SUBTYPING
OVERVIEW

- Subtyping also known as subtype polymorphism.
  - Other polymorphisms:
    - Universal Polymorphism: $\forall A.A \rightarrow A$
    - Existential Polymorphism: $\exists X. \{a: X; f: X \rightarrow \text{int} \rightarrow X\}$
      - ...
  - Commonly found in object-oriented programming.
    - E.g., Java
    - Super-class, sub-class and inheritance

- Subtyping interacts with most of the language features we have discussed so far.

- Key idea: Type $t_1$ is a subtype of $t_2$ if all values with type $t_1$ can be used in operations where values of type $t_2$ are expected.
BASICS

- Type is a collection of values...
- Notation:
  \[ t_1 \leq t_2 \]
- Basic Properties:
  \[ t \leq t \quad \text{(S-Reflexivity)} \]
  \[ t_1 \leq t_2 \quad t_2 \leq t_3 \quad t_1 \leq t_3 \quad \text{(S-Transitivity)} \]
- Extending the type system with Top and Subsumption:
  \[ t ::= \ldots \mid \text{Top} \quad \text{(like the Object class in Java)} \]
  \[ t \leq \text{Top} \quad \Gamma |- e : t_1 \quad t_1 \leq t_2 \quad \Gamma |- e : t_2 \quad \text{(T-Sub)} \]
EXAMPLE TYPING DERIVATION

Program:

```
let f = \x:Top.x in
{f 2, f true}
```

(let G = f:Top→Top)

```
G |- f : Top→Top
G |- 2 : Top
```

If we used universal polymorphism:

```
let f = ∀A. λx: A. x in
{f[int] 2, f[bool] true} : int * bool
```
EXTENDING SUBTYPES TO TUPLES

- Recall:
  \[
  \begin{align*}
  \frac{\text{for each } i : \Gamma | - e_i : t_i}{\Gamma | - \{e_i\} : \{t_i\}} \quad \text{(T-Tuple)} \\
  \frac{\Gamma \vdash e : \{t_i\}}{\Gamma \vdash e.j : t_j} \quad \text{(T-Proj)}
  \end{align*}
  \]

- Widened tuples are more specific, hence subtype of original tuple type.
  \[
  \frac{m \geq n}{\{t_i^{i \in \{1..m\}}\} \subseteq \{t_i^{i \in \{1..n\}}\}} \quad \text{(S-TupWidth)}
  \]

- The reverse is bad:
  \[
  \frac{m \leq n}{\{t_i^{i \in \{1..m\}}\} \subseteq \{t_i^{i \in \{1..n\}}\}} \quad \text{(BAD!)}
  \]

  - The following program will type check but evaluation gets stuck:

    - let l = {1, 2, 3} in l.4
    - {1, 2, 3} : int * int * int <= int * int * int * int
    - l.4 : int
EXTENDING SUBTYPES TO TUPLES

- Covariant Rule:

\[
\forall i : t_i <= t'_i \\
\{t_i \_\_\_\_..n\} <= \{t'_i \_\_\_\_..n\} \quad (S - \text{TupDep})
\]

For example int * bool * int <= Top * Top * Top

- Contra-variant Rule is bad:

\[
\forall i : t'_i <= t_i \\
\{t'_i \_\_\_\_..n\} <= \{t_i \_\_\_\_..n\} \quad (S - \text{TupDep})
\]
EXTENDING SUBTYPES TO SUMS

- Given the typing of n-ary sum:
  \[
  \frac{\Gamma \vdash -e : t_i}{\Gamma \vdash \text{in}_i[t_1 + \ldots + t_n] e : t_1 + \ldots + t_n} \quad \text{(T-Ini)}
  \]
  \[
  \frac{\Gamma \vdash -e : t_1 + \ldots + t_n \quad \forall i \in 1..n : \Gamma, x : t_i \vdash \text{e}_i : t}{\Gamma \vdash \text{case e of (in}_i x \Rightarrow \text{e}_i \mid \ldots \mid \text{in}_n x \Rightarrow \text{e}_n) : t} \quad \text{(T-Case)}
  \]

- First consider this rule:
  \[
  \frac{m \geq n}{t_1 + \ldots + t_m \leq t_1 + \ldots + t_n} \quad \text{(S-SumWid?)}
  \]

- Counter Example:
  case (\text{in}_3[\text{int}+\text{int}+\text{int}] 0) of
  \[
  \begin{align*}
  \text{(in}_1 x & \Rightarrow \text{true} \\
  \mid \text{in}_2 x & \Rightarrow \text{false}
  \end{align*}
  \]
  - Typechecks since \text{int}+\text{int}+\text{int} \leq \text{int} + \text{int} and due to (T-Case)
  - But gets stuck
EXTENDING SUBTYPES TO SUMS

- The correct rule is:

\[
\frac{m \leq n}{t_1 + \ldots + t_m \leq t_1 + \ldots + t_n} \quad (S\text{-SumWid})
\]

- The co-variant rule:

\[
\frac{\forall i : t_i \leq t_i'}{t_1 + \ldots + t_m \leq t_1' + \ldots + t_n'} \quad (S\text{-SumDepth})
\]

- Again contra-variant rule is bad.
  - E.g.,
    
    ```
    case (in_1 {1, 2})
    ( in_1 x => x.3
    | in_2 x => 0)
    )
    int * int * int <= int * int \rightarrow int* int + int <= int * int * int + int
    ```
FUNCTIONS

\[
\frac{t_1 \leq t'_1}{t_1 \rightarrow t_2 \leq t'_1 \rightarrow t'_2} \quad \text{(Bad!)}
\]

\[
\frac{t_1' \leq t_1}{t_1 \rightarrow t_2 \leq t_1' \rightarrow t_2'} \quad \text{(Bad!)}
\]

\[
\frac{t_1' \leq t_1' \leq t_2'}{t_1 \rightarrow t_2 \leq t_1' \rightarrow t_2'} \quad \text{(Bad!)}
\]

\[
\frac{t_1 \leq t_1' \leq t_2' \leq t_2}{t_1 \rightarrow t_2 \leq t_1' \rightarrow t_2'} \quad \text{(S-Func)}
\]

- **Counter examples**
  - \((\\{x\}:\text{int}*\text{int}*\text{int.} \{x.3, x.3, x.3\}) \{2, 3\}
    - int*int*int \leq int*int, rule 1 and 2 are bad!
  - \((\\{x\}:\text{int}*\text{int}*\text{int.} \{x.3, x.3, x.3\}) \{1, 2, 3\}).4
    - int*int*int \rightarrow int*int*int*int \leq int*int*int \rightarrow int*int*int*int*int: rule 3 is bad!

- **Intuition:**
  - if a function \(f\) is of type \(t_1 \rightarrow t_2\)
  - \(f\) accepts elements of type \(t_1\), and also subtype \(t_1'\) of \(t_1\);
  - \(f\) returns elements of type \(t_2\), which also belongs to supertype \(t_2'\).

- To prove Rule (S-Func) is good, we need to prove progress.
**Canonical Forms Lemma**

- Intuition: Given a type, we know the “shape” of its values.
  
  If \( \Gamma \vdash \nu : \tau \) then:
  
  1. If \( \tau = \tau_1 \rightarrow \tau_2 \) then \( \nu = \forall x : \sigma . e \), where \( \tau_1 \leq \sigma \);
  
  2. If \( \tau = \tau_1 \ast \ldots \ast \tau_n \) then \( \nu = (\nu_1, \ldots, \nu_m) \), where \( m \geq n \);
  
  3. If \( \tau = \tau_1 \ast \ldots \ast \tau_n \) then \( \nu = \text{in}_i[\tau_1 + \ldots + \tau_m](\nu) \) where \( m \leq n \), \( 1 \leq i \leq m \).

**Proof:**

By induction on the typing derivation \( \Gamma \vdash \nu : \tau \)

**Case:**

\( \Gamma \vdash \nu : \tau' \) \( \tau' \leq \tau \)

----------------------- (subsumption rule)

\( \Gamma \vdash \nu : \tau \)

**Subcase (1) \( \tau = \tau_1 \rightarrow \tau_2 \)**

1. \( \tau' \leq \tau_1 \rightarrow \tau_2 \) (By assumption)
2. \( \tau' = \tau_1' \rightarrow \tau_2' \) and \( \tau_1 \leq \tau_1' \) and \( \tau_2' \leq \tau_2 \) (By 1 and S-Func)
3. \( \nu = \forall x : \tau'. e \) and \( \tau_1' \leq \tau \) (IH)
4. \( \tau_1 \leq \tau' \) (By 3 and S-Transitivity)

(Rest left as exercise!)
PROGRESS LEMMA

If $e$ is a closed, well-typed expression, then either $e$ is a value or else there is some $e'$ where $e \rightarrow e'$.

Proof: By induction on the derivation of typing relations.

Case T-Var: doesn’t occur because $e$ is closed.

Case T-Abs: already a value.

Case $\Gamma |- e_1 : t_{11} \rightarrow t_{12}$ $\Gamma |- e_2 : t_{11}$ (T-App)

\[ \Gamma |- e_1 e_2 : t_{12} \]

subcase 1: $e_1$ can take a step \hspace{1cm} (By IH)

then $e_1 e_2$ can take a step. \hspace{1cm} (By E-App1)

subcase 2: $e_2$ can take a step \hspace{1cm} (By IH)

then $e_1 e_2$ can take a step \hspace{1cm} (By E-App2)

subcase 3: $e_1$ and $e_2$ are both values \hspace{1cm} (By IH)

$e_1 = \lambda x: s_{11}. e_{12}$ \hspace{1cm} (By canonical forms)

$e_1 e_2$ can take a step \hspace{1cm} (By E-AppAbs)
PROGRESS LEMMA (CONT’D)

Case \( \frac{\text{for each } i : \Gamma \vdash e_i : t_i}{\Gamma \vdash \{e_i^{i \in 1..n}\} : \{t_i^{i \in 1..n}\}} \) (T-Tuple)

subcase 1: there’s an \( e_i \) which can take a step \hspace{1cm} (By IH)

\( e \) can take a step \hspace{1cm} (By E-Tuple)

subcase 2: all \( e_i \)'s are values. \hspace{1cm} (By IH)

then definition, \( \{e_i, i \notin 1..n\} \) is also value.

Case \( \frac{\Gamma \vdash e : \{t_i^{i \in 1..n}\}}{\Gamma \vdash e.j : t_j} \) (T-Proj)

subcase 1: \( e \) can take a step \hspace{1cm} (By IH)

then \( e.j \) can also take a step \hspace{1cm} (By E-ProjTuple1)

subcase 2: \( e \) is already a value \hspace{1cm} (By IH)

then \( e = \{v1, v2, \ldots, vm\}, m \geq n \) \hspace{1cm} (By Canonical forms)

then \( e \) can take a step \hspace{1cm} (By E-ProjTuple)
**Progress Lemma (Cont’d)**

Cases for sums (T-case and T-Ini) are similar.

Case\[\frac{\Gamma|-e:t_1 \quad t_1 \leq t_2}{\Gamma|-e:t_2}\] (T-Sub) is true by IH.
LEMMA: INVERSION OF SUBTYPING

(1) if $t \leq t_1' \rightarrow t_2'$ then $t = t_1 \rightarrow t_2$ and $t_1' \leq t_1$ and $t_2 \leq t_2'$

(2) if $t \leq t_1' * ... * t_n'$ then
   
   $t = t_1 * ... * t_m$ and $m \geq n$
   
   and for $i = 1, ... n$, $t_i \leq t_i'$

(3) if $t \leq$ top then $t$ can be any type

(4) if $t \leq$ bool then $t =$ bool

Prove: By induction on the subtyping relations
**Lemma: Component Typing**

1. If $G |- \forall x: s_1 \cdot e_2 : t_1 \rightarrow t_2$, then $t_1 \leq s_1$ and $G, x : s_1 |- e_2 : t_2$.

2. If $G |- \{ e_1, \ldots, e_m \} : t_1 \ast \ldots \ast t_n$, then $m \geq n$ and $G |- e_i : t_i$, for $1 \leq i \leq m$.

3. If $G |- \ln_i[t_1+\ldots+t_m] e : t_1 + \ldots + t_n$, then $m \leq n$ and $G |- e : t_i$, for $1 \leq i \leq m$.

Proof: Straightforward induction on typing relations, using “Inversion of subtypes” lemma for T-Sub case.
Substitution Lemma

If $G, x:s |- e : t$ and $G |- v : s$, then $G |- e[v/x] : t$.

Proof: By induction on the derivation of typing relations. Similar to the proof of substitution lemma without subtyping.
**Preservation Lemma**

If $G \vdash e : t$, and $e \Rightarrow e'$, then $G \vdash e' : t$.

Proof: By induction on the derivation of typing relations.

Case T-Var and T-Abs are easy.

Case $\frac{\Gamma \vdash e_1 : t_{11} \Rightarrow t_{12} \quad \Gamma \vdash e_2 : t_{11}}{\Gamma \vdash e_1 \ e_2 : t_{12}}$ (T-App)

For $e_1 \ e_2$ to take a step, there are three possible rules, hence three subcases:

**Subcase** $e_1 \Rightarrow e_1'$: result follows. (IH and Hypothesis)

**Subcase** $e_2 \Rightarrow e_2'$: result follows. (IH and Hypothesis)

**Subcase** $e_1 = \lambda x : s_{11}. \ e_{12}, \ e_2 = v, \ e' = e_{12}[v/x]$:  

(1) $t_{11} \leq s_{11}$ and $G, x : s_{11} \vdash e_{12} : t_{12}$ (Component Typing Lemma)

(2) $G \vdash v : s_{11}$ (Assumption & T-Sub)

(3) $G \vdash e' : t_{12}$. (By (2) and Substitution lemma)

QED.
**Preservation Lemma (Cont’d)**

Case \( \frac{\text{for each } i : \Gamma | - e_i : t_i}{\Gamma | - \{ e_i \}_{i \in \ldots n} : \{ t_i \}_{i \in \ldots n}} \) (T-Tuple)

if e takes a step, then it must be
the case that \( e_j \rightarrow e_j' \) for some field \( e_j \). \hspace{1cm} (E-Tuple)

if \( e_j : t_j \), then \( e_j' : t_j \). \hspace{1cm} (IH)

Therefore, \( e' : t_1 \ast \ldots \ast t_n \)

QED.

Case \( \frac{\Gamma | - e : \{ t_i \}_{i \in \ldots n}}{\Gamma | - e.j : t_j} \) (T-Proj)

There are two evaluation rules by which \( e.j \) can take a step.

Subcase E-ProjTuple: \( e = \{ v_1, \ldots, v_n \} \), \( e' = v_j \).

\hspace{1cm} \text{forall } i: v_i : t_i \hspace{1cm} \text{(Component typing)}

\hspace{1cm} \therefore e.j : t_j \text{ and } v_j : t_j \hspace{1cm} \text{(T-Proj)}

Subcase E-ProjTuple1: \( e = e_1.j \), \( e' = e_1'.j \)

\hspace{1cm} \text{result follows.} \hspace{1cm} \text{(IH and T-Proj)}
**Preservation Lemma (cont’d)**

- **Case**
  \[ \Gamma |- c : t \]
  \[ \Gamma |- e : t_i \]
  \[ \Gamma |- \text{in}_i[t_1 + \ldots + t_n] e : t_1 + \ldots + t_n \]

  if \( \text{in}_i[t_1 + \ldots + t_n] e \) takes a step, then it must be \( e \rightarrow e' \).  
  \( \text{E-Ini} \)

  \( e' : t_i \)  
  \( \text{IH} \)

  \( \text{in}_i e' : t_1 + \ldots + t_n \)  
  \( \text{T-Ini} \)

- **Case**
  \[ \Gamma |- e : t_1 + \ldots + t_n \]
  \[ \forall i : \Gamma, x : t_i |- e_i : t \]

  \[ \Gamma |- \text{case } e \text{ of (in}_i x => e_i | \ldots | \text{in}_n x => e_n) : t \]

  Subcase E-CaseIni: result follows  
  \( \text{IH and Substitution IH} \)

  Subcase E-Case: result follows  
  \( \text{IH and T-Case} \)

- **Case**
  \[ \Gamma |- e : t_1 \]
  \[ t_1 \leq t_2 \]

  \[ \Gamma |- e : t_2 \]

  \( e \rightarrow e', e' : t_1 \)  
  \( \text{IH} \)

  \( e' : t_2 \)  
  \( \text{T-Sub} \)

  QED.
**Top and Bottom Types**

- Top is the maximum type in our language.
- It’s not necessary in simply-typed lambda calculus, but we keep it because:
  - Corresponds to Object in Java
  - Convenient technical device in complex system involving subtyping and parametric polymorphism
  - Its behavior is straightforward and useful in examples

- Can we have a minimum type?
  
  \[ t ::= \ldots \mid \text{Bot} \]
  
  \[ \text{Bot} \leq t \quad (S\text{-Bot}) \]
  
  - Bot is empty – no enclosed values
WHAT IF BOT HAS VALUES?

- Say $v$ is a value in Bot.
- By S-Bot, we can derive $\vdash v : \text{Top} \rightarrow \text{Top}$.
  - By Canonical forms, $v = \lambda x : t_1 . e_2$ for some $t_1$ and $e_2$.
- On the other hand, we can also derive $\vdash v : t_1 * t_2$.
  - By Canonical forms, $v = (e_1, e_2)$.
- The syntax of $v$ dictates that $v$ cannot be a function and a tuple at the same time.
- Contradiction!
**PURPOSES OF Bot**

- Express that some operations (e.g. throwing exceptions) are not expected to return.

- Two benefits:
  - Signal the programmer that no result is expected.
  - Signal the typechecker that expression of Bot type can be used in a context expecting any type of value.

- Example:
  \[
  \text{x: t .} \\
  \text{if <check that x is reasonable> then} \\
  \text{<compute result>}
  \text{else}
  \text{error /* error is of type Bot */}
  \]

- Above expression is always well typed no matter what the type of the normal result is, error will be given that type by T-Sub and hence the conditional is well typed.
POLYMORPHISM

- Type systems allowing a single piece of code to be used with multiple types is called *polymorphism* (poly = many, morph = form).

- **Subtype polymorphism**
  - give an expression many types following the subsumption rule
  - Allow us to selectively “forget” information about the expression’s behavior
  - Java class hierarchy

- **Parametric polymorphism**
  - Allows a piece of code to be typed generically
  - Using type variables
  - Instantiated with particular types when needed
  - Generic programming, Java interface, ML modules

- **Ad-hoc polymorphism**
  - Allows a polymorphic value to exhibit different behavior when “viewed” at different types.
  - Provides multiple implementations of the behaviors
  - Overloading in Java/C++:
    - operator + works for int, float, char, string, etc.