

Programming Languages, CPSC 604—Slides 13

Bounded Quantification

Jaakko Järvi

April 9, 2009

Outline

1 Introduction

2 Why F-bounds?

Bounded quantification

- Combining subtyping and polymorphism
- Straightforward but uninteresting idea: keep both orthogonal
- Following two examples demonstrate why this is not adequate

Example 1

```
rec_a_id = λx:{a:nat}. x;
```

```
%> rec_a_id : {a:nat} → {a:nat}
```

```
rec_a_id {a=1}
```

```
%> {a=1} : {a:nat}
```

```
rec_a_id {a=1, b=true} // this is ok,  
                        // {a:nat, b:bool} <: {a:nat}
```

```
%> {a=1, b=true} : {a:nat}
```

```
(rec_a_id {a=1, b=true}).b // error!
```

Example 1

```
rec_a_id = λx:{a:nat}. x;
```

```
%> rec_a_id : {a:nat} → {a:nat}
```

```
rec_a_id {a=1}
```

```
%> {a=1} : {a:nat}
```

```
rec_a_id {a=1, b=true} // this is ok,  
                        // {a:nat, b:bool} <: {a:nat}
```

```
%> {a=1, b=true} : {a:nat}
```

```
(rec_a_id {a=1, b=true}).b // error!
```

- Subtyping allows functions to work with more than one type, but some type information is lost with subsumption

Example 1

- Polymorphic types come to rescue. Now `poly_id` works for all types, and returns the same type as the incoming type:

```
poly_id =  $\Lambda T. \lambda x:T. x$ ;
```

```
%> poly_id :  $\forall T. T \rightarrow T$ 
```

```
poly_id [{a:nat, b=bool}] {a=1, b:true}
```

```
%> {a=1, b=true} : {a:nat, b=bool}
```

Example 1

- Polymorphic types come to rescue. Now `poly_id` works for all types, and returns the same type as the incoming type:

```
poly_id =  $\Lambda T. \lambda x:T. x$ ;
```

```
%> poly_id :  $\forall T. T \rightarrow T$ 
```

```
poly_id [{a:nat, b=bool}] {a=1, b:true}
```

```
%> {a=1, b=true} : {a:nat, b=bool}
```

- However, no operations on the parameterized types are possible...

Example 2

- Consider the following, where a “non-parametric” operation is performed to the argument of the function

```
rec_a_next =  $\lambda x:\{a:\text{nat}\}. \{old=x, next=(succ\ x.a)\};$   
%> rec_a_next : {a:nat}  $\rightarrow$  {old:{a:nat}, next:nat}
```

```
rec_a_next {a=1}  
%> {old={a=1},next=2} : {old:{a:nat}, next:nat}
```

```
rec_a_next {a=1, b=true}  
%> {old={a=1,b=true},next=2} : {old:{a:nat}, next:nat}
```

Example 2

- Consider the following, where a “non-parametric” operation is performed to the argument of the function

```
rec_a_next = λx:{a:nat}. {old=x, next=(succ x.a)};
%> rec_a_next : {a:nat} → {old:{a:nat}, next:nat}
```

```
rec_a_next {a=1}
%> {old={a=1},next=2} : {old:{a:nat}, next:nat}
```

```
rec_a_next {a=1, b=true}
%> {old={a=1,b=true},next=2} : {old:{a:nat}, next:nat}
```

- Attempt to fix this with type abstraction fails:

```
poly_a_next = ΛT. λx:T. {old=x, next=(succ x.a)};
%> error!
```

Example 2

- Consider the following, where a “non-parametric” operation is performed to the argument of the function

```
rec_a_next = λx:{a:nat}. {old=x, next=(succ x.a)};
%> rec_a_next : {a:nat} → {old:{a:nat}, next:nat}
```

```
rec_a_next {a=1}
%> {old={a=1},next=2} : {old:{a:nat}, next:nat}
```

```
rec_a_next {a=1, b=true}
%> {old={a=1,b=true},next=2} : {old:{a:nat}, next:nat}
```

- Attempt to fix this with type abstraction fails:

```
poly_a_next = ΛT. λx:T. {old=x, next=(succ x.a)};
%> error!
```

- We want to qualify “for all” somehow
- This is where we mix subtyping and universal types

Bounded quantification: System $F_{<}$:

```
poly_a_next2 =
   $\Lambda T <: \{a: \text{nat}\}. \lambda x: T. \{old=x, next=(succ\ x.a)\};$ 
```

```
poly_a_next2 :
   $\forall T <: \{a: \text{nat}\}. T \rightarrow \{old: T, next: \text{nat}\};$ 
```

- All quantifiers (universal and existential) can carry subtype constraints
- Two systems: **kernel** $F_{<}$: and **full** $F_{<}$:

Kernel $F_{<}$:

• Syntax

$$t ::= x \mid v \mid t t$$

$$v ::= \lambda x : T. t \mid \Lambda X <: T. t \mid t[T]$$

$$T ::= X \mid T \rightarrow T \mid \forall X <: T. T \mid \text{top}$$

$$\Gamma ::= \emptyset \mid \Gamma, x : T \mid \Gamma, X <: T$$

• Evaluation rules

E-AppFun

$$\frac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2}$$

E-AppArg

$$\frac{t \rightarrow t'}{v t \rightarrow v t'}$$

E-AppAbs

$$(\lambda x : T. t) v \rightarrow [v/x]t$$

E-TypeApp

$$\frac{t_1 \rightarrow t_1'}{t_1[T] \rightarrow t_1'[T]}$$

E-TypeAppAbs

$$(\Lambda X <: T_1. t)[T_2] \rightarrow [T_2/X]t$$

• Typing rules

T-Variable

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

T-Abstraction

$$\frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x : T. u : T \rightarrow U}$$

T-Application

$$\frac{\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : U}{\Gamma \vdash t u : T}$$

T-TypeAbstraction

$$\frac{\Gamma, X <: T_1 \vdash t : T_2}{\Gamma \vdash \Lambda X <: T_1. t : \forall X <: T_1. T_2}$$

T-TypeApplication

$$\frac{\Gamma \vdash t : \forall X <: T_1. T_2 \quad \Gamma \vdash U <: T_1}{\Gamma \vdash t[U] : [U/X]T_2}$$

T-Subsumption

$$\frac{\Gamma \vdash t : S \quad \Gamma \vdash S <: T}{\Gamma \vdash t : T}$$

Kernel $F_{<}$: continues

$$\text{S-Reflexivity} \\ \frac{}{\Gamma \vdash T <: T}$$

$$\text{S-Transitivity} \\ \frac{\Gamma \vdash T <: U \quad \Gamma \vdash U <: V}{\Gamma \vdash T <: V}$$

$$\text{S-Top} \\ \frac{}{\Gamma \vdash T <: \text{top}}$$

$$\text{S-Function} \\ \frac{\Gamma \vdash T_1 <: T_2 \quad \Gamma \vdash U_1 <: U_2}{\Gamma \vdash T_2 \rightarrow U_1 <: T_1 \rightarrow U_2}$$

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

$$\text{S-All} \\ \frac{\Gamma, X <: U \vdash S <: T}{\Gamma \vdash \forall X <: U. S <: \forall X <: U. T}$$

Bounded vs. unbounded quantification

- Is parametric polymorphism still possible?
- What happened to the rules where no quantification is used?

Bounded vs. unbounded quantification

- Is parametric polymorphism still possible?
- What happened to the rules where no quantification is used?
- A type parameter without a bound is just syntactic sugar for a type parameter with `top` as the bound

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

Well typed contexts

- In $\Gamma \vdash t : T$ both t and T can contain **free** type variables, which must be bound in Γ
- Types in Γ can contain free type variables too!
- Different interpretations of scoping within the environment
- The book discusses a restrictive variant
- Practical languages (C#, Java, Eiffel) have adopted more flexible (but each different) rules

Scoping in typing context

- Is this context well typed?

$\Gamma_1 = X < : \text{top}, y : X \rightarrow \text{nat}$

Scoping in typing context

- Is this context well typed? Yes

$\Gamma_1 = X<:\text{top}, y:X \rightarrow \text{nat}$

- Could arise e.g. from typing the following term

$\Lambda X<:\text{top}. \lambda y:X \rightarrow \text{nat}. t$

Scoping in typing context

- Is this context well typed? Yes

$$\Gamma_1 = X<:\text{top}, y:X\rightarrow \text{nat}$$

- Could arise e.g. from typing the following term

$$\Lambda X<:\text{top}. \lambda y:X\rightarrow \text{nat}. t$$

- How about this?

$$\Gamma_2 = y:X\rightarrow \text{nat}, X<:\text{top}$$

Scoping in typing context

- Is this context well typed? Yes

$$\Gamma_1 = X<:\text{top}, y:X\rightarrow \text{nat}$$

- Could arise e.g. from typing the following term

$$\Lambda X<:\text{top}. \lambda y:X\rightarrow \text{nat}. t$$

- How about this? No

$$\Gamma_2 = y:X\rightarrow \text{nat}, X<:\text{top}$$

- Could arise e.g. from typing the following term (where first X is not bound)

$$\lambda y:X\rightarrow \text{nat}. \Lambda X<:\text{top}. t$$

Scoping in typing context

- Are these well-typed contexts?

$$\Gamma_3 = X<:\{a:\text{nat}, b:X\}$$

$$\Gamma_4 = X<:\{a:\text{nat}, b:Y\}, Y<:\{c:\text{bool}, d:X\}$$

$$\Gamma_5 = Y<:\{a:\text{nat}, b:Y\}, X<:Y$$

$$\Gamma_6 = Y<:X, X<:Y$$

Scoping in typing context

- Are these well-typed contexts?

$$\Gamma_3 = X<:\{a:\text{nat}, b:X\}$$

$$\Gamma_4 = X<:\{a:\text{nat}, b:Y\}, Y<:\{c:\text{bool}, d:X\}$$

$$\Gamma_5 = Y<:\{a:\text{nat}, b:Y\}, X<:Y$$

$$\Gamma_6 = Y<:X, X<:Y$$

- Γ_3 is **F-bounded polymorphism**, the rest its extensions

Scoping in typing context

- Are these well-typed contexts?

$$\Gamma_3 = X<:\{a:\text{nat}, b:X\}$$

$$\Gamma_4 = X<:\{a:\text{nat}, b:Y\}, Y<:\{c:\text{bool}, d:X\}$$

$$\Gamma_5 = Y<:\{a:\text{nat}, b:Y\}, X<:Y$$

$$\Gamma_6 = Y<:X, X<:Y$$

- Γ_3 is **F-bounded polymorphism**, the rest its extensions
- C#, Eiffel, Java, Fortress allow all

Scoping in typing context

- Are these well-typed contexts?

$$\Gamma_3 = X<:\{a:\text{nat}, b:X\}$$

$$\Gamma_4 = X<:\{a:\text{nat}, b:Y\}, Y<:\{c:\text{bool}, d:X\}$$

$$\Gamma_5 = Y<:\{a:\text{nat}, b:Y\}, X<:Y$$

$$\Gamma_6 = Y<:X, X<:Y$$

- Γ_3 is **F-bounded polymorphism**, the rest its extensions
- C#, Eiffel, Java, Fortress allow all
- F-bounded quantification crucial for typing **binary methods**

Full $F_{<}$:

- The kernel rule S-All:

$$\frac{\Gamma, X <: U \vdash S <: T}{\Gamma \vdash \forall X <: U. S <: \forall X <: U. T}$$

- Consider bounded-polymorphic types A and B . $A <: B$ if both have the same type parameter and bound, and “body” of A is a subtype of the body B . subtype relation
- The **full** rule S-All:

$$\frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma, X <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: S_1. S_2 <: \forall X <: T_1. T_2}$$

- $A <: B$ if both have the same type parameter and its constraint in A is weaker than in B (bound of B subtype of bound of A), and “body” of A is a subtype of the body B .

Example

- Consider the code:

$$a = \Lambda X < : B. \lambda x : X. \{u=x\};$$

$$b = \Lambda X < : B. \lambda x : X. \{u=x, v=x\};$$

$$a : TA = \forall X < : B. \{u:X\}$$

$$b : TB = \forall X < : B. \{u:X, v:X\}$$

- Here, $TB < : TA$ in both systems
- Assume below that $B < : A$

$$a = \Lambda X < : B. \lambda x : X. \{u=x\};$$

$$b = \Lambda X < : A. \lambda x : X. \{u=x, v=x\};$$

$$a : TA = \forall X < : B. \{u:X\}$$

$$b : TB = \forall X < : A. \{u:X, v:X\}$$

- $TB < : TA$ in full $F_{<}$, but not in the kernel variant

Similar examples in Java

- Bounds do not change

```
public class a<X extends B> {  
    X u;  
}  
  
public class b<X extends B> extends a<X> {  
    X v;  
}
```

- Bounds change

```
public class A {}  
public class B extends A {}  
public class a<X extends B> {  
    X u;  
}  
  
public class b<X extends A> extends a<X> { // error  
    X v;  
}
```

Similar examples in Java

- Bounds do not change

```
public class a<X extends B> {  
    X u;  
}  
  
public class b<X extends B> extends a<X> {  
    X v;  
}
```

- Bounds change

```
public class A {}  
public class B extends A {}  
public class a<X extends B> {  
    X u;  
}  
  
public class b<X extends A> extends a<X> { // error  
    X v;  
}
```

- No feature to demonstrate full vs. kernel really

Full vs. Kernel in Java context

- Replacing the definition of `A1<X extends V>` with `A1<X extends U>` is OK (if the body of `A1` stays well-formed)

```
class U {}
class V extends U {}
class A1<T extends V > {
    public Integer a;
}

// can A2 accept all instantiations that A1 can?
class A2 <T extends U> {
    public Integer a;
}

...
(new A1<V>()).a = 5;
(new A2<V>()).a = 5;
```

Outline

- 1 Introduction
- 2 Why F-bounds?

A long example without a single λ !

- We use the following C++-like syntax to demonstrate why plain bounded quantification is not adequate

```
struct point { int x; int y; };  
struct color_point : public point { int color; };
```

- This establishes

```
color_point <: point
```

- A foundational paper:



Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell.

F-bounded polymorphism for object-oriented programming.
Functional Programming Languages and Computer Architecture,
pages 273–280, ACM, 1989.

Subtyping example

```
struct point { int x; int y; };
struct color_point : public point { int color; };

point move(point a, int dx, int dy)
{ a.x += dx; a.y += dy; return a; }

int main () {

    point p; p.x = 0; p.y = 0;
    p = move(p, 1, 2);
    assert(p.x == 1 && p.y == 2);

    color_point cp; cp.x = 0; cp.y = 0; cp.color = 0;
    p = move(cp, 1, 2);
    assert(p.x == 1 && p.y == 2);

    return 0;
};
```

- With just subtyping, the exact type of `cp` is lost when passed in to and out from `move()`

Let's try

```
struct point { int x; int y; };
struct color_point : public point { int color; };

point move(point a, int dx, int dy)
{ a.x += dx; a.y += dy; return a; }

int main () {
    color_point cp; cp.x = 0; cp.y = 0; cp.color = 0;
    cp = move(cp, 1, 2);
    return 0;
}
```

Let's try

```

struct point { int x; int y; };
struct color_point : public point { int color; };

point move(point a, int dx, int dy)
{ a.x += dx; a.y += dy; return a; }

int main () {
    color_point cp; cp.x = 0; cp.y = 0; cp.color = 0;
    cp = move(cp, 1, 2);
    return 0;
}

```

```

fail_point_color_point_subtyping.cpp: In function 'int main()':
fail_point_color_point_subtyping.cpp:9: error: no match for 'operator=' in 'cp =
move(cp.color_point::<anonymous>, 1, 2)' fail_point_color_point_subtyping.cpp:2:
note: candidates are: color_point& color_point::operator=(const color_point&)

```

Bounded quantification

- We know that a possible fix to this lost of type accuracy is to use parametric polymorphism (instead of subtype polymorphism):

```
template <class T>  
T move(T a, int dx, int dy)  
{ a.x += dx; a.y += dy; return a; };
```

- This works in C++:

```
cp = move(cp, 1, 2);
```

Bounded quantification

- We know that a possible fix to this lost of type accuracy is to use parametric polymorphism (instead of subtype polymorphism):

```
template <class T>
T move(T a, int dx, int dy)
{ a.x += dx; a.y += dy; return a; };
```

- This works in C++:

```
cp = move(cp, 1, 2);
```

- ... but how can it? `move` is not intrinsically a parametric function? Its body's working depends on properties of the type parameters

Bounded quantification

- We know that a possible fix to this lost of type accuracy is to use parametric polymorphism (instead of subtype polymorphism):

```
template <class T>  
T move(T a, int dx, int dy)  
{ a.x += dx; a.y += dy; return a; };
```

- This works in C++:

```
cp = move(cp, 1, 2);
```

- ... but how can it? `move` is not intrinsically a parametric function? Its body's working depends on properties of the type parameters
 - Constraints are implicit

Bounded quantification

- Temporarily, let us invent new syntax for C++ to express subtype constraints...

```
template <class T <: point>
T move(T a, int dx, int dy)
{ a.x += dx; a.y += dy; return a; };
```

- The type of this function would be something like:

$$\forall T <: \text{point}. (T, \text{int}, \text{int}) \rightarrow T$$

- Compare to the type of the non-generic move:

```
point move(point a, int dx, int dy)
{ a.x += dx; a.y += dy; return a; }
```

$$(\text{point}, \text{int}, \text{int}) \rightarrow \text{point}$$

... bounded quantification

- [Cardelli, Wegner 85]
- Bounded quantification

$$\forall t. t <: \sigma. \tau[t]$$

where t does not occur in σ .

- The essence of the restriction is, that constraint is fixed for all instantiations.
- Generalizes to multiple type variables.

$$\forall t_1. t_1 <: \sigma_1. (\forall t_2. t_2 <: \sigma_2. \tau[t_1, t_2])$$

Problems with bounded quantification

- Move over to C# now...

```
interface Movable { Movable move(int x, int y); }
```

- Define function `translate` that takes any `Movable` object, and returns another one of the same type, moved one unit along both axis. `translate` should have (roughly) the type: $\forall T <: \text{Movable}. T \rightarrow T$
- First attempt

```
T translate<T>(T m) where T : Movable
{ return m.move(1, 1); }
```

Problems with bounded quantification

- Move over to C# now...

```
interface Movable { Movable move(int x, int y); }
```

- Define function `translate` that takes any `Movable` object, and returns another one of the same type, moved one unit along both axis. `translate` should have (roughly) the type: $\forall T <: \text{Movable}. T \rightarrow T$
- First attempt

```
T translate<T>(T m) where T : Movable
{ return m.move(1, 1); }
```

```
// typing error! move is (roughly) of type
(Movable, int, int) → Movable
```

Problems with bounded quantification

- Move over to C# now...

```
interface Movable { Movable move(int x, int y); }
```

- Define function `translate` that takes any `Movable` object, and returns another one of the same type, moved one unit along both axis. `translate` should have (roughly) the type: $\forall T <: \text{Movable}. T \rightarrow T$
- First attempt

```
T translate<T>(T m) where T : Movable
{ return m.move(1, 1); }
```

```
// typing error! move is (roughly) of type
(Movable, int, int) → Movable
```

- Essentially, we again hit the problem of subtyping losing information

Binary method problem

- Assume the following interface:

```
interface Comparable { bool eq(Comparable); }
```

- Task: Define a function `noteq` for computing the negation of `eq`. The type should be as follows:

```
noteq : ∀ T <: Comparable. (T, T) → bool
```

- This is what bounded quantification enables:

```
bool noteq<T>(T t, T u) where T : Comparable {
  return !t.eq(u);
}
```

- Now define a class `MyInt` which is `Comparable`

Binary method problem

- Assume the following interface:

```
interface Comparable { bool eq(Comparable); }
```

- Task: Define a function `noteq` for computing the negation of `eq`. The type should be as follows:

```
noteq :  $\forall T <: \text{Comparable}. (T, T) \rightarrow \text{bool}$ 
```

- This is what bounded quantification enables:

```
bool noteq<T>(T t, T u) where T : Comparable {
  return !t.eq(u);
}
```

- Now define a class `MyInt` which is `Comparable`

```
class MyInt : Comparable {
  bool eq(MyInt)      { ... } // not a valid override
  bool eq(Comparable) { ... } // meaningless comparison
}
```

F-bounded quantification

- [Canning, Cook, Hill, Olthoff, Mitchell 89]

- F-bounded quantification:

$$\forall t. t <: F[t]. \tau[t]$$

where t can occur in $F[t]$ (and obviously in $\tau[t]$).

- Again, generalization to more than one type parameter:

$$\forall t_1. t_1 <: F_1[t_1]. (\forall t_2. t_2 <: F_2[t_1, t_2]. \tau[t_1, t_2])$$

\Rightarrow a constraint on type t_i may only refer to type parameters $t_j, j \leq i$.

F-bounds in the translate example

- Define function `translate` that takes any `Movable` object, and returns another one of the same type, moved one unit along both axis. `translate` should have (roughly) the type: $\forall T <: \text{Movable}. T \rightarrow T$
- The unsuccessful translate example:

```
interface Movable { Movable move(int x, int y); }
T translate<T>(T m) where T : Movable
{ return m.move(1, 1); }
```

- can now be written as:

```
interface Movable<T> { T move(int x, int y); }

T translate<T>(T m) where T : Movable<T> {
  return m.move(1, 1);
}
```

- The type of `translate` is: $\forall T <: \text{Movable}<T>. T \rightarrow T$

Binary method problem revisited

- Assume the following interface:

```
interface Comparable { bool eq(Comparable); }
```

- Task: Define a function `noteq` for computing the negation of `eq`. The type should be as follows:

```
noteq :  $\forall T <: \text{Comparable}. (T, T) \rightarrow \text{bool}$ 
```

- This is what bounded quantification enables:

```
bool noteq<T>(T t, T u) where T : Comparable {
  return !t.eq(u);
}
```

- Now define a class `MyInt` which is `Comparable`

```
class MyInt : Comparable {
  bool eq(MyInt)      { ... } // not a valid override
  bool eq(Comparable) { ... } // meaningless comparison
}
```

F-bounds to rescue

- The interface becomes:

```
interface Comparable<T> { bool eq(T); }
```

- `noteq` becomes:

```
bool noteq<T>(T t, T u) where T : Comparable<T> {
    return !t.eq(u);
}
```

- `MyInt` can then be defined as:

```
class MyInt : Comparable<MyInt> {
    bool eq(MyInt) { ... }
}
```

Generalizing F-bounds

- Constrained quantification [Curtis 87]
- A mutually recursive system of subtype constraints.
- Can express cases that F-bounds cannot:

```
interface Node<E> {  
    List<E> out_edges();  
}  
  
interface Edge<N> {  
    N source();  
    N target();  
}  
  
void breadth_first_search<N, E>(N n)  
    where N : Node<E>, E : Edge<N> { ... }
```

- This is what Java, C# (2.0), Eiffel provides!
- Actually a bit more: plain type parameters can be used as bounds.

Summary so far

- Minor generalizations of F-bounded polymorphism are the backbone of Java, C#, and Eiffel generics
- The essential feature in F-bounds is that bounds are generic too — they change along the argument that is being tested for
- Subtyping is one way of expressing constraints, there are others (Haskell type classes, C++ concepts, ML signatures)