Templates and Generics

Edward Z. Yang

Subtype versus parametric polymorphism ?

apply

f(Object x);

to any

C <: Object

apply

generic<T> f(T x)

to any

y : C

Last lecture, we discussed subtyping, and something you might observe is that there are some similarities between subtyping and parametric polymorphism. For example, if I write a function f(Object x), subtyping says that I can call this function with any class which is a subtype of Object (e.g. C). In a similar manner, if I write a function parametric in T, this means that I can specialize the function to any type (including C.)

subtype

parametric

However, subtyping is not a perfect replacement for parametric polymorphism. One of the distinguishing differences between subtyping and parametric polymorphism is "information loss." If I have a function identity which takes an Object and produces that an Object, when I pass it a Boolean, I lose information about what the original type of the object was: I only get an Object which most be unsafely downcast back to be Bool again. In constrast, with parametric polymorphism, when I unify the input argument a with Bool, the return argument also refines to a Bool.

Object id(Object);

Object r = id(true);

to get Boolean, must do checked downcast!

subtype

id :: $a \to a$

id True :: Bool

parametric

Parametric polymorphism motivation

Consequently, even in a language with sub typing, there is often still a good deal of demand for some sort of parametric polymorphism.

- Containers (Primary use-case!)

  Array<T>, Map<K,V>, Pair<A,B>

The most obvious use-case is containers, but there are a wide variety of other classes which can profitably use parametric polymorphism.

- Java uses:

  Reference<T>, WeakReference<T>, Callable<V>, Class<T>, ...

- C++ similar uses: also metaprogramming and specialization

In C++, template instantiation is not parametric (indifferent to the type in question) which opens the possibility for other types of fun features.

There are two big takeaways from lecture today. The first is that when you put subtyping and parametric polymorphism together, you get a lot of complexity. We'll be talking about these issues in this lecture.

Subtyping

\+ Parametric polymorphism

= Complexity

(in the type system!)

The second point is that there are two ways you can handle this complexity. In Java, generics were added as a full-fledged extension to the type system, with the attendant complexity. C++, however, supports generics simply by deferring type checking until after expansion. This is certainly sound, but has lead to C++'s famously bad error messages.

Complexity

C++ punts!

(type errors deferred until after specialization occurs)

Java Generics

(complicated type system extension w/ some warts)

C++ templates

As C++ templates are "simpler" from the perspective of compiler behavior, we'll take a look at them first.

Haskell
Polymorphic functions

```haskell
swap :: IORef a -> IORef a -> IO ()
swap rx ry = do
    x <- readIORef rx
    y <- readIORef ry
    writeIORef y ry
```

C++
Function Templates

```cpp
template <typename T>
void swap(T& x, T& y) {
    T tmp = x;
    x = y;
    y = tmp;
}
```

very different compiled result!

Haskell Polymorphic functions

C++ Function Templates

There are a lot of similarities between Haskell's polymorphic functions and function templates, as this side-by-side comparison shows.

optional

```
swap :: IORef a -> IORef a -> IO ()
swap rx ry = do
    x <- readIORef rx
    y <- readIORef ry
    writeIORef y ry
```

mandatory

```
template <typename T>
void swap(T& x, T& y) {
    T tmp = x;
    x = y;
    y = tmp;
}
```

Some major differences: Haskell's type system allows parameters to be inferred; in C++ they must always be explicitly stated. Additionally, C++ will generate code for each instantiation of the template; Haskell operates on a boxed representation and only needs to be compiled once.

very different compiled result!

one function operating on heap

seperate code for each type, temporary is arbitrary size

Implicit constraints on type parameter

A big thing about C++ templates is that there are implicit constraints on what types can be validly used to instantiate a template, based on how the type is used in the body of the template.

pointer to an array

```
template <typename T>
void sort(int count, T (&a)[]) {
    for(int i=0; i<count-1; i++) {
        for (int j=i+1; j<count; j++) {
            if (A[j]<A[i]) swap(A[i],A[j]);
        }
    }
}
```

meaning of comparison on T

size of pointer arith. on T

In this particular example, T needs to support comparison and "swappability" (what exactly swappable means depends on the definition of swap, which itself is templated.)

Specialization

(overlapping instances)

The fact that C++ is non-parametric means that some interesting performance tricks can be done with templates. For example, we can define a generic template over all types T with an inefficient implementation of some algorithm, but then for specific types specify a different implementation, which makes use of the extra information to do better.

```
template<class T>
void swap(T&x, T&y) {
    T tmp = x;      ← always makes full copy of T
    x = y;
    y = tmp;
}
```

can pick most specialized applicable template

```
template<class T>
void swap(vector<T>&, vector<T>&) {
    // move pointers in vector header
}
```

Specialization (2)

Specialization in C++ is actually quite flexible, and bears some similarities to Haskell type classes. For example, if we have a templated class for sets; we can fully specialize on characters (the resulting instance having no leftover parameters), or we can partially specialize, given an implementation of Set for pointers of type T, for any choice of T

```
template <typename T> class Set {
    // binary tree
}
```

fully specialized

```
template <> class Set<char> {
    // bit-vector
}
```

partially specialized

```
template <typename T> class Set<T*> {
    // hash-table
}
```

C++ templates

- compile time instantiation
- overloading done <u>after</u> substitution

} like macros

- type directed (like type classes)
  picking <u>best match</u> (when specialized)

- very limited "separate compilation"

Standard Template Library

- Provides polymorphic abstract types and operations
- Runtime efficiency! (maybe not code space)
- Does not rely on objects

e.g. sort

Container
Iterator
Algorithm

Adapter
Function object
Allocator

An example function from C++

merge two sorted lists

$$\underbrace{\mathrm{range}(s) \times \mathrm{range}(t)}_{\text{ordered lists of elements of type } s \text{ and } t} \times \underbrace{\mathrm{comparison}(u)}_{\text{boolean valued function on } u} \to \mathrm{range}(u)$$

where $s <: u$ and $t <: u$

```
template<class InIter1, class InIter2,
         class OutIter, class Compare>
OutIter merge(InIter1 fst1, InIter1 last1,
              InIter2 fst2, InIter2 last2,
              Compare comp)
```

C++ template metaprogramming

Instantiate, then typecheck

→ maximal typing flexibility

→ type/template/non-type parameters
integer

The ability to declare specializations means that

Templates are Turing Complete

(C++ concepts?)

```cpp
template<int N> struct Factorial {
    enum{ value = Factorial<N-1>::value * N};
};
template<> struct Factorial<0>{
    enum{ value = 1 };
};

int main(){
    char array[Factorial<4>::value];
    std::cout << sizeof(array);
}
```

```
template <typename T,
          typename LockingPolicy,
          typename RangePolicy>
class Vector : public RangePolicy,
               public LockingPolicy;

T& Vector<T,RangePolicy>::operator[](size_t i) {
    LockingPolicy::Lock lock;
    RangePolicy::CheckRange(i, this->size);
    return this->elems[i];
}
```

mix-ins

Note that the template parametrizes over the parent of the class we want to define

Java Generics

The container problem

The bad old days (Java 1.0)

```
class Stack {
    void push(Object o) {...}
    Object() pop() {...}
}
String s = "Hello";
Stack st = new Stack();
st.push(s)
...
s = (String) st.pop();
```

With generics

```
class Stack<T> {
    void push(    T o) {...}
        T() pop() {...}
}
String s = "Hello";
Stack<String> st = new Stack<String>();
st.push(s)
...
s =             st.pop();
```

no cast needed

An obvious looking thing...

... the details are work!

(Many proposals, backwards compat concerns...)

Java generics are typechecked

```
class PriorityQueue<T> {
    void push(T x) {
        ...
        if (x.less(y)) {...}
        ...
    }
}
```

type error! (T might not support T)

Compare C++: compile it and see if the result typechecks!

Basic generics on a slide

type parameter(s)

```
class Stack<T> {
    void push(T o) {...}
    T() pop() {...}
}
String s = "Hello";
Stack<String> st = new Stack<String>();
st.push(s)
...
s = st.pop();
```

reference type parameters in body

Sometimes, Java can infer this! <>

occurrences of class must "invoke" the generic w/ type(s)

Variance redux

Accepted:

```
class A {}
class B extends A {}
B[] bArray = new B[10];
A[] aArray = bArray;
aArray[0] = new A;
```

$B[] <: A[]$

Variance redux

Java generics are invariant (neither covariant nor covariant)

Rejected!

```
class A {}
class B extends A {}
List<B> bArray = new ArrayList<B>;
List<A> aArray = bArray;
aArray[0] = new A;
```

$ArrayList\langle B\rangle <: List\langle B\rangle$

$List\langle B\rangle \not<: List\langle A\rangle$

Wildcards

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k=0; k< c.size; k++) {
        System.out.println(i.next());
    }
}
```

old

Wildcards

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

Collection<Foo> a;
printCollection(a);
```

Wildcards

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

```
Collection<Foo> a;
printCollection(a);
```

✗

newer... better?

Wildcards

matches any type

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

only allowed because Object is supertype of all types.

```
Collection<Foo> a;
printCollection(a); ✓
```

Using wildcards

P.S. Without Wildcards

generic method

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

Collection<Foo> a;
<Foo>printCollection(a);
```

have to explicitly specify type

Bounded Wildcards

```
void
printCollection(Collection<? extends Showable> c) {
    for (Showable e : c) {
        e.show();
    }
}
```

Also: <? super Subtype>

matching all <u>supertypes</u> of Subtype

(Why is this useful?)

Wildcards serve as use-site variance

invariant

```
class Ref<A> {
  A get() {...}
  void put(A x) {...}
}
```

covariant

```
class Source<A> {
  A get() {...}
}
```

contravariant

```
class Sink<A> {
  void put(A x) {...}
}
```

Wildcards serve as use-site variance

```
Ref<? extends B> source;

source = new Ref<C>();
B a = source.get();
source.put(new B());
```

effectively, only
Source is usable

```
class Ref<A> {
    A get() {...}
    void put(A x) {...}
}
```

```
class A {}
class B extends A {}
class C extends B {}
```

Wildcards serve as use-site variance

```
Ref<? super B> sink;

source = new Ref<A>();
B a = source.get();
source.put(new B());
```

only Sink is usable (well, you can get an Object from get())

```
class Ref<A> {
    A get() {...}
    void put(A x) {...}
}

class A {}
class B extends A {}
class C extends B {}
```

But wait! There's more...

Subtyping

$+$ Parametric polymorphism

$=$ Complexity

Polymorphism <u>with</u> subtyping

Parametric polymorphism

$\mathrm{max} :: \forall t.\ (t \to t \to \mathrm{Bool}) \to t \to t \to t$

for every type t, given a less than function, ...

Bounded polymorphism

$\mathrm{printString} :: \forall t <: \mathrm{Showable}.\ t \to \mathrm{String}$

for every subtype t of Showable, ...

F-bounded polymorphism

$\mathrm{max} :: \forall t <: \mathrm{Comparable}(t).\ t \to t \to t$

for every subtype t of Comparable(t), ...

(pardon the Haskell)

Contravariance redux

```
interface Comparable {
    int compareTo(Comparable);
}

class Foo implements Comparable {
    int compareTo(Foo x) { ... }
}
```

Illegal!

Foo <: Comparable
but in contravariant position

Contravariance redux

```
interface Comparable<T> {
    int compareTo(T);
}
class Foo implements Comparable<Foo> {
    int compareTo(Foo x) {...}
}
```

Contravariance redux

```
interface A { public int compareTo(A);
              int foo(); ... }
```

<:

```
interface Comparable<A>
          { public int compareTo(A); }
```

```
interface Comparable<T> {
    int compareTo(T);
}
```

forall T, s.t. T is a subtype of Comparable<T>

```
public static <T extends Comparable<T>>
                T max(Collection<T> coll) {
    T cand = coll.iterator().next();
    for (T elt : coll) {
        if (cand.compareTo(elt) < 0)
            cand = elt;
    }
    return elt;
}
```

(EX)

```
abstract class Enum<E extends Enum<E>>
```

E must be a subtype of Enum<E>

e.g.

```
A extends Enum<A>
```

(improves type safety inside Enum class)

```
int ordinal();
int compareTo(E x) { ... }
```

Aside: metatheoretic difficulties

Recall: H-M type inference decidable (supporting parametric polymorphism)

[Tate-Amin'16] Java's type system is unsound

[Grigore'16] Type checking with Java generics is undecidable.

```java
class Unsound {
  static class Constrain<A, B extends A> {}
  static class Bind<A> {
    <B extends A>
    A upcast(Constrain<A,B> constrain, B b) {
      return b;
    }
  }
  static <T,U> U coerce(T t) {
    Constrain<U,? super T> constrain = null;
    Bind<U> bind = new Bind<U>();
    return bind.upcast(constrain, t);
  }
  public static void main(String[] args) {
    String zero = Unsound.<Integer,String>coerce(0);
  }
}
```

[Tate-Amin'16]

decidability is overrated!

```
public static interface List<T> {};
public static class C<P>
    implements List<List<? super C<C<P>>>> {}
public void foo(C<Byte> x) {
    List<? super C<Byte>> y = x;
}
```

Superclass gets bigger; infinite proof!

javac stack overflows!

is C<Byte> a subtype of List<? super C<Byte>>

[Kennedy-Pierce '07]

[Tate-Leung-Lerner '11]

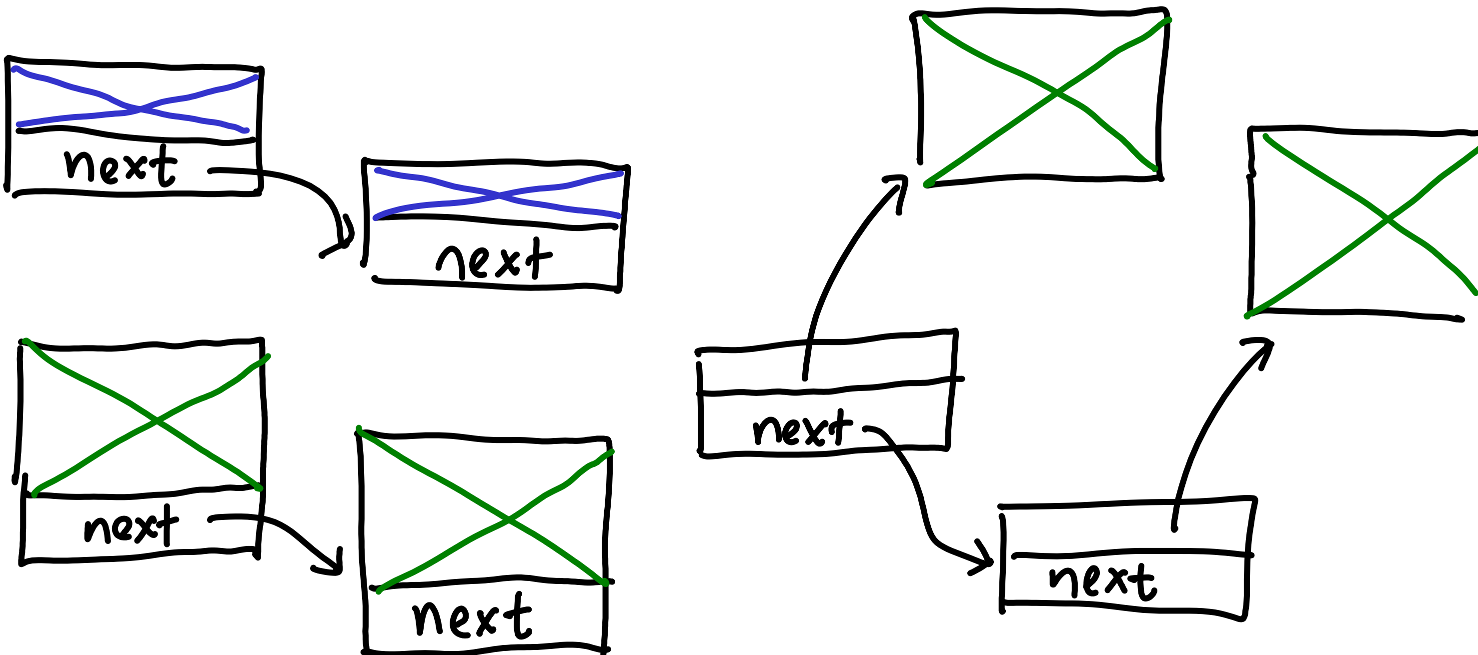
Implementation and all that

Type erasure

(Generics are not templates)

missed opportunity?

Heterogenous vs. Homogenous



Erasure!

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

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

replace parameters w/ Object, insert casts
if <A extends B>, replace with B

primary concern: backwards compatibility

Erasure!

semantics $\longleftarrow$ impl

static variables

```
class G<T> {
    static public int x;
}

G<Int>::x = 2;
G<Bool>::x = 3;
```

only one VM-time class; shared static variable

Erasure!

semantics ← impl

constructors

```
class G<T> {
    void f() {
        T x = new T()
    }
}
```

T erased, no way to resolve constructor at VM time

Erasure!

semantics ← impl

overloading

```
public void f(Collection<A>) { ... }
public void f(Collection<B>) { ... }
```

erased in Java; no way to resolve overload at VM-time

Conclusion

	C++ templates	Java generics
parametrization	classes & functions	classes & methods
flexibility	compile time instantiation, then type checking/resolution	seperate compilation w/ type constraints
specialization	template and partial Specialization	none
non-type params	integer parameters	none
mixins	can inherit from type parameter	none