

## Subtype Polymorphism

In Smalltalk slides, class  $D <: C$  if class  $D$  responds to all messages class  $C$  responds to.

Could replace an instance of  $C$  with an instance of  $D$  and all message sends would still work.

Smalltalk is dynamically typed: type checking performed at runtime.

### General Idea of Subtyping

$\sigma <: \tau$  “ $\sigma$  is a subtype of  $\tau$ ” if you can use a term of type  $\sigma$  in every program context where you could use a term of type  $\tau$ .

### Some Subtyping Rules

$$\frac{}{\sigma <: \sigma} \text{ reflexive} \qquad \frac{\sigma <: \mu \quad \mu <: \tau}{\sigma <: \tau} \text{ transitive}$$

### Tying subtyping to typing: subsumption

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau} \text{ subsume}$$

If term can be shown to have type  $\sigma$  and if terms of type  $\sigma$  can be used wherever terms of type  $\tau$  can be used (def'n of subtyping) then we can generalize type of term to  $\tau$ .

1

## Subtype variations

Subtyping can be used with **structural** typing – types have internal structure, or with **nominative** typing – a type is just a name.

OO programming generally uses nominative typing, declares subtype relationships: “implements”, “inherits”. Type checking of OO programs uses structural typing internally to verify the user’s declared subtype declarations between nominative types.

Subtyping may have a **top** subtype. (type “Object”) with rule

$$\tau <: T$$

May have **bottom** subtype (sometimes useful for exceptions) with rule

$$\perp <: \tau$$

but may get messy...

Also sometimes way of thinking of the constant `null` in Java – but haven’t thought this through.

2

## Some structural subtype rules

### Records

{a=1, b=true}

has operations: `get field a` returning an int  
`get field b` returning a bool

{a=2, b=false, c=3.14}

has operations: `get field a` returning an int  
`get field b` returning a bool

so {a: int, b: bool, c: real} <: {a: int, b: bool}

### General record rules:

$$\frac{}{\{\text{label}_i : \tau_i \mid i \in 1 \dots n + k\} <: \{\text{label}_i : \tau_i \mid i \in 1 \dots n\}} \text{ width}$$

$$\frac{\sigma_i <: \tau_i \quad 1 \leq i \leq n}{\{\text{label}_i : \sigma_i \mid i \in 1 \dots n\} <: \{\text{label}_i : \tau_i \mid i \in 1 \dots n\}} \text{ depth}$$

3

## Some structural subtype rules

### Sums

datatype foo = A of t1 | B of t2 | C of t3  
datatype bar = A of t1 | B of t2

case e of  
  A x => ...  
  B x => ...  
  C x => ...

could type with e: foo or

could type with e: bar (and never execute case C)

So sum with fewer constructors <: sum with more constructors in a case.

Depth rule for sums works the same way as depth rule for records.

**Note:** this rule is for using an existing sum, not for creating a new sum by injection. Similarly, record rules are not for creating a new record out of individual field values.

4

## Some structural subtype rules

**Functions** When can you use  $f$  in place of  $g$ ?

Type of  $f(x) <:$  type of  $g(x)$  for any  $x$  in dom  $g$ .

What about input types?

$f$  must be callable on all inputs that  $g$  can be called on, but  $f$  could be called on with more inputs in places where it does not have to fill in for  $g$ .

Resulting rule:

$$\frac{\tau <: \tau' \quad \sigma' <: \sigma}{\sigma \rightarrow \tau <: \sigma' \rightarrow \tau'} \text{ function}$$

The rule for subtyping functions is said to be **contravariant** in the argument type, and **covariant** in the result type.

**Note** Java up to 1.4 had invariant return type for method overriding – unless return type was type of **this**.

5

## Some structural subtype rules

**Mutable data**

`val x :  $\sigma$  ref = ref initial_value`

Two contexts for use:

`y = ! x`      $y : \tau$ . Need  $\tau <: \sigma$  to use  $y$ .

`x := z`      $z : \mu$ . Need  $\sigma <: \mu$  to store  $z$ .

Resulting rule:

$$\frac{\sigma = \tau}{\sigma \text{ ref } <: \tau \text{ ref}} \text{ ref}$$

References are **invariant**.

Java and C# Arrays are however covariant!

```
using System;
class Program
{
    static void Main(string[] args)
    {
        BaseType[] array = new Derived[10]; // Allowed
        ModifyArray(array);
    }
    static void ModifyArray(BaseType[] array)
    {
        array[0] = new Derived(); // OK
        array[1] = new BaseType(); // throw exception
    }
}
class BaseType {}
class Derived : BaseType {}

Java: ArrayStoreException
C#: ??
```

6

## Subtyping recursive types

To check  $\mu\alpha.\sigma <: \mu\beta.\tau$

(1) Assume that all occurrences of  $\alpha <: \beta$ .

(2) Under that assumption, check whether  $\sigma <: \tau$ .

If  $\alpha$  and / or  $\beta$  appear in both covariant and contravariant positions, then get  $\mu\alpha.\sigma <: \mu\beta.\tau$  only if  $\mu\alpha.\sigma = \mu\beta.\tau$  for appropriate definition of type equality.

$$\frac{\Delta, \alpha <: \beta \vdash \sigma <: \tau}{\Delta \vdash \mu\alpha.\sigma <: \mu\beta.\tau} \text{ rec}$$

7

## Casts

Casting type to supertype (**upcast**) is always safe: just instance of subsumption rule.

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau} \text{ subsume} \qquad \frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash \text{Cast } e \text{ to } \tau : \tau} \text{ upcast}$$

“I don’t want to subclass my container classes for each class of object that I am storing”

Replies:

1. You must – creating more code so less reusability.
2. Upcast to store, **downcast** to retrieve:

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \text{Cast } e \text{ to } \tau : \tau} \text{ downcast}$$

No relation necessary between  $\sigma$  and  $\tau$ . If  $\tau <: \sigma$  and there is some other way to prove that type of  $e <: \tau$  then this works.

Java, C# package  $e$  with a representation of its type so can put in runtime safety checks.

3. Combine subtype polymorphism with parametric polymorphism: Java 5, C# 2.

8

## Real-world problems: equality, cloning...

“equality is symmetric”

```

public class Point {
    private final int x;
    private final int y;
    public Point (int x, int y) {
        this.x = x; this.y = y;}
    public boolean equals (Object o) {
        if (! (o instanceof Point))
            return false;
        Point p = (Point) o;
        return p.x == x && p.y == y;}
}

public class ColorPoint extends Point {
    private Color color;
    public ColorPoint (int x, int y, Color color) {
        super(x,y);
        this.color = color;
    }
    ...
}

```

What should equals look like for ColorPoint?

9

## Equality and inheritance

(1) Inherit equality from Point

```

ColorPoint c1 = new ColorPoint(1,2,Color.RED);
ColorPoint c2 = new ColorPoint(1,2,Color.BLUE);
c1.equals c2 // returns true
c2.equals c1 // returns true

```

equals is reflexive, symmetric, transitive, but not what we think of as equality

(2) Try in ColorPoint

```

public boolean equals (Object o) {
    if (! (o instanceof ColorPoint))
        return false;
    ColorPoint p = (ColorPoint) o;
    return super.equals(o) && p.color.equals(color);}
// Then
Point p = new Point(1,2);
Colorpoint cp = new Colorpoint(1,2,Color.RED)
p.equals cp // returns true
cp.equals p // returns false

```

equals is not symmetric.

10

## Equality and Inheritance

(3) make ColorPoint ignore color if not given ColorPoint

```

In ColorPoint:
... if (! (o instanceof ColorPoint))
    return super.equals(o); ...
// Then
Point p = new Point(1,2);
ColorPoint cp = new Colorpoint(1,2,Color.RED)
ColorPoint cq = new Colorpoint(1,2,Color.GREEN)
p.equals cp // returns true
cp.equals p // returns true
p.equals cq // returns true
cq.equals p // returns true
cp.equals cq // returnd false

```

Symmetric but not transitive.

(4) In Point add

```

... if (o == null || o.getClass().equals(getClass()))
    return false; ...

```

keeps properties of equality: no instance of a subclass of Point can be equal to an instance of Point.

11

## Combining Polymorphism

Combining subtype polymorphism with parametric polymorphism.

(o) Change parametric polymorphism format from

$\Lambda t. \dots$  to  $\Lambda t <: \tau. \dots$

“bounded polymorphism” if  $\tau$  is a constant.

“F-bounded polymorphism” if  $\tau$  can be (recursively) defined in terms of other bounds.

(o) Need  $\top$  to regain pure parametric polymorphism.

(C# almost has this: base types are automatically upcast, but need to be manually downcast. Java 5 claims to pre 5 doesn't have this: base types (such as int) are not in subtype hierarchy. Java 5 claims to have automatic casts to / from base types.)

Can add subtyping rules, keeping type variables and subtyping relations in the kind environment  $\Delta$ .

(o) New rule for  $\forall$  subtypes:

$$\frac{\Delta, t <: \rho \vdash \sigma <: \tau}{\Delta \vdash \forall t <: \rho. \sigma <: \forall t <: \rho. \tau}$$

(o) Subtyping in type application

$$\frac{\Delta, \Gamma \vdash e : \forall t <: \rho. \sigma \quad \Delta \vdash \tau <: \rho}{\Delta, \Gamma \vdash e[\tau] : \sigma[\tau/t]}$$

(type application (tuScheme @) and substitution both use square brackets).

12

## Generics

Word **generic** is overloaded:

Ada, C++ (template): code that is specialized for a parameter. Potentially many copies.

ML and friends: parametric polymorphism. 1 copy of code.

Java 5 uses 1 copy of code.

C# 2.0 uses 1 copy of code for all object (pointer) types, one copy per base type if needed. Copies made on fly (Just in time compilation).

C#, Java 5 notation:

$\langle T \rangle$  for  $\wedge T$ ...

$\langle T \rangle$  for  $\forall T <: \text{Object}$ ...

$T$  for  $\dots[T]$

Turns out to be almost comprehensible with nominative types.

13

## Example

```

class Stack {
    void push(Object o) {...}
    Object pop() {...}
    ...
}

// stack of String
Stack st = new Stack();
st.push("Hello");
...
st.push(new Object());
...
s = (String)st.pop(); //runtime exception

```

versus

```

class Stack<A> {
    void push(<A> o) {...}
    <A> pop() {...}
    ...
}

// stack of String
Stack<String> st = new Stack<String>();
st.push("Hello");
...
st.push(new Object()); //compile-time error
...
s = st.pop(); // no cast needed

```

14

## Gotcha

If  $\text{Foo} <: \text{Bar}$  is  $\text{Stack}\langle \text{Foo} \rangle <: \text{Stack}\langle \text{Bar} \rangle$ ?

$\text{Stack} = \forall \alpha <: \text{Object}. \{ \text{push} : \alpha \rightarrow \text{void}, \text{pop} : \text{void} \rightarrow \alpha \}$

pop would require  $\text{Foo} <: \text{Bar}$

push would require  $\text{Bar} <: \text{Foo}$

For inheritance (reuse) still useful.

C# parameterize classes, interfaces, individual methods, delegates... Parameterised class can inherit from / implement another class / interface with same type variable name.

Java parameterize classes, interfaces, individual methods. Parameterised class can inherit from / implement another class / interface with same type variable name.

15

## Example (from Buechi) Bounded Polymorphism

```

interface Priority {
    int getPriority();
}
class A implements Priority {
    public int getPriority() {...}
}
class PriorityQueue<E> implements Priority {
    E queue[];
    void insert(E e) {
        ...
        if(e.getPriority() < queue[i].getPriority()) {...}
        ...
    }
}
PriorityQueue<A> p = new PriorityQueue<A>();
p.insert(new A());

```

$\forall E <: \text{Priority}$ ...

16

## Example (from Buechi) F-Bounded Polymorphism

```
interface Comparable<I> {
    boolean lessThan(I o);
}
class A implements Comparable<A> {
    public boolean lessThan(A o) {...}
}
class SortedList<E implements Comparable<E>> {
    E list[];
    void insert(E e) {
        ...
        if(e.lessThan(list[i])) {...}
        ...
    }
}
```

Remember the implicit first parameter?

class A written to only compare with other As.

## Example (from Buechi) Mix-ins

Mix-ins extend the class parameterized on

```
interface LessAndEqual<I> {
    boolean lessThan(I o);
    boolean equal(I o);
}
class A implements LessAndEqual {...}
class Relations<C implements LessAndEqual<C>>
    extends C {
    boolean lessThanEqual(Relations<C> a) {
        return lessThan(a) || equal(a);
    }
}
```

```
Relations<A> x = new Relations<A>();
Relations<A> y = new Relations<A>();
if(x.lessThanEqual(y)) {...}
```

Extending the class of the parameter not allowed in current C# proposal. Difficult for separate compilation: Don't know class C.

17

18

## Generic Supertypes

Some use for  $\forall \alpha :>$  bound as well as  $\forall \alpha <:$  bound

Example from Bracha's tutorial:

java.util.TreeSet<E> ordered set. Constructor for TreeSet<E> takes a Comparator<E>.

```
interface Comparator<T> {
    int compare(T fst, T snd);
}
```

For TreeSet<String>, Comparator<String> would work, but so would Comparator<Object>

So constructor method has specification:

```
TreeSet(Comparator<T super E> c) { ... }
```

Another example: public static method

Collections.max

```
public static <T extends Comparable<T>>
    T max(Collection<T> coll) { ... }
```

only allows an comparable object to be compared with one of its own parameterized type.

```
public static <T extends Comparable<U super T>>
    T max(Collection<T> coll) { ... }
```

Allows comparison with superclasses (could get max of a Point and a ColorPoint if comparable).

19

## Java 5 wildcards

What can we do with type  $\exists \alpha <:$  bound.  $\tau$  ?

For any covariant use of  $\alpha$  in  $\tau$  we can take that use of  $\alpha$  and replace it with bound.

In Java 5, ? indicates this existentially-bound variable.

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

(New Java5 syntax for looping over elements of a collection)

As I understand it the cast to Object only affects class method lookup, not instance method lookup. (Is this correct?)

"extends Object" can be removed.

20

Another use of ? – replace any type variable only used once

```
public static <T extends Comparable<U super T>>
    T max(Collection<T> coll) { ... }
```

Officially-preferred notation:

```
public static <T extends Comparable<? super T>>
    T max(Collection<T> coll) { ... }
```

Is there a use, other than abbreviation for  $\exists \alpha :>$  bound.  $\tau$  ?

Classes where all methods return fixed types?

21

## Sources

Structural subtyping: various, Cardelli, “F-bounded polymorphism for object-oriented programming” by Peter Canning et alia, Proceedings FPCA ’89.

“Generics in the Java Programming Language” by Gilad Bracha <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>

“An Introduction to C# Generics” by Juval Lowy [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/csharp\\_generics.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/csharp_generics.asp)

“GJ: Extending the Java Programming Language with type parameters” Gilad Bracha et alia <http://lamp.epfl.ch/pizza/gj/Documentants/#gj-tutorial>

“Making the future safe for the past: Adding Genericity to the Java Programming Language” Gilad Bracha et alia, Proceedings OOPSLA ’98

“Design and Implementation of Generics for the .NET Common Language Runtime” by Andrew Kennedy and Don Syme <http://research.microsoft.com/clrgen/generics.pdf> Proceedings PLDI ’01.

“Generics in Java and Beyond” by Martin Buechi, PPT

23

Implementation of F-bounded polymorphism useful, but can be tricky. How does it interact with other language features?

Attempt to put into Java 5 without changing the JVM caused reflection and serialization to break. Compiler just changed parameters into the bound – reflection only has to deal with classes, not type functions from classes to classes. But source code is full of type functions from classes to classes: reflection can not make distinctions that are obvious in source code.

Other disadvantages to Java 5: compiler is inserting casts internally, knowing (except for interaction with legacy code) that the casts are safe if the program type-checks. Casts still take time in JVM – fixable. If program does not type check, get errors in odd places – harder to fix.

Attempt to put into C# took major enhancements to IL and rewrites to virtual machine.

Current disadvantage to C# generics: not yet released. Miller hints at  $:>$  but not guaranteed?

22

presentation [www.abo.fi/~mbuechi/download/JavaGenerics.ppt](http://www.abo.fi/~mbuechi/download/JavaGenerics.ppt)

“Inheritance Is Not Subtyping” William Cook et alia, ’90

“Adding Wildcards to the Java Programming Language” by Mads Torgensen et alia, Proceedings SAC ’04

“Parametric Polymorphism for Java: Is There Any Hope in Sight?” by Suad Alagić et alia.

24