

## Chap 16 Metatheory of Subtyping

Algorithmic Subtyping Algorithmic Typing Joins and Meets



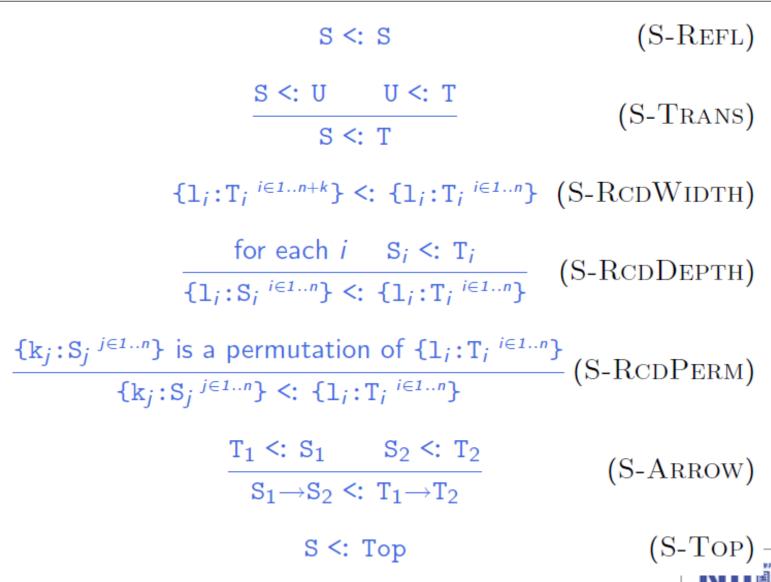


## Developing an algorithmic subtyping relation



#### Subtype Relation





#### **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.





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

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

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



#### Syntax-directed rules



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

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \qquad (T-APP)$$



#### Syntax-directed rules



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

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \qquad (T-APP)$$

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



(T-APP)

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)

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \rightarrow \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1 \ \mathtt{t}_2 : \mathtt{T}_{12}}$$

#### 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*.

 $\implies$  no backtracking!



### Non-syntax-directedness of typing



When we extend the system with *subtyping*, both aspects of syntax-directedness get broken.

1. The set of typing rules now includes *two* rules that can be used to give a type to terms of a given shape (the old one plus T—SUB)

$$\frac{\Gamma \vdash t : S \qquad S \lt: T}{\Gamma \vdash t : T}$$
(T-SUB)

2. Worse yet, the new rule T-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-SUB would cause divergence.)





Moreover, the *subtyping relation* is *not syntax directed* either.

- 1. There are *lots* of ways to derive a given subtyping statement.
- 2. The transitivity rule

 $\frac{S <: U \qquad U <: T}{S <: T} \qquad (S-TRANS)$ 

is *badly non-syntax-directed*: the premises contain a *metavariable* (in an *"input position"*) that does *not appear at all in the conclusion*.

To implement this rule naively, we have to *guess* a value for U!

#### What to do?



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.

- 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**





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 subtypin

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

$$\frac{\{\mathbf{l}_{i} \stackrel{i \in 1..n}{}\} \subseteq \{\mathbf{k}_{j} \stackrel{j \in 1..m}{}\} \qquad \mathbf{k}_{j} = \mathbf{l}_{i} \text{ implies } \mathbf{S}_{j} \leq \mathbf{T}_{i}}{\{\mathbf{k}_{j} : \mathbf{S}_{j} \stackrel{j \in 1..m}{}\} \leq \{\mathbf{l}_{i} : \mathbf{T}_{i} \stackrel{i \in 1..n}{}\}} \qquad (S-RCD)$$



#### Simpler subtype relation



$$\frac{\{l_i \stackrel{i \in 1..n}{} \subseteq \{k_j \stackrel{j \in 1..m}{} \} \quad k_j = l_i \text{ implies } S_j \leq T_i }{\{k_j : S_j \stackrel{j \in 1..m}{} \} < \{l_i : T_i \stackrel{i \in 1..n}{} \}}$$
(S-RCD)

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$
 (S-ARROW)

S <: Top

(S-TOP)



#### Step 2: Get rid of reflexivity



*Observation*: **S-REFL** is unnecessary.

Lemma: S <: S can be derived for every type S without using S-REFL.



$$\frac{S <: U \qquad U <: T}{S <: T} \qquad (S-TRANS)$$

$$\frac{\{\mathbf{l}_{i} \stackrel{i \in 1..n}{}\} \subseteq \{\mathbf{k}_{j} \stackrel{j \in 1..m}{}\} \qquad \mathbf{k}_{j} = \mathbf{l}_{i} \text{ implies } \mathbf{S}_{j} \leq \mathbf{T}_{i}}{\{\mathbf{k}_{j} : \mathbf{S}_{j} \stackrel{j \in 1..m}{}\} \leq \{\mathbf{l}_{i} : \mathbf{T}_{i} \stackrel{i \in 1..n}{}\}} \qquad (S-RCD)$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$
 (S-ARROW)

S <: Top

(S-TOP)



#### Step 3: Get rid of transitivity



*Observation*: S-Trans is unnecessary.

*Lemma*: If S <: T can be derived, then it can be derived without using S-Trans .





$$\frac{\{1_{i}^{i \in 1..n}\} \subseteq \{k_{j}^{j \in 1..m}\} \quad k_{j} = 1_{i} \text{ implies } S_{j} \leq T_{i}}{\{k_{j}: S_{j}^{j \in 1..m}\} \leq \{1_{i}: T_{i}^{i \in 1..n}\}}$$
(S-RCD)
$$\frac{T_{1} \leq S_{1} \quad S_{2} \leq T_{2}}{S_{1} \rightarrow S_{2} \leq T_{1} \rightarrow T_{2}}$$
(S-ARROW)
$$S \leq Top$$
(S-TOP)



#### "Algorithmic" subtype relation

$$[ \underbrace{\mathbb{S}} S \le \operatorname{Top} \qquad (\underbrace{\mathbb{S}} A \operatorname{-} \operatorname{Top})$$

$$\stackrel{[}{\longrightarrow} T_1 \le S_1 \qquad \models S_2 \le T_2 \\ \xrightarrow{[}{\longrightarrow} S_1 \rightarrow S_2 \le T_1 \rightarrow T_2} \qquad (SA \operatorname{-} \operatorname{ARROW})$$

$$\underbrace{\{1_i^{i \in 1..n}\} \subseteq \{k_j^{j \in 1..m}\} \qquad \text{for each } k_j = 1_i, \ \models S_j \le T_i \\ \xrightarrow{[}{\longrightarrow} \{k_j : S_j^{j \in 1..m}\} \le \{1_i : T_i^{i \in 1..n}\}} \qquad (SA \operatorname{-} \operatorname{RCD})$$



#### 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.)





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





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

Is our *subtype* function a decision procedure?





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

Is our *subtype* function a decision procedure?

Since *subtype* is just an implementation of the algorithmic subtyping rules, we have

- 1. if 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)





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

Is our *subtype* function a decision procedure?

Since *subtype* is just an implementation of the algorithmic subtyping rules, we have

- 1. if subtype(S,T) = true, then  $\mapsto S <: T$  (hence, by soundness of the algorithmic rules, S <: T)
- 2. if subtype(S,T) = false, then not  $\mapsto S <: T$ (hence, by completeness of the algorithmic rules, not S <: T)
- Q: What's missing?





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

Is our *subtype* function a decision procedure?

Since *subtype* is just an implementation of the algorithmic subtyping rules, we have

- 1. if 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*?





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

Is our *subtype* function a decision procedure?

Since *subtype* is just an implementation of the algorithmic subtyping rules, we have

- 1. if subtype(S, T) = true, then  $\mapsto S <: T$  (hence, by soundness of the algorithmic rules, S <: T)
- 2. if subtype(S, T) = false, then not  $\mapsto S <: T$  (hence, by completeness of the algorithmic rules, not S <: T)
- Q: What's missing?

A: How do we know that *subtype* is a *total function*?

Prove it!





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

Example:

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

Note that, we are saying nothing about *computability*.





*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) = 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.





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.



#### **Decision Procedures (take 2)**



We want a decision procedure to be a *procedure*.

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



#### Example



 $U = \{1, 2, 3\}$  $R = \{(1, 2), (2, 3)\}$ The function p(x, y) = if x = 2 and y = 3 then trueelse if x = 1 and y = 2 then true else false 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 procedure for R.



#### Example



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

The recursively defined partial function

p(x, y) = if x = 2 and y = 3 then trueelse if x = 1 and y = 2 then trueelse if x = 1 and y = 3 then falseelse p(x, y)



#### Example



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

The recursively defined partial function

p(x, y) = if x = 2 and y = 3 then true else if x = 1 and y = 2 then true else if x = 1 and y = 3 then falseelse p(x, y)

whose graph is

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

is *not* a decision procedure for R.



# Subtyping Algorithm



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

subtype(S, T) =if T = Top, then *true* else if S = S<sub>1</sub>  $\rightarrow$  S<sub>2</sub> and T = T<sub>1</sub>  $\rightarrow$  T<sub>2</sub> then  $subtype(T_1, S_1) \land subtype(S_2, T_2)$ else if S = {k<sub>i</sub>:  $S_i^{j \in 1..m}$ } and T = {l<sub>i</sub>:  $T_i^{i \in 1..n}$ } then  $\{l_i^{i \in 1..n}\} \subseteq \{k_i^{j \in 1..m}\}$ ∧ for all  $i \in 1..n$  there is some  $j \in 1..m$  with  $k_i = l_i$ and  $subtype(S_i, T_i)$ else *false*.



# Subtyping Algorithm



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

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

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 lambdacalculus *with subtyping*?

Given a context  $\Gamma$  and a term t, how do we determine its type T, such that  $\Gamma \vdash t : T$ ?







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

 $\frac{\Gamma \vdash t: S \qquad S <: T}{\Gamma \vdash t: T}$ 

(T-SUB)

Q: where is this rule really needed?







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

 $\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$ 

(T-SUB)

Q: where is this rule really needed?

For applications, e.g., the term

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

is *not typable* without using subsumption.







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

 $\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$  (T-Sub) Q: where is this rule really needed?

For applications, e.g., the term
 (\lambda r: {x: Nat}. r. x) {x = 0, y = 1}
is not typable without using subsumption.

Where else??







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

 $\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$ 

Q: where is this rule really needed?

For applications, e.g., the term
 (\lambda r: {x: 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.



(T-SUB)

#### Plan

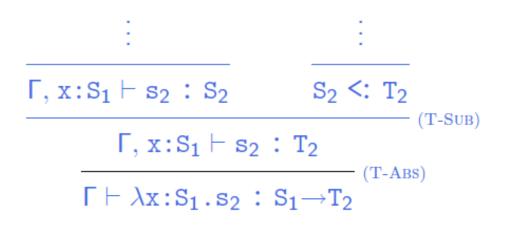


- Investigate how subsumption is used in typing derivations by looking at examples of how it can be "pushed through" other rules
- 2. Use the intuitions gained from these examples to design a new, algorithmic typing relation that
  - Omits subsumption
  - Compensates for its absence by *enriching the application rule*
- *3. Show that* the algorithmic typing relation is essentially equivalent to the original, declarative one



# Example (T-ABS)

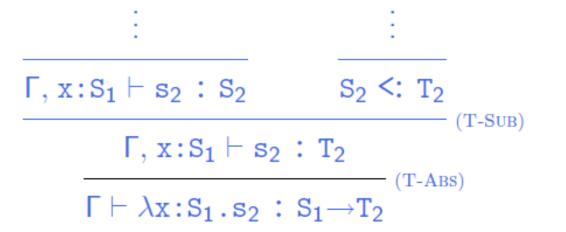




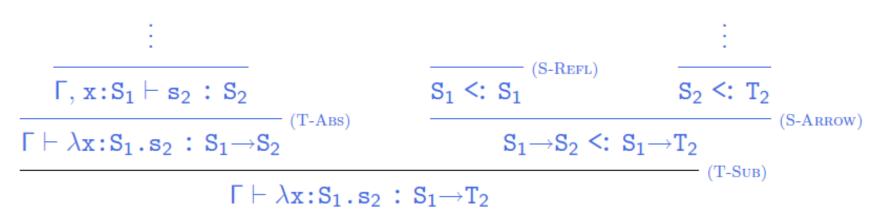


# Example (T-ABS)





becomes





# Intuitions



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

What about T-APP?

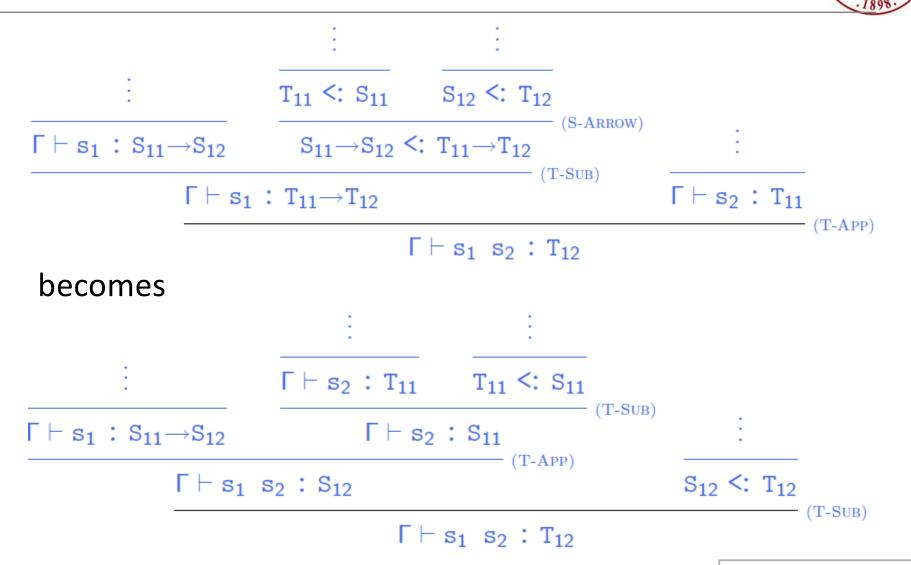
We've already observed that T-SUB is required for typechecking some *applications*.

So we expect to find that we *cannot* play the same game with T-APP as we've done with T-ABS.

Let's see why.

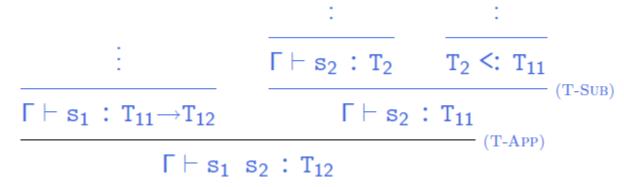


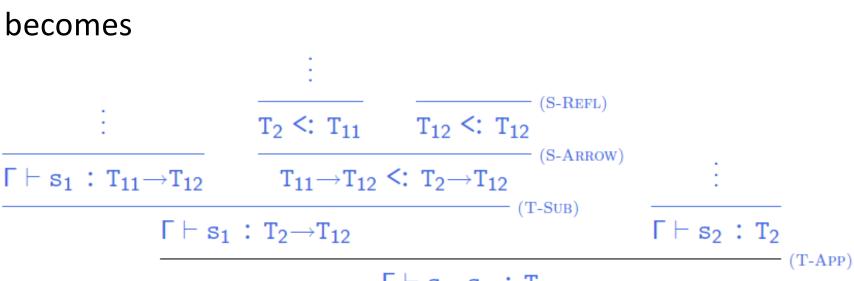
# **Example (T–Sub with T-APP on the left)**





# Example (T—Sub with T-APP on the rig





 $\Gamma \vdash s_1 \ s_2 \ : \ T_{12}$ 



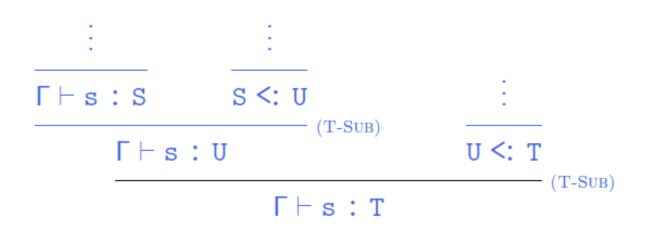
#### **Observations**



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*.

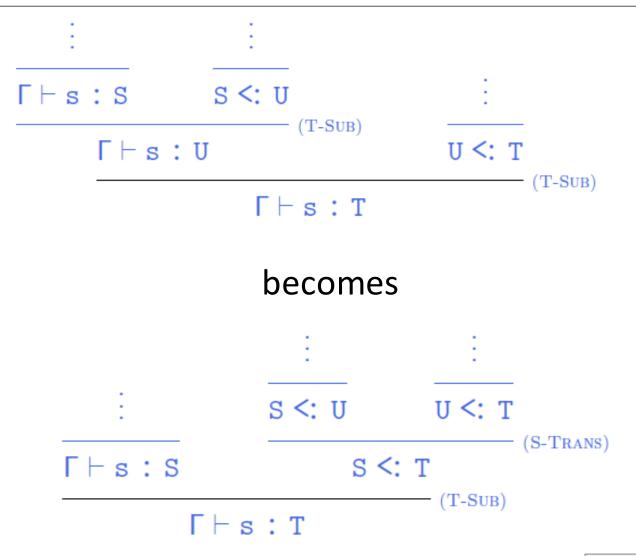


# Example (nested uses of T-Sub)





# Example (nested uses of T-Sub)





# 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.

$$\begin{array}{cccc} \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \rightarrow \mathtt{T}_{12} & \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2 & \vdash \mathtt{T}_2 <: \mathtt{T}_{11} \\ & \\ & \\ & \Gamma \vdash \mathtt{t}_1 \ \mathtt{t}_2 : \mathtt{T}_{12} \end{array}$$

Given any typing derivation, we can now

- 1. normalize it, to move all uses of subsumption to either just before applications (in the right-hand premise) or at the very end
- 2. replace uses of T-App with T-SUB in the right-hand premise by uses of the extended rule above

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



# Minimal Types



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

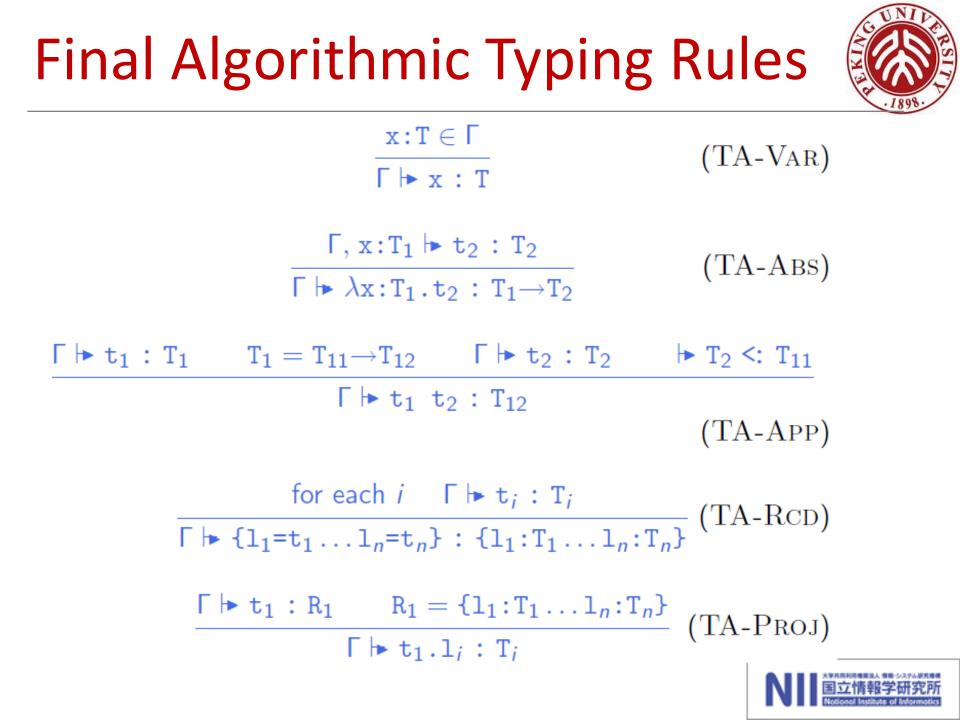
It is just used to give *more* types to terms that have already been shown to have a type.

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

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

For purposes of building a typechecking algorithm, this is enough.







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





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

 $\begin{array}{ll} \label{eq:constraint} \Gamma \vdash \texttt{true} : \texttt{Bool} & (\text{T-TRUE}) \\ \Gamma \vdash \texttt{false} : \texttt{Bool} & (\text{T-FALSE}) \\ \\ \hline \Gamma \vdash \texttt{t}_1 : \texttt{Bool} & \Gamma \vdash \texttt{t}_2 : \texttt{T} & \Gamma \vdash \texttt{t}_3 : \texttt{T} \\ \hline \Gamma \vdash \texttt{if} \ \texttt{t}_1 \ \texttt{then} \ \texttt{t}_2 \ \texttt{else} \ \texttt{t}_3 : \texttt{T} \end{array} \tag{T-IF}$ 





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

What is the minimal type of

*if true then*  $\{x = true, y = false\}$  *else*  $\{x = true, z = ture\}$ ?



# The Algorithmic Conditional Rule



More generally, we can use subsumption to give an expression

#### if $t_1$ then $t_2$ else $t_3$

any type that is a possible type of both  $t_2$  and  $t_3$ .

So the *minimal* type of the conditional is the *least* common supertype (or join) of the minimal type of  $t_2$  and the minimal type of  $t_3$ .



# The Algorithmic Conditional Rule



More generally, we can use subsumption to give an expression

#### if $t_1$ then $t_2$ else $t_3$

any type that is a possible type of both  $t_2$  and  $t_3$ .

So the *minimal* type of the conditional is the *least* common supertype (or join) of the minimal type of  $t_2$  and the minimal type of  $t_3$ .

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







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.,  $Bool \rightarrow Bool$  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  $N \le S$  and  $N \le T$ , 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  $\underline{S}$  and  $\underline{T}$ .



# **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.,  $\frac{M}{S}$  (when it exists) is the largest type that is a subtype of both  $\frac{S}{S}$  and  $\frac{T}{I}$ .

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



# **Calculating Joins**



$$S \lor T = \begin{cases} Bool & \text{if } S = T = Bool \\ M_1 \rightarrow J_2 & \text{if } S = S_1 \rightarrow S_2 & T = T_1 \rightarrow T_2 \\ S_1 \land T_1 = M_1 & S_2 \lor T_2 = J_2 \\ \{j_l: J_l \stackrel{l \in 1..q}{}\} & \text{if } S = \{k_j: S_j \stackrel{j \in 1..m}{}\} \\ T = \{l_i: T_i \stackrel{i \in 1..n}{}\} \\ \{j_l \stackrel{l \in 1..q}{}\} = \{k_j \stackrel{j \in 1..m}{}\} \cap \{l_i \stackrel{i \in 1..n}{}\} \\ S_j \lor T_i = J_l & \text{for each } j_l = k_j = l_i \\ Top & \text{otherwise} \end{cases}$$



# **Calculating Meets**



#### $S \wedge T =$

S	if $T = Top$
Т	if $S = Top$
Bool	if $S = T = Bool$
$J_1 \rightarrow M_2$	$\text{if } S = S_1 {\rightarrow} S_2 \qquad T = T_1 {\rightarrow} T_2$
	$\mathtt{S}_1 \lor \mathtt{T}_1 = \mathtt{J}_1  \mathtt{S}_2 \land \mathtt{T}_2 = \mathtt{M}_2$
${m_{l}: M_{l} \stackrel{l \in 1q}{\longrightarrow}}$	$if S = \{k_j: S_j \stackrel{j \in 1m}{}\}$
	$\mathbf{T} = \{\mathbf{l}_i: \mathbf{T}_i \ ^{i \in 1n}\}$
	$\{\mathbf{m}_{i} \stackrel{i \in 1q}{=} \{\mathbf{k}_{j} \stackrel{j \in 1m}{=} \cup \{\mathbf{l}_{i} \stackrel{i \in 1n}{=} \}$
	$\mathtt{S}_j \wedge \mathtt{T}_i = \mathtt{M}_l$ for each $\mathtt{m}_l = \mathtt{k}_j = \mathtt{l}_i$
	$M_l = S_j$ if $m_l = k_j$ occurs only in S
	$M_l = T_i$ if $m_l = l_i$ occurs only in T
fail	otherwise







- Read and digest chapter 16 & 17
- HW: 16.1.2; 16.2.6

