

# The Return of Jensen's Device

Timothy A. Budd

Department of Computer Science, Oregon State University, Corvallis, Oregon, USA

**Abstract.** In the early 1990's 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 expanding this work, but this time using Java as the language basis. The objective is 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 implemented facilities for functions as first-class values, pass-by-name parameters, ML style pattern matching, and for relational (or logic) programming. In addition, we are currently evaluating a number of other extensions to the language.

## 1 Introduction

In the book *Multiparadigm Programming in Leda* [7], I described a multiparadigm 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 to most users, particularly beginning programmers. 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 program.

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. Many experiments in this area have already been tried. The best known multiparadigm version of Java is Pizza [21]. 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 what James Gosling calls the *feel* of the language [14]. 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 J/mp, our extended Java language. These additions are:

- First class Functions. Functions as values, arguments, and results. Nameless functions created as expressions.
- Pass-by-name parameters. A technique for bidirectional information flow through parameters, and for delaying the evaluation of an argument until it is used (lazy evaluation).
- Operator overloading. Mostly just syntactic sugar used to reduce the size of a program and make it easier to read.
- Pattern Matching. 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.
- 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.

### 1.1 Pass by Name

We began with the assumption that we would need to support functions as first class data types, similar to those found in Pizza. As with Brew [3], and unlike Pizza, the syntax we adopted is closer to the historical C/C++ model, and is to our eyes more in keeping with the feel of Java. Several examples will subsequently be presented.

The next observation was that we needed at least one more parameter passing mechanism, in addition to Java’s approach of passing object references by value. In particular, for many of the applications we envisioned it was desirable to be able to pass information both into and out of a method or function by means of parameters, as well as a means to delay the evaluation of arguments. After considering many of the possibilities (for example, by-reference, copy in-copy out), we arrived at what might be a rather surprising conclusion. The parameter passing mechanism most suited to our purposes was an old and nowadays seldom used technique, pass-by-name [20].

In large part, this decision was influenced by the fact that pass-by-name fits quite easily into the object-oriented philosophy, and hence there is a simple technique for implementing the mechanism. We will describe this in Section 2.

The classic example used to illustrate the effect of pass-by-name parameters is *Jensen’s device*, a procedure for taking a simple sum of a collection of values [11]. (Named after Jørn Jensen, who first explored its properties). Our first version of Jensen’s device is shown in Figure 1, together with a class that illustrates how it can be invoked. The intent that the first and third arguments to the function `Jensen` are to be passed by name is indicated by the plus sign preceding the type. Each time the variable `i` is modified the corresponding actual parameter in the calling procedure will be changed. The fact that `x` is passed by name means that the evaluation of this parameter is delayed until the point where it is used, in

```

public double Jensen(+int i, int max, +double x) {
    double sum = 0.0;
    for (i = 0; i < max; ++i)
        sum += x;
    return sum;
}

class Main {
    static double [ ] data = {1.5, 2.7, 3.2, 4.1, 5.2, 6.3};

    static public void main (String [ ] args) {
        int i;
        System.out.println("Sum " + Jensen(i, 6, data[i]));
        System.out.println("Sum of odds " + Jensen(i, 3, data[1+2*i]));
    }
}

```

**Fig. 1.** Jensens Device, Version 1

the body of the loop. Furthermore, each time it is used, the actual argument is reevaluated.

The sample main program shows how this can be used to compute the sum of an array, or alternatively the sum of the odd-indexed elements in an array. By varying the parameters passed to Jensen's device a wide variety of behaviors can be produced [17]. For example, although seemingly generating only a one dimensional summation, a summation of a two dimensional array could be obtained as follows:

```
Jensen(i, n, Jensen(j, m, a[i][j]));
```

Similarly, a dot product of two vectors could be computed as:

```
Jensen(i, n, a[i]*b[i]);
```

## 1.2 Free Standing Functions

Figure 1 also illustrates the syntax used for free standing functions; that is, functions not associated with any class. As with Pizza, Brew, and similar systems such functions are ultimately first turned into an interface, and then into a class. Later in Section 2 we will show the transformation of this function.

## 1.3 Functions as Values

Many people dislike pass-by-name, although we view its bad reputation is being largely undeserved. Nevertheless, a number of the interesting things one can do

with pass-by-name can also be accomplished using functions as first class values. (This is not surprising. The implementation of pass by name can in a certain sense be considered a special case of functions as arguments). Our second version of Jensen's device (Figure 2) illustrates this. Here there are two formal parameters, the first an integer and the second a function that takes an integer as argument and returns a double. The loop repeatedly invokes the function in order to yield different values.

```
public double Jensen(int max, double (int) f) {
    double sum = 0.0;
    for (int i = 0; i < max; i++)
        sum += f(i);
    return sum;
}

class Main {
    static double [] data = {1.5, 2.7, 3.2, 4.1, 5.2, 6.3};

    static public void main (String [] args ) {
        System.out.println("Sum of odds " +
            Jensen(3, double (int j) { return data[1+2*j];}));
    }
}
```

**Fig. 2.** Jensens Device, Version 2

The syntax used to define the type field of the function parameter largely mirrors the standard Java syntax for function prototypes, although the parameter names are omitted. The invocation (in the method `main`) illustrates the creation of a nameless function argument. Such arguments are often termed *lambda expressions*, a term adapted from the lambda calculus. Alternatively, a named function could have been passed as argument to `Jensen`, although not a named method (functions are first class values, methods are not).

A function that implements a curry illustrates the three common uses for function values; these are functions as arguments, functions as return types, and the creation of an anonymous function value:

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

Note that it has been necessary to declare the arguments as `final` in order for them to be captured in the context for the function. Also notice that the syntax for functions in J/mp requires no additional keywords or operators, and that,

like Brew [3], our syntactic extensions are very similar to existing features in the Java language. The value `curryLeft` could itself be saved in a variable declared as follows:

```
int(int)(int, int(int, int)) c = curryLeft;
```

#### 1.4 Infinite Length Lists

Functions as first class values have many interesting and unusual uses. A good example is the creation of lazy lists; lists that use lazy evaluation and hence do not calculate their elements until needed. Such lists can be used to represent collections that are infinite in length. A simple definition of a lazy list in which each element is computed as a transformation on the prior value can be written as follows:

```
class llist {
    private int hd; // current value
    private int(int) tf; // transformation function

    public llist (int h, int(int) t) { hd = h; tf = t; }

    public int head () { return hd; }
    public llist tail () { return llist(tf(hd), tf); }
}
```

Some would argue that this class is not only lazy but stupid, in that it reevaluates its elements each time a request is made. An alternative, more traditional, lazy list that delays evaluation until necessary, but thereafter remembers values that it has already computed, can be written but is not as concise. Using this class definition, the list containing all natural numbers can be generated as follows:

```
llist naturals = llist(0, int(int x) { return x+1; });
```

One feature to note is the lack of the `new` operator in this expression. If a class name is used in the fashion of a function, an implicit creation is assumed.

Slightly more complicated is the list of all prime numbers. A straightforward algorithm for computing these is the following:

```
llist primes = llist(2, int(int x) { boolean flag = true;
    while (flag) { flag = false; x++;
        for (int y = 2; (y * y <= x) && ! flag; y++)
            flag = (x%y==0);
        }
    return x; });
```

Many interesting effects can be achieved by manipulating infinite length lists, and developing functions that will filter or transform elements from such lists [7].

## 1.5 Pattern Matching

The language ML popularized a style of programming where compound data values were created by means of constructors, and then broken back into their constituent parts by means of pattern matching. The following ML definitions and function illustrate this behavior:

```
datatype Tree =  
  Leaf of int  
  | Node of int * Tree * Tree;  
  
val t = Node(3, Node(4, Leaf(5), Leaf(7)), Leaf(2));  
  
fun sum(Leaf(v)) = v  
  | sum(Node(v, left, right)) = v + sum(left) + sum(right);
```

Constructors in Java serve much the same purpose as constructors in ML. This is particularly true given that we have made the `new` keyword optional, as previously described. A Java class definition similar to the ML code can be written as in Figure 3. Given this definition, we can construct a tree as follows:

```
Tree t = Node(3, Node(4, Leaf(5), Leaf(7)), Leaf(2));
```

While constructors serve the same purpose in Java and ML, there did not seem to be a good substitute in Java for pattern matching. In pondering the possibilities, and remembering our goal of maintaining the spirit of Java and making minimal changes to the language, we determined that the closest operation to pattern matching in the existing language was the `instanceof` operator. By simply adding an optional argument list to this operator we could make it serve the dual purposes of type testing and deconstruction.

```
t instanceof Node(value, left, right)
```

But how to provide semantics to the pattern matching operation? We initially sought to implement this operation outside of the class being tested. However, it is common for Java programmers to protect their data fields, as with the data fields `value`, `left` and `right` in Figure 3. Therefore a compound object represented as an instance of a class cannot easily be broken into its original parts outside of the class definition. For this reason we decided the author of the class must be directly involved in the pattern matching operation. A compound `instanceof` operator will internally be converted into a type test which, if successful, will

```

class Tree {
    protected int value;
}

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

class Node extends Tree {
    private Tree left, right;
    public Node (int v, Tree l, Tree r) { value = v; left = l; right = r; }
    public Node$ (+int v, +Tree l, +Tree r) { v = value; l = left; r = right; }
}

```

**Fig. 3.** Class Definitions for a Binary Tree

invoke the deconstructor method. A deconstructor is written as the name of the class followed by a dollar sign, as shown in Figure 3. Typically the deconstructor uses by-name parameters to pass the values back to the operator invocation. (For an alternative approach to pattern matching in a Java extension, see [21].)

The following function computes the sum of the values in a binary tree, and illustrates the use of the pattern matching operations.

```

public int sum (Tree t) {
    int value;
    Tree left, right;
    if (t instanceof Node(value, left, right))
        return value + sum(left) + sum(right);
    else if (t instanceof Leaf(value))
        return value;
    return 0;
}

```

## 1.6 Operator Overloading

Each of the standard operators is assigned a textual name (plus for +, times for \*, and so on). If the left argument to a standard operator is a class type, then a search will be performed in this class to see if a method has been defined using this textual name, and with a type signature that matches the right argument. If such a method is found then the operator is turned into a standard method invocation. We will see an example of this in the next section. This is the technique we used previously in Leda [7], as well as other languages, such as Python [4]. It is also in keeping with a proposed change to Java [13].

## 1.7 Logic Programming

The combination of by-name parameter passing and functions as values permits a simple implementation of logic programming. Students often have very limited exposure to this useful programming style; often viewing it as merely an obscure practice found only in marginal languages, such as Prolog [10]. Logic programming is sometimes termed *relational programming*, and a classic example is a database of family relations.

Logic programming in J/mp is provided using a supporting class named `Relation`. A relation can be thought of as a generalization of booleans, however the more accurate class definition is shown in Figure 4. The fundamental operation on a relation is `apply`, which takes as argument another relation, and returns a boolean which indicates that both the receiving relation and the argument relation can be satisfied.

Another fundamental operation with relations is the unification operator, named `unify`. `Unify` has two jobs. If both arguments are defined, it ensures that they are equal before testing the relation it is given. Otherwise, if the first argument is undefined, it is set to the second, and then the argument relation is tested. If the argument (the continuation) fails, the assignment is undone, and the unification fails. (Technically, this is not as general as the Prolog style unification, where either or both arguments can be undefined. However, it is sufficient for most purposes, and if desired the more general operation can be written using this simpler version as a basis).

An example to illustrate how these operations can be used is the following:

```
private static Relation eq(+String a, String b) { return Relation.unify(a, b); }

public static Relation progeny(+String f, +String m, +String c) {
    return (eq(f, "Albert") && eq(m, "Victoria") && eq(c, "George"))
    || (eq(f, "George") && eq(m, "Elizabeth") && eq(c, "Elizabeth"))
    || (eq(f, "Phillip") && eq(m, "Elizabeth") && eq(c, "Charles"))
    || (eq(f, "Charles") && eq(m, "Diana") && eq(c, "William"))
    || (eq(f, "Charles") && eq(m, "Diana") && eq(c, "Henry"));
}
```

Here the method `eq` simply provides a convenient shorthand for the invocation of the unification method. The method `progeny` is a typical logic programming database. It takes three by-reference parameters; representing a father, mother and child in the progeny relationship.

Queries in the logic programming style in J/mp are driven by either an `if` or a `while` statement. The conditional `if` seeks a single solution to a given query, while the looping `while` statement seeks all possible bindings. If the actual arguments are already defined they are used as a pattern, and if not defined they are set by the call on the relation. Assuming that `a` and `b` are variables of type `String` that are initially null, the name of a single child of Phillip, for example, could be determined as follows:

```

class Relation {
    public boolean apply (Relation continuation) { return true; }

    public boolean asBoolean() { return apply(new Relation()); }

    public static Relation unify (+Object left, final Object right) {
        return new Relation () {
            public boolean apply (Relation continuation) {
                if (left == null) {
                    left = right;
                    if (continuation.apply(new Relation())) return true;
                    left = null;
                } else if (right != null) {
                    return left.equals(right) && continuation.apply(new Relation());
                }
                return false;
            }
        };
    }

    public Relation and (+Relation right) {
        final Relation me = this;
        return new Relation () {
            public boolean apply (final Relation continuation) {
                return me.apply(new Relation() {
                    public boolean apply(Relation r) {
                        return right.apply(continuation);
                    }
                });
            }
        };
    }

    public Relation or (+Relation right) {
        final Relation me = this;
        return new Relation () {
            public boolean apply (final Relation continuation) {
                return me.apply(continuation) || right.apply(continuation);
            }
        };
    }
}

```

**Fig. 4.** Definition of Relation

```
if (progeny("Phillip", a, b))
    System.out.println("child of Phillip " + b);
```

A characteristic of logic programming is that the same parameters sometimes serve as input and other times as output. For example, the father of Charles can be determined as follows:

```
if (progeny(c, d, "Charles"))
    System.out.println("father of Charles " + c);
```

A while statement can be used to cycle through all bindings of the argument values, as follows:

```
while (progeny("Charles", "Diana", e))
    System.out.println("children of Charles and Diana " + e);
```

New relations are easily defined using existing relations. The definitions for `parentOf` and `grandParentOf`, for example, are as follows:

```
public static Relation parentOf (+String p, +String c) {
    String a; // either father or mother
    return progeny(p, a, c) || progeny(a, p, c);
}

public static Relation grandParentOf (+String g, +String c) {
    String a;
    return parentOf(g, a) && parentOf(a, c);
}
```

Using these one could discover, for example, the grandparent of William using a query such as the following:

```
if (grandParentOf(f, "William"))
    System.out.println("grand Parent of William " + f);
```

It is also easy to integrate conventional boolean expressions with relations, however the details are not shown here.

## 2 Implementation

Like Pizza and other similar systems [2, 3, 6, 12, 19, 21], J/mp is implemented as a source to source translation system. That is, J/mp programs are first processed into equivalent Java programs, which are then compiled using the standard Java

system. In this section we will describe the most significant features of this transformation process.

As was done by the developers of the Pizza system, a function type is represented internally as an interface. A mangling algorithm is used to convert the function type signature into a unique name. (A technique that historically has been used in many different languages [15]). The following, for example, represents a function that takes an integer as argument, and returns a double as result:

```
interface double__int {
    public double call (int $0);
}
```

The simple signature encoding algorithm we use is adequate for primitive types, and for most class types, which are represented simply by the class name. It is legal for a Java program to contain two identically named classes, as long as they come from different packages. This would potentially cause problems with finding unique names, but such examples do not appear to arise in practice.

Similarly, one might believe there could be a subtle interaction between inheritance and assignment of function values (the covariant and contravariant issue for method overriding but now appearing in a different guise). For example, assuming that class `Child` is a subclass of `Parent`, would it ever make sense to assign a function with type signature `void(Child)` to a variable declared as maintaining `void(Parent)`, or vice versa? Fortunately, it is easy to show that such an action, if allowed, could potentially result in non-detectable type errors. This being so, the fact that this behavior is ruled out by the simple expedient of type signature encoding is a benefit, and not a problem.

Using the conversion of function types to interfaces, the second version of Jensen's device (Figure 2) is translated internally into the following Java definition:

```
public class Jensen implements double__int_double__int {
    public double call (int max, double__int f) {
        int i;
        double sum = 0.0;
        for (i = 0; i < max ; i++)
            sum += f.call(i);
        return sum ;
    }
}
```

The use of the interface means that any function that takes an integer as argument and returns a double (that is, any class that implements the interface `double__int`) can be passed as argument to this function.

By-name parameters are implemented by means of an intermediary object that is responsible for accessing and setting a value. In the context of parameter passing such an object is traditionally known as a *Thunk* [16]. A portion of the definition of class `Thunk` is as follows:

```
class Thunk {
    public Object get() { return null; }
    public Object set(Object x) { return null; }

    public int getInt() { return 0; }
    public int setInt(int a) { return a; }
    public double getDouble() { return 0.0; }
    public double setDouble(double a) { return a; }
    ...//
}
```

(As an alternative we could have implemented different classes, one for each primitive data type, but instead elected to define just a single class for all values). Using this class, the implementation of our first definition of Jensen's device (Figure 1) is as follows:

```
public class Jensen implements double__int_int_$double {
    public double call (Thunk i, int max, Thunk x) {
        sum = 0.0 ;
        for (i.setInt(0); i.getInt()<max ; i.setInt(i.getInt()+1))
            sum +=x.getDouble();
        return sum ;
    }
}
```

The majority of work involved in passing parameters by-name is incurred on the calling side, which must create the `Thunk`:

```
class Main {
    static double [ ] data = {1.5 ,2.7 ,3.2 ,4.1 ,5.2 ,6.3 };

    static public void main (String [ ] args ) {
        class Context$2 {
            int i;
        }
        final Context$2 context$3 = new Context$2() ;
        System .out .println ("Sum " + new Jensen ().call (new Thunk() {
            public int getInt () { return context$3 .i ; }
            public int setInt (int ThunkVar$4) {
                return context$3.i = ((int) (ThunkVar$4));
            }
        }));
    }
}
```

```

    } } ,new Thunk() {
    public double getDouble () {
        return data [context$3.i];
    } });
}
}

```

An anonymous inner class is used to create the Thunk. Since anonymous classes in Java do not capture their complete surrounding environment [1] (that is, they are not true closures), it is necessary to create our own data type to maintain the context needed by the Thunk. For this purpose we define an inner class (here named Context\$2) and replace all references to the local variable with references to the context. The anonymous Thunk class then redefines the appropriate get and set methods. If a non-assignable expression is passed as argument only the get method is defined, so that a set on the corresponding formal parameter will have no effect.

Nameless function values (lambda functions) are also implemented using the ability for Java to form anonymous inner classes. Like thunks, they must also capture their surrounding environment in a closure. The main function used in the second Jensen program, for example, is implemented as follows:

```

class Main {
    static double [ ] data = {1.5 ,2.7 ,3.2 ,4.1 ,5.2 ,6.3 };

    static public void main (String [ ] args) {
        System .out .println ("Sum of odds " +new Jensen().call(3,
        new double__int () {
            public double call (int j) { return data[1 +2 *j]; }
        }));
    }
}

```

Pattern matching is implemented using the existing instanceof operator, type-casting and message passing. Deconstructors are converted into a method that returns boolean true, while a statement such as:

```
t instanceof Node(value, left, right)
```

is converted into

```
(t instanceof Node) && ((Node) t).Node$(value, left, right)
```

The backtracking mechanism used by the logic programming system is provided entirely by the methods and and or, shown earlier in Figure 4. (A detailed explanation of how these less-than-obvious functions actually operate can be

found in [7]). To convert a relation to a boolean in the context of an if statement, it is only necessary to append an invocation of `asBoolean`. In this fashion an if statement described earlier in the paper is translated into:

```
if (progeny(c, d, "Charles").asBoolean())
    System.out.println("father of Charles " + c);
```

(This is ignoring the creation of `Thunks` for the parameters `c` and `d`, which is a separate operation). The transformation of a `while` statement is more complex. The statement portion of the loop is transformed into a relation that always fails, which is then passed to the test portion. Since the relation fails, the backtracking system will automatically cycle through all possibilities. Ignoring the creation of `Thunks`, the `while` loop given in the earlier discussion of logic programming is implemented as follows:

```
progeny("Charles", "Diana", e).apply (new Relation() {
    public boolean apply (Relation w) {
        System.out.println("children of Charles and Diana " + e);
        return false;
    }});
```

An unfortunate side effect of this approach is that `return` statements cannot be used within such a loop, as the control would transfer out of the method inside the artificially generated nested class, and not out of the original surrounding function.

An alternative approach to implementing Prolog in Java is given by Engel [12].

### 3 Future Investigations

More than just a static language, we view `J/mp` as a framework for experimenting with language features. `J/mp` is implemented using the antlr compiler-compiler generation system<sup>1</sup>. `J/mp` files are first converted into the standard antlr tree-based representation; then transformations are performed at the intermediate representation level, before transforming the tree back into a conventional Java program. This allows for an extremely flexible and efficient system, making it easy to incorporate new changes.

A considerable percentage of the current effort involves improving the implementation. Two examples of this are final detection and closure elimination. Parameter values will be captured automatically in closures if they are declared as `final`, but programmers seldom bother with this modifier. If we can determine when a parameter can be so declared we can automatically insert the modifier.

---

<sup>1</sup> See [www.antlr.org](http://www.antlr.org).

Similarly we need to refine our algorithm for determining when closures are necessary; the current algorithm is overzealous, creating them even when they are not needed.

In addition to finding novel uses for the programming mechanisms we have described in this paper, there are a large number of alternative programming features we are considering, to determine if they meet our requirements for utility and maintaining the feel of the language. These include the following:

- The addition of generics [6]. Generic classes will be included in the next release of Java (version 1.4). Our interest is in the effect of generics on free standing functions, and the use of bounded generics.
- Type inference. It is sometimes annoying to have to declare and type variables when they appear in expressions, particularly if they are assigned only once. So it would be useful if the system could in some cases infer the type of otherwise undeclared variables. One current modest proposal is to combine this feature with the `final` keyword, so that an assignment such as

```
final x = expression
```

would have a type determined from the expression part, and not need an explicit type declaration.

- Retroactive abstraction by on-the-fly generation of adapters (sometimes known as structural subtyping). Oftentimes it is necessary to combine two class hierarchies that have a similar interface, but do not explicitly share a common ancestor.

```
class OpenLookObject {
    public void display () { ... }
    public int move (int x, int y) { ... }
}
```

```
class MotifObject {
    public void display () { ... }
    public int move (int x, int y) { ... }
}
```

This can be addressed by first creating an interface for the common operations:

```
interface XWindowsObject {
    public void display ();
    public int move (int x, int y);
}
```

We may not be allowed to change the original classes. However, when an instance of the original class is assigned to a variable declared as the interface, we can implicitly construct an adapter. That is, in place of:

```
OpenLookObject v = new OpenLookObject();
...
XWindowsObject xobj = v;
```

We can generate:

```
final OpenLookObject $save = v; // necessary to save context
XWindowsObject xobj = new XWindowsObject() {
    public void display () { $save.display(); }
    public int move (int x, int y) { return $save.move(x, y); }
}
```

- Mixins. Conventional classes provide inheritance of behavior and polymorphic assignment (the assignment of a child value to a parent variable). Interfaces provide polymorphic assignment, but no inheritance of behavior. The third option is the inheritance of behavior, but no assumption of assignability. Such a feature is termed a *mixin* [5]. (A mixin is in this sense very similar to a private inheritance in C++ [8]). Our proposal is to add a third category of modifier to the class heading, *incorporates*. A class that *incorporates* from one or more classes will inherit behavior, but not be allowed any rights of substitution:

```
class A extends B incorporates C implements D {
    ... // inherits behavior from both B and C
}
```

```
B b = new A(); // allowed
D d = new A(); // allowed
C c = new A(); // not allowed
```

To implement this feature in current Java an instance of *C* would be created within each instance of *A*, and the invocation of methods inherited from *C* would be implemented in *A* using this object. Difficulties are dealing with constructors, and the possibility that *A* might want to override methods inherited from *C*. The latter can be handled using inner classes, although the details are somewhat complicated.

- Automatic boxing and unboxing. Primitive types (integer, real and so on) are not true objects in Java. Thus when objects are required (for example, in a data structure) such values must be wrapped in a special class, such as *Integer*. The language C# hides this requirement from the user, by automatically boxing primitive values as objects when necessary, and unboxing

them back into primitives when no longer required [18]. We are exploring how difficult it would be to add this feature to Java.

- Multimethods. Although the receiver is used for dynamic dispatch in Java, it is the static type of argument values that determine the method binding in an overloaded method. If `Child` is a subclass of `Parent`, the following, perhaps surprisingly, will execute the parent method both times, instead of the child method in the second instance:

```
class Test {
    public void test (Parent p) { ... }
    public void test (Child c) { ... }

    public void main () {
        Parent a = new Parent();
        Parent b = new Child();
        test(a);
        test(b); // dynamic type has no effect
    }
}
```

Our proposal is to allow methods with compatible argument type signatures to be combined using the vertical bar (the or operator). Subsequent methods would omit any modifiers or return type, which are not allowed to change.

```
class Test {
    public void test (Parent p) { ... }
    | test (Child c) { ... }
    ...
}
```

The first method would be augmented with the addition of dynamic dispatching code, while the remaining methods are translated as normal. This allows subclasses to override the behavior of such methods, something that is not always possible in other schemes that have been proposed for handling multimethods [3, 9, 21].

- Open classes. Craig Chambers recently described a technique to help circumvent the problem that classes permit easy extension through the addition of new datatypes, while functions permit easy extension through the addition of new operations [9]. His technique allows the addition of new methods to existing classes, using a method heading syntax similar to the following:

```
void Point.print() { ... }
void ColoredPoint.print() { ... }
```

Dispatch on such functions is more complicated than either dispatch on regular methods or dispatch on functions, but appears to be manageable.

## 4 Conclusions

In this paper we have introduced J/mp, which is a multiparadigm extension to Java designed to support programming in the functional and logic programming styles, in addition to the object-oriented style of Java. As with our earlier language Leda [7], our goal in developing J/mp has been to support a natural syntax that requires minimal additions to the base language, and retains the important feel of the original language. Of course, “natural” and “feel” are subjective terms, and hence an evaluation of our success or failure will only come with experience in developing systems using our language extensions.

Currently, J/mp extends Java through the addition of the following facilities:

- Functions as first class values.
- By-name parameters.
- Operator overloading.
- Pattern matching.
- A library for relational (or logic) programming.

Current efforts involve work on the implementation, exploring novel uses for these language features, and exploring new language features that might expand the flexibility or expressiveness of the language, while maintaining the feel of the original Java.

Further information on J/mp, including source code and installation instructions, are available on the authors web site <http://www.cs.orst.edu/~budd/jmp>.

## References

- [1] Ken Arnold, James Gosling and David Holmes, *The Java Programming Language*, 3rd Edition, Addison-Wesley, Reading, MA, 2000.
- [2] Jonathan Bachrach and Keith Payford, “The Java Syntactic Extender”, *Sigplan Notices*, 36(11): 31-42, November 2001.
- [3] Gerald Baumgartner, Martin Jansche, and Christopher D. Peisert, “Support for Functional Programming in Brew”, *Proceedings of the 2001 workshop on Multiparadigm Programming with Object-Oriented Languages*, John von Neumann Institute for Computing, Jülich, Germany, 2001.
- [4] David M. Beazley, *Python Essential Reference*, New Riders Publishing, Indianapolis, IN, 2000.
- [5] Gilad Bracha and William Cook, “Mixin-Based Inheritance”, *Proceedings of the 1986 OOPSLA—Conference on Object-Oriented Programming Systems, Languages and Applications*; Reprinted in *Sigplan Notices*, 25(10): 347–349, 1990.
- [6] Gilad Bracha, Martin Ordersky, David Stoutamire, Philip Adler, “Making the Future Safe for the Past: Adding Genericity to the Java Programming Language,” in *Proceedings of the 1998 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1998, reprinted as *Sigplan Notices*, 33(10):183-200, October 1998.

- [7] Timothy A. Budd, *Multiparadigm Programming in Leda*, Addison-Wesley, Reading, MA, 1995.
- [8] Timothy A. Budd, *An Introduction to Object-Oriented Programming*, 3rd Edition, Addison-Wesley, Reading, MA, 2002.
- [9] Curtis Clifton, Gary Leavens, Craig Chambers, and Todd Millstein, "MultiJava: Modular Symmetric Multiple Dispatch and Open Classes for Java," in *Proceedings of the 2000 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2000.
- [10] William F. Clocksin and Christopher S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1981.
- [11] Edsger W. Dijkstra, "Defense of ALGOL 60," *Communications of the ACM*, 4(11):502-503, November 1961.
- [12] Joshua Engel, *Programming for the Java Virtual Machine*, Addison-Wesley, Reading, MA, 1999.
- [13] James Gosling, "The evolution of numerical computing in Java," Sun Microsystems Laboratories, <http://java.sun.com/people/jag/FP.html>.
- [14] James Gosling, "The Feel of Java", talk at 1996 ACM Conference on Object-Oriented Programming Languages and Applications, 1996.
- [15] Richard G. Hamlet, "High-level Binding with Low-Level Linkers," *Communications of the ACM*, 19:642-644, November 1976.
- [16] Peter Zilahy Ingerman, "Thunks", *Communications of the ACM*, 4(1):55-58, 1961.
- [17] Donald E. Knuth, "The Remaining Troublespots in Algol 60", *Communications of the ACM*, 10(10):611-617, 1967.
- [18] Stanley B. Lippman, *C# Primer*, Addison-Wesley, Reading, MA, 2002.
- [19] Sean McDirmid, Matthew Flatt, Wilson Hsieh, "Jiazzi: New-Age Components for Old-Fashioned Java," in *Proceedings of the 2001 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.
- [20] Peter Naur et al. "Report on the Algorithmic Language Algol 60", *Communications of the ACM*, 6(1):1-17, 1963.
- [21] 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.