

# **MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java**

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein

TR #00-06a

April 2000, Revised July 2000

To appear in *OOPSLA 2000*, Minneapolis, Minnesota, October 2000.

# Outline

- What is MultiJava?
- Openclass and MultiJava
- Symmetric multiple dispatch and MultiJava
- MultiJava source compilation into bytecode
- Related work
- Future work

# What is MultiJava?

- Based on *Dubious Language project*
- Backward-compatible extension to Java
- Supports
  - *open classes*
  - *symmetric multiple dispatch*

# Openclass and MultiJava

- openclass
  - allows to add a method into a class without direct editing of a class
  - allows clients to customize their ifaces to needs of the client's applications
  - does not require existing code to change to create a subclass of referenced class
  - replaces “Visitor pattern”

# Visitor pattern

- deals with adding operations into a class hierarchy
  - idea: reify each operation into a class, thus they create a hierarchy
    - e.g., parsing an arithmetic expression, tree, ...
  - another approach is to create subclasses and/or add methods, but it requires changes into class hierarchy

# Visitor pattern

- drawbacks
  - double-dispatching code is tedious to write and prone to error
  - the need for the Visitor pattern must be anticipated
  - it gives up the ability to add new subclasses to existing classes
    - visitors have to be modified to operate on a new subclass

# MultiJava - Top-Level methods

- definition (e.g., `typeCheck.java`):

```
// compilation unit "typeCheck"
```

```
package oopsla.examples;
```

```
// Methods for typechecking
```

```
public boolean Node.typeCheck() { .... }
```

- usage:

```
rootNode.typeCheck()
```

- example presents

- definition of the *generic function* **typeCheck**

- implementation of the **typeCheck** function

# MultiJava - Top-Level methods

- Top-Level methods
  - can be “external” method
    - not defined in the class compilation unit
  - can be “internal” method
    - defined in its class definition
  - scoping
    - they have own FQN
      - e.g., `oopsla.examples.typeCheck`

# MultiJava - Top-Level methods

- replace Visitor pattern
- encapsulation
  - can use all access attributes (`private/...`)
  - external TL method accesses
    - public members of receiver class
    - non-private members if they are in the same package
  - internal TL method accesses
    - the same rights as members of its receivers class

# MultiJava - Top-Level methods

- restrictions of TL methods
  - external
    - cannot be `abstract`
    - cannot be added to interfaces
  - both
    - T-L must belong to
      - » generic function in the same compilation unit
      - » or must be `internal`
    - cannot be `static`
- TL can
  - use `this`

# Symmetric multiple dispatch and MultiJava

- multiple dispatch
  - found in Common Lisp
    - method invocation depends on
      - pure Java
        - receiver class type and name
      - MultiJava
        - receiver class type, name and argument types
        - such method is called **multimethod**

# Symmetric multiple dispatch and MultiJava

- Multiple dispatch
  - *symmetric* if the rules for method lookup treat all dispatched arguments identically
  - *asymmetric* multiple dispatch typically uses lexicographic ordering, where earlier arguments more important

# Symmetric multiple dispatch and MultiJava (example)

```
public class Shape {
    public boolean intersect(Shape s) { /* ... */ }
}
public class Rectangle extends Shape {
    public boolean intersect(Rectangle r) {
/* efficient code for two Rectangles */
    }
}
```

- method `Rectangle.intersect` does not override `Shape.intersection`
  - violates contravariant typechecking rule for functions

# Symmetric multiple dispatch and MultiJava (example cont.)

```
Rectangle r1, r2;  
Shape s1, s2;  
boolean b1, b2, b3, b4;  
r1 = new Rectangle( /* ... */ );  
r2 = new Rectangle( /* ... */ );  
s1 = r1;  
s2 = r2;  
b1 = r1.intersect(r2); // Rectangle.intersection is called  
b2 = r1.intersect(s2); // Shape.intersection is called  
b3 = s1.intersect(r2); // Shape.intersection is called  
b4 = s1.intersect(s2); // Shape.intersection is called
```

# Symmetric multiple dispatch and MultiJava (solution in Java)

```
public class Rectangle extends Shape {
    /* ... */
    public boolean intersect(Rectangle r) {
        /* efficient code for two Rectangles */
    }
    public class Rectangle extends Shape {
        /* ... */
        public boolean intersect(Shape s) {
            if (s instanceof Rectangle) {
                Rectangle r = (Rectangle) s;
                // efficient code for two Rectangles
            } else {
                super.intersect(s);
            }
        }
    }
}
```

# Symmetric multiple dispatch and MultiJava (solution in MultiJava)

- adding a new Formal Parameter

– *Type @ ClassType VariableDeclaratorId*

```
public class Rectangle extends Shape {  
    /* ... */  
    public boolean intersect(Shape@Rectangle r) {  
        /* efficient code for two Rectangles */  
    }  
}
```

# Symmetric multiple dispatch and MultiJava (solution in MultiJava cont.)

```
public class Circle extends Shape {
    /* ... */
    public boolean intersect(Shape s) {
        /* code for a Circle against any Shape */
    }
    public boolean intersect(Shape@Rectangle r) {
        /* efficient code against a Rectangle */
    }
    public boolean intersect(Shape@Circle c) {
        /* very efficient code for two Circles */
    }
}
```

# Symmetric multiple dispatch and MultiJava (solution in MultiJava cont.)

- super problem
  - in Java `super.xxx()`
    - invoke overridden method
  - in MultiJava `super.intersection(...)`
    - should be invoked overridden method or “matching” method in the receiver’s class?
      - if the target generic function is the same as the sender’s, the overridden method is invoked
      - otherwise available generic function is called

# MultiJava source compilation into bytecode

- no changes to Java bytecode
- internal generic functions

```
public class Square extends Rectangle {  
    public boolean intersect(Shape@Rectangle r) { /* method 1 body */ }  
    public boolean intersect(Shape@Square s) { /* method 2 body */ }  
}
```

## TRANSFORMED TO

```
public class Square extends Rectangle {  
    // the "intersect" dispatch method  
    public boolean intersect(Shape r) {  
        if (r instanceof Square) {  
            Square s_ = (Square) r;  
/* method 2 body, substituting s_ for s */  
            } else if (r instanceof Rectangle) {  
                Rectangle r_ = (Rectangle) r;  
/* method 1 body, substituting r_ for r */  
            } else {  
                return super.intersect(r);  
            }  
        }  
    }  
}
```

# MultiJava source compilation into bytecode

- external generic functions

```
/* compilation unit "rotate" */
public Shape Shape.rotate( float a) { /* method 3 body */ }
public Shape Rectangle.rotate( float a) { /* method 4 body */ }
public Shape Square.rotate(float a) { /* method 5 body */ }

TRANSFORMED TO

public interface rotate$rotate$d { // type of a dispatcher object in this example
    Shape apply(Shape this_, float a);
}

public class rotate$rotate$anchor { // an anchor class
    public static rotate$rotate$d function = new rotate$rotate$dispatcher();
// an inner class implementing a dispatcher object
    private class rotate$rotate$dispatcher implements rotate$rotate$d {
        public Shape apply(Shape this_, float a) {
            if (this_ instanceof Square) {
                Square this2_ = (Square) this_;
/* method 5 body, substituting this2_ for this */
            } else if (this_ instanceof Rectangle) {
                Rectangle this2_ = (Rectangle) this_;
/* method 4 body, substituting this2_ for this */
            } else {
/* method 3 body, substituting this_ for this */
            }
        }
    }
}
}
```

# Related work

- typechecking restrictions in Dubian language
- Encapsulated multimethods design for adding asymmetric multimethods
- Aspect-oriented programming

# Future work

- implementation of MultiJava compiler
- extensions to T-L methods (static,...)
- studying MultiJava expressiveness