CIS 500
Software Foundations
Fall 2006

October 30

Subtyping

Motivation

With our 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

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

is *not* well typed.

Motivation

With our 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

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

is *not* well typed.

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

Polymorphism

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)

Our topic for the next few lectures is *subtype* polymorphism, which is based on the idea of *subsumption*.

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 $\mathtt{S} <: \mathtt{T}$
2. a rule of *subsumption* stating that, if $\mathtt{S} <: \mathtt{T}$, then any value of type $\mathtt{S}$ can also be regarded as having type $\mathtt{T}$

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

Example

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

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

So, by subsumption,

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

and hence

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

is well typed.

The Subtype Relation: Records

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

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

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:

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

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:

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

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

$$\frac{\mathtt{T_1} <: \mathtt{S_1} \qquad \mathtt{S_2} <: \mathtt{T_2}}{\mathtt{S_1}\to\mathtt{S_2} <: \mathtt{T_1}\to\mathtt{T_2}} \tag{S-Arrow}$$

Note the order of $\mathtt{T_1}$ and $\mathtt{S_1}$ 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 $\mathtt{f}$ of type $\mathtt{S_1}\to\mathtt{S_2}$, then we know that $\mathtt{f}$ accepts elements of type $\mathtt{S_1}$; clearly, $\mathtt{f}$ will also accept elements of any subtype $\mathtt{T_1}$ of $\mathtt{S_1}$. The type of $\mathtt{f}$ also tells us that it returns elements of type $\mathtt{S_2}$; we can also view these results belonging to any supertype $\mathtt{T_2}$ of $\mathtt{S_2}$. That is, any function $\mathtt{f}$ of type $\mathtt{S_1}\to\mathtt{S_2}$ can also be viewed as having type $\mathtt{T_1}\to\mathtt{T_2}$.

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.

$$\mathtt{S} <: \mathtt{Top} \tag{S-Top}$$

Cf. `Object` in Java.

The Subtype Relation: General rules

$$\mathtt{S} <: \mathtt{S} \tag{S-Refl}$$

$$\frac{\mathtt{S} <: \mathtt{U} \qquad \mathtt{U} <: \mathtt{T}}{\mathtt{S} <: \mathtt{T}} \tag{S-Trans}$$

Subtype relation

$$\mathtt{S} <: \mathtt{S} \tag{S-Refl}$$

$$\frac{\mathtt{S} <: \mathtt{U} \qquad \mathtt{U} <: \mathtt{T}}{\mathtt{S} <: \mathtt{T}} \tag{S-Trans}$$

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

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

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

$$\frac{\mathtt{T_1} <: \mathtt{S_1} \qquad \mathtt{S_2} <: \mathtt{T_2}}{\mathtt{S_1}\to\mathtt{S_2} <: \mathtt{T_1}\to\mathtt{T_2}} \tag{S-Arrow}$$

$$\mathtt{S} <: \mathtt{Top} \tag{S-Top}$$

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.

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

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?)

Subsumption case

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

Subsumption case

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

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

Subsumption case

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

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

Not hard!

Application case

Case T-App:

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

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.

Application case

Case T-App:

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

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: $\quad t_1 \longrightarrow t_1' \qquad t' = t_1'\ t_2$

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

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

Application case

Case T-App:

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

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: $\quad t_1 \longrightarrow t_1' \qquad t' = t_1'\ t_2$

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

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

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

Case T-App (continued):

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

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

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):

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

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...

Case T-App (continued):

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

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} and $\Gamma, x{:}S_{11} \vdash t_{12} : T_{12}$.

Case T-App (continued):

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

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} and $\Gamma, x{:}S_{11} \vdash t_{12} : T_{12}$.

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

Case T-App (continued):

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

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} 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.

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

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

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: Induction on typing derivations.

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: Induction on typing derivations.

Case T-SUB: $\quad \lambda x{:}S_1.s_2 : U \qquad U <: T_1 \to T_2$

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: Induction on typing derivations.

Case T-SUB: $\quad \lambda x{:}S_1.s_2 : U \qquad U <: T_1 \to T_2$

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

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: Induction on typing derivations.

Case T-SUB: $\quad \lambda x{:}S_1.s_2 : U \qquad U <: T_1 \to T_2$

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

> 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.)*

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: Induction on typing derivations.

Case T-SUB: $\quad \lambda x{:}S_1.s_2 : U \qquad U <: T_1 \to T_2$

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

> 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.)*

By this lemma, we know $U = U_1 \to U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.

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: Induction on typing derivations.

Case T-SUB: $\quad \lambda x{:}S_1.s_2 : U \qquad U <: T_1 \to T_2$

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

> 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.)*

By this lemma, we know $U = U_1 \to U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.
The IH now applies, yielding $U_1 <: S_1$ and $\Gamma, x{:}S_1 \vdash s_2 : U_2$.

Inversion Lemma for Typing

Lemma: If $\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{S_1}.\mathtt{s_2} : \mathtt{T_1}{\to}\mathtt{T_2}$, then $\mathtt{T_1} <: \mathtt{S_1}$ and $\Gamma, \mathtt{x}{:}\mathtt{S_1} \vdash \mathtt{s_2} : \mathtt{T_2}$.
Proof: Induction on typing derivations.

Case T-Sub: $\quad \lambda\mathtt{x}{:}\mathtt{S_1}.\mathtt{s_2} : \mathtt{U} \qquad \mathtt{U} <: \mathtt{T_1}{\to}\mathtt{T_2}$

We want to say "By the induction hypothesis...", but the IH does not apply (we do not know that $\mathtt{U}$ is an arrow type). Need another lemma...

> 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.)*

By this lemma, we know $\mathtt{U} = \mathtt{U_1}{\to}\mathtt{U_2}$, with $\mathtt{T_1} <: \mathtt{U_1}$ and $\mathtt{U_2} <: \mathtt{T_2}$.
The IH now applies, yielding $\mathtt{U_1} <: \mathtt{S_1}$ and $\Gamma, \mathtt{x}{:}\mathtt{S_1} \vdash \mathtt{s_2} : \mathtt{U_2}$.
From $\mathtt{U_1} <: \mathtt{S_1}$ and $\mathtt{T_1} <: \mathtt{U_1}$, rule S-Trans gives $\mathtt{T_1} <: \mathtt{S_1}$.

Inversion Lemma for Typing

Lemma: If $\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{S_1}.\mathtt{s_2} : \mathtt{T_1}{\to}\mathtt{T_2}$, then $\mathtt{T_1} <: \mathtt{S_1}$ and $\Gamma, \mathtt{x}{:}\mathtt{S_1} \vdash \mathtt{s_2} : \mathtt{T_2}$.
Proof: Induction on typing derivations.

Case T-Sub: $\quad \lambda\mathtt{x}{:}\mathtt{S_1}.\mathtt{s_2} : \mathtt{U} \qquad \mathtt{U} <: \mathtt{T_1}{\to}\mathtt{T_2}$

We want to say "By the induction hypothesis...", but the IH does not apply (we do not know that $\mathtt{U}$ is an arrow type). Need another lemma...

> 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.)*

By this lemma, we know $\mathtt{U} = \mathtt{U_1}{\to}\mathtt{U_2}$, with $\mathtt{T_1} <: \mathtt{U_1}$ and $\mathtt{U_2} <: \mathtt{T_2}$.
The IH now applies, yielding $\mathtt{U_1} <: \mathtt{S_1}$ and $\Gamma, \mathtt{x}{:}\mathtt{S_1} \vdash \mathtt{s_2} : \mathtt{U_2}$.
From $\mathtt{U_1} <: \mathtt{S_1}$ and $\mathtt{T_1} <: \mathtt{U_1}$, rule S-Trans gives $\mathtt{T_1} <: \mathtt{S_1}$.
From $\Gamma, \mathtt{x}{:}\mathtt{S_1} \vdash \mathtt{s_2} : \mathtt{U_2}$ and $\mathtt{U_2} <: \mathtt{T_2}$, rule T-Sub gives $\Gamma, \mathtt{x}{:}\mathtt{S_1} \vdash \mathtt{s_2} : \mathtt{T_2}$, and we are done.

Subtyping with Other Features

Ascription and Casting

Ordinary ascription:

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

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

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

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

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

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

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_1}}{\Gamma \vdash \mathtt{<}\mathtt{l_1}\mathtt{=}\mathtt{t_1}\mathtt{>} : \mathtt{<}\mathtt{l_1}{:}\mathtt{T_1}\mathtt{>}} \qquad \text{(T-Variant)}$$

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

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

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

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

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...

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

Subtyping and Arrays

Similarly...

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

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

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 `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

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

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

Subtyping rules

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

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

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

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

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.

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

If we are given some Γ and some $\mathtt{t}$ of the form $\mathtt{t}_1\ \mathtt{t}_2$, we can try to find a type for $\mathtt{t}$ by

1. finding (recursively) a type for $\mathtt{t}_1$
2. checking that it has the form $\mathtt{T}_{11} \to \mathtt{T}_{12}$
3. finding (recursively) a type for $\mathtt{t}_2$
4. checking that it is the same as $\mathtt{T}_{11}$

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.