# Chapter 26: Bounded Quantification

Polymorphism + subtyping

Foundations of OO

# Motivation

- Limitation of Subtyping

```
f = λx:{a:Nat}. x;
ra = {a=0};
rab = {a=0, b=true};
```

```
  f ra;
▸ {a=0} : {a:Nat}

  f rab;
▸ {a=0, b=true} : {a:Nat}
```

by passing rab through the identity function, we have lost the ability to access its b field!

# Motivation

- Could polymorphism help?

```
f = λx:{a:Nat}. x;
    ➔ fpoly = λX. λx:X. x;
ra = {a=0};
rab = {a=0, b=true};
```

do abstraction

```
fpoly [{a:Nat, b:Bool}] rab;
▸ {a=0, b=true} : {a:Nat, b:Bool}
```

But what if we have the following function?

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

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

# Motivation

- Solution: Bounded Quantification

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

# kernel F<:

*Syntax*

| | | | |
|---|---|---|---|
| t | ::= | | *terms:* |
| | | x | *variable* |
| | | λx:T.t | *abstraction* |
| | | t t | *application* |
| | | λX<:T.t | *type abstraction* |
| | | t [T] | *type application* |
| | | | |
| v | ::= | | *values:* |
| | | λx:T.t | *abstraction value* |
| | | λX<:T.t | *type abstraction value* |

# kernel F<:

| | | | |
|---|---|---|---|
| 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* |

# kernel F<:

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

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

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

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

# kernel F<:

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

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

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

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

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

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

# kernel F<:

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

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

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

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

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

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

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

# kernel F<:

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

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

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

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

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

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

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

# Bounded and Unbounded Quantification

- F<: provides only bounded quantification, but it actually covers unbounded quantification of pure System F.

$$\forall X.T \stackrel{\text{def}}{=} \forall X<:Top.T$$

# Scoping of Type Variables

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

| | | | |
|---|---|---|---|
| $\Gamma_1$ | $=$ | $X<:Top,\ y:X\to Nat$ | Ok |
| $\Gamma_2$ | $=$ | $y:X\to Nat,\ X<:Top$ | X |
| $\Gamma_3$ | $=$ | $X<:\{a:Nat,b:X\}$ | X |
| $\Gamma_4$ | $=$ | $X<:\{a:Nat,b:Y\},\ Y<:\{c:Bool,d:X\}$ | X |

Whenever we mention a type T in a context,
the free variables of T should be bound in the portion of
the context to the left of where T appears.

# Full F<:

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

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

# Programming Examples

- Encoding Products

$$\text{Pair } X\ Y = \lambda R.\ (X \to Y \to R) \to R$$

$$\text{pair} = \lambda X.\ \lambda Y.\ \lambda x{:}X.\ \lambda y{:}Y.\ \lambda R.\ \lambda p.\ \ p\ x\ y$$

$$\text{fst} = \lambda X.\ \lambda Y.\ \lambda p.\ p\ [X]\ (\lambda x.\ \lambda y \to x)$$
$$\text{snd} = \lambda X.\ \lambda Y.\ \lambda p.\ p\ [X]\ (\lambda x.\ \lambda y \to y)$$

$$\frac{\Gamma \vdash S_1 <: T_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash \text{Pair } S_1\ S_2 <: \text{Pair } T_1\ T_2}$$

# Programming Examples

- Encoding Records

$$\{T_i\ ^{i\in 1..n}\} \stackrel{\text{def}}{=} \text{Pair } T_1\ (\text{Pair } T_2\ \ldots\ (\text{Pair } T_n\ \text{Top})\ldots).$$

$$\{t_i\ ^{i\in 1..n}\} \stackrel{\text{def}}{=} \text{pair } t_1\ (\text{pair } t_2\ \ldots\ (\text{pair } t_n\ \text{top})\ldots),$$

$$\frac{\Gamma \vdash\ ^{i\in 1..n} S_i <: T_i}{\Gamma \vdash \{S_i\ ^{i\in 1..n+k}\} <: \{T_i\ ^{i\in 1..n}\}}$$

$$\frac{\Gamma \vdash\ ^{i\in 1..n} t_i : T_i}{\Gamma \vdash \{t_i\ ^{i\in 1..n}\} : \{T_i\ ^{i\in 1..n}\}}$$

$$\frac{\Gamma \vdash t : \{T_i\ ^{i\in 1..n}\}}{\Gamma \vdash t.i : T_i}$$

# Programming Examples

- Church Encodings with Subtyping

```
CNat = ∀X. (X→X) → X → X;
```

⬇

```
SNat = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → X;
```

# Programming Examples

- Type Refinement (Subtype)

```
SNat = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → X;

SZero = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → Z;
SPos = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → S;
```

```
  szero = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. z;

▸ szero : SZero
```

```
  spluspp = λn:SPos. λm:SPos.
              λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
                n [X] [S] [S] s (m [X] [S] [Z] s z);

▸ spluspp : SPos → SPos → SPos
```

# Safety

THEOREM [PRESERVATION]: If $\Gamma \vdash \texttt{t} : \texttt{T}$ and $\texttt{t} \longrightarrow \texttt{t}'$, then $\Gamma \vdash \texttt{t}' : \texttt{T}$.

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

# Bounded Existential Types

*New syntactic forms*

| | | | |
|---|---|---|---|
| T | ::= | ... | *types:* |
| | | $\{\exists X <: T, T\}$ | *existential type* |

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

$$\frac{\Gamma, X <: U \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X <: U, S_2\} <: \{\exists X <: U, T_2\}} \quad \text{(S-SOME)}$$

# Bounded Existential Types

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

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2 \qquad \Gamma \vdash U <: T_1}{\Gamma \vdash \{*U, t_2\} \text{ as } \{\exists X <: T_1, T_2\} : \{\exists X <: T_1, T_2\}} \quad \text{(T-PACK)}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X <: T_{11}, T_{12}\} \qquad \Gamma, X <: T_{11}, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2} \quad \text{(T-UNPACK)}$$

# An Example

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

We can use this counter ADT exactly as we did before:

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

▸ 3 : Nat

We are now permitted to use Counter values directly as numbers:

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

▸ 4 : Nat

But we are not able to use numbers as Counters:

```
let {Counter,counter} = counterADT in
counter.inc 3;
```

▸ Error: parameter type mismatch