

CIS 500, 6 November

1

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_1t_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_1t_2 , we need to derive a type for t_1 and a type for t_2 .

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad \text{(T-APP)}$$

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.

CIS 500, 6 November

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.

$$\begin{aligned} \frac{\text{THREE RULES FOR RECORDS}}{\{l_i: T_i^{i\in 1...n+k}\} <: \{l_i: T_i^{i\in 1...n}\} \quad (S-\text{RCDWIDTH})} \\ \frac{\text{for each } i \quad S_i <: T_i}{\{l_i: S_i^{i\in 1...n}\} <: \{l_i: T_i^{i\in 1...n}\}} \quad (S-\text{RCDDEPTH})} \\ \frac{\{k_j: S_j^{i\in 1...n}\} \text{ is a permutation of } \{l_i: T_i^{i\in 1...n}\}}{\{k_j: S_j^{i\in 1...n}\} <: \{l_i: T_i^{i\in 1...n}\}} \quad (S-\text{RCDPERM})} \\ \text{Which rule do we use to decide if } \{k_j: S_j^{j\in 1...m}\} <: \{l_i: T_i^{i\in 1...n}\} <: \{l_i: T_i^{i\in 1...n}\} <: \{l_i: T_i^{i\in 1...m}\} <: \{l_i: T_i^{i\in$$

 $\begin{array}{l} \hline \textbf{ALL-IN-ONE} \\ \hline \textbf{We can replace these three rules by a single rule that does permutation, width and depth subtyping all at once. \\ \{l_i^{i\in 1...n}\} \subseteq \{k_j^{j\in 1...m}\} \\ \hline k_j = l_i \text{ implies } S_j <: T_i \\ \hline \{k_j : S_j^{j\in 1...m}\} <: \{l_i : T_i^{i\in 1...n}\} \quad (S-RCD) \\ \hline \textbf{Lemma 1 } If \ S <: T \ is provable with the 3 separate rules for width, depth and permutation, it is provable with just S-RCD. \end{array}$

CIS 500, 6 November

OTHER RULES
$$S <: S$$
(S-REFL) $\frac{S <: U \quad U <: T}{S <: T}$ (S-TRANS) $S <: Top$ (S-TOP) $\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$ (S-ARROW)What other rules cause difficulty?

 S<: S</th>
 (S-REFL)

 $\frac{S <: U \quad U <: T}{S <: T}$ (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.

CIS 500, 6 November

7

JUST DROP THEM

It turns out that the rest of the subtyping rules are reflexive and transitive, without explicitly declaring that they are.

- **Lemma 2** 1. S <: S can be derived for every type S without using S-REFL.
- 2. If S <: T is derivable, then it can be derived without using S-TRANS.

We don't need these rules in our system.

CIS 500, 6 November

9

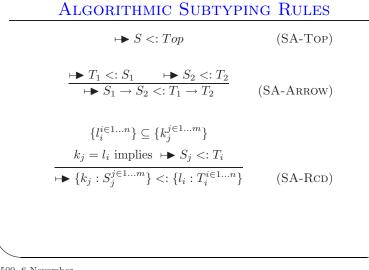
CIS 500, 6 November



The algorithmic rules are sound if every statement derivable from the algorithmic rules is derivable from the declarative rules.

The algorithmic rules are **complete** if every statement derivable from the declarative rules is derivable from the algorithmic rules.

Proposition 3 $S \lt: T$ iff $\mapsto S \lt: T$.



 $\begin{array}{l} \textbf{ALGORITHM} \\ \hline \textbf{subtype}(S,T) = \\ & \text{if } T = Top \text{ then true} \\ & \text{else if } S = S_1 \rightarrow S_2 \text{ and } T = T_1 \rightarrow T_2 \\ & \text{ then subtype}(T_1,S_1) \land \text{ subtype}(S_2,T_2) \\ & \text{else if } S = \{ \ k_j:S_j \ ^{j\in 1...m} \ \} \text{ and } T = \{ \ l_i:T_i \ ^{i\in 1...n} \ \} \\ & \text{ then } \{ \ l_i \ ^{i\in 1...n} \ \} \ \subseteq \ \{ \ k_j \ ^{j\in 1...m} \ \} \\ & \wedge \text{ for all } i \text{ there is some} \\ & j \in 1 \dots m \text{ with } k_j = l_i \\ & \text{ and subtype } (S_j,T_i) \\ & \text{else false} \end{array}$

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.

CIS 500, 6 November

13

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. \vdots \vdots $\vdash \{x:0, y=1\}: \{x:N, y:N\}$ $\{x:N\} \rightarrow N$ $\vdash \{x=0, y=1\}: \{x:N, y:N\}$ $\{x:N\} \rightarrow N$ $\vdash \{x:0, y=1\}: \{x:N\}$ $\{x:N\} \rightarrow N$ $\vdash \{x:N\} \cdot r.x) \{x = 0, y = 1\}: \{x:N\}$ (S-SUB)Do we need subsumption anywhere else?

16.2 Algorithmic Typing

The subsumption rule is not syntax directed:

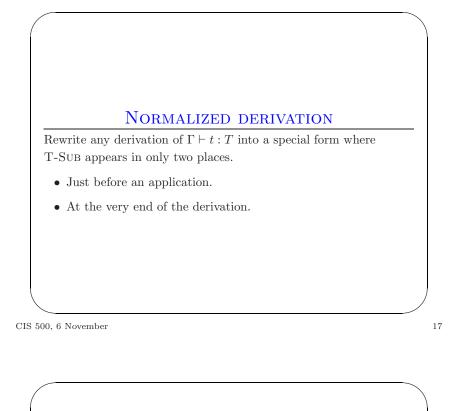
 $\frac{\Gamma \vdash t: S \quad S <: T}{\Gamma \vdash t: T} \quad (\text{T-Sub})$

1. This rule could be used for any t.

2. We have to guess T in the subgoal S <: T.

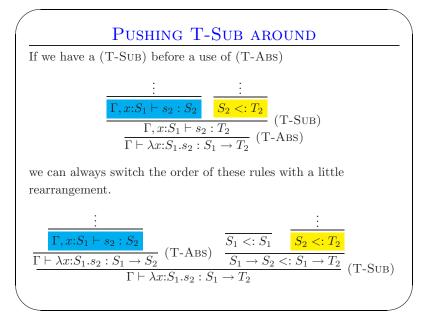
CIS 500, 6 November

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.

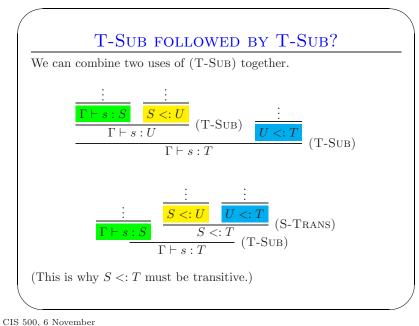




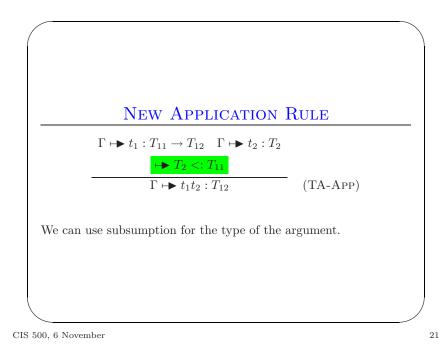
There are similar transformations for T-SUB followed by T-RCD and for T-SUB followed by T-PROJ.

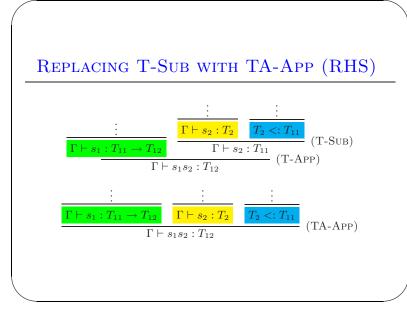


CIS 500, 6 November

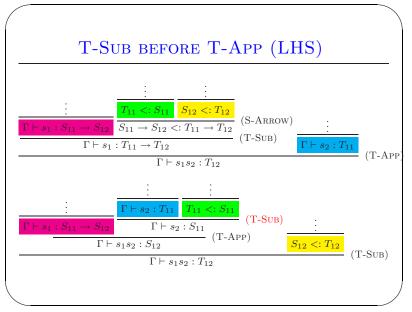


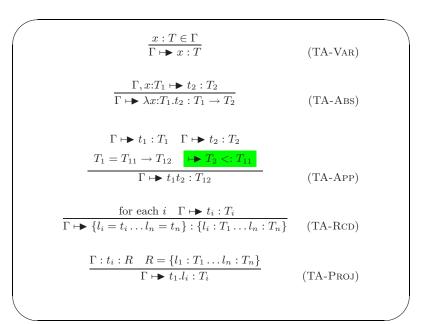
19





CIS 500, 6 November





23

CIS 500, 6 November

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 $\Gamma \mapsto t : T$ 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 $\Gamma \mapsto t : S$ for some S <: T.

CIS 500, 6 November

25

CIS 500, 6 November

Algorithmic rule

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

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

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 .

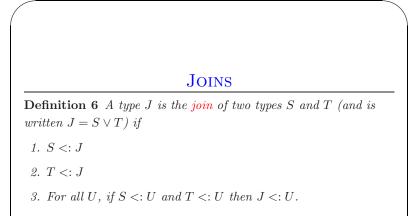
We need to restrict this rule so that it produces the least such T. This T is called the join of T_2 and T_3 .

16.3 Joins and Meets

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

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

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



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 $J = S \wedge T$) if

1. M <: S

2. M <: J

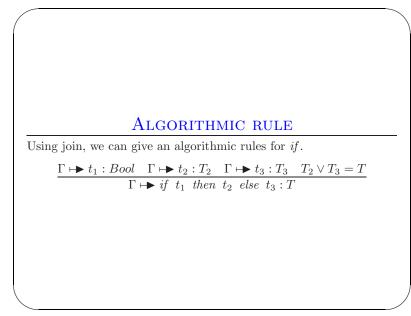
3. For all U, if $U \leq :S$ and $U \leq :T$ then $U \leq :M$.

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.

CIS 500, 6 November $\,$

29



EXISTENCE OF JOINS AND MEETS

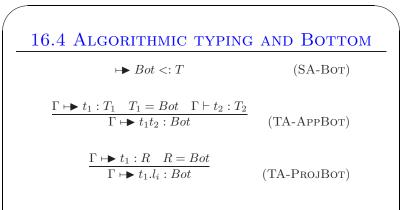
Given a subtype relation, it may or may not be the case that joins and meets exist for every pair of types.

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

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 $S \wedge T = M$.

CIS 500, 6 November



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