Subtyping (Review)

**Subtype Relation**

\[
\{ u \in S \mid \forall i \in I : T_i \subseteq u \} \Downarrow \{ v \in S \mid \forall i \in I : T_i \subseteq v \}
\]

For each \( i \in I \):

- \( S \rightarrow T \)
- \( S \rightarrow U \)
- \( S \rightarrow T \)

(\( S \rightarrow \text{Perm} \))

- \( \{ u \in S \mid \forall i \in I : T_i \subseteq u \} \Downarrow \{ v \in S \mid \forall i \in I : T_i \subseteq v \} \)

(\( S \rightarrow \text{Perm} \))

- \( S \rightarrow U \)
- \( S \rightarrow T \)

(\( S \rightarrow \text{Perm} \))

- \( S \rightarrow S \)

Administrivia

- Homework requests due Monday. Pick up exams next week.
- View your exams with Cheryl Hedley (Lecture 902). (Or Rannula Hanmo.)
- No office hours on Thursday or Friday.
- Class on Wednesday, but no recitations due to Thanksgiving.
- November 30.
- Homework 8 will be on website by end of today. Due Wednesday.
Subtyping and Variants

Subtyping Rule

(T-VarWidth)

\[ \frac{T \vdash \bot}{\bot} \]

(S-Top)

\[ \frac{S \vdash \top}{\top} \]

(S-Arrow)

\[ \frac{T_1 \vdash T_2 \quad S_1 \vdash S_2}{T_1 \cdot S_1 \vdash T_2 \cdot S_2} \]

Subtyping with Other Features

Subtyping and Variants

\[ \frac{\text{S-VariantWidth}}{\text{for each } i \quad S_i \vdash T_i} \]

\[ \frac{\text{S-VariantDepth}}{\text{is a permutation of } S_1} \]

\[ \frac{\text{S-VariantPerm}}{\text{each } i \quad T_i \vdash T_i} \]

Other typing rules as in \( \lambda \)
Subtyping and Lists

\[ S_1 \subseteq T_1 \]

When a reference is read, the context expects a \( T_1 \); so if \( S_1 \subsetneq T_1 \) then an

Why?

I.e., \( S_1 \) is not a covariant (nor a contravariant) type constructor.

\[ \text{(s-Ref)} \]

\[
\begin{array}{c}
\text{Ref } S_1 \subseteq \text{Ref } T_1 \\
S_1 \subseteq T_1 \\
T_1 \subseteq S_1
\end{array}
\]

\[ \text{(s-Ref)} \]

I.e., \( \text{Ref} \) is not a covariant (nor a contravariant) type constructor.

Why?

I.e., \( \text{Ref} \) is not a covariant (nor a contravariant) type constructor.

\[ \text{(s-List)} \]

\[
\begin{array}{c}
\text{List } S_1 \subseteq \text{List } T_1 \\
S_1 \subseteq T_1
\end{array}
\]

\[ \text{(s-List)} \]

By the same reasoning, \( \text{List} \) is a covariant type constructor.

Subtyping and References

Subtyping and Lists
Subtyping and References

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

Subtyping and Arrays

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

\[
\begin{align*}
\text{(S-Array)} & \\
\text{array } S_1 &: array T_1 \\
S_1 &: T_1 \\
S_1 &: T_1 \\
\end{align*}
\]

Subtyping and Arrays

\[
\begin{align*}
\text{(S-ArrayJava)} & \\
\text{array } S_1 &: array T_1 \\
S_1 &: T_1 \\
S_1 &: T_1 \\
\end{align*}
\]

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" capability
- Sink T: reference cell with "write" capability
- Ref T: cell with both capabilities

When a reference is written, the context provides a T, and if the actual T is ok.

When a reference is read, the context expects a T, so if S1 > T1 then an "error".

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

Subtyping and References

\[
\begin{align*}
\text{(S-Ref)} & \\
\text{Ref } S_1 &: Ref T_1 \\
S_1 &: T_1 \\
S_1 &: T_1 \\
\end{align*}
\]

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.
Observation: a value of type $T$ can be used in two different ways: as a source for values of type $S$ or as a sink for values of type $S$. This reflects the idea of splitting the reference type $T$ into three parts:

- **Source $S$**: Reference cell with read capability
- **Sink $S$**: Reference cell with write capability
- **Ref $T$**: Cell with both capabilities

These split the ref $T$ into three parts: source for values of type $T$ and as a sink for values of type $S$.
Syntax-directed rules

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

\[
\begin{align*}
L-\text{Var} & : \quad \overline{\text{L-Var}} \quad \overline{t} \\
L \to S & : \quad \overline{L \to S} \\
L & : \quad \overline{L} \\
S & : \quad \overline{S} \\
\end{align*}
\]

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" and "output" pair that can be used to derive a typing statement involving \( t \), the set of typing rules now includes two rules that can be used to derive a typing statement with subtyping both aspects of \( t \).

Non-syntax-directedness of typing

Syntax-directed sets of rules

For the output position, we can calculate the types of the main arguments directly from the subgoals, and for the input position, we can calculate the types of the terms appearing in the conclusion. Hence, if we are given a type \( \overline{t} \), we can try to find a typing statement involving \( t \) and \( \overline{t} \) that is the same as the term \( t \) and the type \( \overline{t} \). If we succeed, then we have found a type \( \overline{t} \). If it fails, then we know that \( t \) is not typable.

If \( t \) is an application, then we must proceed by using \( \overline{t} \) to see if \( \text{L-App} \) holds for \( t \). If it does, there are rules that can be used to derive a typing statement involving \( t \). And if \( t \) is an application, then the rules that can be used to derive a typing statement involving \( t \) are exactly the same as the rules that can be used to derive a typing statement involving \( \text{L-App} \).

Non-syntax-directedness of subtyping

When we extend the system with subtyping, both aspects of \( t \) and \( \overline{t} \) appear in the conclusions (so we can calculate subtypes from the main arguments directly from the subgoals, and for the input position, we can calculate the subtypes of the terms appearing in the conclusion). Hence, if we are given some \( \overline{t} \) and some \( t \), we can try to find a typing statement involving \( t \) and \( \overline{t} \) that is the same as the term \( t \) and the type \( \overline{t} \). If we succeed, then we have found a type \( \overline{t} \). If it fails, then we know that \( t \) is not typable.
Developing an algorithmic subtyping relation

Non-syntax-directedness of subtyping relation

Moreover, the subtyping relation is not syntax directed either. There are lots of ways to derive a given subtyping judgment.

To implement this rule naïvely, we'd have to guess a value for \( u \), and then substitute it into the premises. However, the subtyping relation is not syntax directed either.

\[ \frac{s \triangleright t}{u \triangleright v} \quad (S-Trans) \]

What to do?

1. Observation: We don't need 1000 ways to prove a given typing or subtyping statement.
   - Think more carefully about the typing and subtyping systems to see where we can get rid of excessive freedom.
   - Use the resulting intuitions to formulate new, "algorithmic" typing and subtyping relations.
   - Prove that the algorithmic relations are the same as the original ones in an appropriate sense.

2. The transitivity rule
   - There are lots of ways to derive a given subtyping statement.

3. To prove that the algorithmic relations are the same as the original ones in a suitable position, think more carefully about the typing and subtyping systems to see where we can get rid of excessive freedom.
   - Use the resulting intuitions to formulate new, "algorithmic" typing and subtyping relations.
   - Prove that the algorithmic relations are the same as the original ones in an appropriate sense.
Subtyping relation

\[ S \xrightarrow{\text{S-Ref}} S \]

\[ S \xrightarrow{\text{S-Trans}} T \]

\[ S \xrightarrow{\text{S-Rcd-Width}} \{ i : T \}_{i \in \text{I}} \]

\[ S \xrightarrow{\text{S-Rcd-Depth}} \{ j : S \}_{j \in \text{J}} \]

\[ S \xrightarrow{\text{S-Rcd-Perm}} \{ k : S \}_{k \in \text{K}} \]

Issues

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

Step 1: Simplify record subtyping

1. \( S \xrightarrow{\text{S-Rcd-Width}}, S \xrightarrow{\text{S-Rcd-Depth}}, \) and \( S \xrightarrow{\text{S-Rcd-Perm}} \) overlap with each other.

2. \( S \xrightarrow{\text{S-Ref}}, S \xrightarrow{\text{S-Trans}} \) overlap with everything.
Step 2: Get rid of transitivity

**Observation:** $S$-HRL is unnecessary.

**Lemma:** If $S \Rightarrow T$ can be derived, then there is a derivation that does not use $S$-Refl.

Even simpler subtype relation

**Step 3: Get rid of transitivity**

**Observation:** $S$-TRANS is unnecessary.

**Lemma:** $S$ can be derived for every type without using $S$-HRL.

Simplest subtype relation
Subtyping Algorithm (pseudo-code)

The algorithmic rules can be translated directly into code:

```plaintext
subtype(S; T) =
if T = Top, then true
elseif S = S_1 \rightleftarrows S_2 and T = T_1 \rightleftarrows T_2
then subtype(T_1; S_1) ^ subtype(S_2; T_2)
elseif S = \{k_{j_1} : S_{j_2} \in S_1::m\} and T = \{l_{i_1} : T_{i_2} \in T_1::n\}
then \forall i \in \text{true} \text{ there is some } j \in \text{true} \text{ with } k_{j_1} = l_{i_2} and
subtype(S_{j_2}; T_{i_2})
else false.
```

Soundness and completeness

**Theorem:**
S \subseteq T \iff \text{true} 

**Proof:**

**Soundness:**

```
\{w_{-1} : T_{11} \subseteq \{w_{-1} : S_{11} \rightleftarrows S_{21}\} \}
\text{ for each } \text{ true } T_{11} && \text{ true } S_{11} \subseteq S_{21} \text{ with respect to the algorithm.}
```

**Completeness:**

```
\{w_{-1} : S_{11} \rightleftarrows S_{21}\} \text{ true } \iff \text{ true }
```

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 if u \in R.

The algorithmic subtyping procedure is a decision procedure (pseudo-code)

```plaintext
true, false
```

1. if subtype(S; T) = true, then `I S \subseteq T` (hence, by soundness of the algorithmic rules, `I S \subseteq T`)
2. if subtype(S; T) = false, then not `I S \subseteq T` (hence, by completeness of the algorithmic rules, not `I S \subseteq T`)

Q: What's missing?

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

Prove it!
A decision procedure for a relation $R$ on $U$ is a total function $p$ from $U$ to a truth value 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 subtype $(S; T) = \text{true}$, then $S \subseteq T$ (hence, by soundness of the algorithmic rules, $S \subseteq T$)
2. If subtype $(S; T) = \text{false}$, then $S \not\subseteq T$ (hence, by completeness of the algorithmic rules, not $S \subseteq T$)

Q: What's missing?

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

Prove it!
A decision procedure for a relation \( R \subset \mathbb{N} \times \mathbb{N} \) is a total function that assigns true or false to pairs of natural numbers. If \( \sigma \) is a decision procedure for \( R \), then

\[
\sigma(\langle n, m \rangle) \begin{cases} 
  \text{true} & \text{if } (n, m) \in R \\
  \text{false} & \text{if } (n, m) \notin R
\end{cases}
\]

For applications, if \( R \) is the relation, then

\[
\sigma(\langle n, m \rangle) = \begin{cases} 
  \text{true} & \text{if } (n, m) \in R \\
  \text{false} & \text{if } (n, m) \notin R
\end{cases}
\]

where \( \sigma \) is the decision procedure. Since \( \sigma \) is a decision procedure for \( R \), we have just one problematic rule to deal with:

\[
\text{subsumption}
\]

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

\[
\text{subsumption}
\]

Prove it!

A: How do we know that \( \sigma \) is a local function?

Q: What is missing?

Hence, by completeness of the algorithmic rules, not \( S \Rightarrow T \)

2. If \( \text{subsume}(S,T) \Rightarrow \text{false} \), then not \( S \Rightarrow T \)

Since \( \text{subsume} \) is just an implementation of the algorithmic subsumption rules, we have:

We need a decision procedure for a relation \( R \subset \mathbb{N} \times \mathbb{N} \) is a local function from \( \mathbb{N} \) to

Decision Procedures
Forthetypingrelation,wehavejustoneproblematicruletodealwith:

\[ \text{Subsumption} \]

\[ t : S \subseteq T \]

Whereisthisrulereallyneeded?

For applications. E.g., the term

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

is not typable without using subsumption.

Where is this rule really needed?

\[ \text{Subsumption} \]

\[ t : T \]

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

\[ \text{T-Sub} \]

\[ \frac{S \subseteq T}{S \subseteq T} \]

\[ \text{T-Abs} \]

\[ \frac{S \subseteq T}{S \subseteq T} \]

\[ \text{T-Refl} \]

\[ \frac{S \subseteq T}{S \subseteq T} \]

\[ \text{T-Arrow} \]

\[ \frac{S \subseteq T}{S \subseteq T} \]