



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 2023



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

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

is the term

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

right?

It is **not** well typed



Motivation

With the usual typing rule for applications

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

the term

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

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.,

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

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 :

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

We will **define subtyping** between **record types**, so that, for example

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

by *subsumption*,

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

and hence

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

is *well* typed.



The Subtype Relation: Records

“*Width subtyping*” : forgetting fields on the right

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

Intuition:

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



The Subtype Relation: Records

“*Width subtyping*” (forgetting fields on the right):

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

Intuition:

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



The Subtype Relation: Records

“*Depth subtyping*” within fields:

$$\frac{\text{for each } i \quad S_i <: T_i}{\{l_j : S_j \mid i \in 1..n\} <: \{l_j : T_j \mid i \in 1..n\}} \quad (\text{S-RCDDEPTH})$$

- 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



$\frac{}{\{a:\text{Nat}, b:\text{Nat}\} <: \{a:\text{Nat}\}} \text{S-RCDWIDTH}$ $\frac{}{\{m:\text{Nat}\} <: \{\}} \text{S-RCDWIDTH}$

$\frac{}{\{x:\{a:\text{Nat}, b:\text{Nat}\}, y:\{m:\text{Nat}\}\} <: \{x:\{a:\text{Nat}\}, y:\{\}} \text{S-RCDDEPTH}$

Examples

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

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



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

$$\{c:\text{Top}, b:\text{Bool}, a:\text{Nat}\} <: \{a:\text{Nat}, b:\text{Bool}, c:\text{Top}\}$$

and

$$\{a:\text{Nat}, b:\text{Bool}, c:\text{Top}\} <: \{c:\text{Top}, b:\text{Bool}, a:\text{Nat}\}$$



The Subtype Relation: Records

Permutation of fields:

$$\frac{\{k_j : S_j^{j \in 1..n}\} \text{ is a permutation of } \{l_i : T_i^{i \in 1..n}\}}{\{k_j : S_j^{j \in 1..n}\} <: \{l_i : T_i^{i \in 1..n}\}} \quad (\text{S-RCDPERM})$$

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*)



The Subtype Relation: Arrow types

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

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



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

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

The Subtype Relation: Arrow types

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

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

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 $S_1 \rightarrow S_2$ can also be viewed as having type $T_1 \rightarrow T_2$



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,

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

Cf. **Object** in Java.



Subtype Relation: General rules

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

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

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

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



Subtype Relation

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

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

$$\{l_i : T_i \mid i \in 1..n+k\} <: \{l_i : T_i \mid i \in 1..n\} \quad (\text{S-RCDWIDTH})$$

$$\frac{\text{for each } i \quad S_i <: T_i}{\{l_i : S_i \mid i \in 1..n\} <: \{l_i : T_i \mid i \in 1..n\}} \quad (\text{S-RCDDEPTH})$$

$$\frac{\{k_j : S_j \mid j \in 1..n\} \text{ is a permutation of } \{l_i : T_i \mid i \in 1..n\}}{\{k_j : S_j \mid j \in 1..n\} <: \{l_i : T_i \mid i \in 1..n\}} \quad (\text{S-RCDPERM})$$

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

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



Properties of Subtyping



Safety

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.

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



Aside: Syntax-directed rules

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 $U <: T_1 \rightarrow T_2$, 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: $U <: W$ $W <: T_1 \rightarrow T_2$
– 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.

Inversion Lemma for Typing

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

Proof: Induction on typing derivations.

Case T-Abs: $T_1 = S_1$, $T_2 = S_2$ $\Gamma, x: S_1 \vdash s_2: S_2$

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 $U_1 <: S_1$ 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 $\Gamma \vdash t : T$ 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: $t : S \quad S <: T$

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

By T-Sub, $\Gamma \vdash t' : T$.

Not hard!



Preservation - Application case

Case T-App :

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

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



Preservation - Application case

Case T-App :

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

Subcase E-App1 : $t_1 \rightarrow t'_1 \quad t' = t'_1 t_2$

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

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

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

Preservation - Application case

Case T-App :

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

Subcase E-App2 : $t_1 = v_1 \quad t_2 \longrightarrow t'_2 \quad t' = v_1 t'_2$

Similar.

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

$$\frac{t_2 \longrightarrow t'_2}{v_1 t_2 \longrightarrow v_1 t'_2} \quad (\text{E-APP2})$$



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-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 ...

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

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

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

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

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

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 have **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:

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \quad (\text{T-ASCRIBE})$$

$$v_1 \text{ as } T \longrightarrow v_1 \quad (\text{E-ASCRIBE})$$

(T) T

up-cast

down-cast

Ascription and Casting

Ordinary ascription:

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \quad (\text{T-ASCRIBE})$$

$$v_1 \text{ as } T \longrightarrow v_1 \quad (\text{E-ASCRIBE})$$

Casting (cf. Java):

$$\frac{\Gamma \vdash t_1 : S}{\Gamma \vdash t_1 \text{ as } T : T} \quad (\text{T-CAST})$$

$$\frac{\vdash v_1 : T}{v_1 \text{ as } T \longrightarrow v_1} \quad (\text{E-CAST})$$

Subtyping and Variants



$$\langle l_i : T_i^{i \in 1..n} \rangle <: \langle l_i : T_i^{i \in 1..n+k} \rangle \quad (\text{S-VARIANTWIDTH})$$

$$\frac{\text{for each } i \quad S_i <: T_i}{\langle l_i : S_i^{i \in 1..n} \rangle <: \langle l_i : T_i^{i \in 1..n} \rangle} \quad (\text{S-VARIANTDEPTH})$$

$$\frac{\langle k_j : S_j^{j \in 1..n} \rangle \text{ is a permutation of } \langle l_i : T_i^{i \in 1..n} \rangle}{\langle k_j : S_j^{j \in 1..n} \rangle <: \langle l_i : T_i^{i \in 1..n} \rangle} \quad (\text{S-VARIANTPERM})$$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \langle l_1 = t_1 \rangle : \langle l_1 : T_1 \rangle} \quad (\text{T-VARIANT})$$



Subtyping and Lists

List is a *covariant type* constructor

$$\frac{S_1 <: T_1}{\text{List } S_1 <: \text{List } T_1} \quad (\text{S-LIST})$$



Subtyping and References

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

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



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 $S_1 <: T_1$ 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 S_1` , someone else may use the T_1 as an S_1 . So we need $T_1 <: S_1$.



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



$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Source } T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}} \quad (\text{T-DEREF})$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Sink } T_{11} \quad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}} \quad (\text{T-ASSIGN})$$

Subtyping rules



$$\frac{S_1 <: T_1}{\text{Source } S_1 <: \text{Source } T_1} \quad (\text{S-SOURCE})$$

$$\frac{T_1 <: S_1}{\text{Sink } S_1 <: \text{Sink } T_1} \quad (\text{S-SINK})$$

$$\text{Ref } T_1 <: \text{Source } T_1 \quad (\text{S-REFSOURCE})$$

$$\text{Ref } T_1 <: \text{Sink } T_1 \quad (\text{S-REFSINK})$$

Subtyping and Arrays

Similarly...

$$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Array } S_1 <: \text{Array } T_1} \quad (\text{S-ARRAY})$$

$$\frac{S_1 <: T_1}{\text{Array } S_1 <: \text{Array } T_1} \quad (\text{S-ARRAY.JAVA})$$

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`



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 .

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

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

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

$$S \rightarrow T_1 \wedge S \rightarrow T_2 <: S \rightarrow (T_1 \wedge T_2) \quad (\text{S-INTER4})$$



Intersection Types

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

$$+ : (\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}) \wedge (\text{Float} \rightarrow \text{Float} \rightarrow \text{Float})$$

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

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).



Varieties of Polymorphism

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



HW for Chap15

- 15.2.2
- 15.3.2
- 15.3.6