

Dagstuhl Workshop

Multiparadigm Extensions to Java

Timothy Budd
Oregon State University

Multiparadigm Extensions to Java

Background

In 1995 my students and I developed Leda

- Based on Pascal
- Friendly syntax
- Worked naturally in functional, object-oriented, or logic programming style
- Current project, try to do the same for Java

Features we have Added

We want a language that has the Java look and feel, and yet slips in some interesting language features.

- Unboxing. Simple compositional unpackaging.
- First Class Functions. Starting point for many interesting programming techniques.
- Pass-by-name Parameters. A tool for delay in evaluation
- Operator Overloading.
- Relation data type, and associated libraries for doing Logic programming.

Boxing and UnBoxing

Since you aren't allowed to change state, but still want a rich set of values, in other functional languages a lot of work is performed by pulling items together into a new entity, and breaking them apart to get their basic parts.

Sometimes called boxing and unboxing.

Can use constructors for boxing. How to do unboxing?

A Typical Type

```
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 { }
```

A Typical Boxing

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

Note: in Leda we get rid of the new operator, which makes it even nicer. But v that would be too radical for Java programmers.

A Typical Unboxing

```
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");
```

How it is Done

`t instanceof Leaf(val)`

could be translated as follows:

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

But of course, you don't have the comma operator in Java, so we fake it a little

Functions as First Class Values

Some examples:

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

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

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

Behind the Scenes

Similar to Wadler's system. Function types become interfaces (which are the closest thing to a type in Java). Function values become implementations of an interface. Must capture local variables and parameters in a closure. This can be tricky.

Pass By Name

Need a mechanism to delay evaluation of argument until used. Lots of different languages do this. We elected to revive an old idea, pass by name. (Originally found in APL)

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

...

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

Implemented using a Thunk

```
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; }});
```

Operator Overloading

Nothing exciting here. We just wanted the ability to give “and” and “or” new meaning and didn’t want to build them into the language.

As a practical matter, performed by overloading a function with a special name “plus” for +.

Logic Programming

Can build logic programming out of the following pieces:

- The relation data type (relation, a function that takes a function as argument and returns a boolean – but you can think of it just as a boolean)
- Pass-by-name so that information can flow both into and out of a parameter
- A unification function (ensures two arguments equal to each other, perhaps by changing one to make this so)
- Overloaded meanings for “and” and “or” to make a backtracking system

A Typical Relation

```
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"));  
}
```

By name arguments allow information to flow either in or out through the same parameter value. Unify tries to make arguments the same. If an entire clause is satisfied, unify undoes its assignments and the next clause is tried.

Invoking a Relation

```
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);
```

(If we introduced implicit conversions could get rid of the method `asBoolean`)

Getting Multiple Solutions out of a Relation

Bundle up whatever action you want to do with the answers, and pass them to
relation:

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

(Of course, assumes we have solved the problem of closures).

Current Status

- Language is defined.
- Work is progressing (far too slowly) on the implementation.
- Not ready for prime-time yet.