Recap on Subtype
Rule of Subsumption

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

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 \( T \)

*a value of one type* can always safely be used where *a value of the other is expected*. 
Subtype Relation

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

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

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

for each \(i\)

\[ S_i <: T_i \]

\[ \{l_i:S_i \mid i \in \{1, \ldots, n\}\} <: \{l_i:T_i \mid i \in \{1, \ldots, n\}\} \quad \text{(S-RcdDepth)} \]

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

\[ \{k_j:S_j \mid j \in \{1, \ldots, n\}\} <: \{l_i:T_i \mid i \in \{1, \ldots, n\}\} \quad \text{(S-RcdPerm)} \]

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

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

\[ S <: \text{Top} \quad \text{(S-Top)} \]
Properties of Subtyping
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.

i.e., 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

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$
**An Inversion Lemma for subtyping**

**Lemma:** If $U <: T_1 \rightarrow T_2$, then $U$ has the form $U_1 \rightarrow U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.

**Proof:** *By induction on subtyping derivations*

**Case S-Arrow:**

$U = U_1 \rightarrow U_2 \quad T_1 <: U_1 \quad U_2 <: T_2$

Immediate.

**Case S-Refl:**

$U = T_1 \rightarrow T_2$

By S-Refl (twice), $T_1 <: T_1$ and $T_2 <: T_2$, as required

**Case S-Trans:**

$U <: W \quad W <: T_1 \rightarrow T_2$

1. Applying the IH to the second subderivation, we find that $W$ has the form $W_1 \rightarrow W_2$, with $T_1 <: W_1$ and $W_2 <: T_2$.

2. Now the IH applies again (to the first subderivation), telling us that $U$ has the form $U_1 \rightarrow U_2$, with $W_1 <: U_1$ and $U_2 <: W_2$.

3. By S-Trans, $T_1 <: U_1$, and, by S-Trans again, $U_2 <: T_2$, as required.
Inversion Lemma for Typing

**Lemma:** 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-ABS:**

$T_1 = S_1 \quad T_2 = S_2 \quad \Gamma, x : S_1 \vdash s_2 : S_2$

**Case T-SUB:**

$\Gamma \vdash \lambda x : S_1 . s_2 : U \quad U : T_1 \rightarrow T_2$

— By the subtyping inversion lemma, $U_1 \rightarrow U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.

— The IH now applies, yielding $U_1 <: S_1$ and $\Gamma, x : S_1 \vdash s_2 : U_2$.

— From $U_1 <: S_1$ and $T_1 <: U_1$, rule S-Trans gives $T_1 <: S_1$.

— From $\Gamma, x : S_1 \vdash s_2 : U_2$ and $U_2 <: T_2$, rule T-Sub gives $\Gamma, x : S_1 \vdash s_2 : T_2$, thus we are done
Preservation

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

Proof: induction on typing derivations.

Which cases are likely to be hard?
Preservation - 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!
Case T-App:

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

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

\textit{E-APP1}, \textit{E-APP2}, and \textit{E-APPABS}.

Proceed by cases.
Preservation - Application case

Case T-App:

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

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

E-APP1, E-APP2, and E-APPABS

Proceed by cases.

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

Case T-App:

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

Subcase E-App1:

\[
t_1 \rightarrow t'_1 \quad t' = t'_1 \ t_2
\]

The result follows from the induction hypothesis and T-APP

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

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

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

Subcase E-App2:
\[ t_1 = v_1 \quad t_2 \rightarrow t'_2 \quad t' = v_1 \ t'_2 \]
Similar.

\[ \frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \]  \hspace{1cm} \text{(T-APP)}

\[ \frac{t_2 \rightarrow t'_2}{v_1 \ t_2 \rightarrow v_1 \ t'_2} \]  \hspace{1cm} \text{(E-APP2)}
Preservation - Application case

**Subcase** E-AppAbs:

\[ t_1 = \lambda x: S_{11}. t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2] t_{12} \]

by the *inversion lemma* for the typing relation ...

\[ T_{11} <: S_{11} \quad \text{and} \quad \Gamma, x: S_{11} \vdash t_{12}: T_{12} \]

By using T-Sub, \( \Gamma \vdash t_2: S_{11} \)

by the *substitution lemma*, \( \Gamma \vdash t': T_{12} \)

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

(T-App)

\[
(\lambda x: T_{11}. t_{12}) \ v_2 \rightarrow [x \mapsto v_2] t_{12}
\]

(E-AppAbs)
Progress

Lemma for Canonical Forms

1. If \( v \) is a closed value of type \( T_1 \rightarrow T_2 \), then \( v \) has the form \( \lambda x: S_1 . t_2 \).

2. If \( v \) is a closed value of type \( \{ l_i : T_i \} \), then \( v \) has the form \( \{ k_j = v_j \} \) with \( \{ l_i \} \subseteq \{ k_a \} \)

Possible shapes of values belonging to arrow and record types.

Based on this Canonical Forms Lemma, we can still has the progress theorem and its proof quite close to what we saw in the simply typed lambda-calculus
Subtyping with Other Features
Ascription and Casting

Ordinary ascription:

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

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

(T) T
up-cast
down-cast
Ascription and Casting

Ordinary ascription:

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

\[
v_1 \text{ as } T \rightarrow v_1 \quad \text{(E-ASCRIBE)}
\]

Casting (cf. Java):

\[
\frac{\Gamma \vdash t_1 : S}{\Gamma \vdash t_1 \text{ as } T : T} \quad \text{(T-CAST)}
\]

\[
\vdash v_1 : T \quad \text{(E-CAST)}
\]
Subtyping and Variants

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

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

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

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

\[ (S\text{-REF}) \]

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

i.e., Ref is not a covariant (nor a contravariant) type constructor.

Why?

– When a reference is read, the context expects a $T_1$, so if $S_1 <: T_1$ then an $S_1$ is ok.

– When a reference is written, the context provides a $T_1$ and if the actual type of the reference is 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 $Ref\ T$ 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 $\text{Ref } T$ 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
Modified Typing Rules

\[
\Gamma \mid \Sigma \vdash t_1 : \text{Source } T_{11} \\
\Gamma \mid \Sigma \vdash !t_1 : T_{11}
\]  \hspace{2cm} (T-DEREF)

\[
\Gamma \mid \Sigma \vdash t_1 : \text{Sink } T_{11} \quad \Gamma \mid \Sigma \vdash t_2 : T_{11} \\
\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}
\]  \hspace{2cm} (T-ASSIGN)
Subtyping rules

\[ S_1 <: T_1 \]

\[ \text{Source } S_1 <: \text{Source } T_1 \]

\[ T_1 <: S_1 \]

\[ \text{Sink } S_1 <: \text{Sink } T_1 \]

\[ \text{Ref } T_1 <: \text{Source } T_1 \]

\[ \text{Ref } T_1 <: \text{Sink } T_1 \]

\[ (S-\text{SOURCE}) \]

\[ (S-\text{SINK}) \]

\[ (S-\text{REFSOURCE}) \]

\[ (S-\text{REFSINK}) \]
Similarly...

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

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

This is regarded (even by the Java designers) as a mistake in the design.
Capabilities

Other kinds of capabilities can be treated similarly, e.g.,

- *send* and *receive* capabilities on communication channels,
- *encrypt/decrypt* capabilities of cryptographic keys,
- ...
Base Types

For language with a rich set of base types, it’s better to introduce primitive subtype relations among them

- e.g., Bool <: Nat
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 (\textit{greatest lower bound}) of \( T_1 \) and \( T_2 \)

\[
T_1 \land T_2 \leq T_1
\]

\( (S\text{-}\text{INTER1}) \)

\[
T_1 \land T_2 \leq T_2
\]

\( (S\text{-}\text{INTER2}) \)

\[
S \leq T_1 \quad S \leq T_2
\]

\[
\underbrace{S \leq T_1 \land T_2}
\]

\( (S\text{-}\text{INTER3}) \)

\[
S \rightarrow T_1 \land S \rightarrow T_2 \leq S \rightarrow (T_1 \land T_2)
\]

\( (S\text{-}\text{INTER4}) \)
Intersection Types

Intersection types permit a very flexible form of finitary overloading, e.g., S-Inter4:

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

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) ordinary union of the set of values belonging to $T_1$ and that of values belonging to $T_2$.

*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 $T_2$.

Note well: 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)
Issues in Subtyping
Typing with Subsumption

Principle of safe substitution

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

a value of one can always safely be used where a value of the other is expected

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 \(T\), i.e.,
A subtyping is a *binary relation* between *types* that is closed under the following rules:

- **(S-REFL)**: \( S <: S \)
- **(S-TRANS)**: \( S <: U \quad U <: T \quad \therefore S <: T \)
- **(S-TOP)**: \( S <: \text{Top} \)
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.
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)

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)

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

- 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

\[ \Rightarrow \text{ no backtracking!} \]
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 two rules that can be used to give a type to terms of a given shape (the old one + \( T \rightarrow \text{SUB} \))

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

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 translate 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 (∵ 8.2.4 /9.3.3 [uniqueness of types] ×)
2. The transitivity rule

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

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?

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*.
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.
Chap 16
Metatheory of Subtyping

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

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

\[ S <: S \]  \hspace{2cm} (S-REFL)

\[ S <: U \quad U <: T \quad \frac{}{S <: T} \]  \hspace{2cm} (S-TRANS)

\[ \{l_i \mid i \in 1..n\} \subseteq \{k_j \mid j \in 1..m\} \quad k_j = l_i \text{ implies } S_j <: T_i \]  \hspace{2cm} (S-RCD)

\[ \{k_j : S_j \mid j \in 1..m\} <: \{l_i : T_i \mid i \in 1..n\} \]  \hspace{2cm} (S-RCD)

\[ T_1 <: S_1 \quad S_2 <: T_2 \quad \frac{}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \]  \hspace{2cm} (S-ARROW)

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

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

Lemma 16.1.2: \( S <: S \) can be derived for every type \( S \) without using \( S\text{-REFL} \).
Even simpler subtype relation

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

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

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

\[
S \ll \text{Top}
\] (S-TOP)
Step 3: Get rid of transitivity

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

*Lemma 16.1.2*: If $S <: T$ can be derived, then it can be derived without using $S$-Trans.
Even simpler subtype relation

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

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

\( T_1 \ll S_1 \quad S_2 \ll T_2 \)

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

\( S \ll \text{Top} \)

\((S-\text{RCD})\)

\((S-\text{ARROW})\)

\((S-\text{TOP})\)
"Algorithmic" subtype relation

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

\[
\{l_i, i \in 1..n\} \subseteq \{k_j, j \in 1..m\} \quad \text{for each } k_j = l_i, \quad \vdash S_j &: T_i
\]

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

**Theorem[16.1.5]:** \( S <: T \) 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)*
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 \texttt{subtype} function a decision procedure?

\texttt{subtype} is just an implementation of the algorithmic subtyping rules, we have

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

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

Q: What’s missing?
Decision Procedures

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 \( S <: T \)
   (hence, by *soundness* of the algorithmic rules, \( S <: T \))

1. if \( \text{subtype}(S, T) = \text{false} \), 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*?
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, \(S <: T\))

1. if `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`?

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.
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

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

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

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

is also 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

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

is a decision function for $R$
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

$$p(u) = *true* \text{ iff } u \in R.$$
Example

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

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

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

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

whose graph is

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

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

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

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

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

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

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*

[16.1.6 Termination Proposition]
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 \textit{just one problematic rule} to deal with: \textit{subsumption rule}

\[
\begin{array}{c}
\Gamma \vdash t : S \\
S <: T \\
\hline
\Gamma \vdash t : T
\end{array}
\]  \hspace{1cm} (T\text{-}SUB)

Q: where is this rule really needed?

For \textit{applications}, e.g., the term

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

is \textit{not typable} without using subsumption.

Where else??

\textbf{Nowhere else!}

Uses of subsumption rule to help typecheck \textit{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)

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

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

(becomes)

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

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

\[
\begin{array}{c}
S_1 \ll S_1 \\
\hline
S_2 \ll T_2
\end{array}
\]

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

(S-REFL)  

(S-A Arrow)  

(T-S SUB)
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
Therefore 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 \((T-\text{Sub} \text{ with } T-\text{APP} \text{ on the left})\)

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

becomes

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

\[
\begin{array}{c}
\Gamma \vdash s_1 : T_{11} \rightarrow T_{12} \\
\Gamma \vdash s_2 : T_2 \\
\end{array}
\quad
\begin{array}{c}
T_2 \ll T_{11} \\
\Gamma \vdash s_2 : T_{11} \\
\end{array}
\]

\[
\quad \frac{\Gamma \vdash s_2 : T_2}{\Gamma \vdash s_2 : T_{12}} \quad \text{(T-APP)}
\]

becomes

\[
\begin{array}{c}
\Gamma \vdash s_1 : T_{11} \rightarrow T_{12} \\
\Gamma \vdash s_2 : T_2 \\
\end{array}
\quad
\begin{array}{c}
T_2 \ll T_{11} \\
T_{12} \ll T_{12} \\
\end{array}
\]

\[
\quad \frac{T_{11} \rightarrow T_{12} \ll T_2 \rightarrow T_{12}}{\Gamma \vdash s_1 \ s_2 : T_{12}} \quad \text{(T-APP)}
\]

\[
\quad \frac{\Gamma \vdash s_1 : T_2 \rightarrow T_{12}}{\Gamma \vdash s_1 \ s_2 : T_{12}} \quad \text{(T-APP)}
\]
Observations

We’ve seen that uses of subsumption rule 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)

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

becomes

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

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 rule in application rules, by changing the T-App rule to incorporate a subtyping premise

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

Given any typing derivation, we can now

1. normalize it, to move all uses of subsumption rule 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

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

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

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

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

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

Theorem [Minimal Typing]:

If $\Gamma \vdash t : T$, then $\Gamma \rightarrow t : S$ 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*.

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

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

What is the minimal type of

\[
\text{if true then } \{x = \text{true}, y = \text{false}\} \text{ else } \{x = \text{true}, z = \text{true}\}?
\]
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 \( t_3 \).

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

\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?
Calculating Joins

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

\[
S \land T = \begin{cases} 
  \text{Top} & \text{if } S = T = \text{Top} \\
  M_1 \rightarrow J_2 & \text{if } S = S_1 \rightarrow S_2 \quad T = T_1 \rightarrow T_2 \\
  \{ j_i : J_i \}_{i \in 1..q} & \text{if } S = \{ k_j : S_j \}_{j \in 1..m} \quad T = \{ l_i : T_i \}_{i \in 1..n} \\
  \text{Top} & \text{otherwise}
\end{cases}
\]
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!
Existence of Meets

**Theorem:** For every pair of types $S$ and $T$, we say that a type $M$ is a meet of $S$ and $T$, written $S \land T = M$ if

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*
Calculating Meets

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

• Read and digest chapter 16 & 17

• HW: 16.1.2; 16.2.6; 16.4.1