Multiparadigm Extensions to Java

Timothy A. Budd
Department of Computer Science
Oregon State University
Corvallis, Oregon, USA

November 1, 2000

Abstract

In 1995 my students and I developed Leda, a multiparadigm language based on the Pascal model. Leda allowed programmers to create abstractions in an object-oriented, functional, or logic programming style. More recently we have been interested in recreating this work, but this time using Java as the language basis. The objective to to add as few new operations as possible, and to make these operations seem as close to Java as possible, so that they seem to fit naturally into the language. To date we have proposed facilities for breaking apart composed objects (sometimes called *unboxing*), for functions as first-class values, for pass-by-name parameters, and for relational (or logic) programming.

1 Introduction

In the book *Multiparadigm Programming in Leda* (published by Addison-Wesley in 1995 [Budd 1995]) I described a multipardigm programming language based on the Pascal model. My students and I purposely selected Pascal for our foundation language because we wanted a syntax that would be non-intimidating (rather than say, Scheme, which to many programmers seems to be an entirely intimidating language). By adding only a few simple features (functions as values, nameless or lambda functions, relations) we were able to demonstrate how applications could be developed in a functional style, in a Prolog logic-programming style, or in an object-oriented style. Furthermore, a synergy quickly develops between the different paradigms, and a great deal of power derives from combining code written in two or more styles in a single function.

As Java has slowly become the dominant language in the academic world, we have become interested in the question of whether or not it would be possible to add similar features to Java. Some experiments in this area have already been tried. The best known multiparadigm version of Java is probably

Pizza [Ordersky 1997]. However, we felt that Pizza (and other similar languages) suffered from one fault that we had taken pains to avoid in the original design of Leda. This was that the additions did not remain true to the original *feel* of the language. With Pizza it is far too easy to tell when you leave Java and start programming in Pizza. We wanted something that was more subtle, so that the features seem to flow naturally into the existing Java syntax, without clear-cut boundaries. The reader will have to judge for themselves whether or not we have achieved our goal.

In this paper we will describe five major modifications we have introduced into Java/MP, our extended Java language. These additions are:

- Unboxing. We have extended the instanceof operator so that in addition to testing the dynamic class of an expression, it also breaks a value that was constructed using composition into its component parts.
- First class Functions. Functions as values, arguments, and results. Nameless functions created as expressions.
- Pass-by-name parameters. A technique for delaying the evaluation of an argument until it is used.
- Operator overloading. Mostly just syntactic sugar used to reduce the size of a program and make it easier to read.
- The relation as a data type, and Prolog-style relational programming. As we did with Leda, this feature is constructed by combining the above features with a library of operations that implement logic programming tasks such as backtracking and unification.

In the remainder of the paper we will describe each of these elements in turn.

2 Boxing and UnBoxing

A style of programming popularized in ML and similar functional languages is based on using composition as a fundamental technique, rather than inheritance. (Naturally, since these languages don't have inheritance). Complex objects are formed by combining together more basic parts, and operations are performed largely by breaking apart complex objects into their elements. These higher-level tasks are sometimes called *boxing* and *unboxing*.

Since constructors in Java build a new entity, we can use the constructor for a boxing primitive. To build an unboxing primitive without introducing major changes to the language we extend the existing instanceof operator. This operator is already testing the class of an object. It is a small addition to have it also test the class, and if it is going to return true extract the public data fields.

For example, a Tree might be defined as either a Leaf or a Node:

```
class Tree { }

class Leaf extends Tree {
    public Leaf (int v) { value = v; }
    public int v;
}

class Node extends Tree {
    public Node (Tree l, Tree r) { left = l; right = r; }
    public Tree left;
    public Tree right;
}

class EmptyNode extends Tree { }
```

Building a tree is performed by constructors.[1]

```
Tree t = new Node(new Leaf(3), new Node(new EmptyNode(), new Leaf(3)));
```

Unboxing then looks something like the following:

```
Tree t = ... ; // as before
int val;
Tree lft, rgt;
if (t instanceof Leaf(val))
    System.out.println("leaf value is " + val);
else if (t instanceof Node(lft, rgt))
    System.out.println("left is " + lft);
else if (t instanceof EmptyNode)
    System.out.println("empty node");
```

To implement the unboxing operation, additional clauses are added to the instanceof operator. It is easiest to describe this by assuming that Java had a comma operator similar to C. Then we would say that an expression such as:

```
t instanceof Leaf(val)
```

could be translated as follows:

[1] We would like to get rid of the need for the new operator, as in Leda, but deemed that would be doing too much damage to the underlying language. In Leda simply using a class name as if it were a function automatically implies the creation of a new value.

```
((t instanceof Leaf) & ((val = ((Leaf) t).val), true))
```

Note the cast, the assignment, and the comma. Since Java does not have a comma operator we simulate this by creating a function that takes an object as argument, ignores it, and returns a boolean true value.

Because we are translating Java/MP into Java, we cannot get around the visibility modifiers in the language. That is, we can only recover those fields that are publicly accessible. This leads to declaring data fields as public, something generally considered to be bad form in the object-oriented world. It also introduces an important semantic meaning to the order in which data fields are declared, something that is generally unimportant in Java. Nevertheless, we believe it is worthwhile to support this feature because of the opportunities for different styles of program development.

3 Functions as First Class Values

Functions as first class values have been implemented before [Ordersky 1997], but it was our feeling that the syntax used by Pizza was too un-Java-like. We wanted something closer to the underlying Java language, so that it would almost be difficult to tell where the new features began and the old features ended.

Here are some of the examples of our syntax. Declaring a variable of type function:

```
int (double, int) x;
```

Note that the parenthesis right after the return type tells you in that this is a function type. We don't run into the problem that C has of not knowing until you see the absence of a function body that you are defining a function name.

A function that takes as argument another function:

```
double map (double identity, double (double, double) binFun) { ... }
```

The following shows both the syntax used for a function that returns another function as argument, and the syntax for anonymous, or lambda functions.

```
double (double) buildAdder (double left) {
   return double(double right)
      { return left + right; };
}
```

A function that implements a curry illustrates all three features, functions as arguments, as return types, and the creation of an anonymous inner function:

```
int (int) curryLeft (int left, int (int, int) theFun) {
   return int(int right) { return theFun(left, right); };
}
```

Note that we require no additional keywords or operators, and that our syntactic extensions are very similar to existing features in the Java language.

Of course, those familiar with the issues involved in the implementation of programming languages will note we need to build closures behind the scenes to do this properly. We are trying to do this entirely in Java, creating closures as actual object values. When it is necessary to create a closure, local variables and parameters are copied into a specially constructed heap-allocated object. Thereafter references to local variables and parameters are replaced with references to this object.

As with the Pizza system, functions are represented as instances of specially-constructed classes. Each different set of argument types creates an unique interface. This feature permits function values to be assigned to function variables. A function is then a value that implements the interface.

4 Pass by Name

A number of the more interesting programming techniques require the ability to postpone evaluation of function arguments until they are needed. There are various different ways to do this. The one we are currently investigating is to revive the old and seldom used programming technique called *pass by name*. In pass by name an expression is evaluated at the point it is used (if ever) and not at the point the function invocation is made.

Our current syntax for pass by name is to use the `@` sign:[2]

```
void translate (Point @ b) {
   b.x = b.x + 10;
   b = new Point(12, 13);
}

...
Point x(42, 12);
translate(x);
```

The implementation of call-by-names is traditionally performed using a device called a *Thunk*. With the exception of the problem of closures, thunks are

[2]We are not entirely satisfied with this syntax, and keep proposing alternatives.

easily constructed using anonymous classes. The translation of the above is as follows:

```
void translate (Thunk b) {
   ((Point) b.get()).x = ((Point) b.get()).x + 10;
   b.set(new Point(12, 13));
}

Point x(42, 12);
translate (new Thunk() {
   Object get() {return x; }
   void set(Object val) { x = (Point) val; }});
```

A thunk is an object with two member functions. The first is invoked when the value of the thunk is used as an r-value; that is, as an expression. The second method is invoked when the value of the think is used as an l-value; that is, as the target of an assignment. When arguments are passed that cannot be used in the latter fashion (that is, cannot be assigned) then the method constructed for set simply does nothing. (Some have proposed that it should throw an exception in this situation).

5 Operator Overloading

Operator overloading is permitted. Operators are treated not as methods but as functions at the global level. This is just a bit of syntactic sugar aimed at making programs easier to read. All operators have a textual name; for example the + operator is named "plus". Overloading is achieved by redefining this function with various arguments. This is the technique used by Leda [Budd 1995] as well as other languages, such as Python.

6 Relations

As the programming language Leda demonstrates (as well as Scheme and other systems), Prolog-style relational programming can be easily constructed once you have functions as true first class values, pass-by-name and operator overloading. A relation data type has the rather curious definition involving a class that takes a relation as argument and returns a boolean:

```
abstract class Relation {
   abstract public boolean apply (Relation continuation);

   public boolean asBoolean ()
```

```
        { return apply(trueRelation); }

    public void forAll (void(void) action) { ... }

    private static class TrueRelation extends Relation {
        public boolean apply (Relation continuation)
            { return true; }
    }

    private static TrueRelation trueRelation;
}
```

The pass-by-name mechanism is necessary in order to build a unification function. The unification function takes two arguments and ensures they are equal, setting one or the other to make them equal if necessary. Note that unification is a function that returns a relation (that is, returns another function). The "and" and "or" operators are given overloaded meanings. When combined with relations they implement the backtracking system. (How this is done is explored in [Budd 1995]).

With unification a prolog style database can be built such as the following:

```
Relation child(String @ mother, String @ father, String @ child) {
    return
    (unify(mother,"mary") & unify(father, "sam") & unify(child, "fred")) |
    (unify(mother,"alice") & unify(father, "bob") & unify(child, "sam")) |
    (unify(mother,"mary") & unify(father, "bob") & unify(child, "alice"));
}
```

For those unfamiliar with the Prolog style of programming, one important way that relations differ from functions is that parameters can sometimes be input and sometimes be output:

```
String a = null;
String b = null;
String c = null;

if (child(a, "sam", "fred").asBoolean())
    System.out.println("mother of fred is " + a);
if (child("alice", "bob", b).asBoolean())
    System.out.println("child of alice and bob is " + b);
if (child("mary", c, "alice").asBoolean())
    System.out.println("father of alice is " + c);
```

Of course, some relations have multiple solutions. (Filtering out solutions and backtracking to prior states in the primary job of the "and" and "or"

operators for the relation data type). The method forAll is similar to the visitor design pattern. This method takes a function as argument and executes the function on each solution of the query:

```
String r = null;
String s = null;
child("mary", r, s).forAll(void(void) {
   System.out.println("child of mary " + s); });
```

7 Current Status

The current status of the project is very preliminary. We have defined the language features we know we want, and outlined the various approaches to implementation. The work is being performed by Master's level students. One student is currently working on the problem of closure analysis. In the majority of situations local variables and parameters must be copied into a dynamic structure before they can be saved as part of the state for a function. Other students are working on the implementation of the remaining language features.

References

[Budd 1995] Timothy A. Budd, *Multiparadigm Programming in Leda*, Addison-Wesley, Reading, MA, 1995.

[Ordersky 1997] Martin Ordersky and Philip Wadler, "Pizza into Java: Translating Theory into Practice", *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.