Why multimethods?

In object-oriented languages the version invoked at run-time depends on the dynamic type of the receiver but not of the arguments

This can be unnatural, for example in binary methods

```
class Shape {
  boolean intersect (Shape s) { ...
    //generic code for two Shapes
  }
}

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

The new method does not override the original

Recall that covariant rule would be unsafe:

```
Shape s1 = new Rectangle(...);
Shape s2 = new Shape(...);
s1.intersect(s2)
```

run time error

Java and C++ consider this overloading

The two `intersect` methods belong to different generic functions

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

not the desired semantics

In Java it is possible to solve this problem by explicit run-time type tests

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

Semantics is ok

BUT:

- the programmer has to explicit code the search for what intersection algorithm to execute (tedious and error-prone)

- code is not easily extensibile (when a new subclass of `Shape` is added the method must be modified)

Another solution: using double dispatching (Visitor pattern)

```
class Shape {
  boolean intersect (Shape s) { s.intersectShape(this); }
  boolean intersectShape (Shape s) {
    //generic code for two Shapes
  }
  boolean intersectRectangle (Rectangle r) {
    //generic code for two Shapes
  }
class Rectangle extends Shape { ...
  boolean intersect (Shape s) { s.intersectRectangle(this); }
  boolean intersectRectangle (Rectangle r) {
    //efficient code for two Rectangles
  }
```

BUT:

- the programmer has to explicit code the double-dispatch (even more tedious)

- still not completely modular, since at least the root class must be modified when a new subclass is added

Multimethods

Methods with dynamic binding for all arguments

```
class Rectangle extends Shape { ...
  boolean intersect (Shape@Rectangle r) {//Shape static type
  //but executed only when argument has dynamic type Rectangle
  //efficient code for two Rectangles
}
```

Semantics: for each method `m(T_1[@S_1] ... T_n[@S_n])` in `S_0`

its specializers are `S_0 S_1 ... S_n`

no specializer = implicit specializer `T_i`

`e_0.m(e_1,...,e_n)`

overloading resolution as usual: assume `m(T_1 ... T_n)`

at run-time assume dynamic types `C_0 m(C_1 ... C_n)`

the most specific among those methods

`m(T_1[@S_1] ... T_n[@S_n])` in `S_0`

with specializers applicable to the dynamic types, i.e., `C_i` $\leq$ `S_i` for all $i = 1..n$

In the example:

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

Shape s1;
Shape s2;
...
boolean b = s1.intersect(s2);
```

compile time: method `intersect(Shape)` is selected

run-time: if `s1`, `s2` have both dynamic type `Rectangle`, specializers `Shape Shape` and `Rectangle Rectangle` are applicable, second is more specific

otherwise only `Rectangle Rectangle` is applicable

Important: generalization of standard Java dynamic binding for receiver; programs with no @ specializers behave as regular Java programs (also semantics of overloading is preserved)

Problems

No applicable method = incomplete function

```
class Shape {
  boolean intersect (Shape@Rectangle s) { ... }
}

Shape s1 = new Shape(...);
Shape s2 = new Shape(...);
...
boolean b = s1.intersect(s2);
```

No most-specific applicable method = ambiguous function

```
class Test {
  boolean test (Shape@Rectangle s1, Shape s2) { ... }
  boolean test (Shape s1, Shape@Rectangle s2) { ... }
}
Shape s1 = new Rectangle(...);
Shape s2 = new Rectangle(...);
...
boolean b = s1.intersect(s2);
```

Incomplete and ambiguous functions will be rejected by the type system

Another example

```
class Shape {
  boolean intersect (Shape s) {...} //SS
}
class Circle extends Shape {
  boolean intersect (Shape s) {...}// CS code for a Circle against any Sh
  boolean intersect (Shape@Rectangle r) {...} //CR eff. code against a Re
  boolean intersect (Shape@Circle c) {...} //CC very eff. code for 2 Circ
}

Shape s1, s2;
...
s1.intersect(s2)
```

if dynamic type of `s1, s2` is `Circle`, then SS, CS, CC are applicable, CC selected

if dynamic type of `s1` is `Circle`, of `s2` is `Triangle`, then SS, CS are applicable, CS is selected

if dynamic type of `s1` is `Rectangle`, of `s2` is `Circle`, then only SS is applicable

Super Sends

In Java a super send allows to

- invoke the directly overriden method
- invoke a method in a different generic function

In MultiJava a method can override many others even in the same class and there is no unique directly overriden method

Solution:

- first statically identify for each super send whether it invokes a method from the same generic function
- if yes, the semantics is as in standard MultiJava send except that the applicable methods are those overriden by the sending method
- if not, the applicable methods are those belonging to the superclass

Example

```
class Shape {
  boolean intersect (Shape s) { ... }//SS
}
class Circle extends Shape {
  boolean intersect (Shape s) {...}//CS
  boolean intersect (Shape@Rectangle r) {...} //CR
  boolean intersect (Shape@Circle c) { ... super.intersect(c) ... }
}
```

compile time: it is a send to `intersect(Shape)`

run-time: SS, CS applicable, CS selected

```
class Circle extends Shape { ...
  boolean intersect (Shape@Circle c) {
    ... super.specialIntersect(c) ...
  }
}
```

compile time: it is a send to `specialntersect(Shape)`

run-time: only methods of `specialntersect(Shape)` belonging to Shape are applicable

Other uses of multimethods

```
class Shape {
  ...
  void displayOn (OutputDevice d) {
    //default display of shape
  }
}
class Rectangle extends Shape {
  void displayO (OutputDevice d) {
    //default display of rectangle
  }
  void displayOn (OutputtDevice@XWindows d) {
    //special display of rectangle on X Windows
  }
  void displayOn (OutputtDevice@FastHardware d) {
    //special display of rectangle using hardware support
  }
}
```

Type system

- on method sends overloading resolution is as in Java except that in `super` sends also methods in the same class must be considered

- on individual method declarations each specializer `S` must be a subtype of the associated static type (otherwise no effect or no applicability)

- on entire generic functions `m (T_1 ... T_n)` for each possible argument types `C_1 ... C_n` there is a most-specific applicable method (considering visible types and methods inside the compilation unit)

- morever there is a problem caused by abstract classes when putting together compilation units (see next slide)

Incompleteness problem with abtract classes and multimethods

```
package My
public abstract class Picture {
  public abstract boolean similar (Picture p);
}

import My.Picture;
public class JPEG extends Picture {
  public boolean similar (Picture@JPEG j ) {...}
}

import My.Picture;
public class GIF extends Picture {
  public boolean similar (Picture@GIF g) {...}
}
```

run-time error with an argument of type `JPEG` and one argument of type `GIF`

Restriction: for each non-receiver position non-concrete visible subtypes of the static type must be considered possible argument types