

# SUBTYPING & POLYMORPHISM

1

# OVERVIEW

- Subtyping also known as subtype polymorphism.
  - Other polymorphisms:
    - Universal Polymorphism:  $\forall A. A \rightarrow A$
    - Existential Polymorphism:  $\exists X. \{a: X; f: X \rightarrow \text{int} \rightarrow X\}$
    - The above called *parametric polymorphism*...
- 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.*

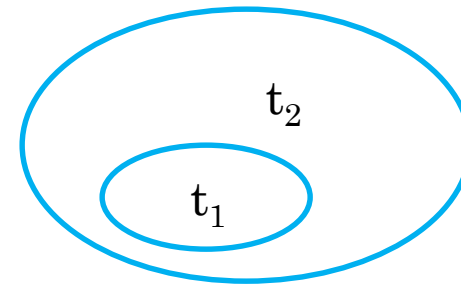
## QUIZ: POLYMORPHISM

- Which one of the following is NOT a type of polymorphism?
  - A) Subtype polymorphism
  - B) Universal polymorphism
  - C) Existential polymorphism
  - D) Constant polymorphism

# BASICS

- Type is a collection of values...
- Notation:

$$t_1 \leq t_2$$



- Basic Properties:

$$\frac{}{t \leq t} \text{ (S-Reflexivity)} \quad \frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3} \text{ (S-Transitivity)}$$

- Extending the type system with Top and Subsumption:

$t ::= \dots \mid \text{Top}$  (like the Object class in Java)

$$\frac{}{t \leq \text{Top}} \text{ (Top)} \quad \frac{\Gamma \mid - e : t_1 \quad t_1 \leq t_2}{\Gamma \mid - e : t_2} \text{ (T-Sub)}$$

# EXAMPLE TYPING DERIVATION

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

(let G = f:Top→Top)

$\frac{G \mid -2:\text{int} \quad \text{int} \leq \text{Top}}{\dots}$	$\frac{G \mid -\text{true}:\text{bool} \quad \text{bool} \leq \text{Top}}{\dots}$
$G \mid - f : \text{Top} \rightarrow \text{Top} \quad G \mid - 2 : \text{top}$	$G \mid - f : \text{Top} \rightarrow \text{Top} \quad G \mid - \text{true} : \text{top}$
$\frac{\dots \quad \dots}{f:\text{Top} \rightarrow \text{Top} \mid - f 2: \text{Top} \quad f:\text{Top} \rightarrow \text{Top} \mid - f \text{true}: \text{Top}}$	
$\frac{\dots \quad \dots}{\cdot \mid - \lambda x:\text{Top}.x : \text{Top} \rightarrow \text{Top} \quad f:\text{Top} \rightarrow \text{Top} \mid - \{f 2, f \text{true}\} : \text{Top}^* \text{Top}}$	
$\cdot \mid - \text{let } f = \lambda x:\text{Top}.x \text{ in } \{f 2, f \text{true}\} : \text{Top} * \text{Top}$	

If we used universal polymorphism:

let f =  $\forall A. \lambda x: A. x$  in  
      {f[int] 2, f[bool] true} : int \* bool

# QUIZ: TYPE DERIVATION

- Write down the type derivation tree for:

```
let swap =  $\lambda p:\text{Top}. \{p.2, p.1\}$   
in  $\{\text{swap } \{\text{true}, \text{false}\}, \text{swap } \{21, 12\}\}$ 
```

# EXTENDING SUBTYPES TO TUPLES

- Recall:

$$\frac{\text{for each } i : \Gamma \vdash e_i : t_i}{\Gamma \vdash \{e_i^{i \in 1..n}\} : \{t_i^{i \in 1..n}\}} \quad (\text{T-Tuple}) \qquad \frac{G \vdash e : \{t_i^{i \in 1..n}\} \quad 1 \leq j \leq n}{G \vdash e.j : t_j} \quad (\text{T-Proj})$$

- Widened tuples are more specific, hence subtype of original tuple type.

$$\frac{m \leq n}{\{t_i^{i \in 1..m}\} \leq \{t_i^{i \in 1..n}\}} \quad (\text{S-TupWidth})$$

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

- 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:

$$\frac{\forall i: t_i \leq t'_i}{\{t_i^{i \in 1..n}\} \leq \{t'_i^{i \in 1..n}\}} \quad (\text{S-TupDep})$$

For example  $\text{int} * \text{bool} * \text{int} \leq \text{Top} * \text{Top} * \text{Top}$

- Contra-variant Rule is bad:

$$\frac{\forall i: t'_i \leq t_i}{\{t_i^{i \in 1..n}\} \leq \{t'_i^{i \in 1..n}\}} \quad (\text{S-TupDep})$$

**Quiz:** Give an example why the contra-variant rule is bad.



# EXTENDING SUBTYPES TO SUMS

- Given the typing of n-ary sum:

$$\frac{\Gamma \mid -e : t_i}{\Gamma \mid -\text{in}_i[t_1 + \dots + t_n] e : t_1 + \dots + t_n} \quad (\text{T-Ini})$$

$$\frac{\Gamma \mid -e : t_1 + \dots + t_n \quad \forall i \in 1..n : \Gamma, x : t_i \mid -e_i : t_i}{\Gamma \mid -\text{case } e \text{ of } (\text{in}_1 x \Rightarrow e_1 \mid \dots \mid \text{in}_n x \Rightarrow e_n) : t} \quad (\text{T-Case})$$

- First consider this rule:

$$\frac{m \geq n}{t_1 + \dots + t_m \leq t_1 + \dots + t_n} \quad (\text{S-SumWid?})$$

- Counter Example:

```

case (in3[int+int+int] 0) of
  (in1 x => true
   | in2 x => false)
    
```

- 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:

$$\frac{m \leq n}{t_1 + \dots + t_m \leq t_1 + \dots + t_n} \quad (\text{S-SumWid})$$

- The co-variant rule:

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

- Again contra-variant rule is bad.

- E.g.,

```
case (in_1 {1, 2}) of
  ( in_1 x => x.3
  | in_2 x => 0
  )
```

`int * int * int <= int * int`  $\rightarrow$  `int* int + int <= int * int * int + int`

# FUNCTIONS

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

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

Contravariant

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

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

Covariant

## Counter examples

- $(\lambda x:\text{int}*\text{int}*\text{int}. \{x.3, x.3, x.3\}) \{2, 3\}$ 
  - $\text{int}*\text{int}*\text{int} \leq \text{int}*\text{int}$ , rule 1 and 2 are bad!
- $((\lambda x:\text{int}*\text{int}*\text{int}. \{x.3, x.3, x.3\}) \{1, 2, 3\}).4$ 
  - $\text{int}*\text{int}*\text{int} \rightarrow \text{int}*\text{int}*\text{int} \leq \text{int}*\text{int}*\text{int} \rightarrow \text{int}*\text{int}*\text{int}*\text{int}$ : 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  $t_2'$ .
- We will make use of S-Func to prove progress lemma.

# CANONICAL FORMS LEMMA

- Intuition: Given a type, we know the “shape” of its values.

If  $\cdot \vdash v : t$  then

- (1) if  $t = t_1 \rightarrow t_2$  then  $v = \lambda x:s_1.e$ , where  $t_1 \leq s_1$ ;
- (2) if  $t = t_1 * \dots * t_n$  then  $v = (v_1, \dots, v_m)$ , where  $m \geq n$ ;
- (3) if  $t = t_1 + \dots + t_n$  then  $v = \text{in}_i[t_1 + \dots + t_m](v)$  where  $m \leq n$ ,  $1 \leq i \leq m$ .

Proof:

By induction on the typing derivation  $\cdot \vdash v : t$

Case:

$\cdot \vdash v : t' \quad t' \leq t$

----- (subsumption rule)

$\cdot \vdash v : t$

subcase (1)  $t = t_1 \rightarrow t_2$

- (1)  $t' \leq t_1 \rightarrow t_2$  (By assumption)
- (2)  $t' = t_1' \rightarrow t_2'$  and  $t_1 \leq t_1'$  and  $t_2' \leq t_2$  (By 1 and S-Func)
- (3)  $v = \lambda x:t''.e$  and  $t_1' \leq t''$  (IH)
- (4)  $t_1 \leq t''$ . (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 
$$\frac{G \mid -e_1 : t_{11} \rightarrow t_{12} \quad G \mid -e_2 : t_{11}}{G \mid -e_1 e_2 : t_{12}} \text{ (T-App)}$$

subcase 1:  $e_1$  can take a step (By IH)

then  $e_1 e_2$  can take a step. (By E-App1)

subcase 2:  $e_2$  can take a step (By IH)

then  $e_1 e_2$  can take a step (By E-App2)

subcase 3:  $e_1$  and  $e_2$  are both values (By IH)

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

$e_1 e_2$  can take a step (By E-AppAbs)

# PROGRESS LEMMA (CONT'D)

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

- subcase 1: there's an  $e_i$  which can take a step (By IH)
- e can take a step (By E-Tuple)
- subcase 2: all  $e_i$ 's are values. (By IH)
- then definition,  $\{e_i, i \in 1..n\}$  is also value.

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

- subcase 1: e can take a step (By IH)
- then  $e.j$  can also take a step (By E-ProjTuple1)
- subcase 2: e is already a value (By IH)
- then  $e = \{v_1, v_2, \dots, v_m\}$ ,  $m \geq n$  (By Canonical forms)
- then e can take a step (By E-ProjTuple)

# PROGRESS LEMMA (CONT'D)

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

Case  $\frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2}$  (T-Sub) is true by IH.

## LEMMA: INVERSION OF SUBTYPING

- (1) if  $t \leq t1' \rightarrow t2'$  then  $t = t1 \rightarrow t2$  and  $t1' \leq t1$   
and  $t2 \leq t2'$
- (2) if  $t \leq t1 * \dots * tn$  then  
 $t = t1 * \dots * tm$  and  $m \geq n$   
and for  $i = 1, \dots, n$ ,  $ti \leq ti'$
- (3) if  $t \leq \text{top}$  then  $t$  can be any type
- (4) if  $t \leq \text{bool}$  then  $t = \text{bool}$

Prove: By induction on the subtyping relations



## LEMMA: COMPONENT TYPING

1. If  $G \vdash \lambda x: s_1. e_2 : t_1 \rightarrow t_2$ , then  $t_1 \leq s_1$  and  $G, x : s_1 \vdash e_2 : t_2$ .
2. If  $G \vdash \{e_1, \dots, e_m\} : t_1^* \dots^* t_n$ , then  $m \geq n$  and  $G \vdash e_i : t_i$ , for  $1 \leq i \leq m$ .
3. If  $G \vdash \text{In}_i[t_1 + \dots + t_m] e : t_1 + \dots + t_n$ , then  $m \leq n$  and  $G \vdash e : t_i$ , for  $1 \leq i \leq m$ .

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

## SUBSTITUTION LEMMA

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

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 ruled out (can't take a step).

$$\text{Case } \frac{G \vdash e_1 : t_{11} \rightarrow t_{12} \quad G \vdash e_2 : t_{11}}{G \vdash e_1 e_2 : t_{12}} \quad (\text{T-App})$$

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 T-App)

Subcase  $e_2 \rightarrow e_2'$ : result follows. (IH and T-App)

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

if  $e$  takes a step, then it must be  
the case that  $e_j \rightarrow e'_j$  for some field  $e_j$ .

(E-Tuple)

if  $e_j : t_j$ , then  $e'_j : t_j$ .

(IH)

Therefore,  $e' : t_1 * \dots * t_n$

(T-Tuple)

QED.

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

There are two evaluation rules by which  $e.j$  can take a step.

Subcase E-ProjTuple:  $e = \{v_1, \dots, v_n\}$ ,  $e' = v_j$ .

forall  $i : v_i : t_i$

(Component typing)

therefore  $e.j : t_j$  and  $v_j : t_j$

(T-Proj)

Subcase E-ProjTuple1:  $e = e_1.j$ ,  $e' = e_1'.j$

result follows.

(IH and T-Proj)

# PRESERVATION LEMMA (CONT'D)

- Case  $\frac{G \mid - e : t_i}{G \mid - \text{in}_i[t_1 + \dots + t_n] e : t_1 + \dots + t_n}$  (T-Ini)  
 if  $\text{in}_i[t_1 + \dots + t_n] e$  takes a step, then it must be  $e \rightarrow e'$ . (E-Ini)  
 $e' : t_i$  (IH)  
 $\text{in}_i e' : t_1 + \dots + t_n$  (T-Ini)

- Case  $\frac{G \mid - e : t_1 + \dots + t_n \quad " i: G, x:t_i \mid - e_i : t}{G \mid - \text{case } e \text{ of } (\text{in}_1 x \Rightarrow e_1 \mid \dots \mid \text{in}_n x \Rightarrow e_n) : t}$  (T-Case)

Subcase E-CaseIni: result follows (IH and Substitution IH)

Subcase E-Case: result follows (IH and T-Case)

- Case  $\frac{\Gamma \mid - e : t_1 \quad t_1 \leq t_2}{\Gamma \mid - e : t_2}$  (T-Sub)

$e \rightarrow e', e' : t_1$  (IH)

$e' : t_2$  (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 straight forward and useful in examples
- Can we have a minimum type?  
 $t ::= \dots \mid \text{Bot}$   
 $\text{Bot} \leq t \quad (\text{S-Bot})$ 
  - 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,  $v = \lambda x : t1 . e2$  for some  $t1$  and  $e2$ .
- On the other hand, we can also derive  $\vdash v : t1 * t2$ .
  - By Canonical forms,  $v = (e1, e2)$ .
- 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:

```
\x:t .  
  if <check that x is reasonable> then  
    <compute result>  
  else  
    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.