

Recap on Subtype

Rule of Subsumption



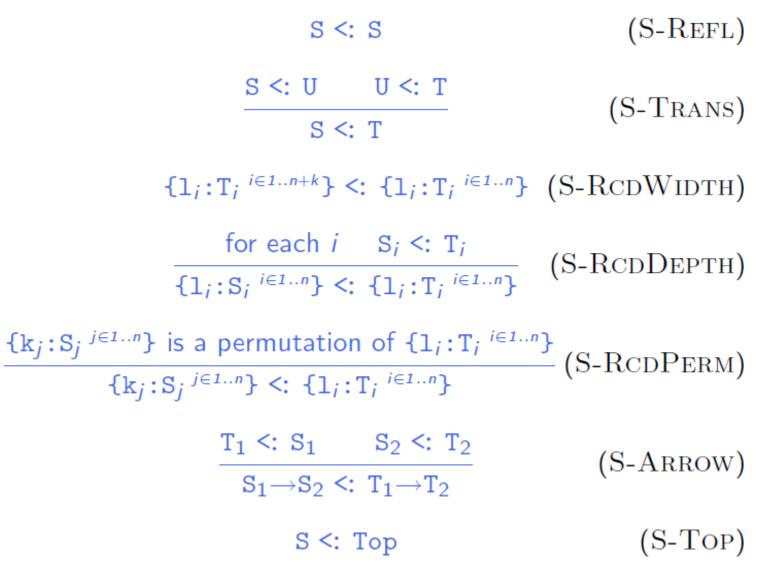
$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$
(T-SUB)

- 1. a *subtyping relation* between types, written S <: T
- 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







Properties of Subtyping





(T-SUB)

Statements of progress and preservation theorems are **unchanged** from λ_{\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

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

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: $U = U_1 \rightarrow U_2$ $T_1 <: U_1$ $U_2 <: T_2$ 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: $U <: W \qquad W <: T_1 \rightarrow T_2$

- 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 $W_2 <: T_2$.
- 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 $U_2 <: W_2$.
- By S-Trans, $T_1 <: U_1$, and, by S-Trans again, $U_2 <: T_2$, as required.



Lemma: if $\Gamma \vdash \lambda x: S_1. s_2: T_1 \longrightarrow T_2$, then $T_1 <: S_1 \text{ and } \Gamma, x: S_1 \vdash s_2: T_2$

Proof: Induction on typing derivations.

Case T-ABS: $T_1 = S_1$ $T_2 = S_2$ Γ , $x:S_1 \vdash s_2:S_2$

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 $U_1 \le S_1$ and Γ , $x:S_1 \vdash s_2 : U_2$.
- From $U_1 \leq S_1$ and $T_1 \leq U_1$, rule S-Trans gives $T_1 \leq S_1$.
- From Γ , $x:S_1 \vdash s_2 : U_2$ and $U_2 <: T_2$, rule T-Sub gives Γ , $x:S_1 \vdash s_2:T_2$, thus we are done



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?



Case T-Sub: $t : S \leq S \leq T$

By the induction hypothesis, $\Gamma \vdash t' : S$. By T-Sub, $\Gamma \vdash t': T$.

Not hard!



Case T-App:

$$t = t_1 \ t_2$$

$$\Gamma \vdash t_1: T_{11} \longrightarrow T_{12}$$

$$\Gamma \vdash t_2: T_{11}$$

$$T = T_{12}$$

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



Case T-App:

$$t = t_1 t_2$$

$$\Gamma \vdash t_1: T_{11} \longrightarrow T_{12}$$

$$\Gamma \vdash t_2: T_{11}$$

$$T = T_{12}$$

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.

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



Case T-App:

$$t = t_1 \ t_2$$

$$\Gamma \vdash t_1 : T_{11} \longrightarrow T_{12}$$

$$\Gamma \vdash t_2 : T_{11}$$

$$T = T_{12}$$

 $\frac{Subcase}{T-APP} E-App1: t_1 \rightarrow t_1' t_1 = t_1' t_2$ The result follows from the induction hypothesis and

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



Case T-App:

$$t = t_1 t_2$$

$$\Gamma \vdash t_1: T_{11} \longrightarrow T_{12}$$

$$\Gamma \vdash t_2: T_{11}$$

$$T = T_{12}$$

<u>Subcase</u> E-App2: $t_1 = v_1$ $t_2 \rightarrow t'_2$ $t' = v_1 t'_2$ Similar.

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

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1 \ \mathtt{t}_2 \longrightarrow \mathtt{v}_1 \ \mathtt{t}_2'} \tag{E-APP2}$$





<u>Subcase</u> E-AppAbs:

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

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

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

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

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

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

 $(\lambda \mathtt{x} : \mathtt{T}_{11} . \mathtt{t}_{12}) \ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2] \mathtt{t}_{12} \quad (\mathrm{E}\text{-}\mathrm{APPABS})$

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 $\{l_i: T_i^{i \in 1..n}\}$, then **v** has the form $\{k_j = v_j^{j \in 1..m}\}$ with $\{l_i^{i \in 1..n}\} \subseteq \{k_a^{a \in 1..m}\}$

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:

$$\frac{\Gamma \vdash t_1 \,:\, T}{\Gamma \vdash t_1 \text{ as } T \,:\, T}$$

(T-ASCRIBE)

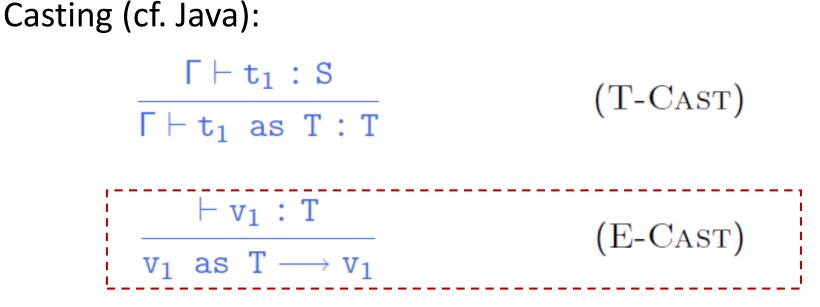
$$v_1$$
 as $T \longrightarrow v_1$

(E-ASCRIBE)

(T) T up-cast down-cast



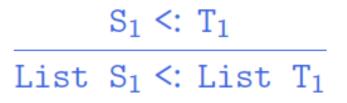
Ascription and Casting Ordinary ascription: $\Gamma \vdash t_1 : T$ (T-ASCRIBE) $\Gamma \vdash t_1$ as T:T(E-ASCRIBE) v_1 as $T \longrightarrow v_1$

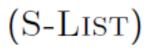


Subtyping and Variants



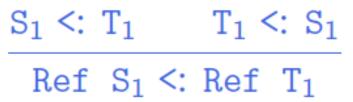






i.e., List is a *covariant type* constructor

Subtyping and References



(S-REF)

i.e., Ref is *not a covariant* (nor *a contravariant*) type constructor, but an *invariant*





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 $S_1 \leq T_1$ 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 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 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



 $\begin{array}{l} \frac{\Gamma \mid \Sigma \vdash t_{1} : \text{Source } T_{11}}{\Gamma \mid \Sigma \vdash !t_{1} : T_{11}} & (\text{T-DEREF}) \\ \\ \frac{\Gamma \mid \Sigma \vdash t_{1} : \text{Sink } T_{11} \quad \Gamma \mid \Sigma \vdash t_{2} : T_{11}}{\Gamma \mid \Sigma \vdash t_{1} : =t_{2} : \text{Unit}} & (\text{T-ASSIGN}) \end{array}$

Subtyping rules



 $S_1 <: T_1$ (S-SOURCE) Source $S_1 \leq Source T_1$ $T_1 <: S_1$ (S-SINK) Sink $S_1 \leq Sink T_1$ Ref $T_1 \leq \text{Source } T_1$ (S-REFSOURCE) (S-RefSink) Ref $T_1 \leq Sink T_1$

Subtyping and Arrays

Similarly...

 $\frac{S_1 <: T_1 \qquad T_1 <: S_1}{Array S_1 <: Array T_1} \qquad (S-ARRAY)$ $\frac{S_1 <: T_1}{Array S_1 <: Array T_1} \qquad (S-ARRAYJAVA)$

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,





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

– e.g., Bool <: Nat</p>



Intersection and Union Types



The inhabitants of $T_1 \wedge T_2$ are terms belonging to both S and T — i.e., $T_1 \wedge T_2$ is an order-theoretic meet (greatest lower

bound) of T_1 and T_2

 $T_1 \wedge T_2 <: T_1$ (S-INTER1)

 $T_1 \wedge T_2 <: T_2$ (S-INTER2)

 $\frac{S <: T_1 \qquad S <: T_2}{S <: T_1 \land T_2}$ (S-INTER3)

 $S \rightarrow T_1 \land S \rightarrow T_2 \iff S \rightarrow (T_1 \land T_2)$ (S-INTER4)



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

- + : (Nat \rightarrow Nat \rightarrow Nat) \land (Float \rightarrow Float \rightarrow 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 are also useful.

 $T_1 \vee T_2$ 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 $T_1 \vee T_2$ 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 $T_1 \vee T_2$ 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



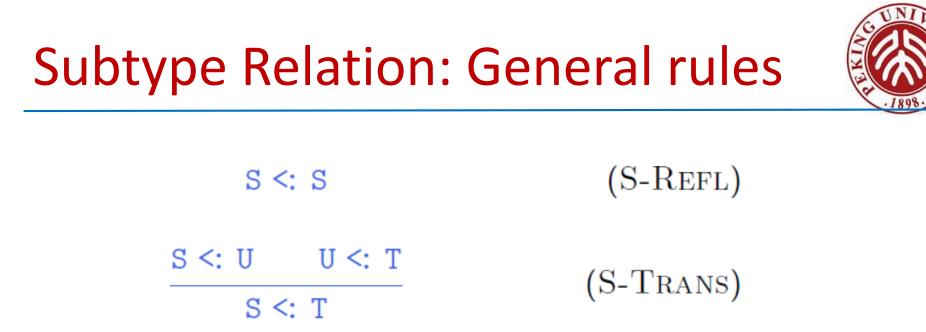
Principle of safe substitution

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

(T-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
- 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



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.

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

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}

Syntax-directed rules



Technically, 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 *"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)

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



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

- 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*
- \implies no backtracking!

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—*SUB*)

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

- 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 translate 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 (:: 8.2.4 /9.3.3 [uniqueness of types] ×)
- 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!



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 $S \leq T$ or $\Gamma \vdash t : T$

Both sets of rules are not *syntax-directed*



- 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.



Chap 16 Metatheory of Subtyping

Algorithmic Subtyping Algorithmic Typing Joins and Meets



Developing an algorithmic subtyping relation



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

$$\frac{\{1_{i} \ ^{i \in 1..n}\} \subseteq \{k_{j} \ ^{j \in 1..m}\} \qquad k_{j} = 1_{i} \text{ implies } S_{j} <: T_{i}}{\{k_{j} : S_{j} \ ^{j \in 1..m}\} <: \{1_{i} : T_{i} \ ^{i \in 1..n}\}}$$
(S-Re

Simpler subtype relation



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

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

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

 $S \leq Top$ (S-TOP)

Step 2: Get rid of reflexivity



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

Lemma 16.1.2: $S \leq S$ can be derived for every type S without using S-REFL.



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

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

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

 $S \leq Top$ (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.



$$\frac{\{1_{i} \stackrel{i \in 1..n}{}\} \subseteq \{k_{j} \stackrel{j \in 1..m}{}\} \quad k_{j} = 1_{i} \text{ implies } S_{j} <: T_{i}}{\{k_{j} : S_{j} \stackrel{j \in 1..m}{}\} <: \{1_{i} : T_{i} \stackrel{i \in 1..n}{}\}} \quad (S-RCD)$$

$$\frac{T_{1} <: S_{1} \qquad S_{2} <: T_{2}}{S_{1} \rightarrow S_{2} <: T_{1} \rightarrow T_{2}} \quad (S-ARROW)$$

$$S <: Top \qquad (S-TOP)$$



$$[\underbrace{F}_{S} \le \operatorname{Top} \qquad (\underbrace{SA}_{-} \operatorname{Top}) \\ \xrightarrow{\models} T_{1} \le S_{1} \implies S_{2} \le T_{2} \\ \xrightarrow{\models} S_{1} \rightarrow S_{2} \le T_{1} \rightarrow T_{2} \qquad (SA-ARROW) \\ \xrightarrow{1_{i} i \in 1..n} \subseteq \{ k_{j} \stackrel{j \in 1..m}{} \text{ for each } k_{j} = 1_{i}, \stackrel{\models}{} S_{j} \le T_{i} \\ \xrightarrow{\models} \{ k_{j} : S_{j} \stackrel{j \in 1..m}{} \} \le \{ 1_{i} : T_{i} \stackrel{i \in 1..n}{} \}$$

Soundness and completeness



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

Terminology:

- The *algorithmic presentation* of subtyping is *sound* with respect to the original, if → 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 → 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 *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
- 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?



- 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!

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)}

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

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)}

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

((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:

subtype(S, T) =if T = Top, then *true* else if S = S₁ \rightarrow S₂ and T = T₁ \rightarrow T₂ 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.

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 = S₁ \rightarrow S₂ and T = T₁ \rightarrow T₂ then $subtype(T_1, S_1) \land subtype(S_2, T_2)$ else if S = $\{k_i: S_i^{i \in 1..m}\}$ 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*.

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 lambdacalculus *with subtyping*?

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





(T-SUB)

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

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

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!

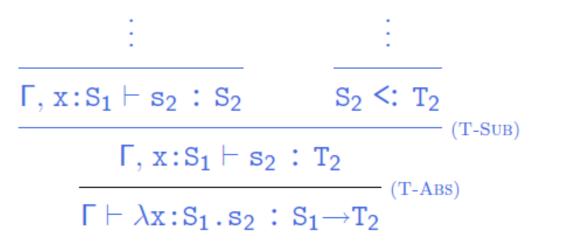
Uses of subsumption rule to help typecheck *applications* are the only interesting ones.

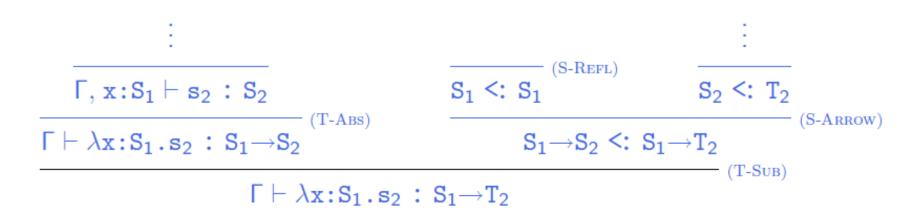


- Investigate how subsumption is used in typing derivations by looking at examples of how it can be "pushed through" other rules;
- 2. Use the intuitions gained from these examples to design a new, algorithmic typing relation that
 - Omits subsumption;
 - Compensates for its absence by *enriching the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

Example (T-ABS)







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*.

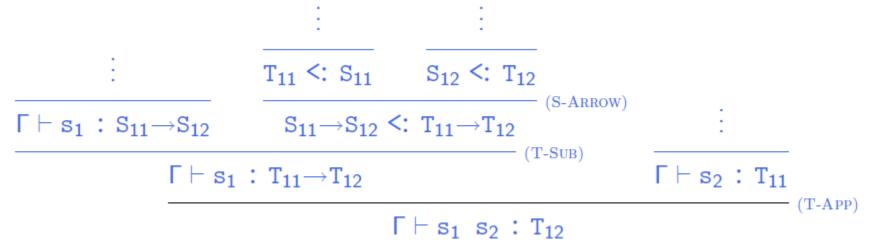
What about *T*-*APP*?

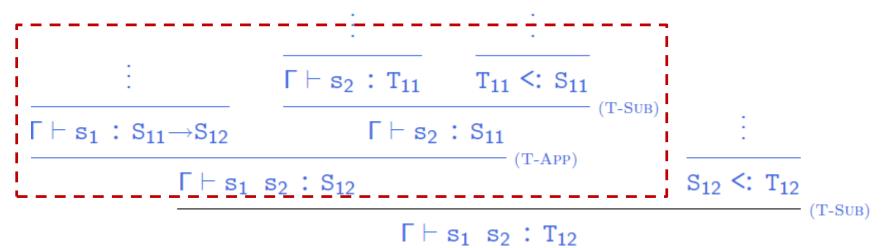
We've already observed that T-SUB is required for typechecking some *applications*

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

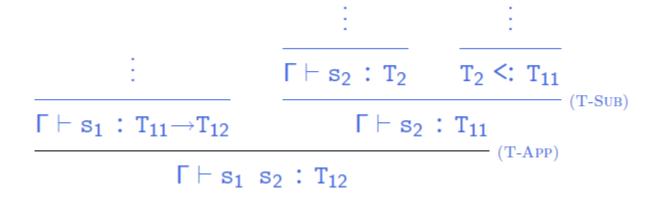
Let's see why.

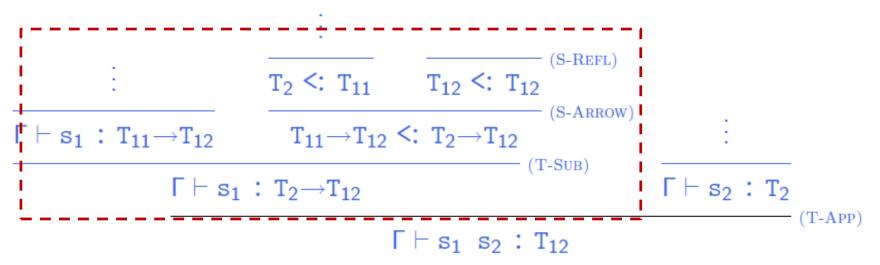
Example (T–Sub with T-APP on the left)





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



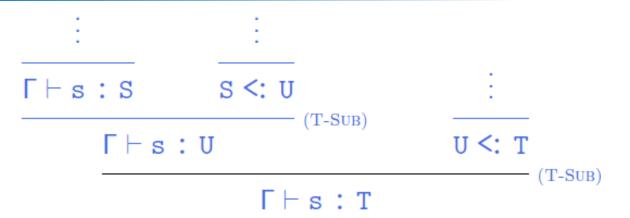


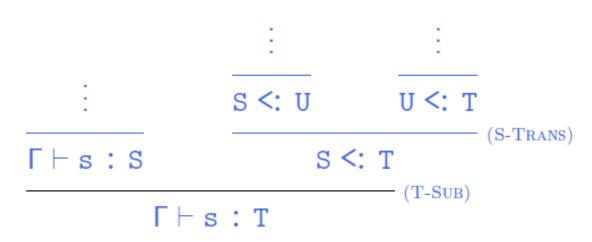




We've seen that uses of subsumption rule can be "pushed" from one of immediately before T-APP's premises to the other, but cannot be completely eliminated

Example (nested uses of T-Sub)





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*

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

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!



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		
$\frac{\mathbf{x}:T\inF}{F\models\mathbf{x}:T}$	(TA-VAR)	
$\frac{\Gamma, \mathbf{x}: \mathbf{T}_1 \models \mathbf{t}_2 : \mathbf{T}_2}{\Gamma \models \lambda \mathbf{x}: \mathbf{T}_1 \cdot \mathbf{t}_2 : \mathbf{T}_1 \rightarrow \mathbf{T}_2}$	(TA-ABS)	
$\label{eq:relation} \Gamma \blacktriangleright \mathtt{t}_1 : \mathtt{T}_1 \qquad \mathtt{T}_1 = \mathtt{T}_{11} {\rightarrow} \mathtt{T}_{12} \qquad \Gamma \trianglerighteq \mathtt{t}_2 : \mathtt{T}_2$	► T ₂ <: T ₁₁	
Γ ⊨ t ₁ t ₂ : T ₁₂	(TA-App)	
for each $i \Gamma \models t_i : T_i$ $\Gamma \models \{l_1 = t_1 \dots l_n = t_n\} : \{l_1 : T_1 \dots l_n :$	T_n (TA-RCD)	
$ \begin{array}{c c} \Gamma \models \mathtt{t}_1 : \mathtt{R}_1 & \mathtt{R}_1 = \{\mathtt{l}_1 : \mathtt{T}_1 \dots \mathtt{l}_n : \mathtt{T}_n \\ & \Gamma \models \mathtt{t}_1 . \mathtt{l}_i : \mathtt{T}_i \end{array} $	(TA-Proj)	

Completeness of the algorithmic 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



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*.

 $\begin{array}{rl} \Gamma \vdash \texttt{true} : \texttt{Bool} & (\text{T-TRUE}) \\ \Gamma \vdash \texttt{false} : \texttt{Bool} & (\text{T-FALSE}) \\ \hline \Gamma \vdash \texttt{t}_1 : \texttt{Bool} & \Gamma \vdash \texttt{t}_2 : \texttt{T} & \Gamma \vdash \texttt{t}_3 : \texttt{T} \\ \hline \Gamma \vdash \texttt{if} \ \texttt{t}_1 \ \texttt{then} \ \texttt{t}_2 \ \texttt{else} \ \texttt{t}_3 : \texttt{T} \end{array}$ (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* $\{x = true, z = true\}$?

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

$$\frac{\Gamma \models t_1 : \text{Bool} \qquad \Gamma \models t_2 : T_2 \qquad \Gamma \models t_3 : T_3}{\Gamma \models \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \lor T_3} \quad \text{(T-IF)}$$

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



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?



$$\begin{split} \mathsf{S} \lor \mathsf{T} &= \left\{ \begin{array}{ll} \mathsf{Bool} & \text{if } \mathsf{S} = \mathsf{T} = \mathsf{Bool} \\ \mathsf{M}_1 {\rightarrow} \mathsf{J}_2 & \text{if } \mathsf{S} = \mathsf{S}_1 {\rightarrow} \mathsf{S}_2 & \mathsf{T} = \mathsf{T}_1 {\rightarrow} \mathsf{T}_2 \\ & \mathsf{S}_1 \land \mathsf{T}_1 = \mathsf{M}_1 & \mathsf{S}_2 \lor \mathsf{T}_2 = \mathsf{J}_2 \\ \{\mathsf{j}_I : \mathsf{J}_I \ ^{I \in 1..q}\} & \text{if } \mathsf{S} = \{\mathsf{k}_j : \mathsf{S}_j \ ^{j \in 1..m}\} \\ & \mathsf{T} = \{\mathsf{l}_i : \mathsf{T}_i \ ^{i \in 1..n}\} \\ & \{\mathsf{j}_I \ ^{I \in 1..q}\} = \{\mathsf{k}_j \ ^{j \in 1..m}\} \cap \{\mathsf{l}_i \ ^{i \in 1..n}\} \\ & \mathsf{S}_j \lor \mathsf{T}_i = \mathsf{J}_I \quad \text{for each } \mathsf{j}_I = \mathsf{k}_j = \mathsf{l}_i \end{array} \right. \end{split}$$

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 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



$S \wedge T =$

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

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





- Read and digest chapter 16 & 17
- HW: 16.1.2; 16.2.6; 16.4.1