Lecture 11: Subtyping

Prof. Liang Huang

huang@qc.cs.cuny.edu
Big Picture

• Part I: Fundamentals
  • Functional Programming and Basic Haskell
  • Proof by Induction and Structural Induction

• Part II: Simply-Typed Lambda-Calculus
  • Untyped Lambda Calculus
  • Simply Typed Lambda Calculus
  • Extensions: Units, Records, Variants
  • References and Memory Allocation

• Part III: Object-Oriented Programming
  • Basic Subtyping
  • Case Study: Featherweight Java

```java
class A extends Object { A() { super(); } }
class B extends Object { B() { super(); } }
class Pair extends Object {
    Object fst;
    Object snd;
    // Constructor:
    Pair(Object fst, Object snd) {
        super(); this.fst=fst; this.snd=snd;}
    // Method definition:
    Pair setfst(Object newfst) {
        return new Pair(newfst, this.snd); }
}
Subtyping
Motivation

With our usual typing rule for applications

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

\((T\text{-APP})\)

the term

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

is *not* well typed.
Motivation

With our usual typing rule for applications

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

(T-APP)

the term

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

is not well typed.

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

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)

Our topic for the next few lectures is *subtype* polymorphism, which is based on the idea of *subsumption*. 

C++ templates
C++ subclass
C++ operator overloading
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 $T$

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

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

Intuition: \( \{ x : \text{Nat} \} \) is the type of all records with \textit{at least} a numeric \( x \) field.

Note that the record type with \textit{more} fields is a subtype of the record type with fewer fields.

Reason: the type with more fields places a \textit{stronger constraint} on values, so it describes \textit{fewer values}.
The Subtype Relation: Records

Permutation of fields:

\[
\{k_j : S_j \ j \in 1..n\} \text{ is a permutation of } \{l_i : T_i \ i \in 1..n\}
\]

\[
\{k_j : S_j \ j \in 1..n\} <: \{l_i : T_i \ i \in 1..n\} \quad (S\text{-RcdPerm})
\]

By using \( S\text{-RcdPerm} \) together with \( S\text{-RcdWidth} \) and \( S\text{-Trans} \) allows us to drop arbitrary fields within records.

\[
S <: U \quad U <: T \\
\hline
S <: T \quad (S\text{-Trans})
\]
The Subtype Relation: Records

“Depth subtyping” within fields:

\[
\text{for each } i \quad S_i <: T_i \\
\{ l_i : S_{i}^{\infty} \} <: \{ l_i : T_{i}^{\infty} \} \quad (S-\text{RcdDepth})
\]

The types of individual fields may change.
Example

\[
\begin{align*}
  \{a: \text{Nat}, b: \text{Nat}\} & \ll \{a: \text{Nat}\} \\
  \{m: \text{Nat}\} & \ll \{\} \\
  \{x: \{a: \text{Nat}, b: \text{Nat}\}, y: \{m: \text{Nat}\}\} & \ll \{x: \{a: \text{Nat}\}, y: \{\}\} \\
  \{a: \text{Nat}, b: \text{Nat}\} & \ll \{a: \text{Nat}\} \\
  \{m: \text{Nat}\} & \ll \{m: \text{Nat}\} \\
  \{x: \{a: \text{Nat}, b: \text{Nat}\}, y: \{m: \text{Nat}\}\} & \ll \{x: \{a: \text{Nat}\}, y: \{m: \text{Nat}\}\} \\
  \{x: \{a: \text{Nat}, b: \text{Nat}\}\} & \ll \{a: \text{Nat}\} \\
  \{x: \{a: \text{Nat}, b: \text{Nat}\}\} & \ll \{x: \{a: \text{Nat}\}\} \\
  \{x: \{a: \text{Nat}, b: \text{Nat}\}, y: \{m: \text{Nat}\}\} & \ll \{x: \{a: \text{Nat}\}\}
\end{align*}
\]
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)
  
  \[ \text{each class member (field or method) can be assigned a single index, adding new indices "on the right" as more members are added in subclasses (i.e., no permutation for classes)} \]

- A class may implement multiple interfaces ("multiple inheritance" of interfaces)
  I.e., permutation is allowed for interfaces.
The Subtype Relation: Arrow types

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

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 $T_1 \rightarrow T_2$. 
Intuition of the S-Arrow rule: “eats less, produces more” is always welcome. :)

\[ \begin{array}{ccc}
C \subseteq A & B \subseteq D & c \subseteq a & b \subseteq d \\
\text{----------------- S-Arrow analogous to -----------------}
\end{array} \]

\[ \begin{array}{ccc}
A \rightarrow B & \subseteq & C \rightarrow D \\
b \bar{a} & < & d \bar{c}
\end{array} \]

\[ f: \{x: \text{Nat}\} \rightarrow \{y: \text{Bool}\} \] can be used as input to \( g \):

\[ g = \lambda p: \{x: \text{Nat}, z: \text{Nat}\} \rightarrow \{}. \]

\( q \) (\( p \) \{x=5,z=2\})

\( (g \ f) \) typechecks because

1. \( f \) can be used in place of \( p \):
   - \( f \) expects \( \{x: \text{Nat}\} \) so \( \{x=5,z=2\} \) is a good input to \( f \).
   - (contra-variant on input).

2. on the other hand, \( (p \ \{x=5,z=2\}) \) returns type \( \{} \) which is the input type of \( q \).
   - \( f \ \{x=5,z=2\} \) outputs type \( \{y: \text{Bool}\} \) which is a good input to \( q \).

eating \( \{x: \text{Nat}\} \) and producing \( \{y: \text{Bool}\} \) is always welcome in a context which expects you to eat \( \{x: \text{Nat}, z: \text{Nat}\} \) and produce \( \{} \).
Motivation

With our usual typing rule for applications

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

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

the term

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

is *not* well typed.
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.

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

Cf. $\text{Object}$ in Java.
The Subtype Relation: General rules

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

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

\[
\begin{align*}
S & \ll S & \text{(S-REFL)} \\
S & \ll U & U \ll T & \Rightarrow S \ll T & \text{(S-TRANS)} \\
\{l_i:T_i \mid i \in \{1, \ldots, n+k\}\} & \ll \{l_i:T_i \mid i \in \{1, \ldots, n\}\} & \text{(S-RcdWidth)} \\
\text{for each } i & & S_i \ll T_i \\
\{l_i:S_i \mid i \in \{1, \ldots, n\}\} & \ll \{l_i:T_i \mid i \in \{1, \ldots, n\}\} & \text{(S-RcdDepth)} \\
\{k_j:S_j \mid j \in \{1, \ldots, n\}\} & \text{is a permutation of } \{l_i:T_i \mid i \in \{1, \ldots, n\}\} & \text{(S-RcdPerm)} \\
\{k_j:S_j \mid j \in \{1, \ldots, n\}\} & \ll \{l_i:T_i \mid i \in \{1, \ldots, n\}\} \\
T_1 & \ll S_1 & S_2 & \ll T_2 & \Rightarrow S_1 \rightarrow S_2 \ll T_1 \rightarrow T_2 & \text{(S-Arrow)} \\
S & \ll \text{Top} & \text{(S-Top)}
\end{align*}
\]
Based on $\lambda_\rightarrow$ (9-1)

**Syntax**

$$t ::= \begin{align*}
& x & \text{variable} \\
& \lambda x : T . t & \text{abstraction} \\
& t \; t & \text{application}
\end{align*}$$

$$v ::= \lambda x : T . t & \text{abstraction value}$$

$$T ::= \begin{align*}
& \text{Top} & \text{maximum type} \\
& T \rightarrow T & \text{type of functions}
\end{align*}$$

$$\Gamma ::= \begin{align*}
& \emptyset & \text{empty context} \\
& \Gamma, x : T & \text{term variable binding}
\end{align*}$$

**Subtyping**

$$S <: S \quad \text{(S-REFL)}$$

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

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

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

**Typing**

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2} \quad \text{(T-ABS)}$$

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

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

**Evaluation**

$$\frac{t_1 \rightarrow t_1'}{t_1 \; t_2 \rightarrow t_1' \; t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \rightarrow t_2'}{v_1 \; t_2 \rightarrow v_1 \; t_2'} \quad \text{(E-APP2)}$$

$$\frac{}{(\lambda x : T_{11} . t_{12}) \; v_2 \rightarrow [x \rightarrow v_2] \; t_{12}} \quad \text{(E-APPABS)}$$

**Figure 15-1:** Simply typed lambda-calculus with subtyping ($\lambda_\rightarrow$)
Properties of Subtyping
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. The rule $T\text{-}S_{\text{UB}}$ could appear anywhere.

\[
\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \quad (T\text{-}S_{\text{UB}})
\]
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 likely to be hard?)
Subsumption case

Case T-Sub: \( t : S \quad S <: T \)
Subsumption case

Case T-Sub: \( t : S \quad S <: T \)

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

**Case T-Sub:** \( t : S \quad S <: T \)

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

Not hard!
**Application case**

*Case T-App:*

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

By the inversion lemma for evaluation, there are three rules by which \( t \longrightarrow t' \) can be derived: \texttt{E-App1}, \texttt{E-App2}, and \texttt{E-AppAbs}. Proceed by cases.
Application case

Case $T$-App:

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

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:

$t_1 \rightarrow t'_1 \quad t' = t'_1 \ t_2$

The result follows from the induction hypothesis and $T$-App.

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

$(T$-App$)$
Application case

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

By the inversion lemma for evaluation, there are three rules by which $t \rightarrow t'$ can be derived: $E\text{-App1}$, $E\text{-App2}$, and $E\text{-AppAbs}$. Proceed by cases.

Subcase $E\text{-App1}$:
\[ t_1 \rightarrow t'_1 \quad t' = t'_1 \ t_2 \]

The result follows from the induction hypothesis and $T\text{-App}$.

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

$(T\text{-App})$

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

$(E\text{-App1})$
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-App2: \[ t_1 = v_1 \quad t_2 \rightarrow t'_2 \quad t' = v_1 \ t'_2 \]
Similar.

\[ \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad \Gamma \vdash t_1 \ t_2 : T_{12} \]  \hspace{1cm} (T-App)

\[ t_2 \rightarrow t'_2 \]

\[ v_1 \ t_2 \rightarrow v_1 \ t'_2 \]  \hspace{1cm} (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:
\[ t_1 = \lambda x:S_{11}. \ t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2]t_{12} \]

By the inversion lemma for the typing relation...
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:
\[ t_1 = \lambda x:S_{11}. \ t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2]t_{12} \]

By the inversion lemma for the typing relation... \( T_{11} <: S_{11} \) and \( \Gamma, x:S_{11} \vdash t_{12} : T_{12} \).
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:
\[ t_1 = \lambda x : S_{11}. \ t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2]t_{12} \]

By the inversion lemma for the typing relation... \( T_{11} <: S_{11} \) and \( \Gamma, x : S_{11} \vdash t_{12} : T_{12} \).
By T-Sub, \( \Gamma \vdash t_2 : S_{11} \).
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:
\[ t_1 = \lambda x : S_{11}. \ t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2]t_{12} \]

By the inversion lemma for the typing relation... \( T_{11} <: S_{11} \) 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.

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

\[
(\lambda x : T_{11}. t_{12}) \ v_2 \rightarrow [x \mapsto v_2]t_{12} \quad (E-AppAbs)
\]
Inversion

Lemma:

1. If $\Gamma \vdash \text{true} : R$, then $R = \text{Bool}$.
2. If $\Gamma \vdash \text{false} : R$, then $R = \text{Bool}$.
3. If $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $\Gamma \vdash t_1 : \text{Bool}$ and $\Gamma \vdash t_2, t_3 : R$.
4. If $\Gamma \vdash x : R$, then $x : R \in \Gamma$.
5. If $\Gamma \vdash \lambda x : T_1 . t_2 : R$, then $R = T_1 \rightarrow R_2$ for some $R_2$ with $\Gamma, x : T_1 \vdash t_2 : R_2$.
6. If $\Gamma \vdash t_1, t_2 : R$, then there is some type $T_{11}$ such that $\Gamma \vdash t_1 : T_{11} \rightarrow R$ and $\Gamma \vdash t_2 : T_{11}$.
Inversion Lemma for Typing

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

Proof: Induction on typing derivations.
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: $\lambda x:S_1.s_2 : U \quad U <: T_1 \rightarrow T_2$
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 $T$-$SUB$: $\lambda x : S_1 . s_2 : U$ $U <: T_1 \rightarrow T_2$

We want to say “By the induction hypothesis...”, but the IH does not apply (we do not know that $U$ 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: $\lambda x:S_1.s_2 : U$ $U <: T_1\rightarrow T_2$

We want to say “By the induction hypothesis...”, but the IH does not apply (we do not know that $U$ 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.)
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: $\lambda x:S_1.s_2 : U$ $U <: T_1\rightarrow T_2$

We want to say “By the induction hypothesis...”, but the IH does not apply (we do not know that $U$ 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 $U_2 <: T_2$. 
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** $T$-$\text{Sub}$: $\lambda x:S_1.s_2 : U$ $U <: T_1\rightarrow T_2$

We want to say “By the induction hypothesis...”, but the IH does not apply (we do not know that $U$ 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:** $\lambda x:S_1.s_2 : U$ $U <: T_1 \rightarrow T_2$

We want to say “By the induction hypothesis...”, but the IH does not apply (we do not know that $U$ 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 $\Gamma \vdash \lambda x:S_1.s_2 : T_1 \rightarrow T_2$, then $T_1 <: S_1$ and $\Gamma, x:S_1 \vdash s_2 : T_2$.

Proof: Induction on typing derivations.

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

We want to say “By the induction hypothesis...”, but the IH does not apply (we do not know that $U$ 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 $T_1 <: S_1$.

From $\Gamma, x:S_1 \vdash s_2 : U_2$ and $U_2 <: T_2$, rule T-SUB gives $\Gamma, x:S_1 \vdash s_2 : T_2$, and we are done.
Subtyping with Other Features
Ordinary ascription:

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

\[
v_1 \text{ as } T \longrightarrow v_1 \quad (E\text{-ASCRIBE})
\]
Ascription and Casting

Ordinary ascription:  
(system)

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

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

Casting (cf. Java):  
(downcasting)

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

\[
\vdash v_1 : T \\
\vdash v_1 \text{ as } T \longrightarrow v_1 \quad \text{(E-CAST)}
\]

trust (at compile time) 

but 

verify (at run time) 

does progress theorem still hold?
Subtyping and Variants

\[
\langle l_1 : T_1 \rangle \quad <: \quad \langle l_1 : T_1 \rangle \\
\langle l_1 : T_i \rangle_{i \in 1..n+k} <: \langle l_1 : T_i \rangle_{i \in 1..n+k} \\
\text{(S-VariantWidth)}
\]

\[
\text{for each } i \quad S_i <: T_i \\
\langle l_1 : S_i \rangle_{i \in 1..n} <: \langle l_1 : T_i \rangle_{i \in 1..n} \\
\text{(S-VariantDepth)}
\]

\[
\langle k_j : S_j \rangle_{j \in 1..n} \quad \text{is a permutation of } \langle l_1 : T_i \rangle_{i \in 1..n} \\
\langle k_j : S_j \rangle_{j \in 1..n} <: \langle l_1 : T_i \rangle_{i \in 1..n} \\
\text{(S-VariantPerm)}
\]

\[
\Gamma \vdash t_1 : T_1 \\
\Gamma \vdash \langle l_1 = t_1 \rangle : \langle l_1 : T_1 \rangle \\
\text{(T-Variant)}
\]
Subtyping and Lists

\[
\begin{array}{c}
S_1 <: T_1 \\
\hline
\text{List } S_1 <: \text{List } T_1
\end{array}
\]

(S-LIST)

I.e., List is a covariant type constructor.
Subtyping and References

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

I.e., \texttt{Ref} is \emph{not} a covariant (nor a contravariant) type constructor.
Why?
Subtyping and References

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

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

- When a reference is \textit{read}, the context expects a \(T_1\), so if \(S_1 <: T_1\) then an \(S_1\) is ok.
Subtyping and References

\[
\begin{array}{c}
S_1 <: T_1 \quad T_1 <: S_1 \\
\hline
\text{Ref } S_1 <: \text{Ref } T_1
\end{array}
\]

(S-Ref)

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

- When a reference is \textit{read}, the context expects a \( T_1 \), so if \( S_1 <: T_1 \) then an \( S_1 \) is ok.

- When a reference is \textit{written}, the context provides a \( T_1 \) and if the actual type of the reference is \textbf{Ref } \( S_1 \), someone else may use the \( T_1 \) as an \( S_1 \). So we need \( T_1 <: S_1 \).
Subtyping and Arrays

Similarly...

array is mutable, list is immutable

\[
\begin{align*}
S_1 &<: T_1 & T_1 &<: S_1 \\
\text{Array } S_1 &<: \text{Array } T_1
\end{align*}
\]  

(S-ARRAY)
Subtyping and Arrays

Similarly...

\[
\frac{S_1 <: T_1}{\text{Array } S_1 <: \text{Array } T_1} \quad (S-\text{ARRAY})
\]

\[
\frac{S_1 <: T_1}{\text{Array } S_1 <: \text{Array } T_1} \quad (S-\text{ARRAY JAVA})
\]

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

in Java syntax, \( S_1[] <: T_1[] \)
Observation: a value of type \texttt{Ref T} can be used in two different ways: as a \textit{source} for values of type \texttt{T} and as a \textit{sink} for values of type \texttt{T}. 
References again

Observation: a value of type $\text{Ref } T$ 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 $\text{Ref } T$ 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

\[
\begin{align*}
\Gamma | \Sigma & \vdash t_1 : \text{Source } T_{11} \\
\quad & \Rightarrow \quad \Gamma | \Sigma \vdash !t_1 : T_{11} \quad \text{(T-DEREF)}
\end{align*}
\]

\[
\begin{align*}
\Gamma | \Sigma & \vdash t_1 : \text{Sink } T_{11} \\
\Gamma | \Sigma & \vdash t_2 : T_{11} \\
\quad & \Rightarrow \quad \Gamma | \Sigma \vdash t_1 := t_2 : \text{Unit} \quad \text{(T-ASSIGN)}
\end{align*}
\]
Subtyping rules

\[
\begin{align*}
S_1 & \ll T_1 & (S\text{-SOURCE}) \\
\text{Source } S_1 & \ll \text{Source } T_1 \\
T_1 & \ll S_1 & (S\text{-SINK}) \\
\text{Sink } S_1 & \ll \text{Sink } T_1 \\
\text{Ref } T_1 & \ll \text{Source } T_1 & (S\text{-REFSOURCE}) \\
\text{Ref } T_1 & \ll \text{Sink } T_1 & (S\text{-REFSINK})
\end{align*}
\]
Algorithmic 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.

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

(T-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* ($T$).

- 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).

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

\[\rightarrow \text{no backtracking!}\]
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)

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

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

\[
\begin{array}{c}
S <: U \\
U <: T
\end{array} \quad \Rightarrow \quad S <: T \quad (S\text{-TRANS})
\]

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.
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

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

\[
S <: U \quad U <: T \quad \Rightarrow S <: T \quad \text{(S-TRANS)}
\]

\[
\{l_i:T_i \mid i \in 1..n+k\} <: \{l_i:T_i \mid i \in 1..n\} \quad \text{(S-RCDWIDTH)}
\]

for each \( i \), \( S_i <: T_i \quad \Rightarrow \quad \{l_i:S_i \mid i \in 1..n\} <: \{l_i:T_i \mid i \in 1..n\} \quad \text{(S-RCDDEPTH)}
\]

\[
\{k_j:S_j \mid j \in 1..n\} \text{ is a permutation of } \{l_i:T_i \mid i \in 1..n\} \quad \Rightarrow \quad \{k_j:S_j \mid j \in 1..n\} <: \{l_i:T_i \mid i \in 1..n\} \quad \text{(S-RCDPERM)}
\]

\[
T_1 <: S_1 \quad S_2 <: T_2 \quad \Rightarrow \quad S_1 \rightarrow S_2 <: T_1 \rightarrow T_2 \quad \text{(S-ARROW)}
\]

\[
S <: \text{Top} \quad \text{(S-TOP)}
\]
Issues

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

1. The conclusions of $S$-$\text{RcdWidth}$, $S$-$\text{RcdDepth}$, and $S$-$\text{RcdPerm}$ overlap with each other.
2. $S$-$\text{Refl}$ and $S$-$\text{Trans}$ overlap with every other rule.
Step 1: simplify record subtyping

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

\[
\{l_i : i \in 1..n\} \subseteq \{k_j : j \in 1..m\} \quad \text{if } k_j = l_i \implies S_j : T_i
\]

\[
\{k_j : S_j : j \in 1..m\} \prec \{l_i : T_i : i \in 1..n\}
\]

(S-RCD)
Simpler subtype relation

\[
\begin{align*}
S \prec S \\
S \prec U & \quad U \prec T \\
\text{implies} & \\
S \prec T
\end{align*}
\]

\[
\{l_i, \, i \in 1..n\} \subseteq \{k_j, \, j \in 1..m\} \quad k_j = l_i \text{ implies } S_j \prec T_i
\]

\[
\{k_j : S_j, \, j \in 1..m\} \prec \{l_i : T_i, \, i \in 1..n\}
\]

\[
\begin{align*}
T_1 \prec S_1 & \quad S_2 \prec T_2 \\
\text{implies} & \\
S_1 \rightarrow S_2 \prec T_1 \rightarrow T_2
\end{align*}
\]

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

(S-Refl)  (S-Trans)  (S-Rcd)  (S-Arrow)  (S-Top)
Step 2: Get rid of reflexivity

Observation: $S$-$Refl$ is unnecessary.

**Lemma:** $S <: S$ can be derived for every type $S$ without using $S$-$Refl$. 
Even simpler subtype relation

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

\[
\begin{align*}
\{l_i, \, i \in 1..n\} & \subseteq \{k_j, \, j \in 1..m\} \\
\begin{aligned}
k_j & = l_i \implies S_j \ll T_i \\
\{k_j : S_j \, j \in 1..m\} & \ll \{l_i : T_i \, i \in 1..n\}
\end{aligned}
\end{align*}
\]  
\text{(S-\textsc{Rcd})}

\[
\begin{align*}
T_1 & \ll S_1 \quad S_2 \ll T_2 \\
\hline
S_1 \rightarrow S_2 & \ll T_1 \rightarrow T_2
\end{align*}
\]  
\text{(S-\textsc{Arrow})}

\[
S \ll \text{Top}
\]  
\text{(S-\textsc{Top})}
Step 3: Get rid of transitivity

Observation: \( S \text{-TRANS} \) is unnecessary.

**Lemma:** If \( S <: T \) can be derived, then it can be derived without using \( S \text{-TRANS} \).
"Algorithmic" subtype relation

\[ \vdash S <: \text{Top} \]  \hspace{1cm} (SA-Top)

\[ \vdash T_1 <: S_1 \quad \vdash S_2 <: T_2 \]
\[ \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2 \]  \hspace{1cm} (SA-Arrow)

\[ \{ l_i \mid i \in 1..n \} \subseteq \{ k_j \mid j \in 1..m \} \quad \text{for each} \ k_j = l_i, \ \vdash S_j <: T_i \]  \hspace{1cm} (SA-RCD)

\[ \vdash \{ k_j : S_j \mid j \in 1..m \} <: \{ l_i : T_i \mid i \in 1..n \} \]
Soundness and completeness

**Theorem:** $S <: T$ iff $\vdash S <: T$.

**Proof:** *(Homework)*

Terminology:

- The algorithmic presentation of subtyping is *sound* with respect to the original if $\vdash S <: T$ implies $S <: T$. *(Everything validated by the algorithm is actually true.)*
- The algorithmic presentation of subtyping is *complete* with respect to the original if $S <: T$ implies $\vdash S <: T$. *(Everything true is validated by the algorithm.)*
Subtyping Algorithm (pseudo-code)

The algorithmic rules can be translated directly into code:

\[ \text{subtype}(S, T) = \]

if \( T = \text{Top} \), then \( \text{true} \)
else if \( S = S_1 \rightarrow S_2 \) and \( T = T_1 \rightarrow T_2 \)
  then \( \text{subtype}(T_1, S_1) \land \text{subtype}(S_2, T_2) \)
else if \( S = \{k_j : S_j \ mid j \in 1..m\} \) and \( T = \{l_i : T_i \ mid i \in 1..n\} \)
  then \( \{l_i \mid i \in 1..n\} \subseteq \{k_j \mid j \in 1..m\} \)
  \land \text{for all } i \in 1..n \text{ there is some } j \in 1..m \text{ with } k_j = l_i \)
  and \( \text{subtype}(S_j, T_i) \)
else \( \text{false} \).
Recall: 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 $\text{subtype}(S, T) = true$, then $IS <: T$ (hence, by soundness of the algorithmic rules, $S <: T$).
2. If $\text{subtype}(S, T) = false$, then not $IS <: T$ (hence, by completeness of the algorithmic rules, not $S <: T$).

Q: What's missing?
A: How do we know that $\text{subtype}$ is a total function? Prove it!
Recall: 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

Recall: 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 $\leadsto S <: T$
   (hence, by soundness of the algorithmic rules, $S <: T$)

2. if $subtype(S, T) = false$, then not $\leadsto S <: T$
   (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!
Recall: 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

Recall: A *decision procedure* for a relation \( R \subseteq U \) is a total function \( p \) from \( U \) to \( \{\text{true, false}\} \) such that \( p(u) = \text{true} \) 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 \( \text{subtype}(S, T) = \text{true} \), then \( \Downarrow S <: T \)
   (hence, by soundness of the algorithmic rules, \( S <: T \))

2. if \( \text{subtype}(S, T) = \text{false} \), then not \( \Downarrow S <: T \)
   (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

Recall: 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 $\triangleright S <: T$
   (hence, by soundness of the algorithmic rules, $S <: T$)

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

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

(T-SUB)

Where is this rule really needed?
For the typing relation, we have just one problematic rule to deal with: subsumption.

\[
\Gamma \vdash t : S \quad S \preceq T \\
\hline
\Gamma \vdash t : T
\]  \hspace{1cm} (T\text{-}Sub)

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.
For the typing relation, we have just one problematic rule to deal with: subsumption.

\[
\Gamma \vdash t : S \quad S \ll T \\
\hline
\Gamma \vdash t : T
\]  

(T-Sub)

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.

Where else??
**Issue**

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

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

Where is this rule really needed?

For applications. E.g., the term

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

is not typable without using subsumption.

Where else??

*Nowhere else!* Uses of subsumption to help typecheck applications are the only interesting ones.
Example (T-Abs)

\[
\begin{array}{c}
\vdash x:S_1 \vdash s_2 : S_2 \\
\hline
\Gamma, x:S_1 \vdash s_2 : S_2 \quad S_2 <: T_2
\end{array}
\]

(\text{T-SUB})

\[
\begin{array}{c}
\vdash x:S_1 \vdash s_2 : T_2
\end{array}
\]

(\text{T-Abs})

\[
\begin{array}{c}
\vdash \lambda x:S_1.s_2 : S_1 \rightarrow T_2
\end{array}
\]
Example (T-Abs)

\begin{align*}
\Gamma, x : S_1 \vdash s_2 : S_2 \quad & \text{(T-Sub)} \\
\Gamma, x : S_1 \vdash s_2 : T_2 \quad & \text{(T-Abs)} \\
\Gamma \vdash \lambda x : S_1 . s_2 : S_1 \rightarrow T_2
\end{align*}

becomes

\begin{align*}
\Gamma, x : S_1 \vdash s_2 : S_2 \quad & \text{(T-Abs)} \\
\Gamma \vdash \lambda x : S_1 . s_2 : S_1 \rightarrow S_2 \quad & \text{(S-Refl)} \\
\Gamma \vdash \lambda x : S_1 . s_2 : S_1 \rightarrow S_2 \quad & \text{(S-Arrow)} \\
\Gamma \vdash \lambda x : S_1 . s_2 : S_1 \rightarrow T_2 \quad & \text{(T-Sub)}
\end{align*}
Example (T-App on the left)

\[ \Gamma \vdash s_1 : S_{11} \rightarrow S_{12} \quad S_{11} \rightarrow S_{12} \subset T_{11} \rightarrow T_{12} \quad (S-Arrow) \quad \vdash s_1 : T_{11} \rightarrow T_{12} \]

\[ \Gamma \vdash s_2 : T_{11} \quad (T-App) \]
Example ($T\text{-App}$ on the left)

\[
\begin{align*}
\Gamma \vdash s_1 : S_{11} \rightarrow S_{12} & \quad S_{11} \rightarrow S_{12} \ll S_{2} \rightarrow T_{12} \\
\Gamma \vdash s_1 : T_{11} \rightarrow T_{12} & \quad (T\text{-Sub}) \quad \Gamma \vdash s_2 : T_{11} \\
\end{align*}
\]

becomes

\[
\begin{align*}
\Gamma \vdash s_1 : S_{11} \rightarrow S_{12} & \quad \Gamma \vdash s_2 : S_{11} \quad S_{11} \ll S_{12} < T_{11} \rightarrow T_{12} \\
\Gamma \vdash s_1 : S_{11} \rightarrow S_{12} & \quad \Gamma \vdash s_2 : S_{11} \quad (T\text{-Sub}) \quad \Gamma \vdash s_2 : T_{12} \\
\end{align*}
\]
Example (\(T\text{-App} \text{ on the right})\)

\[
\begin{align*}
\Gamma \vdash s_1 : T_{11} &\rightarrow T_{12} \\
\Gamma \vdash s_2 : T_2 &<: T_{11} \\
\Gamma \vdash s_1 \; s_2 : T_{12} \\
\end{align*}
\]
Example (T-App on the right)

\[
\begin{array}{c}
\Gamma \vdash s_1 : T_{11} \rightarrow T_{12} \\
\hline \\
\Gamma \vdash s_2 : T_{11} \\
\hline \\
\Gamma \vdash s_1 \ s_2 : T_{12}
\end{array}
\]

becomes

\[
\begin{array}{c}
\hline \\
\Gamma \vdash s_1 : T_{11} \rightarrow T_{12} \\
\hline \\
\Gamma \vdash s_2 : T_{11} \\
\hline \\
\Gamma \vdash s_1 \ s_2 : T_{12}
\end{array}
\]

\[
\begin{array}{c}
\hline \\
T_{12} \lessdot T_{12} \\
\hline \\
T_{11} \lessdot T_{11} \\
\hline \\
\Gamma \vdash s_1 : T_{11} \rightarrow T_{12} \\
\hline \\
\Gamma \vdash s_1 \ s_2 : T_{12}
\end{array}
\]

\[
\begin{array}{c}
\hline \\
T_{2} \lessdot T_{11} \\
\hline \\
\Gamma \vdash s_2 : T_{2}
\end{array}
\]

\[
\begin{array}{c}
\hline \\
\Gamma \vdash s_1 : T_{2} \rightarrow T_{12} \\
\hline \\
\Gamma \vdash s_1 \ s_2 : T_{12}
\end{array}
\]
Example \((T\text{-}SUB)\)

\[
\begin{array}{c}
\vdash s : S \\
\hline
\vdash s : U \quad (T\text{-}SUB) \\
\hline
\vdash s : T
\end{array}
\]

\[
\begin{array}{c}
S <: U \\
\hline
U <: T
\end{array}
\]
Example (T-SUB)

\[
\begin{array}{c}
\Gamma \vdash s : S \\
\hline
S <: U
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash s : U \\
\hline
U <: T
\end{array}
\Rightarrow
\begin{array}{c}
\Gamma \vdash s : T
\end{array}
\]

becomes

\[
\begin{array}{c}
\Gamma \vdash s : S \\
\hline
S <: U
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash s : U \\
\hline
U <: T
\end{array}
\Rightarrow
\begin{array}{c}
\Gamma \vdash s : S
\hline
S <: T
\end{array}
\Rightarrow
\begin{array}{c}
\Gamma \vdash s : T
\end{array}
\]