

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



Part III Chap 15: Subtyping

Subsumption

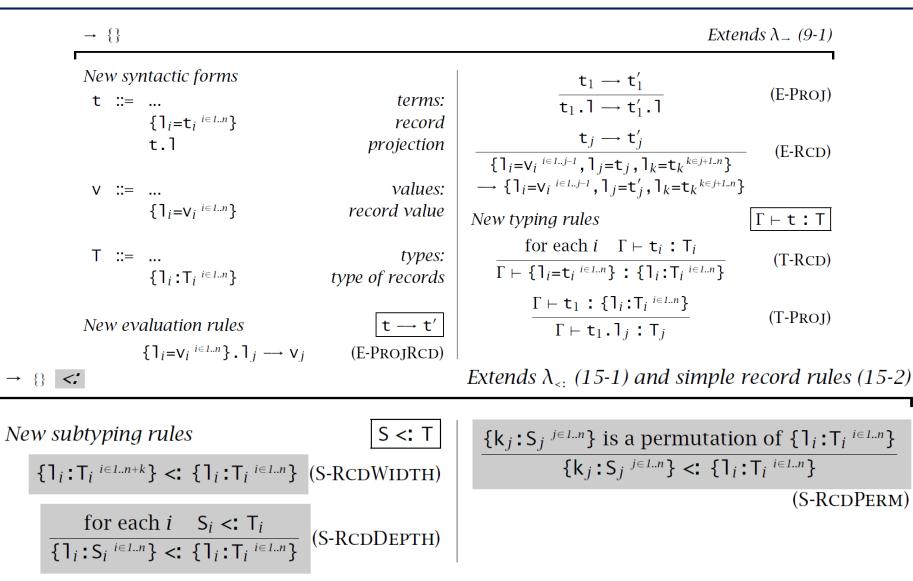
Subtype relation Properties of subtyping and typing Subtyping and other features Intersection and union types



Subtyping

Recap for subtyping





Recap for subtyping



→ <: Top			Based on λ_{\rightarrow} (9-1)
Syntax		Subtyping	S <: T
t ::=	terms:		
Х	variable	S <: S	(S-Refl)
λx:T.t	abstraction	S <: U U <: T	
tt	application	S <: U U <: T S <: T	(S-Trans)
V ::=	values:	S <: Top	(S-TOP)
λx:T.t	abstraction value		
Т ::= Тор	types: maximum type	$\frac{T_1 \boldsymbol{<:} S_1 \qquad S_2 \boldsymbol{<:} T_2}{S_1 \boldsymbol{\rightarrow} S_2 \boldsymbol{<:} T_1 \boldsymbol{\rightarrow} T_2}$	(S-Arrow)
T→T	type of functions	Typing	$\Gamma \vdash t : T$
Γ ::= Ø	contexts: empty context	$\frac{\mathbf{x}:T\in\Gamma}{\Gamma\vdash\mathbf{x}:T}$	(T-VAR)
Г, х: Т	term variable binding	$\frac{\Gamma, \mathbf{x}: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda \mathbf{x}: T_1 \cdot t_2 : T_1 \rightarrow T}$	$_{2}$ (T-ABS)
Evaluation $\frac{t_1 \rightarrow t}{t_1 t_2 \rightarrow t}$	$\begin{array}{c} t \longrightarrow t' \\ 1 \\ t' \\ t' \\ t' \\ t' \\ t' \\ t' \\ $	$\frac{\Gamma \vdash \mathbf{t}_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2}{\Gamma \vdash t_1 \; t_2 : T_{12}}$	2: T ₁₁ (T-APP)
$\frac{t_2 \longrightarrow t}{v_1 \; t_2 \longrightarrow v}$	$\frac{\frac{1}{2}}{1 t_2'} $ (E-APP2)	$\frac{\Gamma \vdash t: S S <: T}{\Gamma \vdash t: T}$	(T-Sub)
$(\lambda \mathbf{x}: T_{11}, t_{12}) v_2 \longrightarrow [x \mapsto v_2] t_{12} (E-APPABS)$			

Design Principles of Programming Languages, Spring 2024

Subtype Relation: General rules



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

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

Subtype Relation



$$S \leq S \qquad (S-REFL)$$

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

$$\{1_i:T_i \stackrel{i \in 1..n+k}{} \leq \{1_i:T_i \stackrel{i \in 1..n}{} (S-RCDWIDTH) \\ \frac{for each i \qquad S_i \leq T_i}{\{1_i:S_i \stackrel{i \in 1..n}{} \leq \{1_i:T_i \stackrel{i \in 1..n}{} (S-RCDDEPTH) \\ \frac{k_j:S_j \stackrel{j \in 1..n}{} is a permutation of \{1_i:T_i \stackrel{i \in 1..n}{} (S-RCDPERM) \\ \frac{K_j:S_j \stackrel{j \in 1..n}{} \leq \{1_i:T_i \stackrel{i \in 1..n}{} (S-RCDPERM) \\ \frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2} \qquad (S-ARROW) \\ S \leq Top \qquad (S-TOP)$$

{



Properties of Subtyping



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*.

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



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.)
 - -e.g., In order to derive a type for $t_1 t_2$, we must use T-App.
- 2. We don't have to "guess" an input (or output) for any rule.
 - *e.g.*, To derive a type for $t_1 t_2$, we need to derive a type for t_1 and a type for t_2 .

An Inversion Lemma for subtyping



Lemma: If $U \le T_1 \rightarrow T_2$, then U has the form $U_1 \rightarrow U_2$, with $T_1 \le U_1$ and $U_2 \le T_2$.

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), $T_1 \leq T_1$ and $T_2 \leq T_2$, as required.

Case S-Trans: $U \lt: W \qquad W \lt: T_1 \rightarrow T_2$

- 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, $T_1 \le U_1$, and, by S-Trans again, $U_2 \le T_2$, as required.

Inversion Lemma for Typing



Lemma: if $\Gamma \vdash \lambda x: S_1. s_2: T_1 \rightarrow T_2$, 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: $\Gamma \vdash \lambda x:S_1$. $s_2: U$ U: $T_1 \rightarrow T_2$

- By the subtyping inversion lemma, U has the form of $U_1 \rightarrow U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.
- The IH now applies, yielding $U_1 \leq S_1$ and Γ , $x:S_1 \vdash S_2 \in U_2$.
- From $U_1 \leq S_1$ and $T_1 \leq U_1$, rule S-Trans gives $T_1 \leq S_1$.
- From Γ , x:S₁ \vdash s₂ : U₂ and U₂ <: T₂, rule T-Sub gives Γ , x: S₁ \vdash s₂: T₂, thus we are done

Preservation



Theorem: If $\Gamma \vdash t$: T and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

Proof: By induction on *typing derivations*.

Which cases are likely to be hard?

Preservation - Subsumption case



Case T-Sub: t : S S <: T

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

Not hard!



Case T-App:

 $t = t_1 t_2 \quad \Gamma \vdash t_1: T_{11} \longrightarrow T_{12} \quad \Gamma \vdash t_2: T_{11} \quad T = T_{12}$

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:

$$t = t_1 \ t_2 \ \Gamma \vdash t_1 : T_{11} \longrightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11} \qquad T = T_{12}$$

<u>Subcase</u> E-App1 : $t_1 \rightarrow t'_1$ $t' = t'_1 t_2$

The result follows from the induction hypothesis and T-App

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

$$\frac{t_1 \longrightarrow t'_1}{t_1 \ t_2 \longrightarrow t'_1 \ t_2} \qquad (E-APP1)$$



Case T-App:

$$\mathbf{t} = \mathbf{t}_1 \ \mathbf{t}_2 \ \Gamma \vdash \mathbf{t}_1 : \mathbf{T}_{11} \longrightarrow \mathbf{T}_{12} \qquad \Gamma \vdash \mathbf{t}_2 : \mathbf{T}_{11} \qquad \mathbf{T} = \mathbf{T}_{12}$$

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

$$\begin{array}{c|c} \vdash t_{1} : T_{11} \rightarrow T_{12} & \Gamma \vdash t_{2} : T_{11} \\ \hline \Gamma \vdash t_{1} & t_{2} : T_{12} \end{array} & (T-APP) \\ \\ \hline \frac{t_{2} \longrightarrow t_{2}'}{v_{1} & t_{2} \longrightarrow v_{1} & t_{2}'} & (E-APP2) \end{array}$$

Design Principles of Programming Languages, Spring 2024

Preservation - Application case



Case T-App:

$$t = t_1 t_2 \qquad \Gamma \vdash t_1: T_{11} \longrightarrow T_{12} \qquad \Gamma \vdash t_2: T_{11} \qquad T = T_{12}$$

Subcase 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, $\Gamma \vdash t': T_{12}$

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

 $(\lambda x:T_{11}.t_{12}) v_2 \longrightarrow [x \mapsto v_2]t_{12}$ (E-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 that in the simply typed lambdacalculus



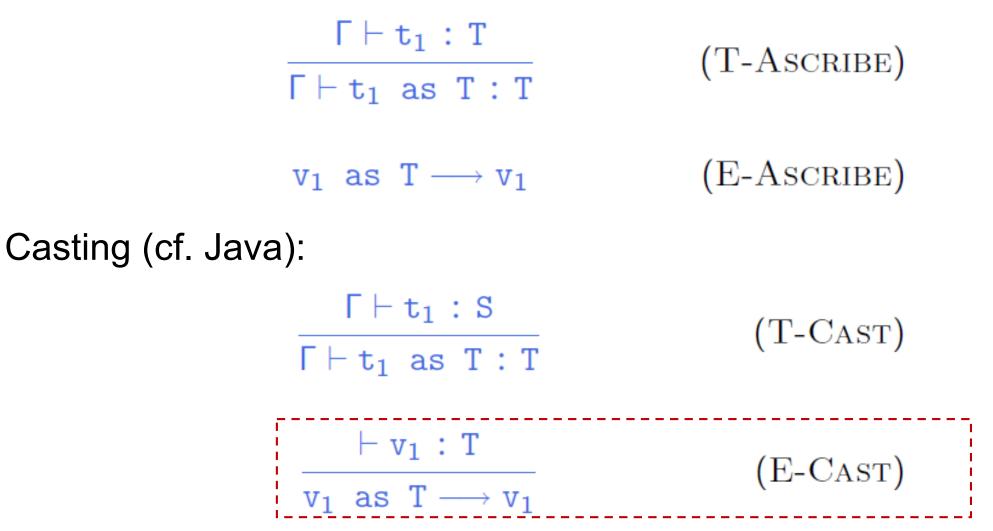
Subtyping with Other Features



Ordinary ascription:



(T) T up-cast down-cast Ordinary ascription:







 $\frac{\langle \mathbf{k}_{j}: \mathbf{S}_{j} \stackrel{j \in 1..n}{} \rangle \text{ is a permutation of } \langle \mathbf{l}_{i}: \mathbf{T}_{i} \stackrel{i \in 1..n}{} \rangle}{\langle \mathbf{k}_{j}: \mathbf{S}_{j} \stackrel{j \in 1..n}{} \rangle} \ll \langle \mathbf{l}_{i}: \mathbf{T}_{i} \stackrel{i \in 1..n}{} \rangle}$ (S-VARIANTPERM)

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \langle l_1 = t_1 \rangle : \langle l_1 : T_1 \rangle}$$

(T-VARIANT)

Subtyping and Lists



List is a covariant type constructor

 $\frac{S_1 <: T_1}{\text{List } S_1 <: \text{List } T_1}$





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

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1}$$

(S-Ref)



Ref is *not a covariant* (nor *a contravariant*) type constructor. Why?

- When a reference is *read*, the context expects a T_1 , so if $S_1 <: T_1$ 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 S₁, someone else may use the T_1 as an S₁. 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 ${\ensuremath{T}}$, and
- as a sink for values of type T



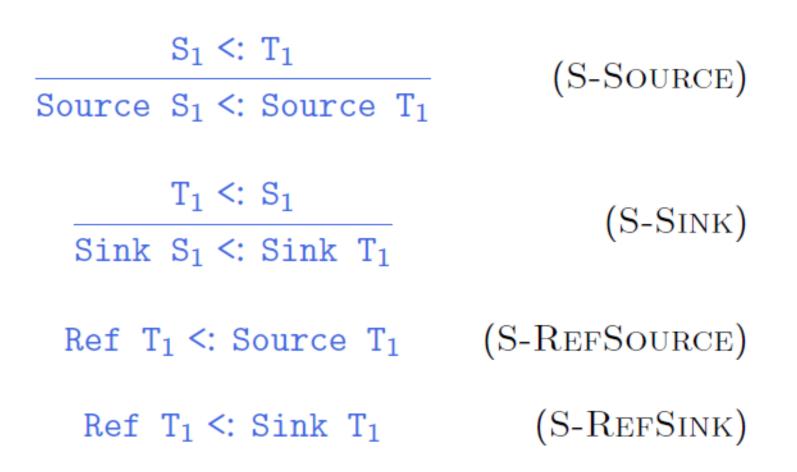
Observation: a value of type *Ref T* can be used in *two different* ways:

- as a source for values of type ${\ensuremath{ T}}$, 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



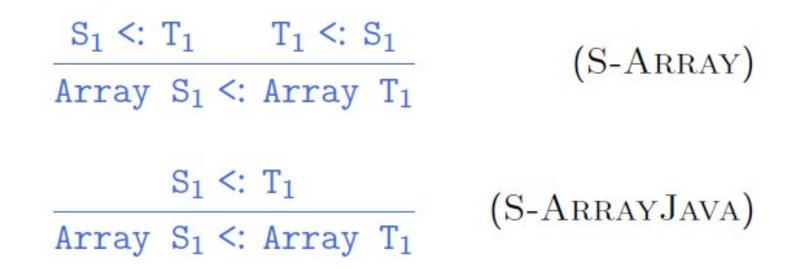
$$\begin{array}{l} \frac{\Gamma \mid \Sigma \vdash t_1 : \text{Source } T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}} & (\text{T-Deref}) \\ \\ \frac{\Gamma \mid \Sigma \vdash t_1 : \text{Sink } T_{11} \quad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 : = t_2 : \text{Unit}} & (\text{T-Assign}) \end{array}$$







Similarly...



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

Base Types



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

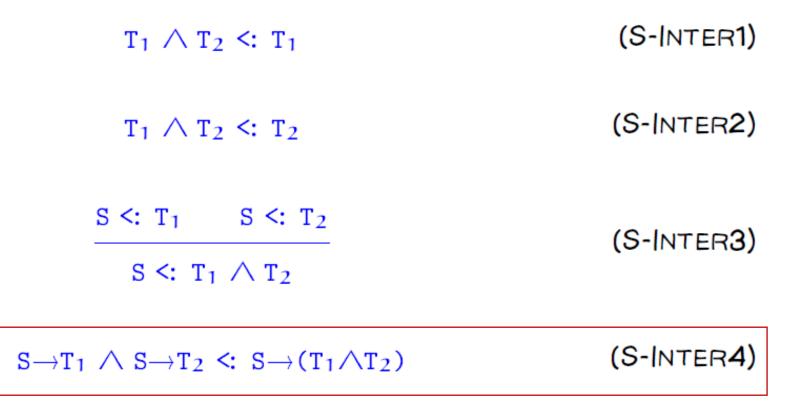
- e.g., Bool <: Nat</p>
- Bool, 0/1



Intersection and Union Types



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





Intersection types permit a very *flexible form* of *finitary overloading*.

+ : (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 (a term is typable iff its evaluation terminates)

type reconstruction problem is undecidable (cf. ch22)

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.

 $T_1 \lor T_2$ is an untagged (non-disjoint) union of T_1 and T_2 . *No tags*: no *case* construct. The only operations we can safely perform on elements of $T_1 \lor T_2$ 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* $T_1 \circ T_2$.

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)

HW for Chap15



- 15.2.2
- 15.3.2
- 15.5.2