v_2 \rightarrow [x \rightarrow v_2] t_{12} \quad \text{(E-APPABS)}
\]
Records

New syntactic forms

\[ t ::= \cdots \begin{array}{l}
\{i_1 = t_{i_1} \}_{i \in \{1, \ldots, n\}} \\
t.1
\end{array} \]

New evaluation rules

\[ \{i_1 = v_{i_1} \}_{i \in \{1, \ldots, n\}} . l_j \rightarrow v_j \]  

New typing rules

\[ \Gamma \vdash t : T \]

\[ \Gamma \vdash \{l_i = t_i \}_{i \in \{1, \ldots, n\}} \rightarrow \{l_i = v_{i_1} \}_{i \in \{1, \ldots, j-1\}}, l_j = t_j, l_k = t_k \}_{k \in \{j+1, \ldots, n\}} \]

\[ t_1 \rightarrow t'_1 \]

\[ t_1.1 \rightarrow t'_1.1 \]  

\[ t_j \rightarrow t'_j \]

\[ \Gamma \vdash t_1 : \{l_i : T_i \}_{i \in \{1, \ldots, n\}} \]

\[ \Gamma \vdash t_1 . l_j : T_j \]  

Extends \( \lambda \) (9-1)
Records & Subtyping

New subtyping rules

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

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

\[ \{l_i:S_i \, i \in 1..n\} <: \{l_i:T_i \, i \in 1..n\} \]  

\[ S <: T \]

Extends \( \lambda <: (15-1) \) and simple record rules (15-2)

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

\( S <: T \)

\[ \{k_j:S_j \, j \in 1..n\} <: \{l_i:T_i \, i \in 1..n\} \]  
(S-RCDPERM)
Properties of Subtyping
Do the Statements of progress and preservation theorems need change?

Statements of progress and preservation theorems are unchanged from $\lambda \rightarrow$. 
Safety

Statements of progress and preservation theorems are unchanged from $\lambda \rightarrow$.

However, Proofs become a bit more involved, because the typing relation is no longer syntax directed.

Given a derivation, we don’t always know what rule was used in the last step.

e.g., the rule T-SUB could appear anywhere.

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

When we say a set of rules is syntax-directed we mean two things:

1. There is *exactly one rule* in the set that applies to each syntactic form. (We can tell by the syntax of a term which rule to use.)
   
   - In order to derive a type for $t_1 t_2$, we must use T-App.

2. We don't have to "guess" an input (or output) for any rule.
   
   - To derive a type for $t_1 t_2$, we need to derive a type for $t_1$ and a type for $t_2$. 
Preservation

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

*Proof:* By induction on *typing derivations*.

*Which cases* are likely to be *hard*?
Subsumption case

Case T-Sub: $t : S \quad S <: T$

By the induction hypothesis, $\Gamma \vdash t' : S$.
By T-Sub, $\Gamma \vdash t' : T$.

Not hard!
Application case

Case T-App:

\[ t = t_1 \, t_2 \quad \Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

By the inversion lemma for evaluation, there are three rules by which \( t \to t' \) can be derived:

E-App1, E-App2, and E-AppAbs.

Proceed by cases.
Application case

Case T-App:
\[ t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

By the evaluation rules in Figure 15-1 and 15-2, there are three rules by which \( t \to t' \) can be derived: E-App1, E-App2, and E-AppAbs.

Proceed by cases.

Subcase E-App1: \( t_1 \to t'_1 \quad t' = t'_1 \ t_2 \)

The result follows from the induction hypothesis and T-App.

\[
\begin{align*}
\Gamma \vdash & t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11} \\
\hline
\Gamma \vdash & t_1 \ t_2 : T_{12} \\
\end{align*}
\]

(T-APP)
Application case

Case T-App:

\[ t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

Subcase E-App2:

\[ t_1 = v_1 \quad t_2 \rightarrow t'_2 \quad t' = v_1 \ t'_2 \]

Similar.

\[
\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \\
\quad \Gamma \vdash t_1 \ t_2 : T_{12} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \qua
Application case

Case T-App:
\[ t = t_1 \cdot t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

Subcase E-AppAbs:
\[ t_1 = \lambda x: S_{11} \cdot t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2] \cdot t_{12} \]
by the inversion lemma for the typing relation ...
\[ T_{11} <: S_{11} \quad and \quad \Gamma, x : S_{11} \vdash t_{12} : T_{12} . \]
By using T-Sub, \( \Gamma \vdash t_2 : S_{11} \).
by the substitution lemma, \( \Gamma \vdash t' : T_{12} . \)

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

\[ (\lambda x : T_{11} \cdot t_{12}) \quad v_2 \longrightarrow [x \mapsto v_2] \cdot t_{12} \quad \text{(E-AppAbs)} \]
Inversion Lemma for Typing

Lemma (15.3.3): If $\Gamma \vdash \lambda x: S_1. s_2: T_1 \rightarrow T_2$, then $T_1 <: S_1$ and $\Gamma, x: S_1 \vdash s_2: T_2$.

Proof: Induction on typing derivations.

Case T–Sub: $\lambda x: S_1. s_2: U$ $U: T_1 \rightarrow T_2$

We want to say “By the induction hypothesis...”, but the IH does not apply (since we do not know that $U$ is an arrow type).

Need another lemma...

Lemma (15.3.2): If $U <: T_1 \rightarrow T_2$, then $U$ has the form of $U_1 \rightarrow U_2$,

with $T_1 <: U_1$ and $U_2 <: T_2$.

(Proof: by induction on subtyping derivations.)
Inversion Lemma for Typing

By this lemma, we know
\[ U = U_1 \rightarrow U_2, \text{ with } T_1 <: U_1 \text{ and } U_2 <: T_2. \]

The IH now applies, yielding
\[ U_1 <: S_1 \text{ and } \Gamma, x: S_1 \vdash s_2: U_2. \]

From \( U_1 <: S_1 \) and \( T_1 <: U_1 \), rule \textbf{S-Trans} gives
\[ T_1 <: S_1. \]

From \( \Gamma, x: S_1 \vdash s_2: U_2 \) and \( U_2 <: T_2 \), rule \textbf{T-Sub} gives
\[ \Gamma, x: S_1 \vdash s_2: T_2, \]
and we are done.
Theorem: If $t$ is a closed, well-typed term, then either $t$ is a value or else there is some $t'$, with and $t \rightarrow t'$

Proof: By induction on typing derivations.

Which cases are likely to be hard?

- case T-APP
- case T-RCD
- case T-PROJ
- case T-SUB
Subtyping

with

Other Features
Ascription and Casting

Ordinary ascription:

\[
\Gamma \vdash t_1 : T \\
\Gamma \vdash t_1 \text{ as } T : T
\]

\[
v_1 \text{ as } T \rightarrow v_1
\]

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

\[
(E-\text{ASCRIBE})
\]
Ascription and Casting

Ordinary ascription:

\[ \Gamma \vdash t_1 : T \]
\[ \frac{}{\Gamma \vdash t_1 \text{ as } T : T} \quad (\text{T-ASCRIBE}) \]
\[ v_1 \text{ as } T \rightarrow v_1 \quad (\text{E-ASCRIBE}) \]

Casting (cf. Java):

\[ \Gamma \vdash t_1 : S \]
\[ \frac{}{\Gamma \vdash t_1 \text{ as } T : T} \quad (\text{T-CAST}) \]
\[ \vdash v_1 : T \]
\[ \frac{}{v_1 \text{ as } T \rightarrow v_1} \quad (\text{E-CAST}) \]
Subtyping and Variants

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

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

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

\[
\Gamma \vdash t_1 : T_1 \\
\Gamma \vdash \langle l_1 = t_1 \rangle : \langle l_1 : T_1 \rangle \\
\text{(T-VARIANT)}
\]
Subtyping and Lists

\[
\frac{S_1 <: T_1}{\text{List } S_1 <: \text{List } T_1} \quad (S\text{-List})
\]

i.e., List is a covariant type constructor.
Subtyping and References

\[
\begin{align*}
S_1 & <: T_1 & T_1 & <: S_1 \\
\text{Ref } S_1 & <: \text{Ref } T_1
\end{align*}
\]

(S-REF)

i.e., \textbf{Ref} is not a \textit{covariant} (nor a \textit{contravariant}) type constructor.
Subtyping and References

\[
\begin{align*}
S_1 <: T_1 & \quad T_1 <: S_1 \\
\text{Ref } S_1 <: \text{Ref } T_1
\end{align*}
\]

(S-REF)

i.e., \textbf{Ref} is not a \textit{covariant} (nor a \textit{contravariant}) type constructor.

Why?

– When a reference is \textit{read}, the context expects a \(T_1\), so if \(S_1 <: T_1\) then an \(S_1\) is ok.
Subtyping and References

\[ \frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1} \]  
(S-REF)

i.e., \text{Ref} is not a \textit{covariant} (nor a \textit{contravariant}) type constructor.

Why?

– When a reference is \textit{read}, the context expects a \( T_1 \), so if \( S_1 <: T_1 \) then an \( S_1 \) is ok.

– When a reference is \textit{written}, the context provides a \( T_1 \) and if the actual type of the reference is \text{Ref } S_1, someone else may use the \( T_1 \) as an \( S_1 \). So we need \( T_1 <: S_1 \).
Observation: a value of type $\text{Ref}\ T$ can be used in two different ways: as a *source* for values of type $T$ and as a *sink* for values of type $T$.

Idea: Split $\text{Ref}\ T$ into three parts:

- **Source** $T$: reference cell with “read capability”
- **Sink** $T$: reference cell with “write capability”
- **Ref** $T$: cell with both capabilities
Subtyping and Arrays

Similarly...

\[
\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Array } S_1 <: \text{ Array } T_1} \quad (S-\text{ARRAY})
\]

\[
\frac{S_1 <: T_1}{\text{Array } S_1 <: \text{ Array } T_1} \quad (S-\text{ARRAY.JAVA})
\]

This is regarded (even by the Java designers) as a mistake in the design.
Observation: a value of type $\text{Ref } T$ can be used in two different ways:

- as a source for values of type $T$, and
- as a sink for values of type $T$. 
Observation: a value of type \textit{Ref} \textit{T} can be used in two different ways:

- as a \textit{source} for values of type \textit{T}, and
- as a \textit{sink} for values of type \textit{T}.

Idea: Split \textit{Ref} \textit{T} into three parts:

- \textbf{Source} \textit{T}: reference cell with “read capability”
- \textbf{Sink} \textit{T}: reference cell with “write capability”
- \textbf{Ref} \textit{T}: cell with both capabilities
Modified Typing Rules

\[
\frac{\Gamma | \Sigma \vdash t_1 : \text{Source } T_{11}}{\Gamma | \Sigma \vdash !t_1 : T_{11}} \quad \text{(T-DEREF)}
\]

\[
\frac{\Gamma | \Sigma \vdash t_1 : \text{Sink } T_{11} \quad \Gamma | \Sigma \vdash t_2 : T_{11}}{\Gamma | \Sigma \vdash t_1 := t_2 : \text{Unit}} \quad \text{(T-ASSIGN)}
\]
Subtyping rules

\[
\begin{align*}
S_1 &<: T_1 \\
\text{Source } S_1 &<: \text{Source } T_1
\end{align*}
\]

\[
\begin{align*}
T_1 &<: S_1 \\
\text{Sink } S_1 &<: \text{Sink } T_1
\end{align*}
\]

\[
\begin{align*}
\text{Ref } T_1 &<: \text{Source } T_1
\end{align*}
\]

\[
\begin{align*}
\text{Ref } T_1 &<: \text{Sink } T_1
\end{align*}
\]

\[
\begin{align*}
(S-\text{SOURCE})
\end{align*}
\]

\[
\begin{align*}
(S-\text{SINK})
\end{align*}
\]

\[
\begin{align*}
(S-\text{REFSOURCE})
\end{align*}
\]

\[
\begin{align*}
(S-\text{REFSINK})
\end{align*}
\]
Capabilities

Other kinds of capabilities can be treated similarly, e.g.,
- send and receive capabilities on communication channels,
- encrypt/decrypt capabilities of cryptographic keys,
- ...
Intersection and Union Types
Intersection Types

The inhabitants of $T_1 \land T_2$ are terms belonging to both $S$ and $T$ — i.e., $T_1 \land T_2$ is an order-theoretic meet (greatest lower bound) of $T_1$ and $T_2$.

\[
T_1 \land T_2 \leq: T_1 \quad \text{(S-INTER1)}
\]

\[
T_1 \land T_2 \leq: T_2 \quad \text{(S-INTER2)}
\]

\[
S \leq: T_1 \quad S \leq: T_2 \quad \therefore S \leq: T_1 \land T_2 \quad \text{(S-INTER3)}
\]

\[
S \rightarrow T_1 \land S \rightarrow T_2 \leq: S \rightarrow (T_1 \land T_2) \quad \text{(S-INTER4)}
\]
Intersection Types

Intersection types permit a very flexible form of finitary overloading.

\[ + : (\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}) \land (\text{Float} \rightarrow \text{Float} \rightarrow \text{Float}) \]

This form of overloading is extremely powerful.

Every strongly normalizing untyped lambda-term can be typed in the simply typed lambda-calculus with intersection types.

Type reconstruction problem is undecidable.

Intersection types have not been used much in language designs (too powerful!), but are being intensively investigated as type systems for intermediate languages in highly optimizing compilers (cf. Church project).
Union types

Union types are also useful.

\( T_1 \lor T_2 \) is an untagged (non-disjoint) union of \( T_1 \) and \( T_2 \).

No tags: no case construct. The only operations we can safely perform on elements of \( T_1 \lor T_2 \) are ones that make sense for both \( T_1 \) and \( T_2 \).

N. B: untagged union types in C are a source of type safety violations precisely because they ignores this restriction, allowing any operation on an element of \( T_1 \lor T_2 \) that makes sense for either \( T_1 \) or \( T_2 \).

Union types are being used recently in type systems for XML processing languages (cf. Xduce, Xtatic).
Varieties of Polymorphism

- Parametric polymorphism (ML-style)
- Subtype polymorphism (OO-style)
- Ad-hoc polymorphism (overloading)
Chap 16

Metatheory of Subtyping

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

\[
S <: S
\]

\[
S <: U \quad U <: T
\]

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

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

\[
\text{for each } i \quad S_i <: T_i
\]

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

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

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

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

\[
S_1 \rightarrow S_2 <: T_1 \rightarrow T_2
\]  \hspace{1cm} (S-\text{ARROW})

\[
S <: \text{Top}
\]  \hspace{1cm} (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.

\[ \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \]
\[ \Gamma \vdash t_1 \ t_2 : T_{12} \]  

(T-APP)
Syntax-directed rules

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

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

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)

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

⇒ 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 $T \rightarrow_{\text{SUB}}$)

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

2. Worse yet, the new rule $T \rightarrow_{\text{SUB}}$ 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 $T \rightarrow_{\text{SUB}}$ 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

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

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

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

(S-RCD)
Simpler subtype relation

\[ S <: S \]  
\[ S <: U \quad U <: T \quad \frac{S <: T}{S <: T} \]  
\[ \{ l_i : i \in \{1..n\} \} \subseteq \{ k_j : j \in \{1..m\} \} \quad k_j = l_i \text{ implies } S_j <: T_i \]  
\[ \{ k_j : S_j : j \in \{1..m\} \} <: \{ l_i : T_i : i \in \{1..n\} \} \]  
\[ T_1 <: S_1 \quad S_2 <: T_2 \quad \frac{T_1 \rightarrow S_2 <: T_1 \rightarrow T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \]  
\[ S <: \text{Top} \]  
\[ (S\text{-Refl}) \]  
\[ (S\text{-Trans}) \]  
\[ (S\text{-Rcd}) \]  
\[ (S\text{-Arrow}) \]  
\[ (S\text{-Top}) \]
Step 2: Get rid of reflexivity

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

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

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

(S-TRANS)

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

(S-RCD)

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

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

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

(S-ARROW)

(S-TOP)

S \ll \text{Top}
Step 3: Get rid of transitivity

*Observation*: $S$-Trans is unnecessary.

*Lemma*: If $S \preceq T$ can be derived, then it can be derived without using $S$-Trans.
“Algorithmic” subtype relation

\[
\begin{align*}
\vdash S & \ll S \ll Top \\
\vdash T_1 & \ll S_1 & \vdash S_2 & \ll T_2 \\
\vdash S_1 \rightarrow S_2 & \ll T_1 \rightarrow T_2
\end{align*}
\]

\[
\begin{align*}
\{ l_i \mid i \in \{1..n\} \} & \subseteq \{ k_j \mid j \in \{1..m\} \} \\
\text{for each } k_j & = l_i, \quad \vdash S_j \ll T_j \\
\vdash \{ k_j : S_j \mid j \in \{1..m\} \} & \ll \{ l_i : T_i \mid i \in \{1..n\} \}
\end{align*}
\]

\begin{align*}
\text{(SA-TOP)} \\
\text{(SA-ARROW)} \\
\text{(SA-RCD)}
\end{align*}
Soundness and completeness

Theorem: \[ S <: T \iff \mapsto S <: T \]

Terminology:

- The algorithmic presentation of subtyping is *sound* with respect to the original if \[ \mapsto S <: T \] 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 \[ \mapsto S <: T \]. (Everything true is validated by the algorithm.)
Decision Procedures

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 $\{\text{true, false}\}$ such that $p(u) = \text{true}$ iff $u \in R$.

Is our subtype function a decision procedure?
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 \( S <: T \) (hence, by soundness of the algorithmic rules, \( S <: T \))
2. if \( subtype(S, T) = false \), then not \( S <: T \) (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 $\rightarrow S <: T$ (hence, by soundness of the algorithmic rules, $S <: T$)

2. if $subtype(S, T) = false$, then not $\rightarrow S <: T$ (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 \( subtype(S, T) = true \), 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?
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 \( \leftrightarrow S <: T \) (hence, by soundness of the algorithmic rules, \( S <: T \))
2. if \( \text{subtype}(S, T) = false \), then not \( \leftrightarrow S <: T \) (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!
Decision Procedures

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

Example:

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

Note that, we are saying nothing about computability.
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 $R$. 
Recall: A *decision procedure* for a relation $R \subseteq U$ is a *total function* $p$ from $U$ to \{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 $R$. 
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

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

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 $\{\text{true, false}\}$ such that $p(u) = \text{true}$ iff $u \in R$. 
Example

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

The function

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

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**

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

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

The recursively defined partial function

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

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:

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

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

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

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 $\Gamma \vdash t : T$?
Issue

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

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

(T-Sub)

Q: where is this rule really needed?
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.
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??
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 this exercise 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)

\[\vdash \Gamma, x : S_1 \vdash s_2 : S_2 \quad S_2 \ll T_2 \]
\[\Gamma, x : S_1 \vdash s_2 : T_2 \quad (\text{T-Sub})\]
\[\Gamma \vdash \lambda x : S_1 . s_2 : S_1 \rightarrow T_2 \quad (\text{T-Abs})\]
Example (T-ABS)

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

becomes

\[
\begin{align*}
\Gamma, x : S_1 & \vdash s_2 : S_2 \\
\hline
\Gamma & \vdash \lambda x : S_1 . s_2 : S_1 \rightarrow S_2 \\
\hline
S_1 & \vdash S_2 : S_1 \rightarrow S_2 \\
\hline
S_1 \rightarrow S_2 & \vdash S_1 \rightarrow T_2 \\
\hline
\Gamma & \vdash \lambda x : S_1 . s_2 : S_1 \rightarrow T_2 \\
\end{align*}
\]
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 \((T-\text{Sub} \text{ with } T-\text{APP} \text{ on the left})\)

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

becomes

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

\[
\begin{align*}
&\vdash s_1 : T_{11} \rightarrow T_{12} \\
&\vdash s_2 : T_2 \\
&\vdash s_2 : T_{11} \\
&\vdash s_1 \ s_2 : T_{12}
\end{align*}
\]  

becomes

\[
\begin{align*}
&\vdash s_1 : T_{11} \rightarrow T_{12} \\
&T_2 \ll T_{11} \\
&T_{12} \ll T_{12} \\
&T_{11} \rightarrow T_{12} \ll T_2 \rightarrow T_{12} \\
&\vdash s_1 \vdash s_2 : T_2 \\
&\vdash s_1 \ s_2 : T_{12}
\end{align*}
\]
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 cannot be completely eliminated.
Example (nested uses of T-Sub)

\[
\vdash s : S \quad S \ll U \\
___________________________^{(T\text{-SUB)}} \\
\vdash s : U \\
\vdash s : T
\]
Example (nested uses of T-Sub)

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

becomes

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

\[ \frac{x: T \in \Gamma}{\Gamma \vdash x : T} \]  
\( (TA-VAR) \)

\[ \frac{\Gamma, x: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x: T_1. t_2 : T_1 \rightarrow T_2} \]  
\( (TA-ABS) \)

\[ \frac{\Gamma \vdash t_1 : T_1 \quad T_1 = T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_2 \quad \vdash T_2 \leq T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \]  
\( (TA-APP) \)

\[ \text{for each } i \quad \frac{\Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_1=t_1 \ldots l_n=t_n\} : \{l_1:T_1 \ldots l_n:T_n\}} \]  
\( (TA-RCD) \)

\[ \frac{\Gamma \vdash t_1 : R_1 \quad R_1 = \{l_1:T_1 \ldots l_n:T_n\}}{\Gamma \vdash t_1. l_i : T_i} \]  
\( (TA-PROJ) \)
Completeness of the algorithmic rules

**Theorem [Minimal Typing]:** If $\Gamma \vdash t : T$, then $\Gamma \Rightarrow t : S$ 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.

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

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

What is the minimal type of

\[
\begin{align*}
\text{if } true \text{ then } & \{x = true, y = false\} \text{ else } \{x = true, z = true\} \\
\end{align*}
\]
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 \( t_3 \).

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

\[
\frac{
\Gamma \mid \! t_1 : \text{Bool} \quad \Gamma \mid \! t_2 : T_2 \quad \Gamma \mid \! t_3 : T_3
}{
\Gamma \mid \! \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \lor T_3
}
\]

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

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

\[
\begin{align*}
S_1 \land T_1 &= M_1 \\
S_2 \lor T_2 &= J_2 \\
\{ j_l \mid l \in 1..q \} &= \{ k_j \mid j \in 1..m \} \cap \{ l_i \mid i \in 1..n \} \\
S_j \lor T_i &= J_l & \text{for each } j_l = k_j = l_i
\end{align*}
\]
Calculating Meets

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

• Read and digest chapter 16 & 17

• HW#1: 16.2.5

• HW#2: Exercises on Slide p107 & P 111