Foundations of Software
Fall 2021

Week 10

Subtyping

Motivation

With our 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

```
(λr:{x:Nat}. r.x) {x=0,y=1}
```

is *not* well typed.

Motivation

With our 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

```
(λr:{x:Nat}. r.x) {x=0,y=1}
```

is *not* well typed.

But this is silly: all we're doing is passing the function a *better* argument than it needs.

Similarly, in object-oriented languages, we want to be able to define hierarchies of classes, with classes lower in the hierarchy having richer interfaces than their ancestors higher in the hierarchy, and use instances of richer classes in situations where one of their ancestors are expected.

Subsumption

We achieve the effect we want by:

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

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

Example

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

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

So, by subsumption,

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

and hence

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

is well typed.

The Subtype Relation: Records

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

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

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 *sub*type of the record type with fewer fields.

Reason: the type with more fields places a *stronger constraint* on values, so it describes *fewer values*.

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.

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.

Example

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

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 super*class* ("single inheritance" of classes)

  $\longrightarrow$ *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

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

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 $S_1 \to S_2$, then we know that f accepts elements of type S_1; clearly, f will also accept elements of any subtype T_1 of S_1. 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. That is, any function f of type $S_1 \to S_2$ can also be viewed as having type $T_1 \to T_2$.

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.

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

Cf. `Object` in Java.

The Subtype Relation: General rules

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

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

Subtype relation

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

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

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

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

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

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

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

Aside: Structural vs. declared subtyping

The subtype relation we have defined is *structural*: We decide whether `S` is a subtype of `T` by examining the structure of `S` and `T`.

By contrast, the subtype relation in most OO languages (e.g., Java) is *explicitly declared*: `S` is a subtype of `T` only if the programmer has stated that it should be.

There are pragmatic arguments for both.

For the moment, we'll concentrate on structural subtyping, which is the more fundamental of the two. (It is sound to *declare* `S` to be a subtype of `T` only when `S` is structurally a subtype of `T`.)

We'll come back to declared subtyping when we talk about Featherweight Java.

Properties of Subtyping

Safety

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

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. The rule T-SUB could appear anywhere.

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

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.

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: $U = U_1 \to U_2 \qquad T_1 <: U_1 \qquad U_2 <: T_2$

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: $U = U_1 \to U_2$ $T_1 <: U_1$ $U_2 <: T_2$

Immediate.

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: $U = U_1 \to U_2$ $T_1 <: U_1$ $U_2 <: T_2$

Immediate.

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

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: $U = U_1 \to U_2$ $T_1 <: U_1$ $U_2 <: T_2$

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.

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: $U = U_1 \to U_2$ $T_1 <: U_1$ $U_2 <: T_2$

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: $U <: W$ $W <: T_1 \to T_2$

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: $U = U_1 \to U_2$ $T_1 <: U_1$ $U_2 <: T_2$

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: $U <: W$ $W <: T_1 \to T_2$

Applying the IH to the second subderivation,

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: $U = U_1 \to U_2$ $T_1 <: U_1$ $U_2 <: T_2$

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: $U <: W$ $W <: T_1 \to T_2$

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 $W_2 <: T_2$.

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: $U = U_1 \to U_2$ $T_1 <: U_1$ $U_2 <: T_2$

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: $U <: W$ $W <: T_1 \to T_2$

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 $U_2 <: W_2$.

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: $U = U_1 \to U_2$ $T_1 <: U_1$ $U_2 <: T_2$

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: $U <: W$ $W <: T_1 \to T_2$

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.

An Inversion Lemma for Typing

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

Proof: By induction on typing derivations.

An Inversion Lemma for Typing

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

Proof: By induction on typing derivations.

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

An Inversion Lemma for Typing

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

Proof: By induction on typing derivations.

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

Immediate.

Case T-Sub: $\Gamma \vdash \lambda x{:}S_1.s_2 : U$ $U <: T_1 \to T_2$

An Inversion Lemma for Typing

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

Proof: By induction on typing derivations.

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

Immediate.

Case T-Sub: $\Gamma \vdash \lambda x{:}S_1.s_2 : U$ $U <: T_1 \to T_2$

By the subtyping inversion lemma, $U = U_1 \to U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.

An Inversion Lemma for Typing

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

Proof: By induction on typing derivations.

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

Immediate.

Case T-Sub: $\Gamma \vdash \lambda x{:}S_1.s_2 : U$ $U <: T_1 \to T_2$

By the subtyping inversion lemma, $U = U_1 \to U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.

The IH now applies, yielding $U_1 <: S_1$ and $\Gamma, x{:}S_1 \vdash s_2 : U_2$.

An Inversion Lemma for Typing

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

Proof: By induction on typing derivations.

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

Immediate.

Case T-Sub: $\Gamma \vdash \lambda x{:}S_1.s_2 : U$ $U <: T_1 \to T_2$

By the subtyping inversion lemma, $U = U_1 \to U_2$, 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 $T_1 <: S_1$.

An Inversion Lemma for Typing

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

Proof: By induction on typing derivations.

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

Immediate.

Case T-Sub: $\Gamma \vdash \lambda x{:}S_1.s_2 : U$ $U <: T_1 \to T_2$

By the subtyping inversion lemma, $U = U_1 \to U_2$, 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 $\Gamma, x{:}S_1 \vdash s_2 : T_2$, and 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.

Preservation — subsumption case

Case T-SUB: $t : S \quad S <: T$

Preservation — subsumption case

Case T-SUB: $t : S \quad S <: T$

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

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:

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

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.

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

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

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

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.

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

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

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

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

Case T-App (continued):
$t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11}{\to}T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$

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

Similar.

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

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

Case T-App (continued):
$t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11}{\to}T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$

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

By the earlier inversion lemma for the typing relation...

Case T-App (continued):
$t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11}{\to}T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$

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

By the earlier inversion lemma for the typing relation... $T_{11} <: S_{11}$ and $\Gamma, x{:}S_{11} \vdash t_{12} : T_{12}$.

Case T-App (continued):

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

Subcase E-AppAbs:

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

By the earlier inversion lemma for the typing relation... $T_{11} <: S_{11}$ and $\Gamma, x{:}S_{11} \vdash t_{12} : T_{12}$.
By T-Sub, $\Gamma \vdash t_2 : S_{11}$.

Case T-App (continued):

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

Subcase E-AppAbs:

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

By the earlier inversion lemma for the typing relation... $T_{11} <: S_{11}$ and $\Gamma, x{:}S_{11} \vdash t_{12} : T_{12}$.
By T-Sub, $\Gamma \vdash t_2 : S_{11}$.
By the substitution lemma, $\Gamma \vdash t' : T_{12}$, and we are done.

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

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

Subtyping with Other Features

Ascription and Casting

Ordinary ascription:

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

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

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

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

I.e., `List` is a covariant type constructor.

Subtyping and References

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

I.e., `Ref` is *not* a covariant (nor a contravariant) type constructor. Why?

Subtyping and References

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

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.

Subtyping and References

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

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

Subtyping and Arrays

Similarly...

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

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.

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 cabability"
- `Sink T`: reference cell with "write cabability"
- `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

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

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

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

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

Algorithmic Subtyping

Syntax-directed rules

In the simply typed lambda-calculus (without subtyping), each rule can be "read from bottom to top" in a straightforward way.

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

If we are given some Γ and some t of the form $t_1\ t_2$, we can try to find a type for t by

1. finding (recursively) a type for t_1
2. checking that it has the form $T_{11}\to T_{12}$
3. finding (recursively) a type for t_2
4. checking that it is the same as T_{11}

Technically, the reason this works is that We can divide the "positions" of the typing relation into *input positions* (Γ and t) and *output positions* (T).

- For the input positions, all metavariables appearing in the premises also appear in the conclusion (so we can calculate inputs to the "subgoals" from the subexpressions of inputs to the main goal)
- For the output positions, all metavariables appearing in the conclusions also appear in the premises (so we can calculate outputs from the main goal from the outputs of the subgoals)

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

Syntax-directed sets of rules

The second important point about the simply typed lambda-calculus is that the *set* of typing rules is syntax-directed, in the sense that, for every "input" Γ and t, there one rule that can be used to derive typing statements involving t.
E.g., if t is an application, then we must proceed by trying to use T-App. If we succeed, then we have found a type (indeed, the unique type) for t. If it fails, then we know that t is not typable.

$\longrightarrow$ no backtracking!

Non-syntax-directedness of typing

When we extend the system with subtyping, both aspects of syntax-directedness get broken.

1. The set of typing rules now includes *two* rules that can be used to give a type to terms of a given shape (the old one plus T-SUB)

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

2. Worse yet, the new rule T-SUB itself is not syntax directed: the inputs to the left-hand subgoal are exactly the same as the inputs to the main goal!
(Hence, if we translated the typing rules naively into a typechecking function, the case corresponding to T-SUB would cause divergence.)

Non-syntax-directedness of subtyping

Moreover, the subtyping relation is not syntax directed either.

1. There are *lots* of ways to derive a given subtyping statement.
2. The transitivity rule

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

is badly non-syntax-directed: the premises contain a metavariable (in an "input position") that does not appear at all in the conclusion.
To implement this rule naively, we'd have to *guess* a value for U!

What to do?

What to do?

1. Observation: We don't *need* 1000 ways to prove a given typing or subtyping statement — one is enough.
$\longrightarrow$ Think more carefully about the typing and subtyping systems to see where we can get rid of excess flexibility
2. Use the resulting intuitions to formulate new "algorithmic" (i.e., syntax-directed) typing and subtyping relations
3. Prove that the algorithmic relations are "the same as" the original ones in an appropriate sense.

Conclusion

Polymorphism

Subtyping is a kind of *polymorphism*, which in Greek means "having many forms".
A *polymorphic* function may be applied to many different types of data.

Varieties of polymorphism:

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