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)

Varieties of Polymorphism
- Parametric polymorphism (ML-style)
- Subtype polymorphism (OO-style)
- Ad-hoc polymorphism (overloading)
**Motivation**

With our usual typing rule for applications

\[
\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash \lambda x : T_{11}. \ y \ x : T_{12}} \quad \text{(T-App)}
\]

the term

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

is not well typed.

**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 \( S \ll T \)

2. a rule of subsumption stating that, if \( S \ll T \), then any value of type \( S \) can also be regarded as having type \( T \)

\[
\frac{\Gamma \vdash t : S \quad S \ll T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}
\]

**Example**

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

\[
\{x: \text{Nat}, y: \text{Nat}\} \ll \{x: \text{Nat}\}
\]

So, by subsumption,

\[
\vdash \{x=0,y=1\} : \{x: \text{Nat}\}
\]

and hence

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

is well typed.
The Subtype Relation: General rules

\[ S < S \quad \text{(S-REFL)} \]

\[
\begin{align*}
S & < U \quad U < T \\
S & < T 
\end{align*}
\quad \text{(S-TRANS)}
\]

The Subtype Relation: Records

“Width subtyping” (forgetting fields on the right):

\[
\{l_i : T_i \mid i \in \mathbb{N}^n \} \lessdot \{l_i : T_i \mid i \in \mathbb{N}^k \} \quad \text{(S-RcdWidth)}
\]

Intuition: \((x : \text{Nat})\) 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:

\[
\text{for each } i \quad S_i < T_i \quad \text{ (S-RcdDepth)}
\]

\[
\{l_i : S_i \mid i \in \mathbb{N}^n \} < \{l_i : T_i \mid i \in \mathbb{N}^n \}
\]

Example

\[
\begin{align*}
\{a : \text{Nat}, b : \text{Nat}\} & < \{a : \text{Nat}\} \quad \text{(S-RcdWidth)} \\
\{a : \text{Nat}\} & < \{\} \quad \text{(S-RcdWidth)} \\
\{x : \{a : \text{Nat}, b : \text{Nat}\}, y : \{m : \text{Nat}\}\} & < \{x : \{a : \text{Nat}\}, y : \{\}\}\quad \text{(S-RcdDepth)}
\end{align*}
\]
The Subtype Relation: Records

Permutation of fields:

\[ \{ k_j : S_j \ | _{1 \leq j \leq n} \} \text{ is a permutation of } \{ l_i : T_i \ | _{1 \leq i \leq n} \} \]  

(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

\[ T_1 \leq S_1 \quad S_2 \leq T_2 \]  

\[ S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \]  

(S-AARROW)

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 \( S_1 \rightarrow S_2 \), 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 \( S_1 \rightarrow S_2 \) can also be viewed as having type \( T_1 \rightarrow T_2 \).

The Subtype Relation: Top

It is convenient to have a type that is a supertype of every type. We introduce a new type constant \( \text{Top} \), plus a rule that makes \( \text{Top} \) a maximum element of the subtype relation.

\[ S \leq \text{Top} \]  

(S-TOP)

Cf. \textbf{Object} in Java.
**Review**

**Subtyping**

Intuitions: $S \ll T$ 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**

- $S \ll S$  \hspace{1cm} \text{(S-REFL)}
- $S \ll U$  \hspace{0.5cm} $U \ll T$  \hspace{0.5cm} $S \ll T$ \hspace{1cm} \text{(S-TRANS)}
- $\{l_1 : T_l \mid l \in E_1, n \}$ $\ll$ $\{l_1 : T_l \mid l \in E_1, n \}$ \hspace{1cm} \text{(S-RCDWIDTH)}
- for each $i$ $S_i \ll T_i$ \hspace{1cm} $\{l_1 : S_l \mid l \in E_1, n \}$ $\ll$ $\{l_1 : T_l \mid l \in E_1, n \}$ \hspace{1cm} \text{(S-RCDDEPTH)}
- $\{k_j : S_j \mid j \in E_1, n \}$ is a permutation of $\{l_1 : T_l \mid l \in E_1, n \}$ \hspace{1cm} $\{k_j : S_j \mid j \in E_1, n \}$ $\ll$ $\{l_1 : T_l \mid l \in E_1, n \}$ \hspace{1cm} \text{(S-RCDPERM)}

\[ T_1 \ll S_1 \quad S_2 \ll T_2 \]
\[ S_1 \rightarrow S_2 \ll T_1 \rightarrow T_2 \] \hspace{1cm} \text{(S-ARROW)}

\[ S \ll \text{Top} \] \hspace{1cm} \text{(S-TOP)}
**Subsumption Rule**

\[
\frac{\Gamma \vdash t : S \quad S \preceq T}{\Gamma \vdash t : T} \quad (T\text{-}Sub)
\]

Other typing rules as in $\lambda \to$.

---

**Safety**

Statements of progress and preservation theorems are unchanged from $\lambda \to$.

Proofs become a bit more involved, because the typing relation is no longer syntax directed.

---

**Preservation**

**Theorem:** If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

**Proof:** By induction on typing derivations.

(Which cases are hard?)
Subsumption case

Case T-SUB: \[ t : S \quad S \subset T \]

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

Not hard!

Application case

Case T-APP:

\[
\begin{align*}
& t = t_1 \quad t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \\
\hline
& \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \\
& \hline
& \Gamma \vdash t_1 \ t_2 : T_{12}
\end{align*}
\]

By the inversion lemma for evaluation, there are three rules by which \( t \rightarrow t' \) can be derived: \( E\text{-App}1 \), \( E\text{-App}2 \), and \( E\text{-AppAbs} \). Proceed by cases.
Application case

Case T-APP:
\[ t \rightarrow t' \quad t' \rightarrow t'' \]
\[ \Gamma \vdash t : T_1 \rightarrow T_2 \quad \Gamma \vdash t : T_1 \quad T = T_2 \]

By the inversion lemma for evaluation, there are three rules by which \( t \rightarrow t' \) can be derived: E-APP1, E-APP2, and E-APPABS. Proceed by cases.

Subcase E-APP1: \( t_1 \rightarrow t' \quad t' \rightarrow t'' \)

The result follows from the induction hypothesis and T-APP.

\[ \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1 \quad (T-APP) \]
\[ \Gamma \vdash t_1 \quad t_2 : T_1 \]
\[ \Gamma \vdash t_1 \rightarrow t' \quad (E-APP1) \]
\[ \Gamma \vdash t_2 \rightarrow t'' \]
\[ t_1 \rightarrow t' \rightarrow t'' \quad t_2 \rightarrow t'' \]

Case T-APP (CONTINUED):
\[ t \rightarrow t_1 \quad t_2 \quad \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1 \quad T = T_2 \]

Subcase E-APP2: \( t_1 \rightarrow v_1 \quad t_2 \rightarrow t'_2 \quad t' \rightarrow v_1 \quad t'_2 \)

Similar:

\[ \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1 \]
\[ \Gamma \vdash t_1 \quad t_2 : T_1 \quad (T-APP) \]
\[ \Gamma \vdash t_2 \rightarrow t'_2 \quad (E-APP2) \]
\[ v_1 \quad t_2 \rightarrow v_1 \quad t'_2 \]

Case T-APP (CONTINUED):
\[ t \rightarrow t_1 \quad t_2 \quad \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1 \quad T = T_2 \]

Subcase E-APPABS: \( t_1 \rightarrow \lambda x : S_1 : t_2 \quad t_2 \rightarrow v_2 \quad t' \rightarrow [x \mapsto v_2] t_{12} \)

By the inversion lemma for the typing relation...

\[ \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1 \quad (T-APP) \]
\[ \Gamma \vdash t_1 \quad t_2 : T_1 \]
\[ \Gamma \vdash [x \mapsto v_2] t_{12} \]

Case T-APP (CONTINUED):
\[ t \rightarrow t_1 \quad t_2 \quad \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1 \quad T = T_2 \]

Subcase E-APPABS: \( t_1 \rightarrow \lambda x : S_1 : t_1 \rightarrow t_2 \quad t_2 \rightarrow v_2 \quad t' \rightarrow [x \mapsto v_2] t_{12} \)

By the inversion lemma for the typing relation... \( T_{11} \rightarrow S_{11} \) and \( \Gamma, x : S_1 \vdash t_1 \rightarrow t_2 : T_1 \).

\[ \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1 \quad (T-APP) \]
\[ \Gamma \vdash t_1 \quad t_2 : T_1 \]
\[ \Gamma \vdash [x \mapsto v_2] t_{12} \]

\[(\lambda x : T_{11} : t_{12}) \rightarrow [x \mapsto v_2] t_{12} \quad (E-APPABS)\]
Case T-APP (CONTINUED):

\[
\begin{align*}
t & \rightarrow t_1 \quad t_2 & \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} & \quad \Gamma \vdash t_2 : T_{11} & \quad T_1 & \rightarrow T_{12}
\end{align*}
\]

Subcase E-APPABS:\n
\[
\begin{align*}
t_1 \rightarrow \lambda x : S_{11} \cdot t_{12} & \quad t_2 \rightarrow v_2 & \quad t' \rightarrow [x \mapsto v_2]_{t_{12}}
\end{align*}
\]

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, \( \Gamma \vdash t_2 : S_{11} \).

**Lemma:** If \( \Gamma \vdash \lambda x : S_1 \cdot s_2 : T_1 \rightarrow T_2 \), then \( T_1 \rightarrow S_1 \) and \( \Gamma, x : S_1 \vdash s_2 : T_2 \).

**Proof:** Induction on typing derivations.

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 $\Gamma \vdash \lambda x : S_1 . s_2 : T_1 \rightarrow T_2$, then $T_1 < S_1$ and $\Gamma, x : S_1 \vdash s_2 : T_2$.

**Proof:** Induction on typing derivations.

**Case T-Sub:** $\lambda x : S_1 . s_2 : U \quad U <: T_1 \rightarrow T_2$

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 $U <: T_1 \rightarrow T_2$, 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.)

By this lemma, we know $U = U_1 \rightarrow U_2$, 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 $\Gamma \vdash \lambda x : S_1 . s_2 : T_1 \rightarrow T_2$, then $T_1 < S_1$ and $\Gamma, x : S_1 \vdash s_2 : T_2$.

**Proof:** Induction on typing derivations.

**Case T-Sub:** $\lambda x : S_1 . s_2 : U \quad U <: T_1 \rightarrow T_2$

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 $U <: T_1 \rightarrow T_2$, 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.)

By this lemma, we know $U = U_1 \rightarrow U_2$, 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 $\Gamma \vdash x : S_1, s_2 : T_1 \rightarrow T_2$, then $T_1 \prec S_1$ and $\Gamma, x : S_1 \vdash s_2 : T_2$.

**Proof:** Induction on typing derivations.

**Case T-SUB:** $\lambda x : S_1.s_2 : U \quad U \prec T_1 \rightarrow T_2$

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 $U \prec T_1 \rightarrow T_2$, then $U$ has the form $U_1 \rightarrow U_2$, with $T_1 \prec U_1$ and $U_2 \prec T_2$. (Proof: by induction on subtyping derivations.)

By this lemma, we know $U = U_1 \rightarrow U_2$, with $T_1 \prec U_1$ and $U_2 \prec T_2$.

The IH now applies, yielding $U_1 \prec S_1$ and $\Gamma, x : S_1 \vdash s_2 : U_2$.

From $U_1 \prec S_1$ and $T_1 \prec U_1$, rule S-TRANS gives $T_1 \prec S_1$.

From $\Gamma, x : S_1 \vdash s_2 : U_2$ and $U_2 \prec T_2$, rule T-SUB gives $\Gamma, x : S_1 \vdash s_2 : T_2$, and we are done.

Ascription and Casting

**Ordinary ascription:**

<table>
<thead>
<tr>
<th>$\Gamma \vdash t_1 : T$</th>
<th>(T-ASCRIBE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash t_1$ as $T : T$</td>
<td></td>
</tr>
<tr>
<td>$v_1$ as $T \rightarrow v_1$</td>
<td>(E-ASCRIBE)</td>
</tr>
</tbody>
</table>

**Casting (cf. Java):**

<table>
<thead>
<tr>
<th>$\Gamma \vdash t_1 : S$</th>
<th>(T-CAST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash t_1$ as $T : T$</td>
<td></td>
</tr>
<tr>
<td>$v_1$ as $T \rightarrow v_1$</td>
<td>(E-CAST)</td>
</tr>
</tbody>
</table>
Subtyping and Variants

\( \langle l_1 : T_{i_1} \rangle^{i_{E_1}, \ldots, i_n} \) \quad \text{for each } i \quad S_i \triangleleft T_i \quad \text{(S-VARIANTWIDTH)}

\( \langle l_1 : S_i \rangle^{i_{E_i}, \ldots, i_n} \) \quad \text{is a permutation of } \langle l_1 : T_{i_1} \rangle^{i_{E_1}, \ldots, i_n} \quad \text{(S-VARIANTPERM)}

\( \Gamma \vdash t_1 : T_1 \)
\( \Gamma \vdash \langle l_1 = t_1 \rangle : \langle l_1 : T_1 \rangle \) \quad \text{(T-VARIANT)}

Subtyping and Lists

\( S_i \triangleleft T_i \)
\( \text{List } S_i \triangleleft \text{List } T_i \) \quad \text{(S-LIST)}

\( \) i.e., List is a covariant type constructor.

Subtyping and References

\( S_i \triangleleft T_i \quad T_i \triangleleft S_i \) \quad \text{(S-REF)}
\( \) i.e., Ref is not a covariant (nor a contravariant) type constructor.

Subtyping and Arrays

\( S_i \triangleleft T_i \quad T_i \triangleleft S_i \) \quad \text{(S-ARRAY)}
\( \) i.e., Array is a covariant type constructor.

Similarly...

\( S_i \triangleleft T_i \quad T_i \triangleleft S_i \) \quad \text{(S-ARRAY)}
\( \) i.e., Array is a covariant type constructor.
Subtyping and Arrays

Similarly...

\[
S_1 < T_1 \quad T_1 < S_1 \quad (S-ARRAY)
\]

Array \(S_1 < \text{Array } T_1\)

\[
S_1 < T_1 \quad (S-ARRAYJAVA)
\]

Array \(S_1 < \text{Array } T_1\)

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 \(T\).

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

Modified Typing Rules

\[
\Gamma | \Sigma \vdash t_1 : \text{Source } T_{11}
\]

\[
\Gamma | \Sigma \vdash !t_1 : T_{11}
\quad (T-DEREF)
\]

\[
\Gamma | \Sigma \vdash t_1 : \text{Sink } T_{11}
\quad \Gamma | \Sigma \vdash t_2 : T_{11}
\]

\[
\Gamma | \Sigma \vdash t_1 := t_2 : \text{Unit}
\quad (T-ASSIGN)
Subtyping rules

\[ S_1 \ll T_1 \quad \text{(S-SOURCE)} \]
\[ \text{Source } S_1 \ll \text{Source } T_1 \]

\[ T_1 \ll S_1 \quad \text{(S-SINK)} \]
\[ \text{Sink } S_1 \ll \text{Sink } T_1 \]

\[ \text{Ref } T_1 \ll \text{Source } T_1 \quad \text{(S-REFSOURCE)} \]
\[ \text{Ref } T_1 \ll \text{Sink } T_1 \quad \text{(S-REFSINK)} \]

Capabilities

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

Intersection Types

The inhabitants of \( T_1 \land T_2 \) are terms belonging to both \( S \) and \( T \)—i.e., \( T_1 \land T_2 \) is an order-theoretic meet (greatest lower bound) of \( T_1 \) and \( T_2 \).

\[ T_1 \land T_2 \ll T_1 \quad \text{(S-INTER1)} \]
\[ T_1 \land T_2 \ll T_2 \quad \text{(S-INTER2)} \]

\[ S \ll T_1 \quad S \ll T_2 \quad \text{(S-INTER3)} \]
\[ S \ll T_1 \land T_2 \]

\[ S \rightarrow T_1 \land S \rightarrow T_2 \ll S \rightarrow (T_1 \land T_2) \quad \text{(S-INTER4)} \]
Intersection Types

Intersection types permit a very flexible form of finitary overloading.

\[ + : (\text{Nat} \to \text{Nat} \to \text{Nat}) \land (\text{Float} \to \text{Float} \to \text{Float}) \]

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.

\[ \rightarrow \text{type reconstruction problem is undecidable} \]

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.

\[ T_1 \lor T_2 \] is an untagged (non-disjoint) union of \( T_1 \) and \( T_2 \).

No tags \( \rightarrow \) no case construct. The only operations we can safely perform on elements of \( T_1 \lor T_2 \) 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 ignore this restriction, allowing any operation on an element of \( T_1 \lor T_2 \) 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).

Syntax-directed rules

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

\[
\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \quad \text{(T-APP)}
\]

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 \( T_{11} \to T_{12} \)
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 ($\Gamma$ and $t$) and output positions ($\Gamma$).

- 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).

$$\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}$$

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

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 is 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.

\[\rightarrow\] 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)

$$\Gamma \vdash t : S \quad S \prec \prec T$$

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

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

$$S \prec \prec U \quad U \prec \prec T$$

$$S \prec \prec T \quad (S-\text{TRANS})$$

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?

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.