

编程语言的设计原理 Design Principles of Programming Languages

Haiyan Zhao, Di Wang

赵海燕,王迪

Peking University, Spring Term 2024



Issues in Subtyping



Principle of safe substitution:

- 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.,

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

Subtype Relation: General rules



A subtyping is a *binary relation* between *types* that is closed under the following rules

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



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.



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 Γ 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}



The reason this works is that we can *divide the* "*positions*" of the typing relation into *input positions* (i.e., Γ 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 *"sub-goals"* from the sub-expressions 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)

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



The *second important point* about the simply typed lambda-calculus is that *the set of typing rules is syntax-directed*:

- For every "*input* " Γ 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 no backtracking!

Non-syntax-directedness of typing



When the system is extended 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-SUB)

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

2. Worse yet, the new rule T-SUB *itself is not syntax directed*: the *inputs* to *the left-hand sub-goal 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-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] \times)
- 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!





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*



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.



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

$$\frac{\{\mathbf{l}_i^{i \in 1..n}\} \subseteq \{\mathbf{k}_j^{j \in 1..m}\} \quad \mathbf{k}_j = \mathbf{l}_i \text{ implies } \mathbf{S}_j <: \mathbf{T}_i \\ \{\mathbf{k}_j : \mathbf{S}_j^{j \in 1..m}\} <: \{\mathbf{l}_i : \mathbf{T}_i^{i \in 1..n}\} \qquad (S-RCD)$$

Lemma 16.1.1: If **S** <: **T** is derivable from the subtyping rules including **S-RcdDepth**, **S-Rcd-Width**, and **S-Rcd-Perm** (but not **S-Rcd**), then it can also be derived using **S-Rcd** (and not **-RcdDepth**, **S-Rcd-Width**, or **S-Rcd-Perm**), and vice versa.



 $S \le S \qquad (S-REFL)$ $\frac{S \le U \qquad U \le T}{S \le T} \qquad (S-TRANS)$

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

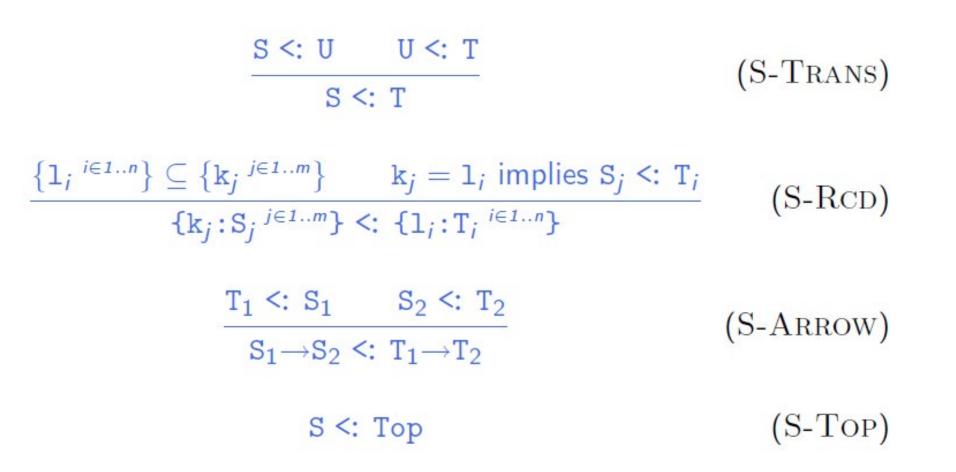
$$\frac{\mathbf{T}_{1} <: \mathbf{S}_{1} \ \mathbf{S}_{2} <: \mathbf{T}_{2}}{\mathbf{S}_{1} \rightarrow \mathbf{S}_{2} <: \mathbf{T}_{1} \rightarrow \mathbf{T}_{2}} \quad (S-ARROW)$$

$$\mathbf{S} <: \mathbf{Top} \quad (S-TOP)$$



Observation: S-REFL is unnecessary.

Lemma 16.1.2: S <: S can be derived for every type S without using S-REFL.





Observation: S-Trans is unnecessary.

Lemma 16.1.2: If S <: T can be derived, then it can be derived without using S-Trans.



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

$$\frac{\mathbf{T}_{1} <: \mathbf{S}_{1} \quad \mathbf{S}_{2} <: \mathbf{T}_{2}}{\mathbf{S}_{1} \rightarrow \mathbf{S}_{2} <: \mathbf{T}_{1} \rightarrow \mathbf{T}_{2}} \quad (S-ARROW)$$

$$\mathbf{S} <: \mathbf{Top} \quad (S-TOP)$$



Definition: The *algorithmic subtyping relation* is the least relation on types closed under the following 3 rules

$$[] \models S <: Top \qquad (\underline{SA}-TOP)$$

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

$$\frac{\{1_i^{i \in 1..n}\} \subseteq \{k_j^{j \in 1..m}\} \quad \text{for each } k_j = 1_i, \ \models S_j <: T_i \\ \models \{k_j : S_j^{j \in 1..m}\} <: \{1_i : T_i^{i \in 1..n}\} \qquad (SA-RCD)$$



Theorem[16.1.5]: $S \le T$ iff $\mapsto S \le T$

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)



subtype(S, T) =if T = Top, then *true* else if $S = S_1 \rightarrow S_2$ and $T = T_1 \rightarrow T_2$ then subtype(T_1, S_1) \land subtype(S_2, T_2) else if S = {k_i: $S_i^{j \in 1..m}$ } and T = {l_i: $T_i^{i \in 1..n}$ } 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.



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?

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?



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)

1. 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*?



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)

1. 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!

Design Principles of Programming Languages, Spring 2024



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:

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



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



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

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

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

whose graph is

{ ((1, 2), *true*), ((2, 3), *true*), ((1, 3), *false*)} is *not* a decision procedure for *R*.



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

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

else false.



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, \Rightarrow S, and T = T, \Rightarrow T.

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

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



How do we implement a *type checker* for the lambda-calculus *with subtyping*?

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

lssue



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

 $\frac{\Gamma \vdash t : S \qquad S \lt: T}{\Gamma \vdash t : T}$ (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??

Nowhere else!

Uses of subsumption rule to help typecheck *applications* are the only interesting ones (where subsumption plays a crucial role in typing)

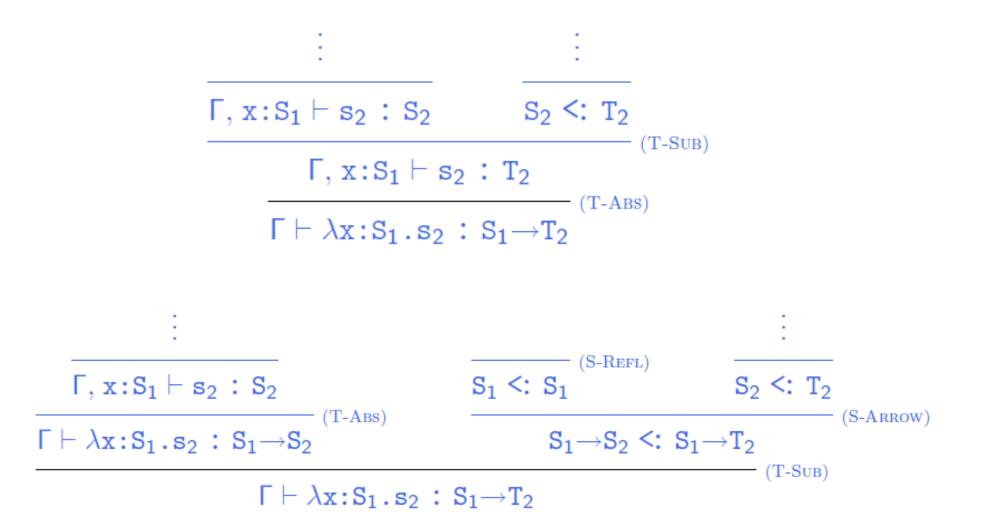




- 1. Investigate *how subsumption is used* in typing derivations by *looking at examples* of how it can be "*pushed through*" other rules;
- 2. Use the intuitions gained from these examples to design a new, algorithmic typing relation that
 - Omits subsumption;
 - Compensates for its absence by *enriching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



becomes



Intuitions



These examples show that we do not need to T-SUB"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*?

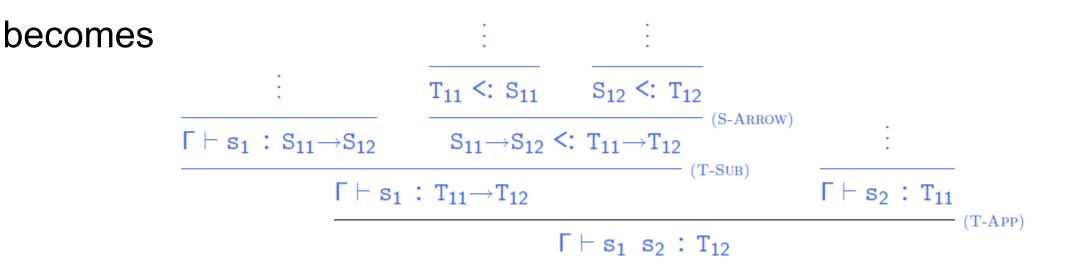
We've already observed that T-SUB is required for typechecking some *applications*

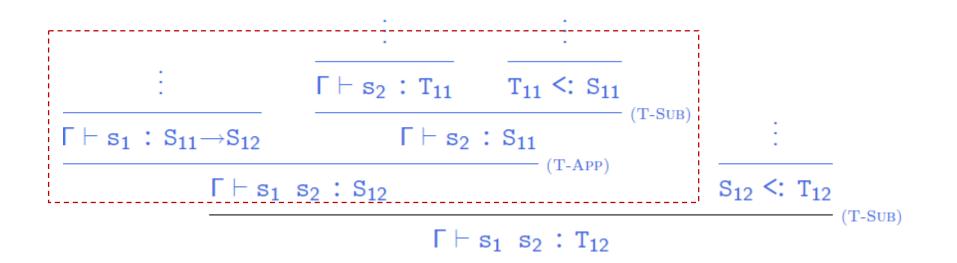
Therefore we expect to find that we *cannot* play the same game with T-APP as we've done with T-ABS

Let's see why.

Example (T—Sub with T-APP on the left)

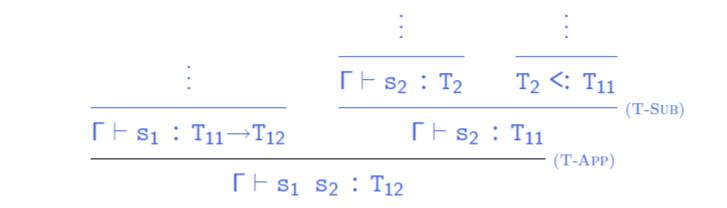


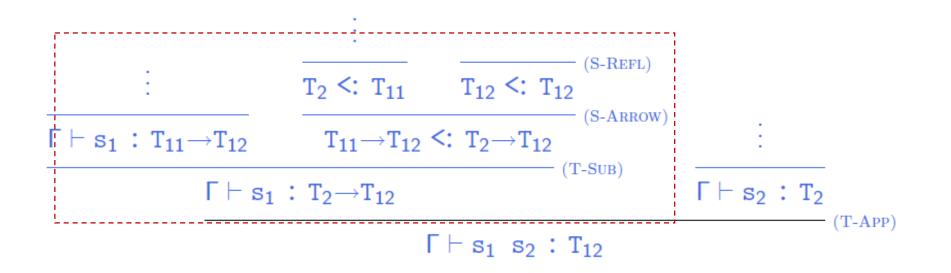




Example (T—Sub with **T**-APP **on the right)**







becomes

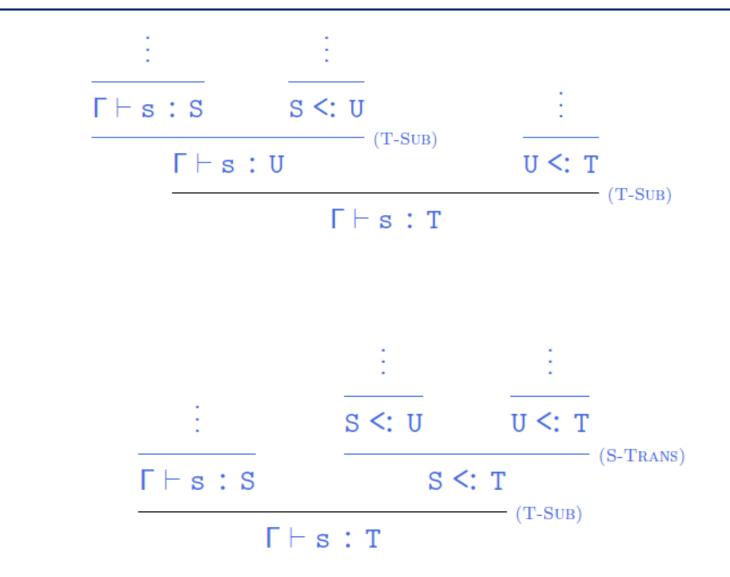
Observations



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)





becomes

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.



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*

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

Given any typing derivation, we can now

- 1. normalize it, to *move all uses of subsumption 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 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.

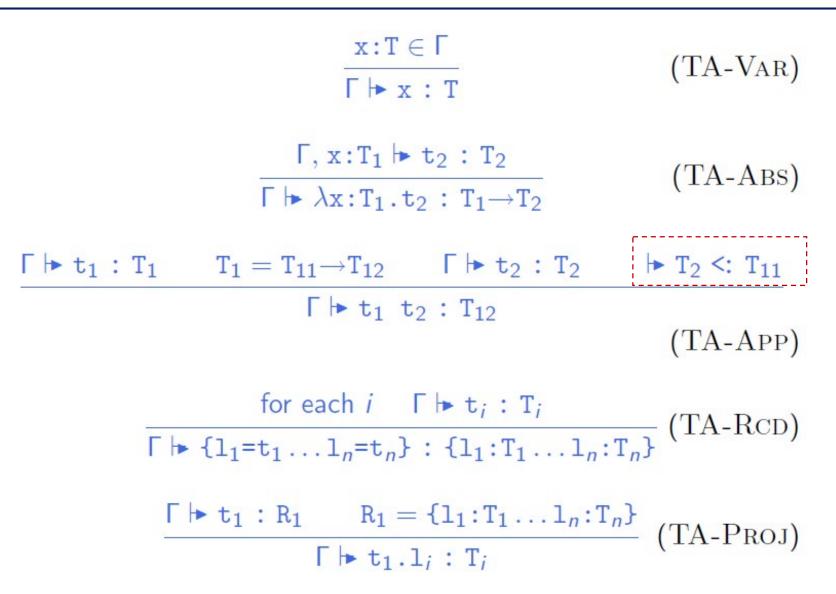
In other words, if we *dropped subsumption completely* (after refining the application rule), we would still be able to give types to exactly the same set of terms — we just would not be able to give as *many types* to some of them.

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







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.

A Problem with Conditional Expressions



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* $\{x = true, z = true\}$?

The Algorithmic Conditional Rule



More generally, we can use subsumption to give an expression

if t₁ then t₂ else t₃

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

 $\begin{array}{c|c} \hline {\mbox{\boldmath${\mbox{\boldmath${${\rm F}$}$}$}} t_1: {\mbox{Bool}} & \mbox{\boldmath${${\rm F}$}$} \flat t_2: T_2 & \mbox{\boldmath${${\rm F}$}$} \flat t_3: T_3 \\ \hline {\mbox{\boldmath${${\rm F}$}$}$} \flat \mbox{if} t_1 \mbox{then} t_2 \mbox{else} t_3: T_2 \lor T_3 \end{array} \end{tabular} \mbox{(T-IF)}$

Q: Does such a type exist for every T_2 and T_3 ??

Design Principles of Programming Languages, Spring 2024



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?



$S \lor T =$	Bool	if $S = T = Bool$
	Bool $M_1 \rightarrow J_2$	$\text{if } \mathtt{S} = \mathtt{S}_1 {\rightarrow} \mathtt{S}_2 \qquad \mathtt{T} = \mathtt{T}_1 {\rightarrow} \mathtt{T}_2$
		$\mathtt{S}_1 \wedge \mathtt{T}_1 = \mathtt{M}_1 \mathtt{S}_2 \vee \mathtt{T}_2 = \mathtt{J}_2$
	${j_{l}: J_{l} \stackrel{l \in 1q}{=} }$	if $S = \{k_j: S_j \in Im\}$
		$\mathbf{T} = \{\mathbf{l}_i: \mathbf{T}_i \ ^{i \in 1n}\}$
		$\{j_{i} {}^{i \in 1q}\} = \{k_{j} {}^{j \in 1m}\} \cap \{l_{i} {}^{i \in 1n}\}$
		$S_j \vee T_i = J_i$ for each $j_i = k_j = l_i$
	Тор	otherwise

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!



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

- 1. M <: S
- 2. M <: T

3. If 0 is a type such that 0 <: S and 0 <: T, then 0 <: 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



 $S \wedge T$ $\begin{cases} S & \text{if } T = \text{Top} \\ T & \text{if } S = \text{Top} \\ \text{Bool} & \text{if } S = T = \text{Bool} \\ J_1 \rightarrow M_2 & \text{if } S = S_1 \rightarrow S_2 \quad T = T_1 \rightarrow T_2 \\ S_1 \lor T_1 = J_1 \quad S_2 \land T_2 = M_2 \\ \{m_l : M_l \ ^{l \in 1 \dots q}\} & \text{if } S = \{k_j : S_j \ ^{j \in 1 \dots m}\} \\ T = \{l_i : T_i \ ^{i \in 1 \dots n}\} \end{cases}$ $\{\mathbf{m}_{l} \mid l \in 1..q\} = \{\mathbf{k}_{i} \mid j \in 1..m\} \cup \{\mathbf{1}_{i} \mid i \in 1..n\}$ $S_i \wedge T_i = M_i$ for each $m_i = k_i = l_i$ $M_l = S_i$ if $m_l = k_i$ occurs only in S $M_{l} = T_{i} \qquad \text{if } m_{l} = l_{i} \text{ occurs only in } T$ ail otherwise





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. $\{x: Bool\} \rightarrow Top and \{y: Bool\} \rightarrow Top?$

Homework[©]



• Read and digest chapter 16 & 17

- HW:
 - 16.1.2; 16.2.6; 16.4.1