

CIS 500

Software Foundations

Fall 2004

22 November

---

## Administrivia

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

Subtyping (Review)

## Subtype relation

(S-REFL)

$$S <: S$$

(S-TRANS)

$$\frac{S <: U \quad U <: T}{S <: T}$$

(S-RCDWIDTH)

$$\{l_i : T_i\}_{i \in I \dots n+k} <: \{l_i : T_i\}_{i \in I \dots n}$$

(S-RCDDEPTH)

$$\frac{\text{for each } i \quad S_i <: T_i}{\{l_i : S_i\}_{i \in I \dots n} <: \{l_i : T_i\}_{i \in I \dots n}}$$

(S-RCDPERM)

$$\frac{\{k_j : S_j\}_{j \in I \dots n} <: \{l_i : T_i\}_{i \in I \dots n}}{\{k_j : S_j\}_{j \in I \dots n} \text{ is a permutation of } \{l_i : T_i\}_{i \in I \dots n}}$$

$$\frac{T_1 \prec S_1 \quad S_2 \prec T_2}{S_1 \leftrightarrow S_2 \prec T_1 \leftrightarrow T_2}$$

S >: Top

(S-Top)

(S-ARROW)

Other typing rules as in  $\lambda \leftarrow$

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

---

Subsumption Rule

Subtyping references

## Subtyping and References

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

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



## Subtyping and References

$$\frac{S_1 \text{ Ref } S_1 < \text{Ref } T_1}{S_1 < T_1}$$

(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

$$\frac{\text{Ref } S_1 < \text{Ref } T_1}{S_1 < T_1 \quad T_1 < S_1} \text{ (S-REF)}$$

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 **Ref**  $S_1$ , someone else may use the  $T_1$  as an  $S_1$ . So we need  $T_1 < S_1$ .

Similarly...

---

## Subtyping and Arrays

$$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Array } S_1 <: \text{Array } T_1}$$

(S-ARRAY)

## Subtyping and Arrays

Similarly...

$$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Array } S_1 <: \text{Array } T_1}$$

(S-ARRAY)

$$\frac{S_1 <: T_1}{\text{Array } S_1 <: \text{Array } T_1}$$

(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

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

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

## Subtyping rules

$$\frac{S_1 < T_1}{\text{Source } S_1 < \text{Source } T_1}$$
$$\frac{T_1 < S_1}{\text{Sink } S_1 < \text{Sink } T_1}$$
$$\text{Ref } T_1 < \text{Source } T_1$$
$$\text{Ref } T_1 < \text{Sink } T_1$$



# Metatheory of 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.

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

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  $T_{11}$

- 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** ( $\mathbb{T}$ ).
- ◆ 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)

$$\frac{\Gamma \vdash t_1 \quad t_2 : \mathbb{T}_{12}}{\Gamma \vdash t_1 : \mathbb{T}_{11} \rightarrow \mathbb{T}_{12} \quad \Gamma \vdash t_2 : \mathbb{T}_{11}}$$

(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”  $\Gamma$  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)

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

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

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

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.



Developing an algorithmic subtyping relation

## Subtype relation

(S-REFL)

$$S <: S$$

(S-TRANS)

$$\frac{S <: U \quad U <: T}{S <: T}$$

(S-RCDWIDTH)

$$\{l_i : T_i\}_{i \in I \dots n+k} <: \{l_i : T_i\}_{i \in I \dots n}$$

(S-RCDDEPTH)

$$\frac{\text{for each } i \quad S_i <: T_i}{\{l_i : S_i\}_{i \in I \dots n} <: \{l_i : T_i\}_{i \in I \dots n}}$$

(S-RCDPERM)

$$\frac{\{k_j : S_j\}_{j \in I \dots n} <: \{l_i : T_i\}_{i \in I \dots n}}{\{k_j : S_j\}_{j \in I \dots n} \text{ is a permutation of } \{l_i : T_i\}_{i \in I \dots n}}$$

(S-TOP)

 $S \rightarrow \text{Top}$ 

(S-ARROW)

$$\frac{T_1 \rightarrow S_1 \quad S_2 \rightarrow T_2 \quad S_1 \rightarrow S_2 \quad T_1 \rightarrow T_2}{S \rightarrow \text{Top}}$$

---

## Issues

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

1. S-RCD-WIDTH, S-RCD-DEPTH, and S-RCD-PERM overlap with each other
2. S-REFL and S-TRANS overlap with everything

## Step 1: simplify record subtyping

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

$$\frac{\{l_i : T_i\}_{i \in I \dots n} \subseteq \{k_j : S_j\}_{j \in J \dots m} \quad k_j = l_i \text{ implies } S_j < T_i}{\{k_j : S_j\}_{j \in J \dots m} < \{l_i : T_i\}_{i \in I \dots n}} \text{(S-RCD)}$$

## Simpler subtype relation

(S-REFL)

$$S <: S$$

(S-TRANS)

$$\frac{S <: U \quad U <: T}{S <: T}$$

(S-RCD)

$$\frac{\{l_i : T_i\}_{i \in I \dots n} \subseteq \{k_j : S_j\}_{j \in I \dots m} \quad k_j = l_i \text{ implies } S_j <: T_i}{\{k_j : S_j\}_{j \in I \dots m} <: \{l_i : T_i\}_{i \in I \dots n}}$$

(S-ARROW)

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

(S-TOP)

$$S <: \text{Top}$$

---

## Step 2: Get rid of reflexivity

Observation: S-REFL is unnecessary.

Lemma:  $S < S$  can be derived for every type without using S-REFL.

## Even simpler subtype relation

(S-TRANS)

$$\frac{S <: U \quad U <: T}{S <: T}$$

(S-RCD)

$$\frac{\{l_i \mid i \in I \dots n\} \subseteq \{k_j \mid j \in I \dots m\} \quad k_j = l_i \text{ implies } S_j <: T_i}{\{k_j : S_j \mid j \in I \dots m\} <: \{l_i : T_i \mid i \in I \dots n\}}$$

(S-ARROW)

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

(S-TOP)

$$S <: \text{Top}$$



---

## Step 3: Get rid of transitivity

Observation: S-TRANS is unnecessary.

**Lemma:** If **S** < **T** can be derived, then there is a derivation that does not use S-TRANS.

“Algorithmic” subtype relation

(SA-TOP)  $\vdash S <: \text{Top}$

(SA-ARROW) 
$$\frac{\vdash T_1 <: S_1 \quad \vdash S_2 <: T_2}{\vdash T_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

(SA-RCD) 
$$\frac{\vdash \{k_j : S_j\}_{j \in I \dots m} <: \{l_i : T_i\}_{i \in I \dots n}}{\text{for each } k_j = l_i, \vdash S_j <: T_i}$$

## Soundness and completeness

---

Theorem:  $S \leq T \text{ iff } S \leq T$ .

Proof: ...

Terminology:

◆ The algorithmic presentation of subtyping is **sound** with respect to the original if  $S \leq T$  implies  $S \leq T$ .

(Everything validated by the algorithm is actually true.)

◆ The algorithmic presentation of subtyping is **complete** with respect to the original if  $S \leq T$  implies  $S \leq T$ .

(Everything true is validated by the algorithm.)

## Subtyping Algorithm (pseudo-code)

The algorithmic rules can be translated directly into code:

```
subtype(S, T) = if T = Top, then true  
              else if S = S1 → S2 and T = T1 → T2  
                 then subtype(T1, S1) ∧ subtype(S2, T2)  
              else if S = {kj: Sj}j ∈ 1..m and T = {li: Ti}i ∈ 1..n  
                 then {li}i ∈ 1..n ⊆ {kj}j ∈ 1..m  
              ∧ for all i there is some j ∈ 1..m with kj = li  
                 and subtype(Sj, Ti)  
              else false.
```

## Decision Procedures

---

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

## Decision Procedures

---

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

Is our *subtype* function a decision procedure?

## Decision Procedures

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

Is our *subtype* function a decision procedure?

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

have

1. if  $subtype(S, T) = true$ , then  $S < T$

(hence, by soundness of the algorithmic rules,  $S < T$ )

2. if  $subtype(S, T) = false$ , then not  $S < T$

(hence, by completeness of the algorithmic rules, not  $S < T$ )

## Decision Procedures

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

Is our *subtype* function a decision procedure?

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

have

1. if  $subtype(S, T) = true$ , then  $S < T$

(hence, by soundness of the algorithmic rules,  $S < T$ )

2. if  $subtype(S, T) = false$ , then not  $S < T$

(hence, by completeness of the algorithmic rules, not  $S < T$ )

Q: What's missing?



## Decision Procedures

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

Is our *subtype* function a decision procedure?

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

have

1. if  $subtype(S, T) = true$ , then  $S < T$

(hence, by soundness of the algorithmic rules,  $S < T$ )

2. if  $subtype(S, T) = false$ , then not  $S < T$

(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?

## Decision Procedures

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

Is our *subtype* function a decision procedure?

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

1. if  $subtype(S, T) = true$ , then  $S < T$

(hence, by soundness of the algorithmic rules,  $S < T$ )

2. if  $subtype(S, T) = false$ , then not  $S < T$

(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!

# Metatheory of Typing

---

## Issue

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

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t : S \quad S <: T}$$

(T-SUB)

Where is this rule really needed?

---

## Issue

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

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t : S \quad S <: T}$$

(T-SUB)

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.

---

## Issue

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

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t : S \quad S <: T}$$

(T-SUB)

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.

Where else??

## Issue

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

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

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.

Where else??

**Nowhere else!** Uses of subsumption to help typecheck applications are the only interesting ones.

## Example

$$\begin{array}{c} \vdots \\ \hline \Gamma, x:S_1 \vdash s_2 : S_2 \\ \hline \Gamma, x:S_1 \vdash s_2 : T_2 \end{array} \quad \begin{array}{c} \vdots \\ \hline S_2 <: T_2 \\ \hline \end{array} \quad \begin{array}{c} \hline \Gamma, x:S_1 \vdash s_2 : T_2 \\ \hline \end{array} \quad \begin{array}{c} \hline \Gamma \vdash \lambda x:S_1. s_2 : S_1 \rightarrow T_2 \\ \hline \end{array} \quad \begin{array}{c} \text{(T-Sub)} \\ \text{(T-Abs)} \end{array}$$



## Example

$$\begin{array}{c}
 \vdots \\
 \vdots \\
 \hline
 \Gamma, x:S_1 \vdash s_2 : T_2 \\
 \hline
 \Gamma, x:S_1 \vdash s_2 : T_2 \\
 \text{(T-SUB)} \\
 \hline
 \Gamma \vdash \lambda x:S_1.s_2 : S_1 \multimap T_2 \\
 \text{(T-ABS)}
 \end{array}$$

becomes

$$\begin{array}{c}
 \vdots \\
 \hline
 \Gamma, x:S_1 \vdash s_2 : S_2 \\
 \hline
 \Gamma \vdash \lambda x:S_1.s_2 : S_1 \multimap S_2 \\
 \text{(T-ABS)} \\
 \hline
 \Gamma \vdash \lambda x:S_1.s_2 : S_1 \multimap T_2 \\
 \text{(T-SUB)} \\
 \hline
 \vdots \\
 \vdots \\
 \hline
 \Gamma, x:S_1 \vdash s_2 : T_2 \\
 \hline
 \Gamma, x:S_1 \vdash s_2 : T_2 \\
 \text{(S-REFL)} \\
 \hline
 \Gamma, x:S_1 \vdash s_2 : T_2 \\
 \hline
 \Gamma \vdash \lambda x:S_1.s_2 : S_1 \multimap T_2 \\
 \text{(S-ARROW)}
 \end{array}$$



# Example

$$\begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash s_1 : S_{11} \rightarrow S_{12} \\
 \hline
 \Gamma \vdash s_1 : T_{11} \rightarrow T_{12} \quad (S\text{-ARROW}) \\
 \hline
 \Gamma \vdash s_1 : T_{11} <: S_{11} \quad \Gamma \vdash s_2 : T_{12} <: S_{12} \\
 \hline
 \Gamma \vdash s_2 : T_{11} \rightarrow T_{12} \quad (T\text{-SUB}) \\
 \hline
 \Gamma \vdash s_1 : T_{11} \rightarrow T_{12} \\
 \hline
 \Gamma \vdash s_1 \ s_2 : T_{12} \quad (T\text{-APP}) \\
 \hline
 \vdots
 \end{array}$$

becomes

$$\begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash s_1 : S_{11} \rightarrow S_{12} \\
 \hline
 \Gamma \vdash s_1 \ s_2 : S_{12} \\
 \hline
 \Gamma \vdash s_1 \ s_2 : T_{12} \quad (T\text{-APP}) \\
 \hline
 \Gamma \vdash s_1 : T_{11} <: S_{11} \quad \Gamma \vdash s_2 : T_{12} <: S_{12} \\
 \hline
 \Gamma \vdash s_1 \ s_2 : T_{12} \quad (T\text{-SUB}) \\
 \hline
 \vdots \\
 \hline
 S_{12} <: T_{12} \quad (T\text{-SUB}) \\
 \hline
 \Gamma \vdash s_1 \ s_2 : T_{12}
 \end{array}$$

## Example

$$\begin{array}{c} \vdots \\ \hline \Gamma \vdash s_1 : T_{11} \rightarrow T_{12} \\ \hline \Gamma \vdash s_1 \ s_2 : T_{12} \end{array} \quad \begin{array}{c} \vdots \\ \hline \Gamma \vdash s_2 : T_2 \\ \hline \Gamma \vdash s_2 <: T_{11} \\ \hline \vdots \end{array} \quad \begin{array}{c} \vdots \\ \hline \Gamma \vdash s_1 \ s_2 : T_{11} \\ \hline \Gamma \vdash s_1 \ s_2 : T_{12} \end{array} \quad \begin{array}{c} \vdots \\ \hline \Gamma \vdash s_1 \ s_2 : T_{11} \\ \hline \Gamma \vdash s_1 \ s_2 : T_{12} \end{array} \quad \begin{array}{c} \vdots \\ \hline \Gamma \vdash s_1 \ s_2 : T_{11} \\ \hline \Gamma \vdash s_1 \ s_2 : T_{12} \end{array}$$



## Example

$$\begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash s : s \\
 \hline
 \Gamma \vdash s : u \\
 \hline
 \vdots \\
 \hline
 \Gamma \vdash s : u \\
 \hline
 \Gamma \vdash s : T
 \end{array}
 \quad
 \begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash s : s \\
 \hline
 \Gamma \vdash s : u \\
 \hline
 \vdots \\
 \hline
 \Gamma \vdash s : u \\
 \hline
 \Gamma \vdash s : T
 \end{array}
 \quad
 \begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash s : s \\
 \hline
 \Gamma \vdash s : u \\
 \hline
 \vdots \\
 \hline
 \Gamma \vdash s : u \\
 \hline
 \Gamma \vdash s : T
 \end{array}$$

# Example

$$\begin{array}{c}
 \vdots \\
 \vdots \\
 \frac{\Gamma \vdash s : s}{\vdots} \\
 \frac{\Gamma \vdash s : u \quad \Gamma \vdash s : s}{\Gamma \vdash s : u} \text{ (T-SUB)} \\
 \frac{\Gamma \vdash s : T}{\vdots} \text{ (T-SUB)}
 \end{array}$$

becomes

$$\begin{array}{c}
 \vdots \\
 \vdots \\
 \frac{\Gamma \vdash s : u \quad \Gamma \vdash s : T}{\Gamma \vdash s : T} \text{ (S-TRANS)} \\
 \frac{\Gamma \vdash s : s \quad \Gamma \vdash s : T}{\Gamma \vdash s : T} \text{ (T-SUB)}
 \end{array}$$