Subtyping (Review)

Subtype relation

\[
\{ u \in L \mid T_i \} \rightarrow \{ u \in L \mid S_i \}
\]

For each \( i \), \( S_i \rightarrow T_i \)

\[
\{ u \in L \mid T_i \} \rightarrow \{ u \in L \mid S_i \}
\]

\[
\{ u \in L \mid T_i \} \rightarrow \{ u \in L \mid S_i \}
\]

\[
\{ u \in L \mid T_i \} \rightarrow \{ u \in L \mid S_i \}
\]

\[
\{ u \in L \mid T_i \} \rightarrow \{ u \in L \mid S_i \}
\]

No office hours on Wednesday and Thursday.
Class on Wednesday, but no recitations due to Thanksgiving.
Homework 9 assigned today, due Dec 1.
Subtyping and References

Why?

1. Ref is not a covariant (nor a contravariant) type constructor.

\[
\begin{align*}
& \text{Ref } S_1 : \text{Ref } T_1 \\
\Rightarrow & \text{Ref } S_1 : \text{Ref } T_1 \\
\Rightarrow & S_1 : T_1 \\
\end{align*}
\]

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

Subtyping and References

Subtyping References

Other type rules as in $\downarrow^V$

(T-Step)

\[
\begin{align*}
& L : T \\
\Rightarrow & T \\
\Rightarrow & \text{Ref } S_1 : S \\
\Rightarrow & L : S \\
\end{align*}
\]

(Top)

\[
\begin{align*}
& S : \text{Top} \\
\Rightarrow & S \Rightarrow T_1 \\
\Rightarrow & S_1 \Rightarrow T_1 \\
\Rightarrow & S_1 \Rightarrow T_2 \\
\Rightarrow & S_2 \Rightarrow T_2 \\
\Rightarrow & T_1 : S_1 \\
\end{align*}
\]
Subtyping and References

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

\[
\text{Array } S I \rightarrow \text{Array } T I \\
S I \rightarrow T I
\]

Similarly...

\[
\text{Array } S I \rightarrow \text{Array } T I \\
S I \rightarrow T I
\]

\[
\text{Ref } S I \rightarrow \text{Ref } T I \\
S I \rightarrow T I
\]

\[
\text{Ref } S I \rightarrow \text{Ref } T I \\
S I \rightarrow T I
\]

Subtyping and Arrays

Similarly...

\[
\text{Array } S I \rightarrow \text{Array } T I \\
S I \rightarrow T I
\]

\[
\text{Ref } S I \rightarrow \text{Ref } T I \\
S I \rightarrow T I
\]

\[
\text{Ref } S I \rightarrow \text{Ref } T I \\
S I \rightarrow T I
\]
Observation: a value of type \( \text{Ref} \) can be used in two different ways: as a source for values of type \( T \) and as a sink for values of type \( T \).

Reference cell with "read" capability

Reference cell with "write" capability

Idea: Split \( \text{Ref} \) into three parts:

- \( \text{SourceT} \): reference cell with "read" capability
- \( \text{SinkT} \): reference cell with "write" capability
- \( \text{RefT} \): cell with both capabilities

**Subtyping Rules**

**Modified Typing Rules**

\[ \text{j} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]

\[ \text{t} \]
Syntax-directed rules

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

If we are given some $\Gamma$ 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 $T_11 \rightarrow T_12$
3. finding (recursively) a type for $t_2$
4. checking that it is the same as $T_11$

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" $\Gamma \vdash t_1 : T_11 \rightarrow T_12$ and $\Gamma \vdash t_2 : T_12$, there is a rule that can be used to derive typing statements involving $t = t_1 \ t_2$.

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.

Technically, the reason this works is that we can divide the "positions" of the typing relation into input positions ($\Gamma$ 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).

... no backtracking!
Non-syntax-directedness of typing

1. When we extend the system with subtyping, both aspectsof syntax-directedness get broken.

2. 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).

Moreover, the subtyping relation is not syntax directed either.

Non-syntax-directedness of subtyping

1. Observation: We don’t need 1000 ways to prove a given typing or subtyping statement.

2. Use the resulting intuition 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.

Moreover, the subtyping relation is not syntax-directed either.

1. There are lots of ways to derive a given subtyping statement.

2. The resulting intuition is not syntax directed either.

Non-syntax-directedness of subtyping

1. What to do?

- Think more carefully about the typing and subtyping systems to see where we can get rid of excess flexibility.

- Use the T-Sub rule 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 type-checking function, the case corresponding to T-Sub would cause divergence.

- Moreover, the new rule T-Sub itself is not syntax directed: the inputs to the left-hand goal of T-Sub are not given subtyping statements.
Developing an algorithmic subtyping relation

Subtype Relation

2. S-Refl and S-Trans overlap with everything.
1. S-Rcd-Width, S-Rcd-Depth, and S-Rcd-Perm overlap with each other.

For a given subtyping statement, there are multiple rules that could be used.

Issues

For a given subtyping statement, there are multiple rules that could be used.

\[
S <: S \quad S <: U \quad U <: T
\]

(S-Refl)

\[
\{l_i : T_i \mid 1 \leq i \leq n\} <: \{l_j : T_j \mid 1 \leq j \leq m\}
\]

(S-Trans)

\[
\{l_i : S_i \mid 1 \leq i \leq n\} <: \{l_j : T_j \mid 1 \leq j \leq m\}
\]

(S-RcdWidth)

\[
\{l_i : S_i \mid 1 \leq i \leq n\} <: \{l_j : T_j \mid 1 \leq j \leq m\}
\]

(S-RcdDepth)

\[
\{l_i : T_i \mid 1 \leq i \leq n\} <: \{l_j : T_j \mid 1 \leq j \leq m\}
\]

(S-RcdPerm)

\[
\{l_i : T_i \mid 1 \leq i \leq n\} <: \{l_j : T_j \mid 1 \leq j \leq m\}
\]

(S-Arrow)

\[
S > T
\]

(S-Top)

\[
S_i > T \quad S_j > T
\]

Issues

1. S-Rcd-Width, S-Rcd-Depth, and S-Rcd-Perm overlap with each other.
2. S-Refl and S-Trans overlap with everything.

For a given subtyping statement, there are multiple rules that could be used.

\[
S <: S \quad S <: U \quad U <: T
\]

(S-Refl)

\[
\{l_i : T_i \mid 1 \leq i \leq n\} <: \{l_j : T_j \mid 1 \leq j \leq m\}
\]

(S-Trans)

\[
\{l_i : S_i \mid 1 \leq i \leq n\} <: \{l_j : T_j \mid 1 \leq j \leq m\}
\]

(S-RcdWidth)

\[
\{l_i : S_i \mid 1 \leq i \leq n\} <: \{l_j : T_j \mid 1 \leq j \leq m\}
\]

(S-RcdDepth)

\[
\{l_i : T_i \mid 1 \leq i \leq n\} <: \{l_j : T_j \mid 1 \leq j \leq m\}
\]

(S-RcdPerm)

\[
\{l_i : T_i \mid 1 \leq i \leq n\} <: \{l_j : T_j \mid 1 \leq j \leq m\}
\]

(S-Arrow)

\[
S > T
\]

(S-Top)
Step 1: Simplify record subtyping

Idea: combine all three record subtyping rules into one "macro rule" that captures all of their effects.

Step 2: Get rid of reflexivity

Observation: S-Refl is unnecessary.

Lemma: S ⊆ T can be derived for every type without using S-Refl.
Step 3: Get rid of transitivity

Observation: S-Trans is unnecessary.

The algorithmic presentation of subtyping is complete with respect to the original if $S \Rightarrow T$ implies $S \Rightarrow T$. Everything true is validated by the algorithm.

Subtyping Algorithm (pseudo-code)

The algorithmic rules can be translated directly into code:

```plaintext
subtype(S; T) =
  if T = Top, then true
  elseif S = S₁ \rightarrow S₂ and T = T₁ \rightarrow T₂ then
    subtype(T₁; S₁)^
    subtype(S₂; T₂)
  elseif S = \{k_j \in k \mid k ∈ K with k \neq T\} and T = \{l_i \in l \mid l ∈ L with l \neq T\} then
    ∃ j_k \in k \subseteq \{k_j \mid j \in K\}, ∃ l_l \in l \subseteq \{l_i \mid i \in L\} such that j_k \neq l_l
    subtype(Sⱼ; Tᵢ)^
  else false.
```

Soundness and completeness

Theorem: $S \Rightarrow T$ if $S \Rightarrow T$.

Proof:

```
\forall j_k \in k \subseteq \{k_j \mid j \in K\}, ∃ l_l \in l \subseteq \{l_i \mid i \in L\} such that j_k \neq l_l

let $f_{k_j}$
```

Every typing line is validated by the algorithm.

The algorithmic presentation of subtyping is complete with respect to the original if $S \Rightarrow T$ implies $S \Rightarrow T$. Everything true is validated by the algorithm.

Soundness:

```
\forall j_k \in k \subseteq \{k_j \mid j \in K\}, ∃ l_l \in l \subseteq \{l_i \mid i \in L\} such that j_k \neq l_l

let $f_{k_j}$
```

The algorithmic presentation of subtyping is sound with respect to the original if $S \Rightarrow T$ implies $S \Rightarrow T$. Everything validated by the algorithm is actually true.
A decision procedure for a relation \( R \subseteq U \) is a total function \( p : U \rightarrow \{ true, false \} \) such that:

\[
p(u) = true \iff u \in R.
\]

Q: What's missing?

Since subtype is just an implementation of the algorithmic subtyping rules, we have:

1. If subtype \( S \rightarrow T \) then \( S \leq T \) (hence, by soundness of the algorithmic rules, \( S \leq T \)).
2. If subtype \( S \nrightarrow T \) then not \( S \leq T \) (hence, by completeness of the algorithmic rules, not \( S \leq T \)).

Q: What's missing?

A: How do we know that subtype is a total function?

Prove it!
A decision procedure for a relation $R \subseteq U$ is a total function $p$ from $U$ to $(\text{true}, \text{false})$ such that $p(u) = \text{true}$ if $u \in R$.

Is our subtype function a decision procedure?

Since subtype is just an implementation of the algorithmic subtyping rules, we have:

1. if $\text{subtype}(S; T) = \text{true}$, then $S \subseteq T$ (hence, by soundness of the algorithmic rules, $S < T$)
2. if $\text{subtype}(S; T) = \text{false}$, then not $S \subseteq T$ (hence, by completeness of the algorithmic rules, not $S < T$)

Q: What's missing?

A: How do we know that subtype is a total function?

Prove it!
For the typing relation, we have just one problematic rule to deal with: subsumption.

\[
\begin{array}{c}
T \vdash \text{false} : S \to T
\
T \vdash \text{false} : T
\end{array}
\]

Where is this rule really needed?

Uses of subsumption to help type check are the only interesting ones.

Example

\[
\begin{array}{c}
L \vdash x : S_1 \vdash x : S_2
\
L \vdash x : T_2
\end{array}
\]

\[
\begin{array}{c}
L \vdash x : S_1 \vdash x : S_2
\
L \vdash x : T_2
\end{array}
\]

\[
\begin{array}{c}
S \vdash \text{false} : S \to S
\
S \vdash \text{false} : S
\end{array}
\]

\[
\begin{array}{c}
S \vdash \text{false} : S \to S
\
S \vdash \text{false} : S
\end{array}
\]
Example

\[
\frac{L \vdash \not\exists z \ T_{12}}{L \vdash \not \exists z \ T_{11}}
\]

Example

\[
\frac{L \vdash \not \exists z \ T_{12}}{L \vdash \not \exists z \ T_{11}}
\]

Example

\[
\frac{L \vdash \not \exists z \ T_{12}}{L \vdash \not \exists z \ T_{11}}
\]

Example