Advances in Programming Languages:
Generics and polymorphism

Stephen Gilmore
The University of Edinburgh
February 1, 2007

Background

Computer programs are laborious and expensive to develop. The cost of this work would be amortised if code developed for one application could be reused for another. One way to promote code reuse is to reduce dependencies of components of the application on specific details of data representations in the program. In this way, it may be the case that if we change the data representation then part of the code can be reused unchanged, or it might even be applicable to other data structures in other programs.

Code which has a formal independence from the type of some of the data which it processes is said to be generic.

In general, generic and re-usable code are different things. Generic code is not always re-usable; and re-usable code is not always generic. However, one advantage of generic code is that it makes clear the extent to which it is independent of the data which it processes.

Examples of generic code include the following.

• A function to reverse a list; this should be applicable to lists of any type.
• A method to flatten a data structure to a printable or serial representation.
• A visitor which walks over a data structure applying a given method.

Polymorphic functions

Polymorphic functions have many forms (this is what “polymorphic” means). Different programming languages have different notions of polymorphism. Caml implements parametric polymorphism.

The Objective Caml top-level loop is a good way of experimenting with functions to see if we understand how to use them. We issue the command ocaml to use the interactive system. (The ocamlc compiler produces bytecode. The ocamlopt compiler produces native code.).

Objective Caml version 3.09.2

# List.rev ;; (* ask about the reverse function *)
- : 'a list -> 'a list = <fun>

# List.rev [ 1 ; 2 ; 3 ] ;; (* apply this function *)
- : int list = [3; 2; 1]
Polymorphic functions such as `List.rev` do not require special treatment from the Caml compiler and run-time. Caml programmers can define and use their own polymorphic functions. One typical example of polymorphic functions (operating over a polymorphic datatype) are list-processing functions.

The empty list is denoted by `[]`. An example of a non-empty list literal is `[1; 2; 3]`. Lists can be built element-by-element using the list constructor (“cons”, denoted by two colons `::`). Thus `3::[]` is the same as `[3]`, a one-element integer list containing the integer 3. The two-element integer list containing first 2 and then 3 could be written as `2::3::[]` or `2::(3::[])` or `2::[3]` or `[2; 3]`. The append function is “@”, so `[1]@[2;3]` is `[1;2;3]`.

If we wished to implement list reversal ourselves, we could write this:

```plaintext
(* File: programs/caml/reverse.ml *)
let rec reverse =
  function [ ] -> [ ]
  | h :: t -> (reverse t) @ [h];;
```

Caml infers that, given a list of `'a` values `reverse` returns a list of `'a` values, for any type `'a`. The `'a` variable is known as a type variable. That is, it is a variable which stands for a type (such as integer, or string, or boolean) instead of a value (such as 13, or “abc”, or true).

The reason that a type variable is introduced is that there is nothing in the code of the function which causes the element type of the list to be specialised. Type inference in Caml computes the most general type for any function or value. The type `'a list` is more general than any monotype such as `int list`.

**Object polymorphism**

Object-oriented languages such as Java do not have parametric polymorphism as their primary means of generalising classes and methods. OO languages instead support object polymorphism.

An OO method is polymorphic because it could be passed an object of the specified class or any of its subclasses.

Parametric polymorphism and object polymorphism are different things. A Java programmer would implement an object-polymorphic reverse routine quite differently from that produced by a functional programmer.

```plaintext
/* File: programs/java/Reverse.java */
class Reverse {

  /* A method which operates on Object arrays */
  static void reverse (Object[] a) {
    /* Iterate through to the mid-point */
    for (int i = 0 ; i < a.length / 2 ; i++) {
      /* Swap two entries in the array */
      Object temp = a[i];
      a[i] = a[a.length - (i+1)];
      a[a.length - (i+1)] = temp;
    }
```

public static void main(String[] args) {
    /* Compute the lengths of the arguments */
    Integer[] lengths = new Integer[args.length];
    for (int i = 0 ; i < args.length ; i++)
        lengths[i] = new Integer(args[i].length());

    reverse(args); /* Reverse a string array */
    for (int i = 0; i < args.length ; i++)
        System.out.println(args[i]);
    reverse(lengths); /* Reverse an integer array */
    for (int i = 0; i < lengths.length ; i++)
        System.out.println(lengths[i]);
}

This program compiles and runs successfully, reversing both of the arrays. In operation it is very different from the functional version because it modifies its argument in place instead of copying it and returning the reversed copy. To compare the two versions more closely, we can write a version of the Java program which returns a (reversed) copy.

/* File: programs/java/Reverse2.java */
class Reverse2 {

    /* A method which operates on Object arrays */
    static Object[] reverse (Object[] a) {
        /* Accumulate the answer in a local variable */
        Object[] a_rev = new Object[a.length];
        /* Iterate all the way through the array */
        for (int i = 0 ; i < a.length ; i++) {
            a_rev[a.length - (i+1)] = a[i];
        }
        return a_rev;
    }

    public static void main(String[] args) {
        /* Compute the lengths of the arguments */
        Integer[] lengths = new Integer[args.length];
        for (int i = 0 ; i < args.length ; i++)
            lengths[i] = new Integer(args[i].length());

        /* Reverse arrays; then downcast */
        args = (String[])reverse(args);
        lengths = (Integer[])reverse(lengths);
        for (int i = 0; i < args.length ; i++)
            System.out.println(args[i]);
        for (int i = 0; i < lengths.length ; i++)
            System.out.println(lengths[i]);
    }
}
This program compiles without generating any warnings from the Java compiler. It fails at run-time with a class cast exception.

```
[slim]stg: java Reverse2 a bb ccc ddd
Exception in thread "main" java.lang.ClassCastException
  at Reverse2.main(Reverse2.java:21)
```

The problematic command in the program code is the one to downcast the result of the method from `Object[]` to `String[]`.

Java uses a *covariant* rule for typing array uses. That is, if `A` is a subtype of `B` then `A[]` is a subtype of `B[]`.

Concretely, in our example `String` is a subtype of `Object` so `String[]` is a subtype of `Object[]`, so the call to the `reverse` method is well-typed. However, the local variable whose result is returned from the method is an `Object[]`, which cannot be downcast to `String[]`. But this points the way to repairing the program: just iterate through the array downcasting each element in turn.

```java
/* File: programs/java/Reverse3.java */
class Reverse3 {
    /* Reverse method as before. */
    static Object[] reverse (Object[] a) {
        Object[] a_rev = new Object[a.length];
        for (int i = 0 ; i < a.length ; i++)
            a_rev[a.length - (i+1)] = a[i];
        return a_rev;
    }

    public static void main(String[] args) {
        Integer[] lengths = new Integer[args.length];
        for (int i = 0 ; i < args.length ; i++)
            lengths[i] = new Integer(args[i].length());
        /* Reverse arrays of strings and integers */
        Object[] objectArgs = reverse(args);
        Object[] objectLengths = reverse(lengths);
        /* Downcast both object arrays appropriately */
        for (int i = 0; i < args.length ; i++) {
            args[i] = (String)objectArgs[i];
            lengths[i] = (Integer)objectLengths[i];
        }
        for (int i = 0; i < args.length ; i++)
            System.out.println(args[i]);
        for (int i = 0; i < lengths.length ; i++)
            System.out.println(lengths[i]);
    }
}
```

This program compiles without warnings and runs successfully. Each object in the array is a string in the first case, and an integer in the second case. So each per-element cast is successful.

However, we have done quite a bit of violence to the intention of the program just to get it to work. In particular, every use of the `reverse` method returns a result of the wrong type which needs to be repaired later.
Cast operations

The previous two attempts to implement this (simple) function needed to use either implicit coercions or explicit cast operations. The parameters to the reverse method were implicitly coerced up to Object[] at the point of call. The result from the reverse method was explicitly cast down from Object[]. Up casts (up the class hierarchy) are always safe but down casts are not.

Cast operations are unhelpful to compile-time checking of programs. In a strongly-typed language such as Java, an incorrect downcast raises an exception. This will either be handled by the application, or terminate its execution. Casts are not unregulated in Java. An attempt to cast across the class hierarchy (say, from String to Integer) will be faulted at compile-time.

```java
// File: programs/java/CompileTimeCastChecking.java
class CompileTimeCastChecking {
    Integer i = (Integer) "1";
}
```

This is the result of compiling this (intentionally) flawed class definition.

```
[sлим]stg: javac CompileTimeCastChecking.java
CompileTimeCastChecking.java:3: inconvertible types
found  : java.lang.String
required: java.lang.Integer
    Integer i = (Integer) "1";
^
1 error
```

In a weakly-typed language such as C (without a class hierarchy) casts which can never succeed are still passed at compile time. A cast error may result in a program crash, or continuation with unpredictable future behaviour. Reducing the number of times that programmers need to write cast operations in their code would lead to more reliable programs. This is one of the aims of generic programming.

Generic Java

The Java programming language has developed over several versions. So far there have been seven significant releases of Java (Java 1.0, 1.1, 1.2, 1.3, 1.4, 5.0 and 6.0). Some releases have incorporated language extensions (inner classes, strict floating point, assertions, . . . ) and others have improved library support (Swing, XML, regular expressions, . . . ). A relatively recent release of the Java programming language (Java 5.0) includes generic classes and methods, along with many other language extensions. This is generic Java. The intention of this extension of the language is to allow programming with parametric polymorphism in addition to the existing object polymorphism.

Syntactically, the notation for generics in Java is similar to the notation for templates in C++, so it might be not-too-horrifying for many Java developers.

As our first example of a generic class we can look at generic stacks in generic Java. But first we recall stacks in (non-generic) Java.

---

1 Somewhat confusingly for programming language historians of the future, Java 1.2 was also known as Java 2 and Java 5.0 is also called Java 1.5.
import java.util.Stack;

class SimpleStack {
    public static void main (String[] args) {
        /* Declare a stack of objects */
        Stack s = new Stack();
        s.push("a"); s.push("b"); s.push("c");
        while (!s.isEmpty()) {
            /* Cast needed to downcast to String */
            String st = (String)s.pop();
            System.out.println(st);
        }
    }
}

Stacks in (non-generic) Java are heterogeneous in the usual OO way. They contain instances of objects of different classes. We need to downcast the objects which we retrieve from a heterogeneous stack. Stacks in generic Java are homogeneous like lists in functional programming languages. They contain instances of objects of the same class. We instantiate their type variable when we declare a stack. We do not need to downcast the objects which we retrieve from a homogeneous stack.

import java.util.Stack;

class GenericStack {
    public static void main (String[] args) {
        /* Declare a stack of strings */
        Stack<String> s = new Stack<String>();
        s.push("a"); s.push("b"); s.push("c");
        while (!s.isEmpty()) {
            /* No cast needed here! */
            String st = s.pop();
            System.out.println(st);
        }
    }
}

Now, if we incorrectly attempt to store a non-string into a stack of strings the compiler will fault our program at compile-time. In contrast, non-generic Java would allow us to store non-strings in a stack which we had intended to use only for strings. Then, an incorrect cast operation would cause a run-time exception. Many people would argue that it is better to catch errors in programs at compile-time than to let them occur at run-time.
String st = s.pop();
System.out.println(st);
}
}
}

The compiler (correctly) reported the error in the program.

CheckStack.java:7:
push(java.lang.String) in
java.util.Stack<java.lang.String>
cannot be applied to (java.lang.Integer)
s.push(new Integer(1)); /* ERROR! */
^
1 error

Array reversal revisited: generically

Now that we have seen how to define generic data structures in generic Java we can revisit our example of a polymorphic array reversal method. It might seem that all we need to do is make the reverse method generic over the type of the elements of the array, say E.

/* File: programs/java/GenericReverse.java */
class GenericReverse {
    static <E> E[] reverse (E[] a) {
        E[] a_rev = new E[a.length];
        for (int i = 0 ; i < a.length ; i++) {
            a_rev[a.length - (i+1)] = a[i];
        }
        return a_rev;
    }
    public static void main(String[] args) {
        Integer[] lengths = new Integer[args.length];
        for (int i = 0 ; i < args.length ; i++)
            lengths[i] = new Integer(args[i].length());
        args = reverse(args);
        for (int i = 0; i < args.length ; i++)
            System.out.println(args[i]);
        lengths = reverse(lengths);
        for (int i = 0; i < lengths.length ; i++)
            System.out.println(lengths[i]);
    }
}

The above program is faulted at compile time. It is not legal to allocate a generic array in generic Java.

[beetgreens]stg: javac GenericReverse.java
GenericReverse.java:3: generic array creation
    E[] a_rev = new E[a.length];
    
1 error
The reason why this is an error is that generics are implemented in Java by *erasure*. That is, they are compiled out (for backwards compatibility). If we consider this program in this light, then it is simply another way of writing the object polymorphic version which we saw previously. That was a flawed program, which gave rise to a run-time class cast exception. In rejecting this program at compile time, the Java type system is doing the correct thing for a static type system to do: rejecting early rather than late.

**Generics in C#**

Generics have been added to C#2.0. At present we can experiment with generics in C# using the **gmcs** compiler from the Mono project. I used version 1.1.13.0 of that compiler together with v2.0.50727 of the Microsoft.NET Framework. Generics have been added to the Common Language Runtime which executes compiled code in .NET, so generics are not compiled out (to Object). Accordingly, the Generic C# version of this program compiles and runs as expected.

```csharp
/* File: programs/cs/GenericReverse.cs */
using System;

class GenericReverse {
    static E[] Reverse<E> (E[] a) {
        E[] a_rev = new E[a.Length];
        for (int i = 0 ; i < a.Length ; i++) {
            a_rev[a.Length - (i+1)] = a[i];
        }
        return a_rev;
    }

    public static void Main(string[] args) {
        int[] lengths = new int[args.Length];
        for (int i = 0 ; i < args.Length ; i++)
            lengths[i] = args[i].Length;
        args = Reverse(args);
        for (int i = 0; i < lengths.Length ; i++)
            Console.WriteLine(args[i]);
        lengths = Reverse(lengths);
        for (int i = 0; i < lengths.Length ; i++)
            Console.WriteLine(lengths[i]);
    }
}
```

This program executes as below:

```
 gmcs GenericReverse.exe abc defgh
defgh
 abc
 5
 3
```
We could attempt to force our Java program to compile by inserting a cast, as shown below.

```java
/* File: programs/java/GenericReverse2.java */
class GenericReverse2 {
    static <E> E[] reverse (E[] a) {
        E[] a_rev = (E[]) new Object[a.length];
        for (int i = 0 ; i < a.length ; i++) {
            a_rev[a.length - (i+1)] = a[i];
        }
        return a_rev;
    }

    public static void main(String[] args) {
        Integer[] lengths = new Integer[args.length];
        for (int i = 0 ; i < args.length ; i++)
            lengths[i] = new Integer(args[i].length());
        args = reverse(args);
        for (int i = 0; i < args.length ; i++)
            System.out.println(args[i]);
        lengths = reverse(lengths);
        for (int i = 0; i < lengths.length ; i++)
            System.out.println(lengths[i]);
    }
}
```

As is often the case, a cast is the wrong solution. The Java compiler generates a compile-time warning and refers us to a “lint” tool.

```bash
[hefty] stg: javac GenericReverse2.java
Note: GenericReverse2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
[hefty] stg: javac -Xlint:unchecked GenericReverse2.java
GenericReverse2.java:4:
warning : [unchecked] unchecked cast
found : java.lang.Object[
required: E[]
E[] a_rev = (E[]) new Object[a.length];
```

1 warning

The cast is wrong, so the program fails at run-time.

```bash
[hefty] stg: java GenericReverse2 a bb ccc ddd eeeee
Exception in thread "main"
    java.lang.ClassCastException: [Ljava.lang.Object;
at GenericReverse2.main(GenericReverse2.java:15)
```

Java’s generics have shown themselves to be safe, but seem to be less expressive than the parametric polymorphism in O’Caml. Now let’s see an example where they work well, and allow us to define a parametric polymorphic routine.
List reversal revisited: generically

We now try to implement list reversal in generic Java. The first step in implementing this is to define a generic List class. We will use Java’s null value to represent the empty list. Any non-empty list has a head and a tail so these will be fields of the List class. The constructor for the class will initialise these two fields.

The rev function can be a static method in the class. This will be accessed by List.rev, as in Caml. Its operation will be as in Caml: we pass in a list to be reversed and the method allocates and builds a new list which is the reversal of the list passed in as a parameter. This newly-allocated list is returned as the result of the method.

```java
/* File: programs/java/List.java */
/* A generic list class with type variable "E" */
public class List<E> {
    /* Fields to hold the head of the list and the tail */
    public E head;
    public List<E> tail;
    
    /* A constructor for generic lists (Cons) */
    public List (E head, List<E> tail) {
        this.head = head;
        this.tail = tail;
    }
    
    /* The generic list reverse method */
    public static <A> List<A> rev (List<A> lst) {
        List<A> res = null;
        while (lst != null) {
            res = new List<A>(lst.head, res);
            lst = lst.tail;
        }
        return res;
    }
}
```

Recall that Java passes object references by value so the assignments to the “lst” variable in the “rev” method do not update the actual parameter. We can now implement a test harness which successfully reverses [“a”, “b”, “c”] to [“c”, “b”, “a”].
class GenericListReverse {
    public static void main (String[] args) {
        List<String> l =
            new List<String>(
                "a",
                new List<String>(
                    "b",
                    new List<String>(
                        "c", null)));
        List<String> l_rev = List.rev(l);
        while (l_rev != null) {
            System.out.println(l_rev.head);
            l_rev = l_rev.tail;
        }
    }
}

This program compiles without warnings and runs without errors.

**Re-use in Ruby**

The Ruby language does not have static type checking as O’Caml and Java do, so how would generic programming work in Ruby? The answer is that it seems to work very simply, with the obvious implementation working as expected.

```ruby
# File: programs/ruby/Reverse.rb
class Reverse
def Reverse.rev(a)
    a_rev = Array.new
    for i in 0..a.length - 1
        a_rev[a.length - (i+1)] = a[i]
    end
    return a_rev
end
end
```

The method `Reverse.rev` is a class method (a static method of the class). As might be expected, `Array.new` allocates a new array (size initially 0).

```ruby
# File: programs/ruby/ReverseTest.rb
a1 = [1, 2, 3]
puts Reverse.rev(a1) == [3, 2, 1] # prints true

a2 = ["a", "b", "c"]
puts Reverse.rev(a2) == ["c", "b", "a"] # prints true

a3 = [1, "b", [3, 4]]
puts Reverse.rev(a3) == [[3, 4], "b", 1] # prints true
```

Ruby is really working with something akin to `Object` arrays in Java, so we have the flexibility of this kind of programming but none of the checking applied by generics.
Summary

- Caml, like many other functional programming languages, allows programmers to develop (parametric) polymorphic functions.

- Classical OO languages such as Java use object polymorphism instead of parametric polymorphism.

- Object polymorphism occasionally requires programmers to cast the result down to the correct class. These operations are not checked at compile time and can lead to run-time errors.

- Casting arrays may not work as expected. Casting an array is not the same as casting all of its elements. Downcasting an array can involve writing an iterator.

- Generic Java offers parametric polymorphism via generics in addition to the object polymorphism offered by the class system.

- Generic methods can be checked at compile time and generic collection classes are better for representing homogeneous collections than non-generic collection classes. Homogeneous collections occur frequently in practice.