CIS 500
Software Foundations
Fall 2003

3 November

Administrivia

- Reminder: Midterm II is next Wednesday, November 12th.

  Covering all material we've seen so far, up through Chapter 16 of TAPL (but omitting Chapter 12 and Section 15.6).
- Schedule:
  - Last week: Chapter 14 (references) and Chapters 13 (exceptions)
  - This week: Chapter 15 (subtyping) and 16 (metatheory of subtyping)
  - Next week: review session, Midterm II
- Change of BCP's office hours, next two weeks:
  - This wednesday 5-6, as usual
  - No office hour this Thursday, Nov. 6
  - Next week: Monday, Nov. 10, 3-5 (only)

Subtyping

Varieties of Polymorphism

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

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.

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

Subsumption

More generally: some types are better than others, in the sense that a value of one can always safely be used where a value of the other is expected.

We can formalize this intuition by introducing

1. a subtyping relation between types, written $S <: T$
2. a rule of subsumption stating that, if $S <: T$, then any value of type S can also be regarded as having type T

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

Example

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

`{x:Nat, y:Nat} <: {x:Nat}`

So, by subsumption,

`⊢ {x=0,y=1} : {x:Nat}`

and hence

`(λr:{x:Nat}. r.x) {x=0,y=1}`

is well typed.

The Subtype Relation: General rules

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

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

The Subtype Relation: Records

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

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

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 a stronger constraint on values, so it describes fewer values.

"Depth subtyping" within fields:

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

Example

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

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, we can 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

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

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.

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

Cf. `Object` in Java.

Review

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.
- the set of elements of S is a subset of the set of elements of T.
- S is more informative / richer than T.

Subtype relation

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

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

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

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

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

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

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

Subsumption Rule

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

Other typing rules as in $\lambda_{\rightarrow}$

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.

Preservation

Theorem: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathtt{T}$.

Proof: By induction on typing derivations.

(Which cases are hard?)

Subsumption case

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

Subsumption case

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

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

Subsumption case

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

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

Not hard!

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.

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

Application case

Case T-App:

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

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

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: $t_1 = v_1 \qquad t_2 \longrightarrow t_2' \qquad t' = v_1\ t_2'$

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 inversion lemma for the typing relation...

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

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

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

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

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

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, $\Gamma \vdash t_2 : S_{11}$.

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

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

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: $\lambda x{:}S_1.s_2 : U \qquad U$ <: $T_1 \to T_2$

We want to say "By the induction hypothesis...", but the IH does not apply (we do not know that S 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: $\lambda x{:}S_1.s_2 : U \qquad U <: T_1 \to T_2$

We want to say "By the induction hypothesis...", but the IH does not apply (we do not know that S 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: $\lambda x{:}S_1.s_2 : U \qquad U <: T_1 \to T_2$

We want to say "By the induction hypothesis...", but the IH does not apply (we do not know that S 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: $\lambda x{:}S_1.s_2 : U \qquad U <: T_1 \to T_2$

We want to say "By the induction hypothesis...", but the IH does not apply (we do not know that S 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 $\Gamma \vdash \lambda x{:}S_1.s_2 : T_1 \to T_2$, then $T_1 <: S_1$ and $\Gamma, x{:}S_1 \vdash s_2 : T_2$.

Proof: Induction on typing derivations.

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

We want to say "By the induction hypothesis...", but the IH does not apply (we do not know that S 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 $\Gamma, x{:}S_1 \vdash s_2 : U_2$.

From $U_1 <: S_1$ and $T_1 <: U_1$, rule S-TRANS gives $T_1 <: S_1$.

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: $\lambda x{:}S_1.s_2 : U \qquad U <: T_1 \to T_2$

We want to say "By the induction hypothesis...", but the IH does not apply (we do not know that S 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 $\Gamma, x{:}S_1 \vdash s_2 : U_2$.

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:

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

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

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

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

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

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

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

Subtyping and Lists

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

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

Subtyping and References

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

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

Subtyping and Arrays

Similarly...

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

Subtyping and Arrays

Similarly...

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

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

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

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

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

Subtyping rules

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

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

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

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

Capabilities

Other kinds of capabilities (e.g., send and receive capabilities on communication channels, encrypt/decrypt capabilities of cryptographic keys, ...) can be treated similarly.

Coercion semantics

(We're skipping this section.)

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.

$$T_1 \wedge T_2 <: T_1 \quad \text{(S-Inter1)}$$

$$T_1 \wedge T_2 <: T_2 \quad \text{(S-Inter2)}$$

$$\frac{S <: T_1 \qquad S <: T_2}{S <: T_1 \wedge T_2} \quad \text{(S-Inter3)}$$

$$S \to T_1 \wedge S \to T_2 <: S \to (T_1 \wedge T_2) \quad \text{(S-Inter4)}$$

Intersection Types

Intersection types permit a very flexible form of finitary overloading.

$+ : (Nat \to Nat \to Nat) \wedge (Float \to Float \to Float)$

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.

$\longrightarrow$ 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 $\longrightarrow$ no `case` construct. The only operations we can safely perform on elements of $T_1 \vee T2$ are ones that make sense for both T_1 and T_2.

N.b.: 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).

Metatheory of Subtyping (Preview)

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.

   ⟶ 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.