CIS 500
Software Foundations
Fall 2006

November 1

Notes from Yesterday's Email Discussion

Some lessons

- This is generally a crunch-time in the semester
  - Slow down a little and give people a chance to catch up
- Once you're confused, it's hard to know what to ask
  - So not necessarily a problem if people are not asking many questions, but definitely a sign to slow down more
- Working simple examples in class is good
  - ... in part because it makes people think of other questions
- Some hard homework problems have been too vague
  - Not enough information ⟶ need to look at the solution to see what is wanted ⟶ hard to think independently any more
- Big picture has been getting a little lost in the details

Big Picture

Plan for the rest of the semester

- **This week:** Basic subtyping
- **Next week:** Review and midterm
- **Nov 13,15:** Algorithmics of subtyping
- **Nov 20,22:** Modeling OO languages in typed lambda-calculus
- **Nov 27,29:** Featherweight Java
- **Dec 4,6:** *To be decided* (Parametric polymorphism? ML module system? ...)
- **Dec 20:** Final exam

What's it all for

- Techniques and notations for formalizing languages and language constructs
  - inductive definitions, operational semantics, typing and subtyping relations, etc.
  - Records, exceptions, etc. as case studies
- Strong intuitions about fundamental safety properties
  - Especially: Healthy scepticism and good investigative skills for how things can be broken!
- Some specific fundamental building blocks of languages
  - Variables, scope, and binding
  - Functions and their types (higher-order programming)
  - References (mutable state, aliasing)
  - Subtyping
  - Objects and classes

Ultimately, the goal is to give you the ability to put all this together and formalize your own languages or language features.

Subtyping (again)

Motivation

We want terms like

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

to be well typed.

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. defining a new *subtyping* relation between types, written $S <: T$
2. adding a new rule of *subsumption* to the typing relation:

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

Subtype relation

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

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

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

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

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

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

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

Example

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

Another example

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

(board)

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 $\lambda_{\rightarrow}$.

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

Proof: By induction on subtyping derivations.

An Inversion Lemma for Subtyping

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

Proof: By induction on subtyping derivations.

Case S-Arrow: $U = U_1 \rightarrow U_2 \qquad T_1 <: U_1 \qquad U_2 <: T_2$

An Inversion Lemma for Subtyping

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

Proof: By induction on subtyping derivations.

Case S-Arrow: $U = U_1 \rightarrow U_2 \qquad T_1 <: U_1 \qquad U_2 <: T_2$

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

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

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

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

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

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,

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

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 $W_2 <: T_2$.

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

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 $U_2 <: W_2$.

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

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.

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: $T_1 = S_1$ $\quad T_2 = S_2$ $\quad \Gamma, x{:}S_1 \vdash s_2 : S_2$

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: $T_1 = S_1$ $\quad T_2 = S_2$ $\quad \Gamma, x{:}S_1 \vdash s_2 : S_2$

Immediate.

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

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: $T_1 = S_1$ $\quad T_2 = S_2$ $\quad \Gamma, x{:}S_1 \vdash s_2 : S_2$

Immediate.

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

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: $T_1 = S_1$ $\quad T_2 = S_2$ $\quad \Gamma, x{:}S_1 \vdash s_2 : S_2$

Immediate.

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

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: $t : S$ $S <: T$

Preservation — subsumption case

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

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

Preservation — application case

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

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:

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

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.

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

Preservation — 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.

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

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

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.

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

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

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 earlier 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 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):

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

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 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):

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

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 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.

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

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

Subtyping with Other Features

Ascription and Casting

Ordinary ascription:

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

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

Ascription and Casting

Ordinary ascription:

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

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

Casting (cf. Java):

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

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

Subtyping and Variants

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

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

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

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

Subtyping and Lists

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

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

Subtyping and References

$$\frac{\texttt{S}_1 <: \texttt{T}_1 \qquad \texttt{T}_1 <: \texttt{S}_1}{\texttt{Ref } \texttt{S}_1 <: \texttt{Ref } \texttt{T}_1} \quad \text{(S-REF)}$$

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

Subtyping and References

$$\frac{\texttt{S}_1 <: \texttt{T}_1 \qquad \texttt{T}_1 <: \texttt{S}_1}{\texttt{Ref } \texttt{S}_1 <: \texttt{Ref } \texttt{T}_1} \quad \text{(S-REF)}$$

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

- When a reference is *read*, the context expects a $\texttt{T}_1$, so if $\texttt{S}_1 <: \texttt{T}_1$ then an $\texttt{S}_1$ is ok.

Subtyping and References

$$\frac{\texttt{S}_1 <: \texttt{T}_1 \qquad \texttt{T}_1 <: \texttt{S}_1}{\texttt{Ref } \texttt{S}_1 <: \texttt{Ref } \texttt{T}_1} \quad \text{(S-REF)}$$

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

- When a reference is *read*, the context expects a $\texttt{T}_1$, so if $\texttt{S}_1 <: \texttt{T}_1$ then an $\texttt{S}_1$ is ok.
- When a reference is *written*, the context provides a $\texttt{T}_1$ and if the actual type of the reference is $\texttt{Ref } \texttt{S}_1$, someone else may use the $\texttt{T}_1$ as an $\texttt{S}_1$. So we need $\texttt{T}_1 <: \texttt{S}_1$.

Subtyping and Arrays

Similarly...

$$\frac{\texttt{S}_1 <: \texttt{T}_1 \qquad \texttt{T}_1 <: \texttt{S}_1}{\texttt{Array } \texttt{S}_1 <: \texttt{Array } \texttt{T}_1} \quad \text{(S-ARRAY)}$$

Subtyping and Arrays

Similarly...

$$\frac{\texttt{S}_1 <: \texttt{T}_1 \qquad \texttt{T}_1 <: \texttt{S}_1}{\texttt{Array } \texttt{S}_1 <: \texttt{Array } \texttt{T}_1} \quad \text{(S-ARRAY)}$$

Compare this with the Java rule for array subtyping:

$$\frac{\texttt{S}_1 <: \texttt{T}_1}{\texttt{Array } \texttt{S}_1 <: \texttt{Array } \texttt{T}_1} \quad \text{(S-ARRAYJAVA)}$$

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

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

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

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.

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

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)

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

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.

⟶ 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)

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

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

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

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.
⟶ 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.

We'll come back to this discussion in (much) more detail after the midterm.