G22.2110-003 Programming Languages - Fall 2012

Week 14 - Part 1

Thomas Wies

New York University

Review

Last lecture

- Exceptions

Outline

Today:

- Generic Programming

Sources for today's lecture:

- PLP, ch. 8.4
- Programming in Scala, ch. 19, 20.6

Generic programming

Subroutines provide a way to abstract over *values*.

Generic programming lets us abstract over *types*.

Examples:

- A sorting algorithm has the same structure, regardless of the types being sorted
- Stack primitives have the same semantics, regardless of the objects stored on the stack.

One common use:

- algorithms on containers: updating, iteration, search

Language models:

- C: macros (textual substitution) or unsafe casts
- Ada: generic units and instantiations
- C++, Java, C#, Scala: generics (also called templates)
- ML: parametric polymorphism, functors

Parameterizing software components

Construct	Parameter(s):
array	bounds, element type
subprogram	values (arguments)
Ada generic package	values, types, packages
Ada generic subprogram	values, types
C++ class template	values, types
C++ function template	values, types
Java generic	classes
Scala generic	types (and implicit values)
ML function	values (including other functions)
ML type constructor	types
ML functor	values, types, structures

Templates in C++

```
template <typename T>
class Array {
public:
  explicit Array (size_t);  // constructor
  T& operator[] (size_t);   // subscript operator
  ...  // other operations
private:
  ...  // a size and a pointer to an array
};

Array<int> V1(100);        // instantiation
Array<int> V2;             // use default constructor

typedef Array<employee> Dept; // named instance
```

Type and value parameters

```
template <typename T, unsigned int i>
class Buffer {
  T v[i];               // storage for buffer
  unsigned int sz;      // total capacity
  unsigned int count;   // current contents
public:
  Buffer () : sz(i), count(0) { }
  T read ();
  void write (const T& elem);
};

Buffer<Shape *, 100> picture;
```

Template Does not Guarantee Success

```
template <typename T>
class List {
  struct Link { // for a list node
    Link *pre, *succ;  // doubly linked
    T val;
    Link (Link *p, Link *s, const T& v)
      : pre(p), succ(s), val(v) { }
  };
  Link *head;
public:
  void print (std::ostream& os) {
    for (Link *p = head; p; p = p->succ)
      // will fail if operator<< does
      // not exist for T
      os << p->val << "\n";
  }
};
```

Function templates

Instantiated implicitly at point of call:

```
template <typename T>
void sort (vector<T>&) { ... }

void testit (vector<int>& vi) {
  sort(vi);  // implicit instantiation
             // can also write sort<int>(vi);
}
```

Functions and function templates

Templates and regular functions overload each other:

```
template <typename T> class Complex {...};

template <typename T> T sqrt (T); // template
template <typename T> Complex<T> sqrt (Complex<T>);
                               // different algorithm
double sqrt (double);  // regular function

void testit (Complex<double> cd) {
  sqrt(2);    // sqrt<int>
  sqrt(2.0); // sqrt (double): regular function
  sqrt(cd);  // sqrt<Complex<double> >
}
```

Iterators and containers

- Containers are data structures to manage collections of items
- Typical operations: insert, delete, search, count
- Typical algorithms over collections use:
  - imperative languages: iterators
  - functional languages: map, fold

```
interface Iterator<E> {
  boolean hasNext (); // returns true if there are
                      // more elements
  E next ();          // returns the next element
  void remove ();     // removes the current element
                      // from the collection
};
```

The Standard Template Library

The *Standard Template Library (STL)* is a set of useful data structures and algorithms in C++, mostly to handle collections.

- *Sequential containers*: `list`, `vector`, `deque`
- *Associative containers*: `set`, `map`

We can *iterate* over these using (what else?) *iterators*.
Iterators provided (for `vector<T>`):

```
vector<T>::iterator
vector<T>::const_iterator
vector<T>::reverse_iterator
vector<T>::const_reverse_iterator
```

Note: Almost no inheritance used in STL.

Iterators in C++

For standard collection classes, we have member functions `begin` and `end` that return iterators.

We can do the following with an iterator `p` (subject to restrictions):

`*p`	"Dereference" it to get the element it points to
`++p, p++`	Advance it to point to the next element
`--p, p--`	Retreat it to point to the previous element
`p+i`	Advance it `i` times
`p-i`	Retreat it `i` times

A sequence is defined by a pair of iterators:

- the first points to the first element in the sequence
- the second points to *one past* the last element in the sequence

There are a variety of operations that work on sequences.

Iterator example 1

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
  vector<int> v;
  for (int i = 0; i < 10; ++i) v.push_back(i);
  // Print list
  vector<int>::iterator it;
  for (it = v.begin(); it != v.end(); ++it) {
    cout << *it << "␣";
  }
  cout << endl << endl;
  // Use reverse iterator to print in reverse order
  vector<int>::reverse_iterator rit;
  for (rit = v.rbegin(); rit != v.rend(); ++rit) {
    cout << *rit << "␣";
  }
  cout << endl;
}
```

Iterator example 2

```
#include <vector>
#include <string>
#include <iostream>

using namespace std;

int main () {
  vector<string> ss(20); // initialize to 20 empty strings
  for (int i = 0; i < 20; i++)
    ss[i] = string(1, ’a’+i); // assign "a", "b", etc.
  vector<string>::iterator loc =
    find(ss.begin(), ss.end(), "d"); // find first "d"
  cout << "found:␣" << *loc
       << "␣at␣position␣" << loc - ss.begin()
       << endl;
}
```

STL algorithms, part 1

STL provides a wide variety of standard *algorithms* on sequences.

Example: finding an element that matches a given condition

```
// Find first 7 in the sequence
list<int>::iterator p = find(c.begin(), c.end(), 7);
```

```
#include <algorithm>

// Find first number less than 7 in the sequence
bool less_than_7 (int v) {
   return v < 7;
}

list<int>::iterator p = find_if(c.begin(), c.end(),
                                less_than_7);
```

STL algorithms, part 2

Example: doing something for each element of a sequence

It is often useful to pass a function or *something that acts like a function*:

```
#include <iostream>
#include <algorithm>

template <typename T>
class Sum {
   T res;
public:
   Sum (T i = 0) : res(i) { }                 // initialize
   void operator() (T x) { res += x; }  // accumulate
   T result () const { return res; }     // return sum
};

void f (list<double>& ds) {
   Sum<double> sum;
   sum = for_each(ds.begin(), ds.end(), sum);
   cout << "the␣sum␣is␣" << sum.result() << "\n";
}
```

C++ templates are Turing complete

Templates in C++ allow for arbitrary computation to be done *at compile time!*

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

template <>
struct Factorial<1> {
   enum { V = 1 };
};

void f () {
   const int fact12 = Factorial<12>::V;
   cout << fact12 << endl; // 479001600
}
```

Generics in Java

Only class parameters

Implementation by *type erasure*: all instances share the same code

```
interface Collection <E> {
  public void add (E x);
  public Iterator<E> iterator ();
}
```

`Collection <Thing>` is a parametrized type

`Collection` (by itself) is a raw type!

Generic methods in Java

```
class Collection <A extends Comparable<A>> {
  public A max () {
    Iterator<A> xi = this.iterator();
    A biggest = xi.next();
    while (xi.hasNext()) {
      A x = xi.next();
      if (biggest.compareTo(x) < 0)
        biggest = x;
    }
    return biggest;
  }
  ...
}
```

Generic Programming in Scala

Scala supports two orthogonal generic programming concepts:

Type parameters

- allowed in traits, classes, objects, and methods
- implementation by type erasure like in Java
- no notion of raw types, generic classes are only *type constructors*
- issues related to subtype polymorphism
  - variance annotations
  - lower and upper bounds, view bounds
- can simulate type classes with context bounds

Abstract types

- traits and classes can have types as members
- like other members, types can be abstract

Generic classes in Scala

A simple generic functional queue implementation:

```
class Queue[T] private (private val elems: List[T]) {
  def enqueue(x: T) = new Queue(x :: elems)
  def dequeue() =
    (elems.last, new Queue(elems dropRight 1))
}

object Queue {
  def apply[T](xs: T*) = new Queue(xs.toList.reverse)
}
```

```
scala> val intQueue = Queue(1,2,3)
q: Queue[Int] = Queue@58804a77
scala> q.dequeue
res0: Int = 1
```

Generic classes and subtyping

Consider a generic class

```
class C[T] { ... }
```

If `S` is a subtype of `U` (denoted `S <: U`), what does this mean for the types `C[S]` and `C[U]`?

Is it safe to use values of type `C[S]` in place of values of type `C[U]` or vice versa?

- if `C[S] <: C[U]`, then `C` is said to be *covariant* in `T`
- if `C[U] <: C[S]`, then `C` is said to be *contravariant* in `T`
- otherwise `C` is said to be *invariant* in `T`.

Variance annotations

Unlike Java, Scala allows the programmer to specify the variance of type parameters.

- `class C[+T] { ... }` specifies that `C` is covariant in `T`
- `class C[-T] { ... }` specifies that `C` is contravariant in `T`
- `class C[T] { ... }` specifies that `C` is invariant in `T`

The correctness of these *variance annotations* is checked by the compiler.

In Java, generic classes are always invariant (with the exception of `Array`, which is covariant in the element type).

This restriction in Java can be alleviated using raw types but their correct usage can only be checked at run time.

When is covariance safe?

`C` is covariant in a type parameter `T` means that a value of type `C[S]` is usable as a `C[U]` if each values of type `S` is useable as a `U`.

This is not always possible:

```
class Cell[T](init: T) {
  private[this] var current = init
  def get = current
  def set(x: T) { current = x }
}
```

Suppose `Cell` was covariant in `T`. Then we could do the following:

```
val c1 = new Cell[String]("abc")
val c2: Cell[Any] = c1 // OK, because Cell is covariant
c2.set(1)              // OK, because Int <: Any
val s: String = c1.get // Bzzzt! - c1 now stores an Int
```

Java arrays revisited

An array is essentially an indexed sequence of cells.

Java's arrays are covariant in their element type. This is unsafe:

```
class A { ... }
class B extends A { ... }

B[] b = new B[5];
A[] a = b; // allowed (a and b are now aliases)

a[1] = new A(); // Bzzzt! (ArrayStoreException)
```

Therefore, the JVM has to check the correctness of array stores at run time, which is expensive.

In Scala, arrays are invariant in their element type.

Checked Variance Annotations

The SCALA type checker ensures the safety of all variance annotations.

This gives stronger correctness guarantees at compile time and avoids expensive run-time checks.

In particular, a covariant `Cell` class will be rejected by the compiler:

```
class Cell[+T](init: T) {
  private[this] var current = init
  def get = current
  def set(x: T) { current = x }
}
```

error: covariant type T occurs in contravariant position in type T of value x

Covariant queues

What about `Queue`?

Is it covariant in its type parameter?

```
class Queue[+T] private (private val elems: List[T]) {
  def enqueue(x: T) = new Queue(x :: elems)
  def dequeue() =
    (elems.last, new Queue(elems dropRight 1))
}
```

Covariant queues

What about `Queue`?

Is it covariant in its type parameter?

```
class Queue[+T] private (private val elems: List[T]) {
  def enqueue(x: T) = new Queue(x :: elems)
  def dequeue() =
    (elems.last, new Queue(elems dropRight 1))
}
```

It seems like this should be OK, since there is no mutable state.

A hypothetical counterexample

```
class StrangeQueue extends Queue[Int] {
  override def enqueue(x: Int) {
    println(math.sqrt(x))
    super.enqueue(x)
  }
}

val x: Queue[Any] = new StrangeQueue
  // OK, because StrangeQueue <: Queue[Int] <: Queue[Any]
x.enqueue("abc") // Bzzzt! - Int expected
```

The compiler will reject the covariance annotation in `Queue`:

```
scala> class Queue[+T] (...) { def enqueue(x: T) = ... }
```

error: covariant type T occurs in contravariant position in type T of value x

Lower bounds

Method `enqueue` is safe, as long as the given value is of a supertype `U` of type parameter `T`.

We can encode this using a *lower bound* constraint.

```
class Queue[+T] private (private val elems: List[T]) {
  def enqueue[U >:  T](x: U) = new Queue[U](x :: elems)
  ...
}
```

Now we can use `Queue` covariantly and the compiler will reject the class `StrangeQueue`.

Contravariance

Contravariance annotations are useful for type parameters that only occur in contravariant positions:

```
trait OutputChannel[-T] {
  def write(x: T)
}
```

It is safe to substitute an `OutputChannel[AnyRef]` for an `OutputChannel[String]`.

Co- and contravariance annotations may also be used in combination:

```
trait Function1[-S, +T] {
  def apply(x: S): T
}
```

The `Ordered` trait

SCALA provides a trait for representing ordered types:

```
trait Ordered[T] extends java.lang.Comparable[T] {
  abstract def compare(that: T): Int
  def <(that: T) = (this compare that) < 0
  def >(that: T) = (this compare that) > 0
  def <=(that: T) = (this compare that) <= 0
  def >=(that: T) = (this compare that) >= 0
  ...
}
```

The Ordered trait

Ordered can be mixed into other classes to enable convenient comparison of values:

```
class Person(val surName: String, val lastName: String)
extends Ordered[Person] {
  def compare(that: Person) =
    (lastName + surName) compareToIgnoreCase
      (that.lastName + that.surName)
  override def toString = surName + "␣" + lastName
}
```

```
scala> val robert = new Person("Robert", "Jones")
robert: Person = Robert Jones
scala> val sally = new Person("Sally", "Smith")
sally: Person = Sally Smith
scala> robert < sally
res0: Boolean = true
```

Upper bounds

We can use *upper bounds* to constrain type parameters.

```
def mergeSort[T <:  Ordered[T]](xs: List[T]): List[T] = {
  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if (x < y) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }
  val n = xs.length / 2
  if (n == 0) xs else {
    val (ys, zs) = xs splitAt n
    merge(mergeSort(ys), mergeSort(zs))
  }
}
```

Limitations of upper bounds

Upper bounds can be quite restrictive:

```
scala> mergeSort(List(3,1,2))
error: inferred type arguments [Int] do
  not conform to method mergeSort's type
    parameter bounds [T <: Ordered[T]]
```

The type `Int` does not extend `Ordered[Int]`, but we can convert an `Int` to an `Ordered[Int]`.

View bounds

Define a view that implicitly converts `Int` to `Ordered[Int]`

```
implicit def int2ordered(x: Int): Ordered[Int] =
   new Ordered[Int] {
     override def compare(that: Int) =
       if (x < that) -1 else if (x == that) 0 else -1
   }
```

and replace the upper bound in `mergeSort` by a *view bound*

```
def mergeSort[T <% Ordered[T]](xs: List[T]): List[T] = ...
```

The view bound specifies that `T` can be viewed as an `Ordered[T]`.

```
scala> mergeSort(List(3,1,2))
res0: List[Int] = List(1,2,3)
```

The `Ordering` trait

What if we have more than one ordering on a type `T`?

Scala's API provides a trait `Ordering`.

An object of type `Ordering[T]` defines one strategy of ordering `T`.

For many basic types of Scala orderings are already implicitly defined.

```
trait IntOrdering extends Ordering[Int] {
  override def compare(x: Int, y: Int) =
    if (x < y) -1
    else if (x == y) 0
    else 1
}
implicit object Int extends IntOrdering
```

Context bounds

We can use a *context bound* to express that the type parameter `T` of `mergeSort` has an associated implicit object of type `Ordering[T]`

```
def mergeSort[T : Ordering](xs: List[T]): List[T] = {
  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if (implicitly[Ordering[T]].lt(x, y))
          x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }
  val n = xs.length / 2
  if (n == 0) xs else {
    val (ys, zs) = xs splitAt n
    merge(mergeSort(ys), mergeSort(zs))
  }
}
```

Abstract types

Disadvantage of type parameters

- Parameterization over many types tends to lead to an explosion of bound parameters for encoding variances.
- Also, type parameters cannot be partially instantiated.

Alternative to type parameters

- Scala allows types as members of classes and traits.
- Type members can also be abstract.
- *Abstract types* are useful for encoding complex variance constraints.

Cows don't eat fish

```scala
class Food
abstract class Animal {
  def eat(food: Food)
}
class Grass extends Food
class Cow extends Animal {
  override def eat(food: Grass) {} // this won't compile,
                                   // but if it did, ...
}
class Fish extends Food
val bessy: Animal = new Cow
bessy eat (new Fish) // ... you could feed fish to cows
```

Abstract types in action

```
class Food
abstract class Animal {
  type SuitableFood <: Food
  def eat(food: SuitableFood)
}
class Grass extends Food
class Fish extends Food
class Cow extends Animal {
  type SuitableFood = Grass
  override def eat(food: Grass) {}
}
```

```
scala> val bessy: Animal = new Cow
bessy: Animal = Cow@2e3919
scala> bessy eat (new Fish)
error: type mismatch;
 found   : Fish
 required: bessy.SuitableFood
```

Functors in ML

Why functors, when we have parametric polymorphic functions and type constructors (e.g. containers)?

- Functors can take *structures* as arguments. This is not possible with functions or type constructors.
- Sometimes a type needs to be parameterized on a *value*. This is not possible with type constructors.

Priority queues revisited

```
datatype order = LESS | EQUAL | GREATER

signature PRIORITY_QUEUE =
sig
  type 'a prio_queue
  exception EmptyQueue
  val empty : ('a * 'a -> order) -> 'a prio_queue
  val isEmpty : 'a prio_queue -> bool
  val insert : 'a * 'a prio_queue -> 'a prio_queue
  val min : 'a prio_queue -> 'a option
  val delMin : 'a prio_queue -> 'a prio_queue
end
```

Problem:

Dependence of type `'a queue` on the ordering on `'a` is not made explicit.

Modified priority queue signature

First step: make element type part of the signature

```
signature PRIORITY_QUEUE =
sig
  type elem
  type prio_queue
  exception EmptyQueue
  val empty : prio_queue
  val isEmpty : prio_queue -> bool
  val insert : elem * prio_queue -> prio_queue
  val min : prio_queue -> elem option
  val delMin : prio_queue -> prio_queue
end
```

PriorityQueue functor

Second step: abstract over element type and compare function

```
functor PriorityQueue(type elem
                      val compare : elem * elem -> order)
  :> PRIORITY_QUEUE where type elem = elem =
struct
  type elem = elem
  type prio_queue = elem list
  exception EmptyQueue
  val empty = []
  ...
  fun insert (y, []) = [y]
    | insert (y, x :: xs) =
      if compare (y, x) = GREATER then x :: insert (y, xs)
      else y :: x :: xs
  ...
end
```

Functor instantiation

Third step: instantiate the functor

```
structure IntPQ =
    PriorityQueue (type elem = int
                   compare = Int.compare)

structure StringPQ =
    PriorityQueue (type elem = string
                   compare = String.compare)

fun cmp (x, y) = case Int.compare (x, y) of
     GREATER => LESS
   | LESS => GREATER
   | EQUAL => EQUAL

structure RevIntPQ = PriorityQueue (type elem = int
                                    compare = cmp)
```

More on functors

Functors can also abstract over entire structures:

```
signature ORDERING =
sig
  type elem
  val compare: elem * elem -> order
end

functor PriorityQueue (structure Elem : ORDERING) :>
  PRIORITY_QUEUE =
struct
  type elem = Elem.elem
  ...
end
```

Higher-order functors

SML/NJ in addition supports higher-order functors

```
signature DI_GRAPH =
sig
  type vertex
  type label
  type graph
  ...
end

funsig SHORTEST_PATHS_FN
  (structure DiGraph : DI_GRAPH where type label = int) =
sig
  type graph = DiGraph.graph
  type vertex = DiGraph.vertex
  type cost = int
  val shortestPaths : graph * vertex -> (vertex * cost) list
end
```

Higher-order functors (Cont'd)

```
funsig PRIORITY_QUEUE_FN
   (type elem
    val compare : elem * elem -> order) =
  PRIORITY_QUEUE where type elem = elem

functor Dijkstra
  (functor PqFn : PRIORITY_QUEUE_FN) : SHORTEST_PATHS_FN =
struct
  functor (structure G : DI_GRAPH where type label = int) =
  struct
    fun cmp ((_,c1), (_,c2)) = Int.compare (c1, c2)
    structure Pq = PqFn(type elem = G.vertex * int
                        val compare = cmp)
    ...
  end
end
```