In this class 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**

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 $t_2$.

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

**16.1 Algorithmic Subtyping**

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.
Three rules for records

{\ell_i : T^{i \in 1 \ldots n+k}} \prec \{\ell_i : T^{i \in 1 \ldots n}\} \quad \text{(S-RcdWidth)}

for each \( i \) \( S_i \prec T_i \)

{\ell_i : S^{i \in 1 \ldots n}} \prec \{\ell_i : T^{i \in 1 \ldots n}\} \quad \text{(S-RcdDepth)}

\{k_j : S^{j \in 1 \ldots m}\} \text{ is a permutation of } \{\ell_i : T^{i \in 1 \ldots n}\}

\{k_j : S^{j \in 1 \ldots m}\} \prec \{\ell_i : T^{i \in 1 \ldots n}\} \quad \text{(S-RcdPerm)}

Which rule do we use to decide if \{k_j : S^{j \in 1 \ldots m}\} \prec \{\ell_i : T^{i \in 1 \ldots n}\}?

All-in-one

We can replace these three rules by a single rule that does permutation, width and depth subtyping all at once.

\{\ell_i^{1 \ldots n}\} \subseteq \{k_j^{j \in 1 \ldots m}\}

\( k_j = l_i \) implies \( S_j \prec T_i \)

\{k_j : S^{j \in 1 \ldots m}\} \prec \{\ell_i : T^{i \in 1 \ldots n}\} \quad \text{(S-Rcd)}

Lemma 1 If \( S < : T \) is provable with the 3 separate rules for width, depth and permutation, it is provable with just S-Rcd.

Other rules

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

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

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

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

What other rules cause difficulty?

Non-syntax-directedness of subtyping

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

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

- Both reflexivity and transitivity apply to many inputs, regardless of their syntactic form.
- Furthermore, transitivity requires that we “guess” a value for \( U \). This metavariable appears in the premises of the rule (in an input position) but not in the conclusion.
**Algorithmic Subtyping Rules**

\[ \begin{align*}
\text{SA-Top} & : S : T & \quad \text{(SA-Top)} \\
\text{SA-Arrow} & : S_1 : T_1 \rightarrow S_2 : T_2 & \quad \text{(SA-Arrow)} \\
\text{SA-Rcd} & : \{k_j \in 1 \ldots m\} \subseteq \{l_i \in 1 \ldots n\} & \quad \text{(SA-Rcd)}
\end{align*} \]

**Algorithm**

\[
\text{subtype}(S,T) = \\
\begin{cases} 
\text{true} & \text{if } T = \text{Top} \\
\text{false} & \text{else if } S = S_1 \rightarrow S_2 \text{ and } T = T_1 \rightarrow T_2 \\
\text{subtype}(S_1, S_2) \land \text{subtype}(T_1, T_2) & \text{else if } S = \{k_j : S_j \in 1 \ldots m\} \text{ and } T = \{l_i : T_i \in 1 \ldots n\} \\
\text{false} & \text{else}
\end{cases}
\]

\[
\begin{align*}
&\text{then } \{l_i \in 1 \ldots n\} \subseteq \{k_j \in 1 \ldots m\} \\
&\text{and for all } i \text{ there is some } \\
&\quad j \in 1 \ldots m \text{ with } k_j = l_i
\end{align*}
\]

Does this pseudocode match the algorithmic rules? Is this function total?
**Why declarative rules?**

Why not use the algorithmic rules as the “official definition” of subtyping?

This does not save us much work as we still need to know that the rules are reflexive and transitive when we use them.

---

**16.2 Algorithmic Typing**

The subsumption rule is not syntax directed:

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

(T-SUB)

1. This rule could be used for any \( t \).

2. We have to guess \( T \) in the subgoal \( S << T \).

---

**Where is subsumption used?**

We can’t just get rid of it. A term like

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

is not typeable without subsumption.

\[
\vdash \lambda r. r.x : \{x:N\} \rightarrow N \\
\vdash \{x = 0, y = 1\} : \{x:N, y:N\} \\
\vdash \{x:N, y:N\} << \{x:N\} \\
\vdash (\lambda r. r.x)\{x = 0, y = 1\} : N
\]

(S-Sub)

Do we need subsumption anywhere else?

---

**Combine T-Sub with T-App**

- Application is the only situation where subsumption is important.
- Why? It is the only rule where two types must match.
- Every other use of subsumption can be “postponed”. If we use subsumption before any other rule, we can always rewrite the derivation so that subsumption is used after that rule.
- Therefore, we can incorporate subsumption with the application rule, and not lose any expressiveness.
**Normalized Derivation**

Rewrite any derivation of $\Gamma \vdash t : T$ into a special form where $T$-Sub appears in only two places.

- Just before an application.
- At the very end of the derivation.

---

**Pushing T-Sub around**

If we have a $(T-Sub)$ before a use of $(T-Abs)$

$$\cdots \cdots \frac{\Gamma, x : S_1 \vdash s_2 : S_2 \quad S_2 <: T_2}{\Gamma, x : S_1 \vdash s_2 : T_2} \quad (T-Sub)$$

$$\frac{\Gamma \vdash \lambda x : S_1, s_2 : S_1 \rightarrow T_2}{\Gamma \vdash \lambda x : S_1, s_2 : S_1 \rightarrow T_2} \quad (T-Abs)$$

we can always switch the order of these rules with a little rearrangement.

$$\cdots \cdots \frac{\Gamma, x : S_1 \vdash s_2 : S_2 \quad S_2 <: T_2}{\Gamma \vdash \lambda x : S_1, s_2 : S_1 \rightarrow T_2} \quad (T-Abs)$$

$$\frac{\Gamma \vdash \lambda x : S_1, s_2 : S_1 \rightarrow T_2}{\Gamma \vdash \lambda x : S_1, s_2 : S_1 \rightarrow T_2} \quad (T-Sub)$$

---

**Other rules**

There are similar transformations for $T$-Sub followed by $T$-Rcd and for $T$-Sub followed by $T$-Proj.

---

**T-Sub followed by T-Sub?**

We can combine two uses of $(T-Sub)$ together.

$$\cdots \cdots \frac{\Gamma \vdash s : S \quad S <: U}{\Gamma \vdash s : U} \quad (T-Sub) \quad \frac{\Gamma \vdash s : U \quad U <: T}{\Gamma \vdash s : T} \quad (T-Sub)$$

$$\cdots \cdots \frac{\Gamma \vdash s : S \quad S <: U \quad U <: T}{\Gamma \vdash s : T} \quad (T-Sub)$$

(This is why $S <: T$ must be transitive.)
New Application Rule

\[
\begin{align*}
\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} & \quad \Gamma \vdash t_2 : T_2 \\
\Gamma \vdash \Gamma_\triangleleft T_{11} & \\
\Gamma \vdash t_1 t_2 : T_{12}
\end{align*}
\]

We can use subsumption for the type of the argument.

Replacing T-Sub with TA-App (RHS)

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

T-Sub before T-App (LHS)

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

\[
\begin{align*}
\Gamma \vdash t_1 : T_1 & \quad \Gamma \vdash t_2 : T_2 \\
\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2 & \quad (TA-VAR) \\
\Gamma \vdash \Gamma_\triangleleft T_1 & \\
\Gamma \vdash t_1 t_2 : T_{12} & \quad (T-APP) \\
\Gamma \vdash \Gamma_\triangleleft T_1 & \\
\Gamma \vdash t_2 : T_2 & \quad (TA-App) \\
\Gamma \vdash s_1 : S_{11} & \\
\Gamma \vdash s_2 : S_{11} & \\
\Gamma \vdash s_1 s_2 : T_{12} & \quad (T-Sub) \\
\Gamma \vdash s_1 s_2 : T_{12} & \quad (T-App) \\
\Gamma \vdash s_1 s_2 : T_{12} & \quad (T-Sub)
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t_i : T_i & \quad \Gamma \vdash s_1 : S_{11} & \quad \Gamma \vdash s_2 : S_{11} & \\
\Gamma \vdash \{ t_i : T_i \}_{i = 1}^{n} & \quad \{ t_i : T_i \}_{i = 1}^{n} & \quad \{ t_i : T_i \}_{i = 1}^{n} & \\
\Gamma \vdash \Gamma_\triangleleft t_i & \\
\Gamma \vdash t_i : T_i & \quad \Gamma \vdash s_1 s_2 : T_{12} & \quad (T-RCD) \\
\Gamma \vdash \Gamma_\triangleleft t_i & \\
\Gamma \vdash s_1 s_2 : T_{12} & \quad (T-PROJ) \\
\Gamma \vdash \Gamma_\triangleleft t_i & \\
\Gamma \vdash s_1 s_2 : T_{12} & \quad (T-PROJ)
\end{align*}
\]
Soundness and Completeness

As before, we need to argue that the algorithmic rules are sound and complete with respect to the declarative rules.

**Lemma 4 (Soundness)** If $\Gamma \vdash t : T$ then $\Gamma \vdash t : T$.

We can’t prove the straightforward converse of the above lemma because while the declarative rules could assign many types to $t$, the algorithmic rules assign only one. For completeness, we can prove that the algorithmic rules give us the smallest or (minimal) possible type.

**Lemma 5 (Completeness)** If $\Gamma \vdash t : T$ then $\Gamma \triangleright t : S$ for some $S < : T$.

Algorithmic Rule

Because the two branches must be the same type, we need to incorporate subsumption in an algorithmic version of this rule.

How?

$$
\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3 \quad T_2 <: T \quad T_3 <: T
\quad \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T
$$

However, with this rule, we may not produce the minimal type for the expression. We can choose any $T$ that is greater than $T_2$ and $T_3$.

We need to restrict this rule so that it produces the least such $T$. This $T$ is called the join of $T_2$ and $T_3$.

16.3 Joins and Meets

If we have conditionals or case expressions, we need additional machinery to support algorithmic subtyping.

$$
\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T
\quad \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T
$$

With subsumption, there may be many ways of giving the two branches the same type.

**Definition 6** A type $J$ is the join of two types $S$ and $T$ (and is written $J = S \lor T$) if

1. $S <: J$
2. $T <: J$
3. For all $U$, if $S <: U$ and $T <: U$ then $J <: U$.

A join is a generalization of a least upper bound.
Meets

Definition 7 A type $M$ is the meet of two types $S$ and $T$ (and is written $J = S \land T$) if

1. $M <: S$
2. $M <: J$
3. For all $U$, if $U <: S$ and $U <: T$ then $U <: M$.

A meet is a generalization of a greatest lower bound.

Note: do not confuse joins and meets with the intersection types and union types that we saw on Monday.

Existence of Joins and Meets

Given a subtype relation, it may or may not be the case that joins and meets exist for every pair of types.

- In fact, the subtype relation does not have meets. For example, there is no meet for the types $\{}$ and $\text{Top} \rightarrow \text{Top}$.

Proposition 8 (Joins Exist) For every pair of types $S$ and $T$, there is some type $J$ such that $S \lor T = J$.

Proposition 9 (Bounded Meets Exist) For every pair of types $S$ and $T$ with a common subtype, there is some type $M$ such that $S \land T = M$.

Algorithmic rule

Using join, we can give an algorithmic rules for \textit{if}.

\[
\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3 \quad T_2 \lor T_3 = T
\]

\[
\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T
\]

16.4 Algorithmic typing and Bottom

- $\text{Bot} <: T$ (SA-Bot)

\[
\Gamma \vdash t_1 : T_1 \quad T_1 = \text{Bot} \quad \Gamma \vdash t_2 : T_2
\]

\[
\Gamma \vdash t_1 t_2 : \text{Bot} \quad (\text{TA-AppBot})
\]

\[
\Gamma \vdash t_1 : R \quad R = \text{Bot}
\]

\[
\Gamma \vdash t_1 . l : \text{Bot} \quad (\text{TA-ProjBot})
\]

In a declarative system, we can apply something of type $\text{Bot}$ to an argument of absolutely any type (by using subsumption to promote the $\text{Bot}$ to whatever function type we like), and assume that the result has any other type.