Design Principles of Programming Languages

Bounded Quantification

Zhenjiang Hu, Haiyan Zhao, Yingfei Xiong

Peking University, Spring Term, 2016

Print a list of exceptions

```
void printCollection(Collection<Exception> c) {
    for (Exception e : c) {
        System.out.println(e.getMessage());
    }
}
```

- Problem: Collection<ArgumentException> cannot be passed.

Print a list of exceptions

```
void <T> printCollection(Collection<T> c) {
    for (T e : c) {
        System.out.println(e.getMessage());
    }
}
```

- Compilation error at “e.getMessage()”

Print a list of exceptions

```
void <T extends Exception>
printCollection(Collection<T> c) {
    for (T e : c) {
        System.out.println(e.getMessage());
    }
}
```

Print a list of exceptions

```
void printCollection(Collection<? extends Exception> c) {
    for (Exception e : c) {
        System.out.println(e.getMessage());
    }
}
```

Bounded Quantification

- Confine a type variable to be a subtype of some other type

Another Motivating Example

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};

f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}

rab = {a=0, b=true};
```

- What is the type of “f2 rab”?

Another Motivating Example

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};
```

```
f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}
```

```
rab = {a=0, b=true};
```

- What is the type of “(f2 rab).orig”?
  - {a=0, b=true} : {a:Nat}

Another Motivating Example

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};
```

```
f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}
```

```
rab = {a=0, b=true};
```

- What is the type of “(f2 rab).orig”?
  - {a=0, b=true} : {a:Nat}
- What is the type of “(f2 rab).orig as {a:Nat, b:Bool}”?

Another Motivating Example

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};

f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}

rab = {a=0, b=true};
```

- What is the type of “(f2 rab).orig”?
  - {a=0, b=true} : {a:Nat}
- What is the type of “(f2 rab).orig as {a:Nat, b:Bool}”?
  - typing error

Another Motivating Example

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};

f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}

rab = {a=0, b=true};
```

- Unbounded polymorphism does not help either

```
f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)};
▸ Error: Expected record type
```

Another Motivating Example

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};

f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}

rab = {a=0, b=true};
```

- Solution: bounded quantification

```
  f2poly = λX<:{a:Nat}. λx:X. {orig=x, asucc=succ(x.a)};
▸ f2poly : ∀X<:{a:Nat}. X → {orig:X, asucc:Nat}
```

Formalizing bounded quantification

- Modifying typing rules

$$\frac{\Gamma, X \boxed{<: T_1} \vdash t_2 : T_2}{\Gamma \vdash \lambda X \boxed{<: T_1} . t_2 : \forall X \boxed{<: T_1} . T_2} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X \boxed{<: T_{11}} . T_{12} \qquad \boxed{\Gamma \vdash T_2 <: T_{11}}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TAPP)}$$

A problem on subtyping

- What is the subtyping relation between A, B and C?
  - Even <: Nat
  - A = λX<:Even. λx: X. {x}
  - B = λX<:Nat. λx: X. {x}
  - C = λX<:Even. λx: X. {x, x}

A problem on subtyping

- What is the subtyping relation between A, B and C?
  - Even <: Nat
  - A = λX<:Even. λx: X. {x}
  - B = λX<:Nat. λx: X. {x}
  - C = λX<:Even. λx: X. {x, x}
- Kernel: only terms with the same bound are comparable
  - C <: A
- Full: Quantification are compared similar to functions
  - B <: A, C <: A

System $F_{<:}$

Syntax

$t ::=$		*terms:*
	x	*variable*
	$\lambda x{:}T.t$	*abstraction*
	$t\ t$	*application*
	$\lambda X{<:}T.t$	*type abstraction*
	$t\ [T]$	*type application*
$v ::=$		*values:*
	$\lambda x{:}T.t$	*abstraction value*
	$\lambda X{<:}T.t$	*type abstraction value*
$T ::=$		*types:*
	X	*type variable*
	Top	*maximum type*
	$T \to T$	*type of functions*
	$\forall X{<:}T.T$	*universal type*
$\Gamma ::=$		*contexts:*
	$\varnothing$	*empty context*
	$\Gamma, x{:}T$	*term variable binding*
	$\Gamma, X{<:}T$	*type variable binding*

Evaluation

$$(\lambda X{<:}T_{11}.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \qquad \text{(E-TAPPTABS)}$$

Subtyping $\boxed{\Gamma \vdash S <: T}$

$$\Gamma \vdash S <: S \qquad \text{(S-REFL)}$$

$$\frac{\Gamma \vdash S <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash S <: T} \qquad \text{(S-TRANS)}$$

$$\Gamma \vdash S <: Top \qquad \text{(S-TOP)}$$

$$\frac{X{<:}T \in \Gamma}{\Gamma \vdash X <: T} \qquad \text{(S-TVAR)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(S-ARROW)}$$

$$\frac{\Gamma, X{<:}U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X{<:}U_1.S_2 <: \forall X{<:}U_1.T_2} \qquad \text{(S-ALL)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma, X{<:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X{<:}T_1.t_2 : \forall X{<:}T_1.T_2} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X{<:}T_{11}.T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \qquad \text{(T-TAPP)}$$

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

Only show the modified rules from System F

Exercise

- Can you write S-ALL rule for the full system?

Exercise

- Can you write S-ALL rule for the full system?

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X<:T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<:S_1 . S_2 <: \forall X<:T_1 . T_2} \quad \text{(S-ALL)}$$

Preservation: If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.

- Proof: Induction on the typing rules

Evaluation $\boxed{t \longrightarrow t'}$

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

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TAPP)}$$

$$(\lambda X <: T_{11} . t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TAPPTABS)}$$

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

Typing $\boxed{\Gamma \vdash t : T}$

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

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

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

$$\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1 . t_2 : \forall X <: T_1 . T_2} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X <: T_{11} . T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TAPP)}$$

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

Preservation Proof

- No evaluation:
  - T-VAR, T-ABS, T-TABS
- T-APP
  - E-APP1, E-APP2: induction hypothesis
  - E-APPABS: narrowing
- T-TAPP
  - E-TAPP: induction hypothesis
  - E-TAPPTABS: narrowing
- T-SUB
  - Induction hypothesis

Narrowing

- If
  $\Gamma, X <: Q, \Delta \vdash S <: T$
  and
  $\Gamma \vdash P <: Q$,
  then
  $\Gamma, X <: P, \Delta \vdash S <: T$.

- If
  $\Gamma, X <: Q, \Delta \vdash t : T$
  and
  $\Gamma \vdash P <: Q$,
  then
  $\Gamma, X <: P, \Delta \vdash t : T$.

Progress

- If t is closed, well-typed $F_{<:}$ term, then either t is a value or else there is some t' with $t \longrightarrow t'$.

- Proof: Induction on the typing rule

Bounded Existential Types

New syntactic forms

$\mathsf{T} ::= \ldots$ — *types:*

$\{\exists \mathsf{X} \boxed{<:\mathsf{T}}, \mathsf{T}\}$ — *existential type*

New subtyping rules $\boxed{\Gamma \vdash \mathsf{S} <: \mathsf{T}}$

$$\frac{\Gamma, \mathsf{X}<:\mathsf{U} \vdash \mathsf{S}_2 <: \mathsf{T}_2}{\Gamma \vdash \{\exists \mathsf{X}<:\mathsf{U}, \mathsf{S}_2\} <: \{\exists \mathsf{X}<:\mathsf{U}, \mathsf{T}_2\}} \quad \text{(S-SOME)}$$

New typing rules $\boxed{\Gamma \vdash \mathsf{t} : \mathsf{T}}$

$$\frac{\Gamma \vdash \mathsf{t}_2 : [\mathsf{X} \mapsto \mathsf{U}]\mathsf{T}_2 \qquad \Gamma \vdash \mathsf{U} <: \mathsf{T}_1}{\Gamma \vdash \{{*}\mathsf{U}, \mathsf{t}_2\} \ \mathtt{as}\ \{\exists \mathsf{X} <:\mathsf{T}_1, \mathsf{T}_2\} : \{\exists \mathsf{X} <:\mathsf{T}_1, \mathsf{T}_2\}} \quad \text{(T-PACK)}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \{\exists \mathsf{X} <:\mathsf{T}_{11}, \mathsf{T}_{12}\} \qquad \Gamma, \mathsf{X} <:\mathsf{T}_{11}, \mathsf{x}:\mathsf{T}_{12} \vdash \mathsf{t}_2 : \mathsf{T}_2}{\Gamma \vdash \mathtt{let}\ \{\mathsf{X}, \mathsf{x}\} = \mathsf{t}_1\ \mathtt{in}\ \mathsf{t}_2 : \mathsf{T}_2} \quad \text{(T-UNPACK)}$$

Review: Encoding Abstract Data Types

```
counterADT =
   {*{x:Nat},
    {new = {x=1},
     get = λi:{x:Nat}. i.x,
     inc = λi:{x:Nat}. {x=succ(i.x)}}}
 as {∃Counter,
     {new: Counter, get: Counter→Nat, inc: Counter→Counter}};
```

```
▸ counterADT : {∃Counter,
                 {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

```
let {Counter,counter} = counterADT in
counter.get (counter.inc counter.new);
```

```
▸ 2 : Nat
```

Exercise: Can you define a sub type ResetCounter?

```
counterADT =
   {*{x:Nat},
    {new = {x=1},
     get = λi:{x:Nat}. i.x,
     inc = λi:{x:Nat}. {x=succ(i.x)}}}
 as {∃Counter,
     {new: Counter, get: Counter→Nat, inc: Counter→Counter}};

▸ counterADT : {∃Counter,
                 {new:Counter,get:Counter→Nat,inc:Counter→Counter}}

let {Counter,counter} = counterADT in
counter.get (counter.inc counter.new);

▸ 2 : Nat
```

1. "reset" method to set x to 0
2. ResetCounter can be used as Counter:
   counter.inc resetCounter.reset

Key to excercise

```
let {Counter, counter} = counterADT in
let ResetCounterADT =
{*{x:Nat},
    {new = counter.new, get = counter.get, inc=counter.inc,
     reset= {x=0}}
    as {∃ResetCounter <: Counter,
          {new: ResetCounter, get: ResetCounter->Nat,
           inc:ResetCounter->ResetCounter,
           reset: ResetCounter->ResetCounter}}} in
let {ResetCounter, resetCounter} = ResetCounterADT in
counter.inc resetCounter.reset
```