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 2026

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

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

is the term

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

right?

It is not well typed

Motivation

With the usual typing rule for applications

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

the term

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

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.,

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

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

by introducing 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:

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

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

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

$$\frac{}{\texttt{\{a:Nat,b:Nat\}} <: \texttt{\{a:Nat\}}}\ \text{S-RcdWidth} \qquad \frac{}{\texttt{\{m:Nat\}} <: \texttt{\{\}}}\ \text{S-RcdWidth}$$

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

Examples

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

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

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: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:

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

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

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

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

$$\{l_i{=}v_i\ ^{i\in 1..n}\}.l_j \longrightarrow v_j \quad \text{(E-ProjRcd)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.l \longrightarrow t_1'.l} \quad \text{(E-Proj)}$$

$$\frac{t_j \longrightarrow t_j'}{\{l_i{=}v_i\ ^{i\in 1..j-1}, l_j{=}t_j, l_k{=}t_k\ ^{k\in j+1..n}\} \longrightarrow \{l_i{=}v_i\ ^{i\in 1..j-1}, l_j{=}t_j', l_k{=}t_k\ ^{k\in j+1..n}\}} \quad \text{(E-Rcd)}$$

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

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i{=}t_i\ ^{i\in 1..n}\} : \{l_i{:}T_i\ ^{i\in 1..n}\}} \quad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash t_1 : \{l_i{:}T_i\ ^{i\in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j} \quad \text{(T-Proj)}$$

→ {} *<:*

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

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

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

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

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

Subtyping for functional types

The Subtype Relation: Arrow types

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

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

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

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

The Subtype Relation: Arrow types

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

Intuition: if we have a function $\mathbf{f}$ of type $S_1 \to S_2$,

1. $\mathbf{f}$ accepts elements of type S_1; clearly, $\mathbf{f}$ will also accept elements of any subtype T_1 of S_1
2. the type of $\mathbf{f}$ 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 $\mathbf{f}$ of **type** $S_1 \to S_2$ can also be viewed as having **type** $T_1 \to T_2$

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 $t \longrightarrow t'$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

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

Subtyping $S <: T$

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

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

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

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

Typing $\Gamma \vdash t : T$

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

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

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

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

Subtype Relation

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

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

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

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

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

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

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

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

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

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

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 $U <: T_1 \to T_2$, then U has the form $U_1 \to U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.

Proof: *By induction on subtyping derivations.*

Case S-ARROW: $U = U_1 \to U_2 \quad T_1 <: U_1,\ U_2 <: T_2$

Immediate.

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

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

Case S-TRANS: $U <: W \quad W <: T_1 \to T_2$

- Applying the IH to the second subderivation, we find that W has the form $W_1 \to W_2$, 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 $U_1 \to U_2$, 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: $T_1 = S_1,\ T_2 = S_2 \quad \Gamma,\ x{:}\ S_1 \vdash s_2{:}\ S_2$

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

- 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 $\Gamma \vdash t : T$ *and* $t \longrightarrow t'$, *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 :

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

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 :

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

Subcase E-APP1 : $t_1 \to t'_1 \quad t' = t'_1\ t_2$

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

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

$$\frac{t_1 \longrightarrow t'_1}{t_1\ t_2 \longrightarrow t'_1\ t_2} \quad (\text{E-App1})$$

Preservation – Application case

Case T-APP :

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

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

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 :

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

Subcase E-APPABS :

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

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

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

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

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

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

- *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:

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \qquad \text{(T-Ascribe)}$$

$$v_1 \text{ as } T \longrightarrow v_1 \qquad \text{(E-Ascribe)}$$

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

(T) t

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

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

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

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

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

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

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

Subtyping and Lists

List is a covariant type constructor

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

Subtyping and References

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

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

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 (as a capability to read values of type T from a cell), and
- as a *sink* for values of type T (a capability to write to a cell)

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 $\mathrm{Ref}\ T$ into three parts:

- $\mathrm{Source}\ T$: reference cell with "read capability"
- $\mathrm{Sink}\ T$: reference cell with "write capability"
- $\mathrm{Ref}\ T$: cell with both capabilities

Modified Typing Rules

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

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

Subtyping rules

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

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

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

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

Subtyping and Arrays

Similarly...

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

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

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.

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

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

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

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

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 ::= ...
Bot

types:
minimum type

New subtyping rules

Bot <: T

S <: 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)

Homework☺

- Read and digest chapter 15.

- HW:
  - 15.2.5
  - 15.5.3