A slice of Pizza:
A quick introduction to Pizza, a dialect of Java

Martin Odersky
University of Southern Australia

Philip Wadler
Lucent Technologies

10 October 1997

Abstract

Pizza is a superset of Java that incorporates four additional features.

- *Parametric polymorphism*: the ability to parameterize classes over types, similar to (but safer than) templates in C++.
- *Higher-order functions*: the ability to treat functions as values, similar to inner classes in Java 1.1, but more convenient and more efficient.
- *Algebraic data types*: a convenient way of representing trees with support for pattern matching, providing advantages similar to the well-known 'Composition' and 'Visitor' design patterns.
- *Tail calls*: don't grow the stack when the last operation in one methods is to call another method; this enables one to replace iteration by recursion, and makes it easy to simulate finite-state machines.

Pizza contains Java as a subset, interworks smoothly with Java, and compiles into the Java Virtual Machine (JVM): so it runs wherever Java runs, including web browsers. Pizza is freely available from the Pizza website.

`http://www.cs.bell-labs.com/~wadler/pizza`

The Pizza compiler can be used to translate Pizza into Java or to generate JVM code directly; it runs faster than Sun's own `javac` compiler.

1 Introduction

Underneath the Java hype, there is substance: a language that is relatively easy to learn and use, that provides a modicum of portability and a general-purpose GUI library, and that lets one attach interactive programs to web pages [AG96, GJS96].

One of the strengths of Java is that it is a relatively simple design. Arguably, it is too simple on at least two counts, as it omits two widely useful programming language features.

- *Parametric polymorphism*: the ability to parameterize classes over types, similar to (but safer than) templates in C++.

- *Higher-order functions*: the ability to treat functions as values.

There are also some features which, while not general purpose, are convenient for certain applications.

- *Algebraic data types*: a convenient way of representing trees with support for pattern matching, providing advantages similar to the well-known 'Composition' and 'Visitor' design patterns.

- *Tail calls*: don't grow the stack when the last operation in one methods is to call another method; this enables one to replace iteration by recursion, and makes it easy to simulate finite-state machines.

Pizza is a superset of Java that includes these four features. Pizza contains Java as a subset, interworks smoothly with Java, and compiles into the Java Virtual Machine (JVM): so it runs wherever Java runs, including web browsers. Pizza is freely available from the Pizza website.

`http://www.cs.bell-labs.com/~wadler/pizza`

The Pizza compiler can be used to translate Pizza into Java or to generate Java Virtual Machine code directly.

The Java designers themselves have accepted the arguments that higher-order functions and parametric polymorphism are a good thing. The new Java 1.1 release includes *inner classes*, which provide similar functionality to higher-order functions. And there is a mailing list moderated by Sun to discuss ways of adding parametric types to Java, and the Pizza designers are engaged in dialogue with the Java designers about how to smoothly extend Java to include parametric polymorphism. Pizza was originally designed based on Java 1.0; the new version supports full Java 1.1 (including inner classes) and also supports higher-order functions directly. The higher-order functions are slightly less flexible than inner classes, but are easier to use and have a more efficient implementation.

Pizza features are defined by translation into Java. The Pizza compiler also produces JVM output directly, for convenience. However, the Java language and the JVM support pretty much exactly the same abstractions, so little is lost by translating Pizza into Java and thence into the JVM rather than compiling to the JVM directly.

The Pizza compiler is itself implemented in Pizza, and was written by Martin Odersky. Since Pizza contains Java as a subset, it can be used as a replacement for a Java compiler. It accepts files with either the `pizza` or `java` extension, and in the latter case accepts only legal Java, disabling the Pizza extensions. Further, it takes less time to compile Java programs than Sun's own `javac` compiler.

2 Parametric polymorphism

Reusable components act on many types. In Java, this is often supported by operating on the class Object, but this requires a great deal of casting, which can be awkward and error prone. To illustrate, here is a sorter in Java, followed by the same program in Pizza.

In the Java program, the marked line will fail at run time, because a comparator for integers is used with strings. In the corresponding Pizza program, the same line will cause an error at compile time (and so the whole line itself is commented out).

Also, note that in the Java program, for sorting purposes one must use an array of `Integer` not an array of `int`, since `Integer` (but not `int`) is a subclass of `Object`. Pizza can work with the `int` type directly. (It does this by using a special representation for arrays of polymorphic type, and by automatically packing an `int` as an `Integer` when an `Object` is expected.)

Example 2.1 Sort in Java.

```
abstract class Ord {
  public abstract boolean leq (Object x, Object y);
}

class OrdInteger extends Ord {
  public boolean leq (Object x, Object y) {
    return ((Integer)x).intValue() <= ((Integer)y).intValue();
  }
}
```

```
class OrdString extends Ord {
  public boolean leq (Object x, Object y) {
    return (((String)x).compareTo((String)y) <= 0);
  }
}

class Sort {
  static void sort (Ord p, Object[] a) {
    for (int i = 0; i < a.length; i++) {
      for (int j = i+1; j < a.length; j++) {
        if (!p.leq(a[i],a[j])) {
          Object t=a[i]; a[i]=a[j]; a[j]=t;
  } } } }
  public static void main(String[] args) {
    Integer[] a = { new Integer(2), new Integer(1) };
    sort(new OrdInteger(), a);
    String[] b = {"data", "ditto", "dada"};
    sort(new OrdString(), b);
    sort(new OrdInteger(), b);  /* RUN-TIME ERROR */
  }
}
```

Example 2.2 Sort in Pizza

```
abstract class Ord<T> {
  public abstract boolean leq (T x, T y);
}

class OrdInteger extends Ord<int> {
  public boolean leq (int x, int y) {
    return x <= y;
  }
}

class OrdString extends Ord<String> {
  public boolean leq (String x, String y) {
    return x.compareTo(y) <= 0;
  }
}
```

```
class Sort {
  static <T> void sort (Ord<T> p, T[] a) {
    for (int i = 0; i < a.length; i++) {
      for (int j = i+1; j < a.length; j++) {
        if (!p.leq(a[i],a[j])) {
          T t=a[i]; a[i]=a[j]; a[j]=t;
  } } } }
  public static void main(String[] args) {
    int[] a = {2, 1};
    sort(new OrdInteger(), a);
    String[] b = {"data", "ditto", "dada"};
    sort(new OrdString(), b);
    /* sort(new OrdInteger(), b);  COMPILE-TIME ERROR */
  }
}
```

3 Higher-order functions

In Java 1.1, one may use inner classes to slightly simplify the above program, by letting one move the definitions of the classes to compute orderings on integers and strings directly to the point where they are used. It can be further simplified by using the higher-order functions of Pizza.

Example 3.1 Sort in Java, with inner classes.

```
abstract class Ord {
  public abstract boolean leq (Object x, Object y);
}

class Sort {
  static void sort (Ord p, Object[] a) {
    for (int i = 0; i < a.length; i++) {
      for (int j = i+1; j < a.length; j++) {
        if (!p.leq(a[i],a[j])) {
          Object t=a[i]; a[i]=a[j]; a[j]=t;
  } } } }
  public static void main(String[] args) {
    Integer[] a = { new Integer(2), new Integer(1) };
```

```
    sort(new Ord {
      public boolean leq (Object x, Object y) {
        return ((Integer)x).intValue() <= ((Integer)y).intValue();
      } }, a);
    String[] b = {"data", "ditto", "dada"};
    sort(new Ord {
      public boolean leq (Object x, Object y) {
        return (((String)x).compareTo((String)y) <= 0);
      } }, b);
  }
}
```

Example 3.2 Sort in Pizza, with higher-order functions

```
class Sort {
  static <T> void sort ((T,T)->boolean p, T[] a) {
    for (int i = 0; i < a.length; i++) {
      for (int j = i+1; j < a.length; j++) {
        if (!p(a[i],a[j])) {
          T t=a[i]; a[i]=a[j]; a[j]=t;
  } } } }
  public static void main(String[] args) {
    int[] a = {2, 1};
    sort(fun(int x,int y)->boolean{return x<=y;},a);
    String[] b = {"data", "ditto", "dada"};
    sort(fun(String x,String y)->boolean{return x.compareTo(y)<=0;},b);
  }
}
```

4 Algebraic data types

Object types and inheritance are complementary to algebraic types and matching. Object types and inheritance make it easy to extend the set of constructors for the type, so long as the set of operations is relatively fixed. Conversely, algebraic types and matching make it easy to add new operations over the type, so long as the set of constructors is relatively fixed. The former might be useful for building a prototype interpreter for a new programming language, where one often wants to add new language constructs, but the set of operations is small and fixed (evaluate, print). The latter might be useful for building an optimising compiler for a mature language, where one often wants to add new passes, but the set of language constructs is fixed.

Here is a simple example for a data type that represents linked lists, that also demonstrates polymorphism and higher-order functions. Note that the `zip` routine decomposes *two* lists in parallel, and is not very easy to write in the traditional object-oriented style, where one breaks a method into parts for each subclass.

Example 4.1 Polymorphism, higher-order functions, and algebraic types

```
class Pair<A,B> {
  case Pair(A fst,B snd);
}

class List<A> {
  case Nil;
  case Cons(A head, List<A> tail);
  <B> List<B> map ((A)->B f) {
    switch (this) {
      case Nil:
        return Nil;
      case Cons(A x, List<A> xs):
        return Cons(f(x), xs.map(f));
    }
  }
  <B> List<Pair<A,B>> zip (List<B> ys) {
    switch (Pair(this,ys)) {
      case Pair(Nil,Nil):
        return Nil;
      case Pair(Cons(A x, List<A> xs),
                Cons(B y, List<B> ys)):
        return Cons(Pair(x,y), xs.zip(ys));
    }
  }
}
```

5 Tail calls

A *tail call* is one that occurs as the very last step in a method. A clever compiler can implement tail calls without growing the stack; instead the current call frame is replaced by the frame for the tail call. The Java Virtual Machine does not support tail calls, but they can be simulated. (This is pretty ironic, considering that tail calls were popularised by Guy Steele, one of the three principal designers of Java [SS76].) The simulation is expensive in time: in Pizza, tail calls cost an order of magnitude more than ordinary calls. But since tail calls don't add to the stack, they can be nested arbitrarily deep.

For instance, one can simulate a finite-state machine by defining method for each state, which calls the method for the next state with a tail call. This can be very flexible: you don't need to decide in advance what different states exist, you just define one method for each state as required. To achieve the same effect without tail calls, you need to have each routine return an object that contains the next method to be called, which is how tail calls are implemented in Pizza.

Tail calls are indicated by placing `goto` in front of the call. Methods that may contain a tail call or be the target of a tail call are indicated by writing `continue` in the declaration. (Both `goto` and `continue` are already reserved words in Java.)

As an example, here is a routine that processes text, counting for each pair of parentheses the number of items (separated by commas) between them, and signalling an error if any item is empty or if parentheses are nested or do not match.

Example 5.1 Tail calls in Pizza.

```
class Counter {
  final char EOF = (char)0;
  InputStream in;
  Counter (InputStream i) { in = i; }
  char next () {
    int c = in.read();
    if (c == -1) return EOF; else return (char)c;
  }
  continue void normal () {
    switch (next()) {
      case EOF: goto done ();
      case '(': goto startitem (0);
```

```
      default: goto normal ();
    }
  }
  continue void startitem (int count) {
    switch (next()) {
      case EOF: goto error ();
      case ')': goto error ();
      case ',': goto error ();
      default: goto insideitem (count);
    }
  }
  continue void insideitem (int count) {
    switch (next()) {
      case EOF: goto error ();
      case ')': System.out.println("count " + count); goto normal ();
      case ',': goto startitem (count+1);
      default: goto insideitem (count);
    }
  }
  continue void done {
    System.out.println("done");  return;
  }
  contine void error {
    System.out.println("error");  return;
  }
  static public void main (String[] args);
    new Counter(System.in).normal();
  }
}
```

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* Java Series, Sun Microsystems, 1996. ISBN 0-201-63455-4.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Java Series, Sun Microsystems, 1996. ISBN 0-201-63451-1.

[SS76] Guy Lewis Steele, Jr. Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMDBA: The Ultimate GOTO. AI Memo 443, MIT AI Lab, October 1977.