Subtyping & Polymorphism

OVERVIEW

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

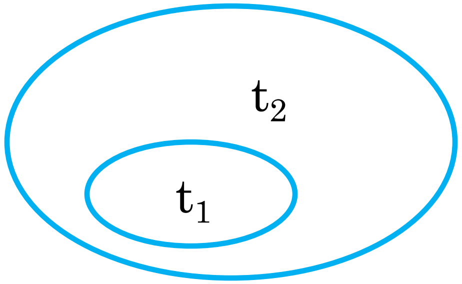
- Which one of the following is NOT a type of polymorphism?

A) Subtype polymorphism
B) Dynamic polymorphism
C) Universal polymorphism
D) Existential polymorphism

Basics

- Type is a collection of values…
- Notation:

  $t_1 <= t_2$



- Basic Properties:

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

- Extending the type system with Top and Subsumption:

t ::= … | Top (like the Object class in Java)

$$\frac{}{t <= \mathrm{Top}}\ \text{(Top)} \qquad \frac{\Gamma \mid\!- e : t_1 \quad t_1 <= t_2}{\Gamma \mid\!- e : t_2}\ \text{(T-Sub)}$$

Example Typing Derivation

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

(let G = f:Top→Top)

```
                              G|-2:int  int<= Top                   G|-true:bool   bool<= Top
                              ---------------------------           --------------------------------
              G|- f : Top→Top      G|-2 : top        G|- f : Top→Top      G|- true : top
            -----------------------------------------------------  -----------------------------------------------
                         f:Top→Top |- f 2: Top           f:Top→Top |- f true:Top
---------------------------------  ---------------------------------------------------------------------------
. |- \x:Top.x : Top→Top          f:Top→Top |- {f 2, f true} : Top* Top
------------------------------------------------------------------------------------------
 . |- let f = \x:Top.x in {f 2, f true} : Top * Top
```

If we used universal polymorphism:

let f = ∀A. λx: A. x in

    {f[int] 2, f[bool] true} : int * bool

Quiz: Type Derivation

- Write down the type derivation tree for:

```
let swap = λp:Top. {p.2, p.1}
in {swap {true, false}, swap {21, 12}}
```

Extending Subtypes to Tuples

- Recall:

$$\frac{\text{for each } i : \Gamma \mid\!- e_i : t_i}{\Gamma \mid\!- \{e_i^{\,\mathrm{i}\in 1..\mathrm{n}}\} : \{t_i^{\,i\in 1..n}\}} \quad \text{(T - Tuple)} \qquad \frac{\Gamma \mid\!- e : \{t_i^{\,i\square 1..n}\} \qquad 1 \le j \le n}{\Gamma \mid\!- e.j : t_j} \quad \text{(T-Proj)}$$

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

$$\frac{m \,\square\, n}{\{t_i^{\,i\square 1..m}\} <= \{t_i^{\,i\square 1..n}\}} \quad \text{(S-TupWidth)}$$

- The reverse is bad: $$\frac{m \le n}{\{t_i^{\,i\square 1..m}\} <= \{t_i^{\,i\square 1..n}\}} \quad \text{(BAD!)}$$

  - 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:

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

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

- Contra-variant Rule is bad:

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

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

Extending Subtypes to Sums

- Given the typing of n-ary sum:

$$\frac{\Gamma \mid\!- e : t_i}{\Gamma \mid\!- in_i[t_1 + \ldots + t_n]\, e : t_1 + \ldots + t_n} \quad \text{(T-Ini)}$$

$$\frac{\Gamma \mid\!- e : t_1 + \ldots + t_n \qquad \forall i \in 1..n : \Gamma, x : t_i \mid\!- e_i : t}{\Gamma \mid\!- \text{case } e \text{ of } (in_1\ x \Rightarrow e_1 \mid \ldots \mid in_n\ x \Rightarrow e_n) : t} \quad \text{(T-Case)}$$

- First consider this rule:

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

- Counter Example:

  case (in$_3$[int+int+int] 0) of

  (in$_1$ x => true

  | in$_2$ x => false)

  - Typechecks since int+int+int <= int + int and due to (T-Case)
  - But gets stuck

Extending Subtypes to Sums

- The correct rule is:

$$\frac{m <= n}{t_1 + \ldots + t_m <= t_1 + \ldots + t_n} \quad \text{(S-SumWid)}$$

- The co-variant rule:

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

- 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 ➔ int* int + int <= int * int * int + int
```

FUNCTIONS

$$\frac{t_1 <= t_1' \quad t_2 <= t_2'}{t_1 \to t_2 <= t_1' \to t_2'} \quad \text{(Bad!)} \qquad \frac{t_1 <= t_1' \quad t_2' <= t_2}{t_1 \to t_2 <= t_1' \to t_2'} \quad \text{(Bad!)}$$

Contravariant

$$\frac{t_1' <= t_1 \quad t_2' <= t_2}{t_1 \to t_2 <= t_1' \to t_2'} \quad \text{(Bad!)} \qquad \frac{t_1' <= t_1 \quad t_2 <= t_2'}{t_1 \to t_2 <= t_1' \to t_2'} \quad \text{(S-Func)}$$

Covariant

- Counter examples
  - (\x:int*int*int. {x.3, x.3, x.3}) {2, 3}
    - int*int*int <= int*int, rule 1 and 2 are bad!
  - ((\x:int*int*int. {x.3, x.3, x.3}) {1, 2, 3}).4
    - int*int*int→int*int*int <= int*int*int→int*int*int*int: rule 3 is bad!
- Intuition:
  - if a function f is of type t1→t2
  - f accepts elements of type t1, and also subtype t1' of t1;
  - f returns elements of type t2, which also belongs to supertype t2'.
- 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 . |- v : t then

(1) if $t = t_1 \rightarrow t_2$ then $v = \backslash x{:}s_1.e$, where $t_1 <= s_1$;

(2) if $t = t_1 * \ldots * t_n$ then $v = (v_1, \ldots, v_m)$, where $m>=n$;

(3) if $t = t_1 + \ldots + t_n$ then $v = in\_i[t_1+\ldots t_m]\ (v)$ where $m<=n$, $1 <= i <= m$.

Proof:

By induction on the typing derivation |- v: t

Case:

```
|- v : t'    t' <= t
---------------------    (subsumption rule)
   |- v : t
```

subcase (1) $t = t1 \rightarrow t2$

(1) $t' <= t1 \rightarrow t2$	(By assumption)
(2) $t' = t1' \rightarrow t2'$ and $t1 <= t1'$ and $t2' <= t2$	(By 1 and S-Func)
(3) $v = \backslash x{:}t''.e$ and $t1' <= t''$	(IH)
(4) $t1 <= t''$.	(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 → 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 $$\frac{\Gamma \mid\!- e_1 : t_{11} \to t_{12} \qquad \Gamma \mid\!- e_2 : t_{11}}{\Gamma \mid\!- e_1\ e_2 : t_{12}}$$ (T-App)

subcase 1: e1 can take a step	(By IH)
then e1 e2 can take a step.	(By E-App1)
subcase 2: e2 can take a step	(By IH)
then e1 e2 can take a step	(By E-App2)
subcase 3: e1 and e2 are both values	(By IH)
e1 = \x:s_{11}.e_{12}	(By canonical forms)
e1 e2 can take a step	(By E-AppAbs)

Progress Lemma (cont'd)

Case $\dfrac{\text{for each } i : \Gamma \mid- e_i : t_i}{\Gamma \mid- \{e_i^{i\Box 1..n}\} : \{t_i^{i\Box 1..n}\}}$ (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 by definition, {e_i, i \in 1..n} is also value.

Case $\dfrac{\Gamma \mid- e : \{t_i^{i\in 1..n}\}}{\Gamma \mid- e.j : t_j}$ (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 = {v1, v2, …, vm}, m>= 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 $\dfrac{\Gamma \mid\!- e : t_1 \quad t_1 <= t_2}{\Gamma \mid\!- e : t_2}$ (T-Sub) is true by IH.

Lemma: Inversion of Subtyping

(1) if t <= t1’ $\rightarrow$ t2' then t = t1 $\rightarrow$ t2 and t1' <= t1
          and t2 <= t2’

(2) if t <= t1 * … * tn then

  t = t1 * … * tm and m >= n

  and for i = 1, … n, ti <= ti’

(3) if t <= top then t can be any type

(4) if t <= bool then t = bool

Prove: By observation on the subtyping relations

Lemma: Component Typing

1. If G |- \x: s_1. e_2 : $t_1 \rightarrow t_2$, then t_1 <= s_1 and G, x : s_1 |- e_2 : t_2.
2. If G |- {e_1, …, e_m} : t_1* … * t_n, then m>= n and G |- e_i : t_i, for 1 <= i <= m.
3. If G |- ln_i[t_1+…+t_m] e : t_1 + … + t_n, then m<=n and G |- e : t_i, for 1<=i<=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 |- e : t, and e → e', then G |- 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).

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

For e1 e2 to take a step, there are three possible rules, hence three subcases:

Subcase e1→ e1': result follows. (IH and T-App)

Subcase e2→ e2': result follows. (IH and T-App)

Subcase e1 = \x : s11. e12, e2 = v, e' = e12[v/x]:

(1) t11<=s11 and G, x:s11 |- e12 : t12 (Component Typing Lemma)

(2) G |- v : s11 (Assumption & T-Sub)

(3) G |- e' : t12. (By (2) and Substitution lemma)

QED.

Preservation Lemma (cont'd)

Case $\dfrac{\text{for each } i : \Gamma \mid - e_i : t_i}{\Gamma \mid - \{e_i^{\,i \in 1..n}\} : \{t_i^{\,i \in 1..n}\}}$ (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 *…* t_n (T-Tuple)
QED.

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

There are two evaluation rules by which e.j can take a step.
Subcase E-ProjTuple: e = $\{v_1, \ldots, v_n\}$, 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 $\dfrac{\Gamma \mid\!- e : t_i}{\Gamma \mid\!- in_i[t_1+\ldots+t_n]\ e : t_1 + \ldots + t_n}$ (T-Ini)

  if $in_i[t_1+\ldots+t_n]e$ takes a step, then it must be $e \rightarrow e'$. (E-Ini)

  $e' : t_i$ (IH)

  $in_i\ e' : t_1 + \ldots + t_n$ (T-Ini)

- Case $\dfrac{\Gamma \mid\!- e : t_1 + \ldots + t_n \qquad \forall i: \Gamma, x{:}t_i \mid\!- e_i : t}{\Gamma \mid\!- \text{case } e \text{ of } (in_1\ x \Rightarrow e_1 \mid \ldots \mid in_n\ x \Rightarrow e_n) : t}$ (T-Case)

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

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

- Case $\dfrac{\Gamma \mid\!- e : t_1 \quad t_1 <= t_2}{\Gamma \mid\!- e : t_2}$ (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 ::= … | Bot

  Bot <= t (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 |- v : Top → Top.
  - By Canonical forms, v = \x : t1 . e2 for some t1 and e2.
- On the other hand, we can also derive |- 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.