Part III
Chapter 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

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

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

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

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:

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 $S <: T$

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.,

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

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*)

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, we will define subtyping between record types so that, for example

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

by *subsumption*,

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

Example

We will define 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

```
(λr:{x:Nat}. r.x) {x=0,y=1}
```

is *well* typed.

The Subtype Relation: Records

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

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

Intuition:

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

The Subtype Relation: Records

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

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

Intuition:

$\{x : Nat\}$ 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:

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

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

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

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

Examples

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

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

Order of fields in Records

The order of fields in a record does *not 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:

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

By using S-RcdPerm together with S-RcdWidth and S-Trans allows us to *drop arbitrary fields* within records.

Variations

Real languages often choose *not to adopt all of these record subtyping rules*. For example, 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

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

The Subtype Relation: Arrow types

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

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 and *covariant* in the right-hand sides.

The Subtype Relation: Arrow types

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

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 and *covariant* in the right-hand sides.

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

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; we can also view these results 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 $T_1 \longrightarrow T_2$.

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.

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

The Subtype Relation: Top

IIt 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.

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

Cf. Object in Java.

Subtype Relation: General rules

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

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

Subtype Relation: General rules

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

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

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

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

Subtype Relation

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

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

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

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

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

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

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

Properties

of

Subtyping

Safety

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

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 rule T-SUB could appear anywhere.

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

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.

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* ?

Subsumption case

Case T-Sub: $t : S \qquad S <: T$

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

Not hard!

Application case

Case T-App :

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

By the inversion lemma for evaluation, there are

three rules

by which $\mathrm{t} \longrightarrow \mathrm{t}'$ can be derived:

E-App1, E-App2, and E-AppAbs .

Proceed by cases.

Application case

Case T-App :

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

By the inversion lemma for evaluation, there are *three rules* by which $\mathsf{t} \longrightarrow \mathsf{t}'$ can be derived:

E-App1, E-App2, and E-AppAbs.

Proceed by cases.

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

Application case

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-App1: $t_1 \longrightarrow t'_1$ $t' = t'_1 \ t_2$

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

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

Application case

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 : $t_1 = v_1 \quad t_2 \longrightarrow t'_2 \quad t' = v_1 \; t'_2$

Similar.

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

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

Application case

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

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

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

Inversion Lemma for Typing

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

Inversion Lemma for Typing

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

Proof: *Induction on typing derivations*.

Case T–Sub: $\lambda x{:}S_1.s_2{:}U$ $U{:}\ T_1 \longrightarrow T_2$

We want to say "By the induction hypothesis...", but the IH does not apply (*since we do not know that U is an arrow type*).

Need another lemma...

Lemma: If $U <: T_1 \longrightarrow T_2$, then U has the form of $U_1 \longrightarrow U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.

(*Proof*: by *induction on subtyping derivations*.)

Inversion Lemma for Typing

By this lemma, we know

$$U = U_1 \longrightarrow U_2, \text{ with } T_1 <: U_1 \text{ and } U_2 <: T_2.$$

The IH now applies, yielding

$$U_1 <: S_1 \quad \text{and} \quad \Gamma, x{:}\, S_1 \vdash s_2 : U_2.$$

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

$$\Gamma, x{:}\, S_1 \vdash s_2 : T_2,$$

and we are done.

Subtyping
with
Other Features

Ascription and Casting

Ordinary ascription:

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

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

(T) T

up-cast

down-cast

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

$$\texttt{<}\texttt{l}_i\texttt{:}\texttt{T}_i\ ^{i\in 1..n}\texttt{>} \quad <: \quad \texttt{<}\texttt{l}_i\texttt{:}\texttt{T}_i\ ^{i\in 1..n+k}\texttt{>} \qquad \text{(S-VariantWidth)}$$

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

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

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1}{\Gamma \vdash \texttt{<}\texttt{l}_1\texttt{=}\texttt{t}_1\texttt{>} : \texttt{<}\texttt{l}_1\texttt{:}\texttt{T}_1\texttt{>}} \qquad \text{(T-Variant)}$$

Subtyping and Lists

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

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

Subtyping and References

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

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

Subtyping and References

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 $\mathrm{S}_1 <: \mathrm{T}_1$ then an S_1 is ok.

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 $\mathrm{S}_1 <: \mathrm{T}_1$ then an S_1 is ok.

Subtyping and References

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

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 $S_1 <: T_1$ 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 $\text{Ref } S_1$, someone else may use the T_1 as an S_1. So we need $T_1 <: S_1$.

Subtyping and Arrays

Similarly...

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

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

This is regarded (even by the Java designers) as a mistake in the design.

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 $\mathrm{Ref\ T}$ 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

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

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

Subtyping rules

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

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

$$\mathtt{Ref\ T_1} <: \mathtt{Source\ T_1} \qquad \text{(S-RefSource)}$$

$$\mathtt{Ref\ T_1} <: \mathtt{Sink\ T_1} \qquad \text{(S-RefSink)}$$

Capabilities

Other kinds of capabilities can be treated similarly, e.g.,

- *send* and *receive* capabilities on communication channels,
- *encrypt*/*decrypt* capabilities of cryptographic keys,
- ...

Intersection and Union Types

Intersection 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.

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

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

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

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

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.

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

- 15.3.2
- 15.3.6