**Administrivia**

Reminder: Midterm II is next Wednesday, November 17th.

Covering all material we’ve seen so far, up through Chapter 14 of TAPL (but omitting Chapters 6, 7, 9 and 12). Emphasizing material covered since the last midterm.

Exams from last two years on website. Ignore questions on subtyping.

---

**Polymorphism**

The T-App rule is very restrictive.

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

A **polymorphic** function may be applied to many different types of data.

Varieties of polymorphism:

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

With our usual typing rule for applications

\[
\begin{align*}
\Gamma \vdash t_1 &: T_{11} \rightarrow T_{12} & \Gamma \vdash t_2 &: T_{11} \\
\Gamma \vdash t_1 \; t_2 &: T_{12}
\end{align*}
\] (T-App)

the term

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

is not well typed.

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

Example

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

\[
\{x : \text{Nat}, \; y : \text{Nat}\} < : \{x : \text{Nat}\}
\]

So, by subsumption,

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

and hence

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

is 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 \( S < : T \)

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

\[
\begin{align*}
\Gamma \vdash t &: S & S < : T \\
\Gamma \vdash t &: T
\end{align*}
\] (T-SUB)

Example

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

\[
\{x : \text{Nat}, \; y : \text{Nat}\} < : \{x : \text{Nat}\}
\]

So, by subsumption,

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

and hence

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

is well typed.
The Subtype Relation: Records

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

\[ \{l_i : T_i \mid i \in \{1, \ldots, n\} \} \subset \{l_i : T_i \mid i \in \{1, \ldots, n+k\} \} \quad \text{(S-RcdWidth)} \]

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.

The Subtype Relation: General rules

\[
\begin{align*}
S & \ll S & \text{(S-Refl)} \\
S & \ll U & U \ll T & \text{(S-Trans)}
\end{align*}
\]

“Depth subtyping” within fields:

\[
\begin{align*}
\text{for each } i & & S_i \ll T_i & \text{(S-RcdDepth)}
\end{align*}
\]
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: Records

Permutation of fields:

\[
\{k_1:S_{i_1} \ldots i_n\} \text{ is a permutation of } \{l_1:T_{i_1} \ldots i_n\} \quad \text{(S-RcdPerm)}
\]

By using S-RcdPerm together with S-RcdWidth and S-Trans, we can drop arbitrary fields within records.

The Subtype Relation: Arrow types

\[
\frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2} \quad \text{(S-Arrow)}
\]

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 \(S_1 \rightarrow S_2\), 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.

\[
S \leq \text{Top} \quad \text{(S-Top)}
\]

Cf. \texttt{Object} in Java.
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.

Given a derivation, we don’t always know what rule was used in the last step.

Subsumption case

Case T-Sub: $t : S$ $S \subseteq T$

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: \( t : S \quad S \preceq T \)

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

Not hard!

Application case

Case T-APP:

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

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: \( t_1 \rightarrow t'_1 \quad t' = t'_1 \cdot t_2 \)

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

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

\[
\begin{array}{c}
\Gamma \vdash t_1 \quad t_2 : T_{12} \\
\hline
\Gamma \vdash t_1 \quad t'_1 \\
\hline
\Gamma \vdash t_1 \quad t'_2
\end{array}
\]

(T-APP)

\[
\begin{array}{c}
\Gamma \vdash t_1 \quad t'_1 \\
\hline
\Gamma \vdash t_1 \quad t_2 \rightarrow t'_1 \quad t_2
\end{array}
\]

(E-APP1)
Case T-App (continued):
\[ t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

Subcase E-AppAbs: \[ t_1 = \lambda x : T_{11} \cdot \ t_2 \quad t_2 = v_2 \quad t' = [x \mapsto v_2] \ t_1 \]

By the inversion lemma for the typing relation...

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

\[ \Gamma \vdash t_1 \ t_2 : T_{12} \]

\[ (\lambda x : T_{11} \cdot t_{12}) \ v_2 \rightarrow [x \mapsto v_2] t_{12} \]  

(E-AppAbs)

---

Case T-App (continued):
\[ t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

Subcase E-AppAbs: \[ t_1 = \lambda x : T_{11} \cdot \ t_2 \quad t_2 = v_2 \quad t' = [x \mapsto v_2] \ t_1 \]

By the inversion lemma for the typing relation... \[ T_{11} \ll T_{12} \]

By T-Sub, \[ \Gamma \vdash t_2 : T_{11} \]

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

\[ \Gamma \vdash t_1 \ t_2 : T_{12} \]

\[ (\lambda x : T_{11} \cdot t_{12}) \ v_2 \rightarrow [x \mapsto v_2] t_{12} \]

(E-AppAbs)

---

Case T-App (continued):
\[ t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

Subcase E-App2: \[ t_1 = v_1 \quad t_2 \rightarrow t'_2 \quad t' = v_1 \ t'_2 \]

Similar.

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

\[ \Gamma \vdash t_1 \ t_2 : T_{12} \]

\[ t_2 \rightarrow t'_2 \]

\[ (\lambda x : T_{11} \cdot t_{12}) \ v_1 \ t_2 \rightarrow v_1 \ t'_2 \]  

(E-App2)

---

Case T-App (continued):
\[ t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

Subcase E-AppAbs: \[ t_1 = \lambda x : T_{11} \cdot \ t_2 \quad t_2 = v_2 \quad t' = [x \mapsto v_2] \ t_1 \]

By the inversion lemma for the typing relation... \[ T_{11} \ll T_{12} \]

By T-Sub, \[ \Gamma \vdash t_2 : T_{11} \]

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

\[ \Gamma \vdash t_1 \ t_2 : T_{12} \]

\[ (\lambda x : T_{11} \cdot t_{12}) \ v_2 \rightarrow [x \mapsto v_2] t_{12} \]  

(E-AppAbs)

---
**InversionLemma for Typing**

**Lemma:** If $\Gamma \vdash \lambda x:S_1.s_2 : T_1 \rightarrow T_2$, then $T_1 \ll S_1$ and $\Gamma, x:S_1 \vdash s_2 : T_2$.

**Proof:** Induction on typing derivations.

---

**Case T-App:**

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

**Subcase E-AppAbs:**

$t_1 = \lambda x:S_{11} \cdot t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2]t_{12}$

By the inversion lemma for the typing relation... $T_{11} \ll S_{11}$ 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.

$$
\begin{array}{c}
\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \\
\Gamma \vdash t_2 : T_{11} \\
\hline
\Gamma \vdash t_1 t_2 : T_{12}
\end{array}
$$

**(T-App)**

$$
(\lambda x:T_{11}. t_{12}) v_2 \mapsto [x \mapsto v_2]t_{12}
$$

**(E-AppAbs)**
Inversion Lemma for Typing

**Lemma:** If $\Gamma \vdash \lambda x:S_1.s_2 : T_1 \rightarrow T_2$, then $T_1 \ll S_1$ and $\Gamma, x:S_1 \vdash s_2 : T_2$.

**Proof:** Induction on typing derivations.

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

We want to say “By the induction hypothesis...”, but the IH does not apply (we do not know that $U$ is an arrow type). Need another lemma...

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

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

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

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

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

Ordinary ascription:

\[ \Gamma \vdash t_1 : T \]
\[ \Gamma \vdash t_1 \text{ as } T : T \]  
(T-ASCRIBE)

\[ v_1 \text{ as } T \rightarrow v_1 \]  
(E-ASCRIBE)

Casting (cf. Java):

\[ \Gamma \vdash t_1 : S \]
\[ \Gamma \vdash t_1 \text{ as } T : T \]
\[ \vdash v_1 : T \]
\[ v_1 \text{ as } T \rightarrow v_1 \]  
(T-CAST)

(E-CAST)

Subtyping and Variants

\[ \langle l_1 : T_1 \rangle^{i \in I \ldots n} \lesssim \langle l_1 : T_1 \rangle^{i \in I \ldots n + k} \]  
(S-VARIANTWIDTH)

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

\[ \langle k_i : S_i \rangle^{i \in I \ldots n} \]  
is a permutation of \[ \langle l_1 : T_1 \rangle^{i \in I \ldots n} \]  
(S-VARIANTPERM)

\[ \Gamma \vdash t_1 : T_1 \]
\[ \Gamma \vdash \langle l_1 = t_1 \rangle : \langle l_1 : T_1 \rangle \]  
(T-VARIANT)

Ascription and Casting

Ordinary ascription:

\[ \Gamma \vdash t_1 : T \]
\[ \Gamma \vdash t_1 \text{ as } T : T \]  
(T-ASCRIBE)

\[ v_1 \text{ as } T \rightarrow v_1 \]  
(E-ASCRIBE)

Casting (cf. Java):

\[ \Gamma \vdash t_1 : S \]
\[ \Gamma \vdash t_1 \text{ as } T : T \]
\[ \vdash v_1 : T \]
\[ v_1 \text{ as } T \rightarrow v_1 \]  
(T-CAST)

(E-CAST)
Subtyping and Lists

\[
S_1 \triangleleft T_1 \quad T_1 \triangleleft S_1 \\
\text{List } S_1 \triangleleft \text{List } T_1
\]

(S-List)

I.e., \textit{List} is a covariant type constructor.

Subtyping and References

\[
S_1 \triangleleft T_1 \quad T_1 \triangleleft S_1 \\
\text{Ref } S_1 \triangleleft \text{Ref } T_1
\]

(S-Ref)

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

Why?

\begin{itemize}
  \item When a reference is \textit{read}, the context expects a \textit{T}, so if \textit{S} \triangleleft \textit{T} then an \textit{S} is ok.
  \item When a reference is \textit{written}, the context provides a \textit{T} and if the actual type of the reference is \textit{Ref S}, someone else may use the \textit{T} as an \textit{S}. So we need \textit{T} \triangleleft \textit{S}.
\end{itemize}
Subtyping and Arrays

Similarly...

\[ S_1 <: T_1 \quad T_1 <: S_1 \]

\[ \text{Array } S_1 <: \text{Array } T_1 \]  

(S-ARRAY)

\[ S_1 <: T_1 \]

\[ \text{Array } S_1 <: \text{Array } T_1 \]  

(S-ARRAY.JAVA)

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

References again

Observation: a value of type \texttt{Ref T} can be used in two different ways: as a source for values of type \texttt{T} and as a sink for values of type \texttt{T}.

Idea: Split \texttt{Ref T} into three parts:

- \texttt{Source T}: reference cell with “read capability”
- \texttt{Sink T}: reference cell with “write capability”
- \texttt{Ref T}: cell with both capabilities
Subtyping rules

\[ S_1 <: T_1 \]

\[ \text{Source } S_1 <: \text{Source } T_1 \quad (S-\text{Source}) \]

\[ T_1 < S_1 \]

\[ \text{Sink } S_1 <: \text{Sink } T_1 \quad (S-\text{Sink}) \]

\[ \text{Ref } T_1 <: \text{Source } T_1 \quad (S-\text{RefSource}) \]

\[ \text{Ref } T_1 <: \text{Sink } T_1 \quad (S-\text{RefSink}) \]

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 \( T_2 \).

\[ T_1 \land T_2 <: T_1 \quad (S-\text{Inter1}) \]

\[ T_1 \land T_2 <: T_2 \quad (S-\text{Inter2}) \]

\[ S <: T_1 \quad S <: T_2 \]

\[ S <: T_1 \land T_2 \quad (S-\text{Inter3}) \]

\[ S \rightarrow T_1 \land S \rightarrow T_2 <: S \rightarrow (T_1 \land T_2) \quad (S-\text{Inter4}) \]

Modified Typing Rules

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

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

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

\[ \Gamma | \Sigma \vdash t_2 : T_{11} \]

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

Capabilities

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

T₁ ∨ T₂ is an untagged (non-disjoint) union of T₁ and T₂.

No tags → no case construct. The only operations we can safely perform on elements of T₁ ∨ T₂ are ones that make sense for both T₁ and T₂.

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 T₁ ∨ T₂ that makes sense for either T₁ or T₂.

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.

\[ \Gamma \vdash t₁ : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t₂ : T_{11} \]
\[ \Gamma \vdash t₁ \ t₂ : T_{12} \]

(T-App)

If we are given some Γ and some \( t \) of the form \( t₁ \ t₂ \), we can try to find a type for \( t \) by
1. finding (recursively) a type for \( t₁ \)
2. checking that it has the form \( T_{11} \rightarrow T_{12} \)
3. finding (recursively) a type for \( t₂ \)
4. checking that it is the same as \( T_{11} \)

Intersection types permit a very flexible form of finitary overloading.

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

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.

→ 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).
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)

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

\[ \Gamma \vdash t_1 \cdot t_2 : T_{12} \quad \text{(T-App)} \]

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)

\[ \Gamma \vdash t : S \quad S <: T \]

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

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.)
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.