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 2025

Part III

Chap 15:   Subtyping

Subsumption

Subtype relation

Properties of subtyping and typing

Subtyping and other features

Intersection and union types

Subtyping

Motivation

With the *usual typing rule* for applications

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

is the term

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

right?

It is not well typed

Motivation

With the usual typing rule for applications

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad (\text{T-App})$$

the term

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

is *not* well typed.

This is silly: what we're doing is passing the function *a better argument* than it needs

Subsumption

More generally: some types *are better* than others, in the sense that *a value of one* can *always safely be used* where *a value of the other* is expected

We can *formalize this intuition* by introducing:

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.,

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

Principle of safe substitution

Subtyping

Intuitions: S <: T means ...

“An element of S *may safely be used* wherever an element of T is expected” (*Official*)

- S is “*better than*” T
- S is a *subset* of T
- S is *more informative* / richer than T

Example

Back to the example：

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

with subtyping between record types, so that, for example

$$\{x{:}\,Nat, y{:}\,Nat\} <: \{x{:}\,Nat\}$$

by *subsumption*

$$\vdash \{x = 0, y = 1\} : \{x{:}\,Nat\}$$

and hence

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

is *well* typed.

Subtype Relation

The Subtype Relation: Top

It is *convenient* to have a type that is a

supertype of every type

We introduce a new *type constant* Top, plus *a rule* that makes Top a *maximum element* of the subtype relation

i.e,

$$S <: \mathtt{Top} \qquad \text{(S-TOP)}$$

Cf. Object in Java.

Subtype Relation: General rules

$$S <: S \qquad \text{(S-Refl)}$$

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

- Following directly from the intuition of safe substitution, Subtyping should be **reflexive**, and **transitive**

Subtyping
for
Record Types

The Subtype Relation: Records

"***Width subtyping***" : *forgetting fields on the right*

$$\{l_i : T_i^{i \in 1..n+k}\} <: \{l_i : T_i^{i \in 1..n}\} \qquad \text{(S-RcdWidth)}$$

The supertype *has fewer fields* than its subtypes

Intuition:

$\{x: Nat\}$ is the type of ***all records*** with *at least* a *numeric* x field

e.g.,

{x = 5} ; {x = 10}

{x = 5, y=12} ; {x =10, a = true, b = 2}

The Subtype Relation: Records

"***Width subtyping***" (forgetting fields on the right):

$$\{l_i : T_i^{\,i \in 1..n+k}\} <: \{l_i : T_i^{\,i \in 1..n}\} \qquad \text{(S-RcdWidth)}$$

Intuition:

- ***Note that*** the record type with *more* fields is a *subtype* of the record type with *fewer* fields
- **Reason**: the type with more fields places *stronger constraints* on values, so it describes *fewer values*

This rule applies only to record types where the common fields are identical

The Subtype Relation: Records

“***Depth subtyping***” within fields:

$$\frac{\text{for each } i \quad \mathtt{S}_i <: \mathtt{T}_i}{\{\mathtt{l}_i\mathtt{:S}_i\ ^{i\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \quad (\textsc{S-RcdDepth})$$

The types of *individual fields* may change, *as long as* the type of each *corresponding field* in the two records are in the *subtype relation*

Examples

- We can use these rules to infer the subtype relation between given types

$$\dfrac{\dfrac{}{\mathtt{\{a:Nat,b:Nat\}} <: \mathtt{\{a:Nat\}}}\;\textsc{S-RcdWidth} \qquad \dfrac{}{\mathtt{\{m:Nat\}} <: \mathtt{\{\}}}\;\textsc{S-RcdWidth}}{\mathtt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \mathtt{\{x:\{a:Nat\},y:\{\}\}}}\;\textsc{S-RcdDepth}$$

Examples

We can also use S-RcdDepth to refine the type of *just a single record field* (instead of refining *every field*), by using a so called S-REFL to obtain trivial **subtyping derivations** for other fields.

$$\dfrac{\dfrac{}{\{a:Nat,\ b:Nat\} <: \{a:Nat\}}\ \text{S}-\text{RCDWIDTH} \qquad \boxed{\dfrac{}{\{m:Nat\} <: \{m:Nat\}}}\ \text{S}-\text{REFL}}{\{\boxed{x:\{a:Nat,\ b:Nat\},}\ y:\{m:Nat\}\} <: \{\boxed{x:\{a:Nat\}},\ y:\{m:Nat\}\}}\ \text{S}-\text{RcdDepth}$$

Order of fields in Records

The *order of fields* in a record *doesn't make any difference* to *how we can safely use it*, since the only thing that we can do with records (*projecting their fields*) is *insensitive* to *the order of fields*

S-RcdPerm tells us that

{c:Top, b: Bool, a: Nat} <: {a: Nat, b: Bool, c:Top}

and

{a: Nat, b: Bool, c:Top} <: {c:Top, b: Bool, a: Nat}

The Subtype Relation: Records

Permutation of fields:

$$\frac{\{\mathtt{k}_j\mathtt{:}\mathtt{S}_j\ ^{j\in 1..n}\}\text{ is a permutation of }\{\mathtt{l}_i\mathtt{:}\mathtt{T}_i\ ^{i\in 1..n}\}}{\{\mathtt{k}_j\mathtt{:}\mathtt{S}_j\ ^{j\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:}\mathtt{T}_i\ ^{i\in 1..n}\}}\ (\text{S-RcdPerm})$$

Using S-RcdPerm together with S-RcdWidth & S-Trans allows us to *drop arbitrary fields* within records

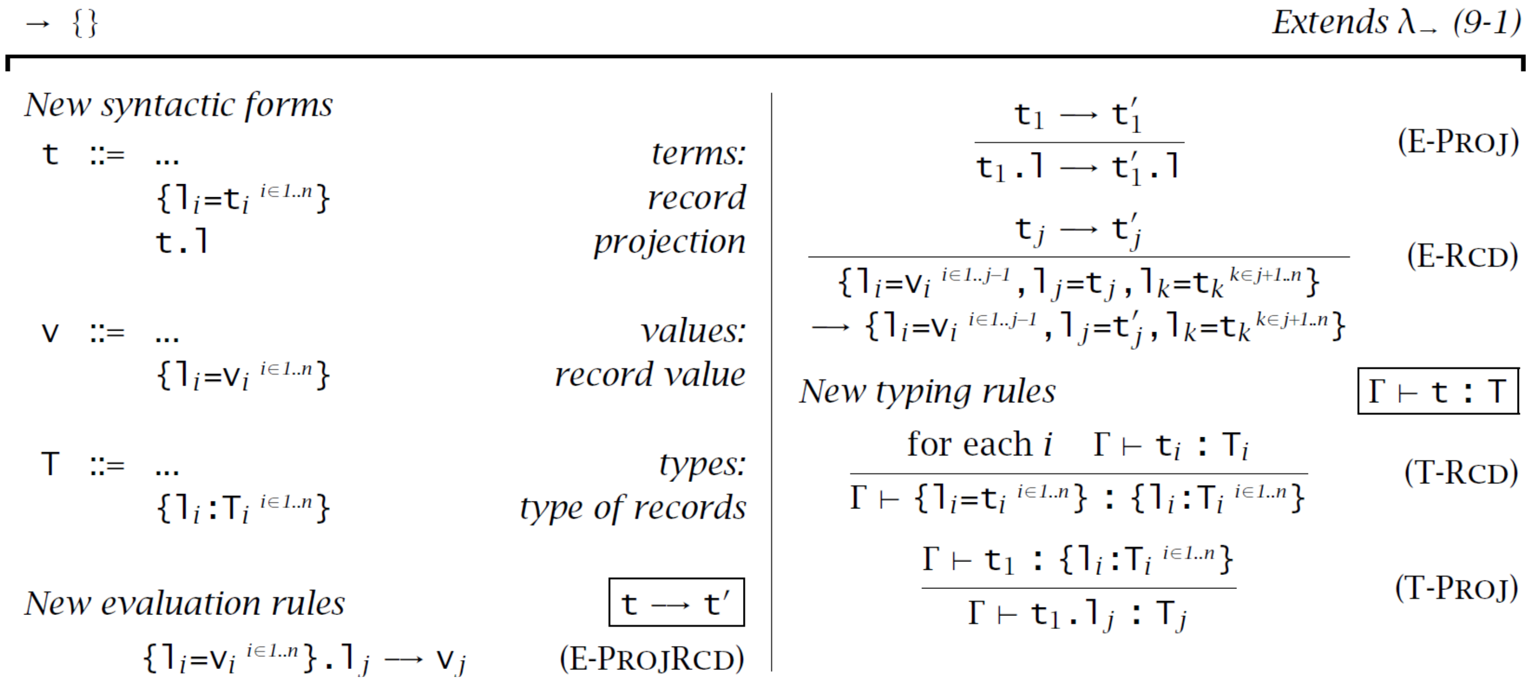
Variations

Real languages often choose *not to adopt all of these record subtyping rules,* e.g., in Java,

- A subclass may not change the argument or result types of a method of its superclass (i.e., *no depth subtyping*)
- Each class has just one superclass ("*single inheritance*" of classes)

  each class member (field or method) can be assigned a single index, adding new indices "on the right" as more members are added in subclasses (i.e., no permutation for classes)
- A class may implement multiple interfaces ("*multiple inheritance*" of interfaces) (i.e., *permutation* is allowed for *interfaces*)

Recap for subtyping



→ {} *Extends* $\lambda_{\to}$ *(9-1)*

New syntactic forms

$\mathtt{t}$	::=	...	*terms:*
		$\{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\}$	*record*
		$\mathtt{t.l}$	*projection*
$\mathtt{v}$	::=	...	*values:*
		$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}$	*record value*
$\mathtt{T}$	::=	...	*types:*
		$\{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}$	*type of records*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}.\mathtt{l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjRcd)}$$

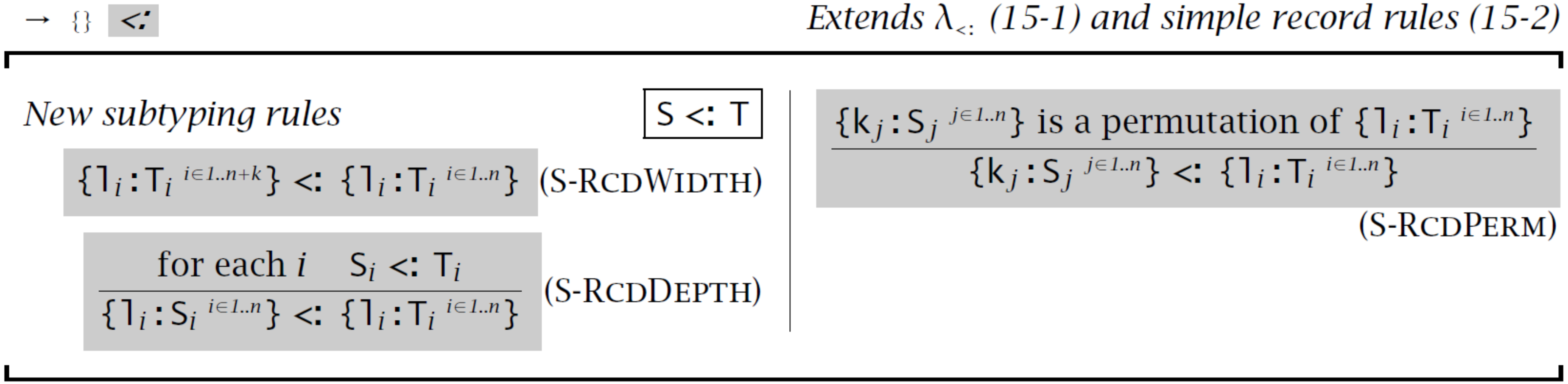
$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1.\mathtt{l} \longrightarrow \mathtt{t}_1'.\mathtt{l}} \qquad \text{(E-Proj)}$$

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\begin{array}{l}\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1},\mathtt{l}_j\mathtt{=t}_j,\mathtt{l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\}\\ \longrightarrow \{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1},\mathtt{l}_j\mathtt{=t}_j',\mathtt{l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\}\end{array}} \qquad \text{(E-Rcd)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\text{for each } i \quad \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1.\mathtt{l}_j : \mathtt{T}_j} \qquad \text{(T-Proj)}$$



→ {} <: *Extends* $\lambda_{<:}$ *(15-1) and simple record rules (15-2)*

New subtyping rules $\boxed{\mathtt{S} <: \mathtt{T}}$

$$\{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n+k}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\} \qquad \text{(S-RcdWidth)}$$

$$\frac{\text{for each } i \quad \mathtt{S}_i <: \mathtt{T}_i}{\{\mathtt{l}_i\mathtt{:S}_i\ ^{i\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(S-RcdDepth)}$$

$$\frac{\{\mathtt{k}_j\mathtt{:S}_j\ ^{j\in 1..n}\} \text{ is a permutation of } \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}}{\{\mathtt{k}_j\mathtt{:S}_j\ ^{j\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(S-RcdPerm)}$$

Subtyping
for
functional types

The Subtype Relation: Arrow types

A high-order language, *functions can be passed as arguments to other functions*

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

The Subtype Relation: Arrow types

Note the *order* of T_1 and S_1 in the first premise.

The subtype relation is

- *contravariant* in the left-hand sides of arrows
- *covariant* in the right-hand sides of arrows

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad (\text{S-Arrow})$$

The Subtype Relation: Arrow types

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

Intuition: if we have a function **f** of type $S_1 \longrightarrow S_2$,

1. **f** accepts elements of type S_1; clearly, **f** will also accept elements of any subtype T_1 of S_1
2. the type of **f** also tells us that it returns elements of type S_2; then these results can be viewed as belonging to any supertype T_2 of S_2

i.e., any function **f** of **type** $S_1 \longrightarrow S_2$ can also be viewed as having **type** $T_1 \longrightarrow T_2$

Recap for subtyping

→ <: Top Based on *λ→ (9-1)*

Syntax

t	::=		*terms:*
		x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
v	::=		*values:*
		λx:T.t	*abstraction value*
T	::=		*types:*
		Top	*maximum type*
		T→T	*type of functions*
Γ	::=		*contexts:*
		∅	*empty context*
		Γ, x:T	*term variable binding*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

Subtyping $\boxed{S <: T}$

$$S <: S \quad \text{(S-REFL)}$$

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

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

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

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

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

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

Subtype Relation

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

$$\mathtt{S} <: \mathtt{S} \qquad (\textsc{S-Refl})$$

$$\frac{\mathtt{S} <: \mathtt{U} \qquad \mathtt{U} <: \mathtt{T}}{\mathtt{S} <: \mathtt{T}} \qquad (\textsc{S-Trans})$$

$$\{\mathtt{l}_i\mathtt{:T}_i{}^{i \in 1..n+k}\} <: \{\mathtt{l}_i\mathtt{:T}_i{}^{i \in 1..n}\} \qquad (\textsc{S-RcdWidth})$$

$$\frac{\text{for each } i \qquad \mathtt{S}_i <: \mathtt{T}_i}{\{\mathtt{l}_i\mathtt{:S}_i{}^{i \in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i{}^{i \in 1..n}\}} \qquad (\textsc{S-RcdDepth})$$

$$\frac{\{\mathtt{k}_j\mathtt{:S}_j{}^{j \in 1..n}\} \text{ is a permutation of } \{\mathtt{l}_i\mathtt{:T}_i{}^{i \in 1..n}\}}{\{\mathtt{k}_j\mathtt{:S}_j{}^{j \in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i{}^{i \in 1..n}\}} \qquad (\textsc{S-RcdPerm})$$

$$\frac{\mathtt{T}_1 <: \mathtt{S}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1 \to \mathtt{S}_2 <: \mathtt{T}_1 \to \mathtt{T}_2} \qquad (\textsc{S-Arrow})$$

$$\mathtt{S} <: \mathtt{Top} \qquad (\textsc{S-Top})$$

Properties of Subtyping

Safety

Statements of progress and preservation theorems are ***unchanged*** from $\lambda_{\rightarrow}$.

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 following rule could appear anywhere

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

Aside: Syntax-directed rules

When we say a set of rules is s*yntax-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 $U <: T_1 \longrightarrow T_2$, then U has the form $U_1 \longrightarrow U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.

Proof: *By induction on subtyping derivations.*

Case S-ARROW: $U = U_1 \longrightarrow U_2 \qquad T_1 <: U_1,\ U_2 <: T_2$

Immediate.

Case S-REFL: $U = T_1 \longrightarrow T_2$

- By S-REFL (twice), $T_1 <: T_1$ and $T_2 <: T_2$, as required.

Case S-TRANS: $U <: W \qquad W <: T_1 \longrightarrow T_2$

- Applying the IH to the second subderivation, we find that W has the form $W_1 \longrightarrow W_2$, 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 $U_1 \longrightarrow U_2$, 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.

Inversion Lemma for Typing

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: $T_1 = S_1,\ T_2 = S_2 \qquad \Gamma, x{:}\,S_1 \vdash s_2 : S_2$

Case T-SUB: $\Gamma \vdash \lambda x{:}S_1.\,s_2 : U \qquad U : T_1 \longrightarrow T_2$

- By the subtyping inversion lemma, U has the form of $U_1 \longrightarrow U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.
- The IH now applies, yielding $U_1 <: S_1$ and $\Gamma, x{:}S_1 \vdash s_2 : U_2$.
- From $U_1 <: S_1$ and $T_1 <: U_1$, rule S-Trans gives $T_1 <: S_1$.
- From $\Gamma, x{:}S_1 \vdash s_2 : U_2$ and $U_2 <: T_2$, rule T-Sub gives $\Gamma, x{:}\,S_1 \vdash s_2 : T_2$, thus we are done

Preservation

Theorem: If $\Gamma \vdash \mathrm{t} : \mathrm{T}$ *and* $\mathrm{t} \longrightarrow \mathrm{t}'$, *then* $\Gamma \vdash \mathrm{t}' : \mathrm{T}$.

Proof: By induction on *typing derivations*.

Which cases are likely to be *hard* ?

Preservation - Subsumption case

Case T-SUB: $\mathrm{t} : \mathrm{S} \quad \mathrm{S} <: \mathrm{T}$

By the induction hypothesis, $\Gamma \vdash \mathrm{t}' : \mathrm{S}$.

By T-SUB, $\Gamma \vdash \mathrm{t}' : \mathrm{T}$.

Not hard!

Preservation - Application case

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 $t \longrightarrow t'$ can be derived:

E-APP1, E-APP2, and E-APPABS.

Proceed by cases

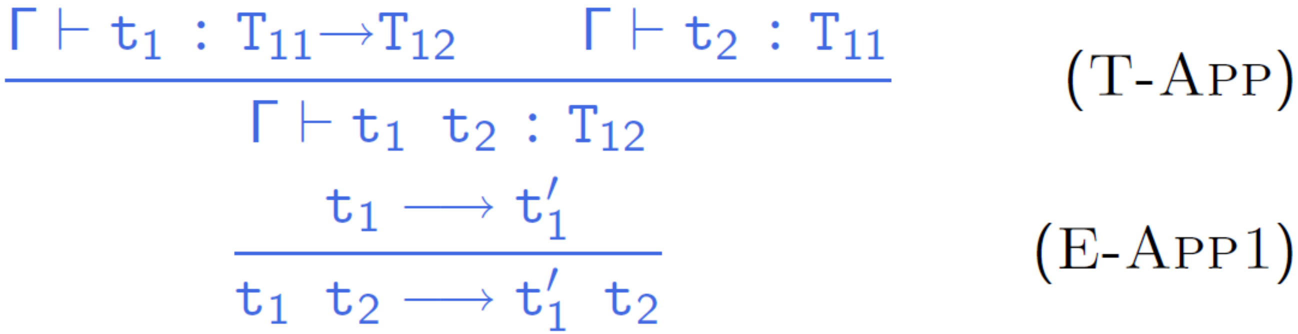
Preservation - Application case

Case T-APP :

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

Subcase E-APP1 : $t_1 \longrightarrow t'_1 \qquad t' = t'_1\ t_2$

The result follows from **the induction hypothesis** and T-APP



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

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

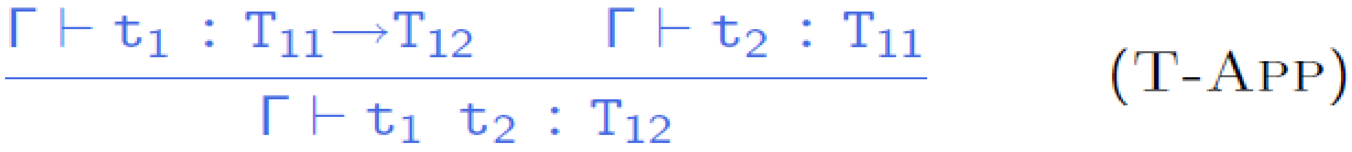
Preservation - Application case

Case T-APP :

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

Subcase E-APP2 : $t_1 = v_1 \quad t_2 \longrightarrow t'_2 \quad t' = v_1 \ t'_2$

Similar.



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

$$\frac{t_2 \longrightarrow t'_2}{v_1 \ t_2 \longrightarrow v_1 \ t'_2} \qquad \text{(E-App2)}$$

Preservation - Application case

Case T-APP :

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

Subcase E-APPABS :

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

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

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

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-APPABS)}$$

Progress

Lemma for Canonical Forms

1. If v is a closed value of type $T_1 \longrightarrow T_2$, 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

$$\{k_j = v_j^{j\in 1..m}\} \text{ with } \{l_i^{i\in 1..n}\} \subseteq \{k_a^{a\in 1..m}\}$$

- *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 lambda-calculus

Subtyping with Other Features

Ascription and Casting

Ordinary ascription:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}}{\Gamma \vdash \mathtt{t}_1 \ \mathtt{as}\ \mathtt{T} : \mathtt{T}} \qquad \text{(T-Ascribe)}$$

$$\mathtt{v}_1 \ \mathtt{as}\ \mathtt{T} \longrightarrow \mathtt{v}_1 \qquad \text{(E-Ascribe)}$$

In languages with subtyping (e.g., Java/ C++), it is often called casting, and written as

(T) t

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{S}}{\Gamma \vdash \mathtt{t}_1 \ \mathtt{as}\ \mathtt{T} : \mathtt{T}} \qquad \text{(T-Downcast)}$$

up-cast : a term is ascribed a supertype of the type

down-cast: to assign types to terms that the typechecker cannot derive statically, and need to involve dynamic type-testing

Ascription and Casting

Ordinary ascription:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}}{\Gamma \vdash \mathtt{t}_1 \ \mathtt{as}\ \mathtt{T} : \mathtt{T}} \qquad \text{(T-Ascribe)}$$

$$\mathtt{v}_1 \ \mathtt{as}\ \mathtt{T} \longrightarrow \mathtt{v}_1 \qquad \text{(E-Ascribe)}$$

Casting (cf. Java):

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{S}}{\Gamma \vdash \mathtt{t}_1 \ \mathtt{as}\ \mathtt{T} : \mathtt{T}} \qquad \text{(T-Cast)}$$

$$\frac{\vdash \mathtt{v}_1 : \mathtt{T}}{\mathtt{v}_1 \ \mathtt{as}\ \mathtt{T} \longrightarrow \mathtt{v}_1} \qquad \text{(E-Cast)}$$

Subtyping and Variants

$$\boxed{\langle \mathtt{l}_i : \mathtt{T}_i \ {}^{i\in 1..n}\rangle \quad <: \quad \langle \mathtt{l}_i : \mathtt{T}_i \ {}^{i\in 1..n+k}\rangle \qquad \text{(S-}\textsc{VariantWidth}\text{)}}$$

$$\frac{\text{for each } i \qquad \mathtt{S}_i <: \mathtt{T}_i}{\langle \mathtt{l}_i : \mathtt{S}_i \ {}^{i\in 1..n}\rangle \quad <: \quad \langle \mathtt{l}_i : \mathtt{T}_i \ {}^{i\in 1..n}\rangle} \qquad \text{(S-}\textsc{VariantDepth}\text{)}$$

$$\frac{\langle \mathtt{k}_j : \mathtt{S}_j \ {}^{j\in 1..n}\rangle \text{ is a permutation of } \langle \mathtt{l}_i : \mathtt{T}_i \ {}^{i\in 1..n}\rangle}{\langle \mathtt{k}_j : \mathtt{S}_j \ {}^{j\in 1..n}\rangle \quad <: \quad \langle \mathtt{l}_i : \mathtt{T}_i \ {}^{i\in 1..n}\rangle} \qquad \text{(S-}\textsc{VariantPerm}\text{)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \langle \mathtt{l}_1 = \mathtt{t}_1\rangle : \langle \mathtt{l}_1 : \mathtt{T}_1\rangle} \qquad \text{(T-}\textsc{Variant}\text{)}$$

Subtyping and Lists

List is a covariant type constructor

$$\frac{S_1 <: T_1}{\texttt{List}\ S_1 <: \texttt{List}\ T_1} \qquad (\textsc{S-List})$$

Subtyping and References

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

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\mathtt{Ref}\ S_1 <: \mathtt{Ref}\ T_1} \qquad \text{(S-REF)}$$

Subtyping and References

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

Why?

- When a reference is *read*, the context expects a T_1, so if $\mathrm{S}_1 <: \mathrm{T}_1$ 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 $\mathrm{Ref}\ \mathrm{S}_1$, someone else may use the T_1 as an S_1. So we need $\mathrm{T}_1 <: \mathrm{S}_1$.

References again

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

References again

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

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Source}\ \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{!t}_1 : \mathtt{T}_{11}} \qquad (\text{T-Deref})$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Sink}\ \mathtt{T}_{11} \qquad \Gamma \mid \Sigma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{t}_1\mathtt{:=}\mathtt{t}_2 : \mathtt{Unit}} \qquad (\text{T-Assign})$$

Subtyping rules

$$\frac{\mathtt{S}_1 \mathrel{<:} \mathtt{T}_1}{\mathtt{Source}\ \mathtt{S}_1 \mathrel{<:} \mathtt{Source}\ \mathtt{T}_1} \qquad (\textsc{S-Source})$$

$$\frac{\mathtt{T}_1 \mathrel{<:} \mathtt{S}_1}{\mathtt{Sink}\ \mathtt{S}_1 \mathrel{<:} \mathtt{Sink}\ \mathtt{T}_1} \qquad (\textsc{S-Sink})$$

$$\mathtt{Ref}\ \mathtt{T}_1 \mathrel{<:} \mathtt{Source}\ \mathtt{T}_1 \qquad (\textsc{S-RefSource})$$

$$\mathtt{Ref}\ \mathtt{T}_1 \mathrel{<:} \mathtt{Sink}\ \mathtt{T}_1 \qquad (\textsc{S-RefSink})$$

Subtyping and Arrays

Similarly...

$$\frac{\mathtt{S}_1 \mathrel{\mathtt{<:}} \mathtt{T}_1 \qquad \mathtt{T}_1 \mathrel{\mathtt{<:}} \mathtt{S}_1}{\mathtt{Array}\ \mathtt{S}_1 \mathrel{\mathtt{<:}} \mathtt{Array}\ \mathtt{T}_1} \qquad (\textsc{S-Array})$$

$$\frac{\mathtt{S}_1 \mathrel{\mathtt{<:}} \mathtt{T}_1}{\mathtt{Array}\ \mathtt{S}_1 \mathrel{\mathtt{<:}} \mathtt{Array}\ \mathtt{T}_1} \qquad (\textsc{S-ArrayJava})$$

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
- ...

Base Types

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

- e.g., in many languages the boolean values true and false are actually represented by the numbers 1 and 0.
- Bool <: Nat
- if b then 5 else 0 => 5*b

Intersection and Union Types

Intersection 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.

$$T_1 \wedge T_2 <: T_1 \qquad \text{(S-INTER1)}$$

$$T_1 \wedge T_2 <: T_2 \qquad \text{(S-INTER2)}$$

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

$$\boxed{S \to T_1 \wedge S \to T_2 <: S \to (T_1 \wedge T_2) \qquad \text{(S-INTER4)}}$$

Intersection Types

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

$$+ : (\mathtt{Nat}\to\mathtt{Nat}\to\mathtt{Nat}) \wedge (\mathtt{Float}\to\mathtt{Float}\to\mathtt{Float})$$

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

Union types are also useful.

$T_1 \vee T_2$ 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 $T_1 \vee T_2$ are ones *that make sense for both* T_1 *and* T_2.

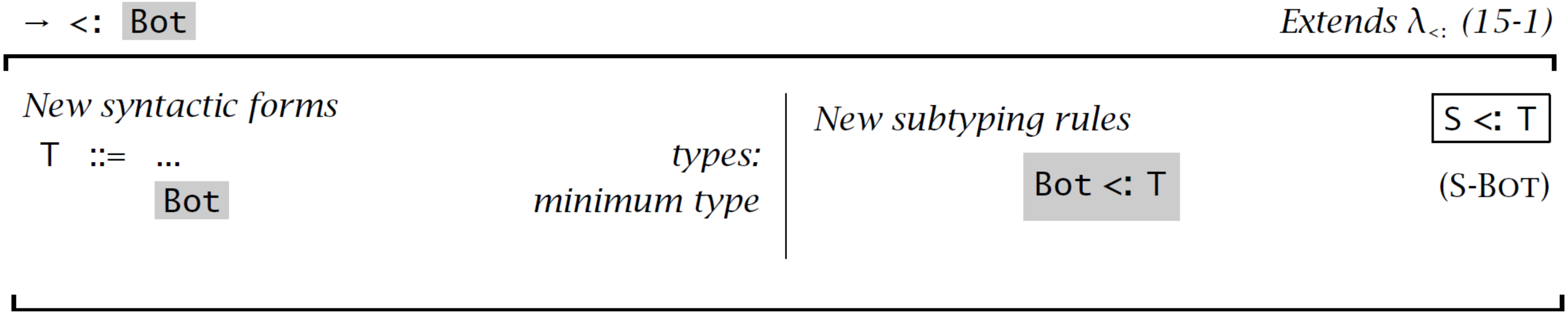
Note well: untagged union types in C are a source of *type safety violations* precisely because they ignores this restriction, allowing any operation on an element of $T_1 \vee T_2$ that makes sense for *either* T_1 *or* T_2.

Union types are being used recently in type systems for XML processing languages (cf. Xduce, Xtatic).

Bottom Type

Can we have a type that is a *subtype of every type* ?

Sure. a *type constant* Bot, plus *a rule* that makes Bot a *minimal element* of the subtype relation



→ <: Bot Extends $\lambda_{<:}$ (15-1)

New syntactic forms

T ::= ... *types:*

Bot *minimum type*

New subtyping rules $\boxed{S <: T}$

Bot <: T (S-Bot)

Bot is empty—there are no closed values of type Bot.

The emptiness of Bot provides a very convenient way of expressing the fact that some operations are not intended to return

Varieties of Polymorphism

- Parametric polymorphism (ML-style)
- Subtype polymorphism (OO-style)
- Ad-hoc polymorphism (overloading)

HW for Chap15

- 15.2.2
- 15.3.6