Efficient Multimethods in a Single Dispatch Language

Brian Foote, Ralph E. Johnson and James Noble

Dept. of Computer Science, University of Illinois at Urbana-Champaign
201 N. Goodwin, Urbana, IL 61801, USA
foote@cs.uiuc.edu johnson@cs.uiuc.edu

School of Mathematical and Computing Sciences, Victoria University of Wellington
P.O. Box 600, Wellington, New Zealand
kjx@mcs.vuw.ac.nz

Abstract. Smalltalk-80 is a pure object-oriented language in which messages are dispatched according to the class of the receiver, or first argument, of a message. Object-oriented languages that support multimethods dispatch messages using all their arguments. While Smalltalk does not support multimethods, Smalltalk's reflective facilities allow programmers to efficiently add them to the language. This paper explores several ways in which this can be done, and the relative efficiency of each. Moreover, this paper can be seen as a lens through which the design issues raised by multimethods, as well as by using metaobjects to build them, can be more closely examined.

1. Introduction

The designers of object-oriented languages usually consider multimethods and single dispatch to be competing alternatives. This paper describes a variety of ways to implement multimethods in single-dispatch languages such as Smalltalk. It is not surprising that multimethods can be implemented in Smalltalk, because it is a reflective language that has been extended in many ways. However, it is surprising how well multimethods can work with single dispatch. This paper develops a simple extended syntax that makes it easy to mix multimethods and normal methods. The semantics of multimethods are simple, they have no syntactic or performance cost if they are not used, they interoperate well with Smalltalk's metaobjects, and they are as efficient to execute as comparable hand-written code.

Our results show that there is no inherent conflict between multi-methods and single dispatch, at least for Smalltalk.

Introducing multimethods into a single-dispatch language like Smalltalk raises a range of issues: incorporating multimethods into Smalltalk syntax and the programming environment; implementing multimethods using the reflective facilities without changing the underlying virtual machine; and ensuring that multimethods provide good performance, without incurring additional overhead if they are not used.

This paper makes the following contributions:

– A core language design for multimethods in Smalltalk, demonstrating that a multimethod facility inspired by the CLOS Metaobject Protocol [Bobrow 1998] can be added to Smalltalk in a seamless, backwards compatible manner within the spirit of the language.

- An extensible implementation of the core language design, written in Smalltalk, that uses only the language's reflective features and requires no changes to the Smalltalk virtual machine
- An analysis of the performance of a range of implementations based on our framework, demonstrating that this approach is practical.

2. Multiple Dispatch

Like most object-oriented languages, Smalltalk provides single dispatch: a method call (in Smalltalk referred to as a *message send*) considers the dynamic type of one argument: the class of the object to which the message is sent. For example, consider the classical example of a graphical display system, where `GraphicScreen` and `GraphicPrinter` classes are subclasses of the abstract `GraphicalDisplay` class. The `GraphicalDisplay` class can define a number of messages such as `drawLine`, `drawRectangle`, `fillRectangle`, `drawArc`, and so on; then each subclass can implement these messages to display on a screen or a printer respectively.

This design has objects for the graphical displays but not for the graphical entities themselves. An obvious refinement of this design is then to introduce a further series of classes to represent the graphical objects: an abstract `GraphicalObject` class with `Line`, `Rectangle`, `FilledRectangle`, and `Arc` subclasses. This should allow programmers to simply their programs: code such as `aScreen draw: aRectangle` or `aPrinter draw: aLine` should allow any kind of graphical display to draw any kind of object. The problem is that this draw method requires *multiple dispatch*— the method body to be invoked must now depend upon *both* arguments to the message: the graphical display doing the drawing, and the graphical object which is being drawn.

The `GraphicalDisplay` classes can each provide an implementation of the draw method, but these cannot depend on the types of the graphical object arguments: a complementary design could swap the methods' receiver and argument objects (so programmers would write `GraphicalObjects drawOn: GraphicalDisplay`) this would allow different messages for each graphical object but not for different kinds of graphical displays. This problem is actually more common that it may seem in object-oriented designs. The visitor pattern, for example has a composite structure that accepts a visitor object embodying an algorithm to carry out over the composite (e.g. `Composite accept: aVisitor`): implementations of the accept method must depend upon the types of both composite and visitor [Gamma 1995].

Overloading in languages like Java or C++ can partially address this problem under certain circumstances. For example, Java allows methods to be distinguished based on the class of their arguments, so that a `ScreenDisplay` object can have different draw methods for displaying `Lines`, `Rectangles`, or `Arcs`:

```
abstract class GraphicalDisplay {
 public void draw(Line l) {
         // draw a line on some kind of display };
 public void draw(Rectangle r} {
         // draw a rectangle on some kind of display };
 public void draw(Arc a) {
         // draw an arc on some kind of display };
}

class ScreenDisplay extends GraphicalDisplay {
 public void draw(Line l) {
         // draw a line on a screen };
 public void draw(Rectangle r} {
         // draw a rectangle on a screen };
 public void draw(Arc a) {
         // draw an arc on a screen };
}
```

The problem here is that overriding is only resolved statically. Java will report an error in the following code:

```
Display d = new ScreenDisplay();
GraphicalObject g = new Line();
d.draw(g)
```

because the screen display class does not implement a `draw(GraphicalObject)` method.

The usual solution to this problem, in both Smalltalk and Java, is *double dispatch* [Ingalls 1986, Hebel 1990]: rather than implementing messages directly, method bodies send messages back to their arguments so that the correct final method body can depend on both classes. In this case, the `GraphicalDisplay` subclasses would each implement the draw methods differently, by asking their argument (the graphical object to be drawn) to draw themselves on a screen or on a printer:

```
ScreenDisplay>>draw: aGraphicalObject
         aGraphicalObject drawOnScreen: self

PrinterDisplay>>draw: aGraphicalObject
         aGraphicalObject drawOnPrinter: self
```

The key idea is that these methods encode the class of the receiver (`Screen` or `Printer`) into the name of the message that is sent. The `GraphicalObject` class can then implement these messages to actually draw:

```
Line>>drawOnScreen: aScreen
         "draw this line on aScreen"

Line>>drawOnPrinter: aPrinter
         "draw this line on aPrinter"
```

Each message send — that is, each dispatch — resolves the type of one argument. Statically overloaded implementations often generate “mangled” names for statically overloaded variants that similarly add type annotations to the names the virtual machine sees under the hood for compiled methods.

A few object-oriented languages, notably CLOS and Dylan [Bobrow 1998a, Keene 1989, Feinberg 1996], and various research extensions to Java [Boyland 1997, Clifton 2000] solve this design problem directly by supporting *multimethods*. A multimethod is simply a method that provides multiple dispatch, that is, the method body that is chosen can depend upon the type of more than one argument In this case code very similar to the Java code above could provide various different versions of the draw methods (one for each kind of `GraphicalObject`) within the `Display` classes, but the languages will choose the correct method to execute at runtime, based on the types of *all* the arguments in the message. The remainder of this paper describes how we implemented efficient multimethods as a seamless extension to Smalltalk.

3. Multimethods for Smalltalk

The first issue we faced in designing Smalltalk multimethods is that we wanted multimethods to fit in with the style or spirit of Smalltallk. Compared with most multimethod languages (especially CLOS) Smalltalk is lightweight, with a minimalist language design philosophy. A program is seen as a community of objects that communicate via message sends, and even “if” statements are technically implemented as messages to objects like true and false. An important aim of our design is that it should not change the basis of the language, and that multimethods should not affect Smalltalk programmers who choose not to write them.

The second issue is simply that Smalltalk, like Common Lisp, is a dynamically typed language, so that the language syntax does not, by default, include any specification of, or notation for, method types. As we’ve seen above, in many other object-oriented languages (such as Java and C++) method definitions must include type declarations for all their arguments even though the message sends will be dispatched in terms of just one distinguished argument.

Furthermore, in Smalltalk, programmers interact with programs on a per-method basis, using Smalltalk browsers. Source descriptions of these method objects are edited directly by programmers, and are compiled whenever methods are saved. Even when code is saved to files, these files are structured as “chunks” of code [Krasner 1983] that are written as sends to Smalltalk objects that can in turn, when read, reconstitute the code. Because of the way Smalltalk's browsers and files are set up, method bodies need not explicitly specify the class to which a method belongs. The class is implicitly given the context in which the message is defined.

Finally, Smalltalk provides reflective access to runtime *metaobjects* that represent the classes and methods of a running program, and allows a program to modify itself by manipulating these objects to declare new classes, change existing ones, compile or recompile methods, and so on. This arrangement is circular, rather than a simple layering, so that, for example, the browsers can be used to change the implementation of the metaobjects, even when those metaobjects will then be used to support the implementation of the browsers.

A language design to provide multimethods for Smalltalk must address all four of these issues: it must define how multimethods fit into Smalltalk’s language model, it must provide a syntax programmers can use to define multimethods, browser support so that programmers can write those methods, and the metaobjects to allow programmers to inspect and manipulate multimethods. A key advantage of the Smalltalk architecture is that these three levels are not independent: the metaobjects can be used to support both the browsers and language syntax.

Design: Symmetric vs. Encapsulated Multimethods

There are two dominant designs for multimethods in object-oriented programming languages. Languages following CLOS or Dylan [Bobrow 1988, Feinberg 1996] provide *symmetric* multimethods, that is, where every argument of the multimethod is treated in the same way. One consequence of this is that multimethods cannot belong to particular classes (because object-oriented methods on classes treat the receiver (`self` or `this`) differently from all the other arguments. *Encapsulated* or *asymmetric* multimethods [Boyland 1997, Castagna 1995, Bruce 1995] are an alternative to symmetric multimethods: as the name implies, these messages belong to a class and are in some sense encapsulated within one class, generally the class of the receiver.

We consider that asymmetric multimethods are a better fit for Smalltalk than symmetric multimethods. Smalltalk's existing methods obviously rely on a single dispatch with a distinguished receiver object; its syntax and virtual machine support are all tied to that programming style. Similarly, Smalltalk being class-based can naturally attach encapsulated multimethods to a single class.

Syntax and Semantics

A Multimethod will differ from a singly dispatched method in two ways. First, *specializers* that describe the types for which the methods are applicable must be specified for their formal arguments. Second, it must be possible to provide multiple definitions (generally with different specializers) for a single message name. This is similar to the way in which Java allows a single method name to have multiple overloaded definitions with different argument types.

There are two ways this might be done in Smalltalk. The first is to change the parser and compiler to recognize a new *syntax* for multimethod specializers. The second is to allow method *objects* to be changed or converted programmatically using runtime messages, perhaps with browser support, such as pull-down specializer lists, or, with additional arguments to the metaobjects that create the method object itself. The first approach is based on the text-based, linguistic tradition of programming language design, while the second is based on a more modern, browser/builder approach that supplants the classical notion of syntax with the more contemporary approach of direct manipulation of first-class language objects.

While we used elements of both approaches to build our multimethods, we relied, in this case, primarily on the more traditional text-based approach of the sort taken by CLOS [Bobrow 1998a], Dylan [Feinberg 1996], and Cecil [Chambers 1992]. In CLOS, a type specializer is represented as a two element list:

```
(defmethod speak ((who animal))
(format t "I'm an animal: ~A~%" who))
```

Dylan, by contrast, uses :: to denote specialization:

```
define method main (argv0 :: <byte-string>, #rest noise)
          puts("Hello, World.\n");
end;
```

The angle brackets are part of the type name in Dylan. Dylan, as with other languages in the Lisp tradition, is permissive about the sorts of characters that may make up names. Cecil uses an @ sign to indicate that an argument is *constrained* (which is how they refer to their brand of specializers).

```
x@smallInt + y@smallInt
         { ^primAdd(x,y, {&errorCode | … })}
```

As a completely dynamically typed language, Smalltalk does not require type declarations for variables or method arguments. However, Smalltalk programmers have long used a type syntax using angle brackets, either before or after the qualified argument, even though such declarations have no effect on the execution of a program using them. The VisualWorks 2.5x Smalltalk compiler can recognize an "extended language" syntax in which method arguments are followed by angle-bracketed type specifiers. This trailing angle-bracketed type designation notation was first suggested for Smalltalk by Borning and Ingalls over twenty years ago [Borning 1982]. A similar syntax was used in Typed Smalltalk [Johnson 1988a], and is used in the Visual Works documentation as well as the Smalltalk Standard. These specifiers can contain Smalltalk literals, symbols, or expressions.

We have adopted this syntax to support multimethods. The necessary adaptation is quite simple, comparable with Boyland and Castagna's Parasitic Multimethods for Java [Boyland 1997]. To declare a multimethod, a programmer simply adds a class name within angle brackets after any method argument. This method body will then only be called when the message is sent with an argument that is (a subclass of) the declared argument type, that is via a multiple dispatch including any argument with a type specializer. Here is an example of this syntax for the Graphical Display problem:

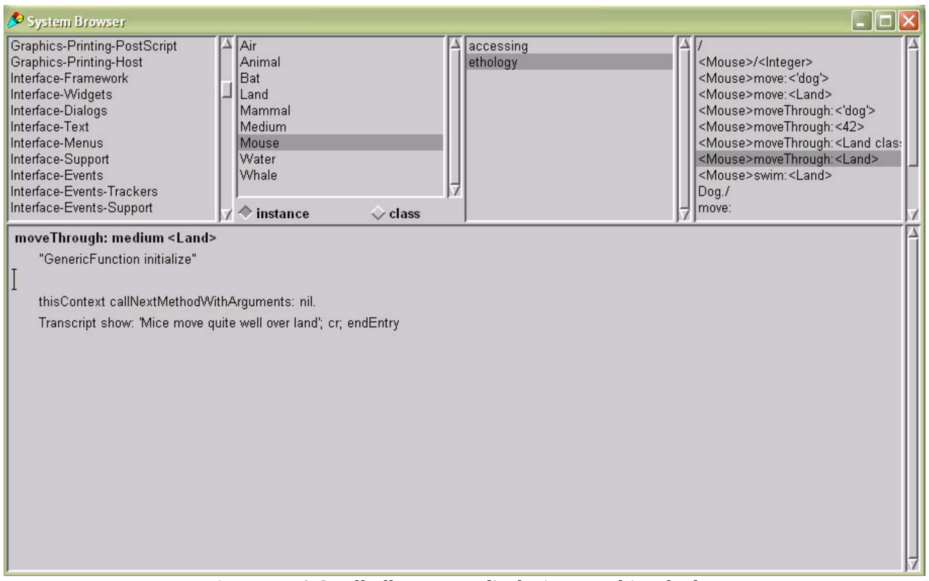
```
ScreenDisplay>>draw: aGraphicalObject <Line>
 "draw a line on a screen"

ScreenDisplay>>draw: aGraphicalObject <Arc>
 "draw an arc on a screen"
```

When a draw: message is sent to a `ScreenDisplay` object, the appropriate draw method body will now be invoked at runtime, with the decision of which message to invoke depending on the runtime classes of the object receiving the message, and any arguments with specializers. If no method matches, the message send will raise an exception, in the same way that Smalltalk raises a `doesNotUnderstand:` exception when an object receives a message it does not define.

These multimethods interoperate well with Smalltalk's standard methods and with inheritance, primarily because they are first sent (asymmetrically) to a receiver `(self)` object so their semantics are a direct extension of Smalltalk's standard method semantics. Multimethods may access instance and class variables based on their receiver, just as with standard Smalltalk methods. A multimethod defined in a subclass will be invoked for all arguments that match; otherwise an inherited method or multimethod in a superclass will be invoked. A multimethod can use a `super` send to invoke a standard method defined in a superclass, and vice versa. From this perspective, a “normal” Smalltalk method is treated exactly the same as a single multimethod body, where all arguments (other than the receiver) are specialized to `Object`. Our base multimethod design does not support one multimethod body delegating a message to another multimethod body defined in the same class, however, such common code can be refactored into a separate method and then called normally We have also experimented with a more flexible “call-next-method” scheme modeled after CLOS.

Browser Support

Languages that support multimethods have long been regarded as needing good programming environment support [Rosseau 1993]. Unlike most other programming languages, Smalltalk-80 has had an excellent integrated programming environment [Goldberg 1984] (and arguably has had one from before the start [Goldberg 1976]). Because this environment is itself written in Smalltalk we were able to exploit it to support Smalltalk multimethods.



```
moveThrough: medium <Land>
    "GenericFunction initialize"

    thisContext callNextMethodWithArguments: nil.
    Transcript show: 'Mice move quite well over land'; cr; endEntry
```

Figure 1 – A Smalltalk Browser displaying a multimethod

In fact, due to the design of the VisualWorks browsers, very few changes were required. For instance, while the Smalltalk `Parser` is selective about method selector syntax, the browsers are not. Any Smalltalk `Symbol` object (and perhaps other printable objects as well) can be used to index a method in the browsers. We exploited this fact to allow `MultiMethod` objects to appear with bracketed type specializers where their specialized arguments are to appear. Normal methods appear unchanged.

Metaobjects

Smalltalk is a computationally reflective language, that is to say, it is implemented in itself. The objects and classes that are used to implement Smallltalk are otherwise completely normal objects (although a few may be treated specially by the VM) but because they are used to implement other objects they are known as *metaobjects* or *metaclasses* respectively. Smalltalk programs are made up of metaobjects — Smalltalk methods are represented by instances of `Method` or `CompiledMethod` metaobjects, and Smalltalk classes by instances of `Class` metaobjects. The Smalltalk compiler (itself an instance of the `Compiler` class) basically translates Smalltalk language strings into constellations of these metaobjects. To implement multimethods in Smalltalk, we installed our own modified version of `Compiler` that understood the multimethod syntax and produced new or specialized metaobjects to implement multimethods.

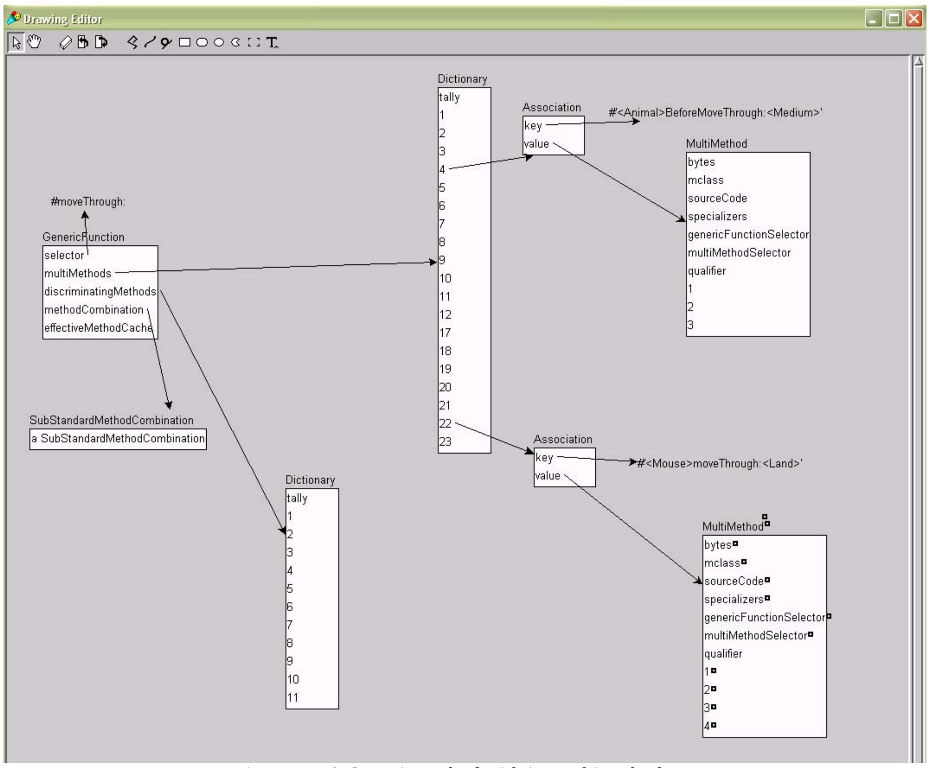


Figure 2 – A GenericMethod with its Multimethods

Our first implementation of multimethods was based on the CLOS MetaObject Protocol [Kiczales 1991]. It consists of the following metaobjects: `Multimethods`, `Specializers`, `GenericMessages`, `MethodCombinations`, and `DiscriminatingMethods`. Fig. 2 shows the way these objects collaborate to represent multimethods.

A `GenericMessage` (`GenericFunction` in the figure) contains a `Dictionary` mapping specialized message selectors to their respective multimethod bodies. The `GenericMessage’s` associated `MethodCombination` object orders these multimethod bodies to determine the correct method to invoke. A `GenericMessage` also contains a list of the `DiscriminatingMethods` that intercept method calls and start the multimethod dispatch. We describe each of these objects in turn below.

Multimethods
A `MultiMethod` metaobject represents one multimethod body. That is, it represents a method that can be dispatched with any or all of its arguments being taken into account, instead of just the first one. A multimethod must have one or more argument `Specializers` that determine the kinds of arguments to which the multimethod will respond. A multimethod can determine if is *applicable* to a series of arguments (via its specializers) and, if so, can run the code in its body when required.

Specializers
`Specializers` represent the argument to which a particular `Multimethod` applies. When a specializer is invoked, it determines if the argument passed to the multimethod matches that multimethod, or not. We currently use two different kinds of `Specializers`: `ClassSpecializers`, and `EqualSpecializers`. `ClassSpecializers` indicate that a multimethod applies when the corresponding argument is a member of the indicated class, or one of its subclasses. `EqualSpecializers` (which are modeled after CLOS's EQL specializers), match when an argument is equal to a particular object. Cecil [Chambers 1992], a prototype-based dynamic language with multimethods, gets `EqualSpecializers` for free, since all specializers are instances, not classes.

Generic Messages
A `GenericMessage` represents the set of all multimethods with the same name. (The name `GenericMessage` is derived by analogy with the similar Generic Function object in CLOS). When a `GenericMessage` is called, its job is to select, from among all its `MultiMethods`, only those that are consistent with the arguments it was called with (the *applicable* methods). These must also be sorted in the correct order, that is, from the most specific multimethod to the least specific multimethod.

Method Combinations:
A `MethodCombination` object defines the order in which methods are called, and how different kinds of methods are treated. Again, our `MethodCombinations` are modeled after those in CLOS. Their job is to take the set of applicable methods that was determined by the `GenericMessage` to be "in-play" given the current arguments, and execute these in the manner that their qualifiers and the `MethodCombination` itself prescribe. When a generic message finds more than one applicable method, these are sorted from most specific to least specific. This situation is analogous to a normal message send, where a call finds the most specific subclass's version of a method.

Multiple methods can apply because some will match more precisely, or specifically, at one or more argument sites. For example, consider the following two multimethods on a Stencil class (that draws multiple copies of an image along a path):

```
Stencil>>drawUsingShape: rectangle<Rectangle>
            OnDisplay: display <GraphicalDisplay>

Stencil>>drawUsingShape: shape<GraphicalObject>
            OnDisplay: display <ScreenDisplay>
```

Both of these multimethods would match a call where the first (shape) argument was a `Rectangle` and the second (display) argument a `ScreenDisplay`. In this case, the `MethodCombination` will sort these in the order shown, because the specifier on the first multimethods's first argument is more specific that the specifier on the first argument of the second multimethod.

`MethodCombination` objects can be thought of as examples of the Strategy design pattern [Gamma 1995]. `MethodCombination` objects represent the rules for combining and calling a multimethod's bodies. A `GenericMessage` can change the way that its methods are dispatched by designating a new `MethodCombination` object. Of course, the multimethods themselves must be written carefully in order to allow changes in the combination scheme to make sense. That is to say, methods are normally written without having to concern themselves with the possibility of being combined in exotic, unexpected ways.

DiscriminatingMethods
Any message send in a Smalltalk program needs to be able to invoke a multimethod. Whether a multimethod or a “normal” Smalltalk method will be invoked depends only upon whether any multimethod bodies (i.e. any methods with specializers) have been defined for that message name. That is (as with normal Smalltalk methods) the implementation of the method is solely the preserve of the receiver of the message (or, from another perspective, the classes implementing that method). This design has the short-term advantage that no performance overhead will be introduced in Smalltalk programs that do not use multimethods, or for sends of “normal” methods in programs that also include multimethods; and the longer-term advantage that classes can turn their methods into multimethods (by adding specialized versions), or vice versa, without any concern for the clients of those classes, or the callers of those messages.

In practice, this design means that our multimethod dispatch must intercept the existing Smalltalk message dispatch. Our last metaobject, the `DiscriminatingMethod`, performs this interception. Smalltalk cannot intercept an incoming message until one dispatch, on the first argument, has already been done. A `DiscriminatingMethod` (again named after the analogous discriminating functions in CLOS) is a `MethodWrapper` [Brant 1998] that acts as a decorator around the standard Smalltalk `CompiledMethod` object.

Fig. 3 shows how the multimethods and `DiscriminatingMethods` hook into standard Smalltalk classes. All standard Smalltalk class metaobjects (including, in this figure, the `Mouse` class) contain a `MethodDictionary` implemented as two parallel arrays. The first array contains method selectors. For our multimethods, as well as the standard selectors (`#moveThrough` in the example in this figure) we include specialized selectors (`#<Mouse>moveThrough:<Land>`). The second array normally contains method bodies: in our implementation the standard selector (`#moveThrough` that will actually be sent by program code) maps to a `DiscriminatingMethod` that will invoke the multimethod dispatch, while the specialized selector maps to the object representing the `MultiMethod` body. Because this method includes two specializers (`<Mouse>` and `<Land>`) it is linked to two `ClassSpecializer` objects.

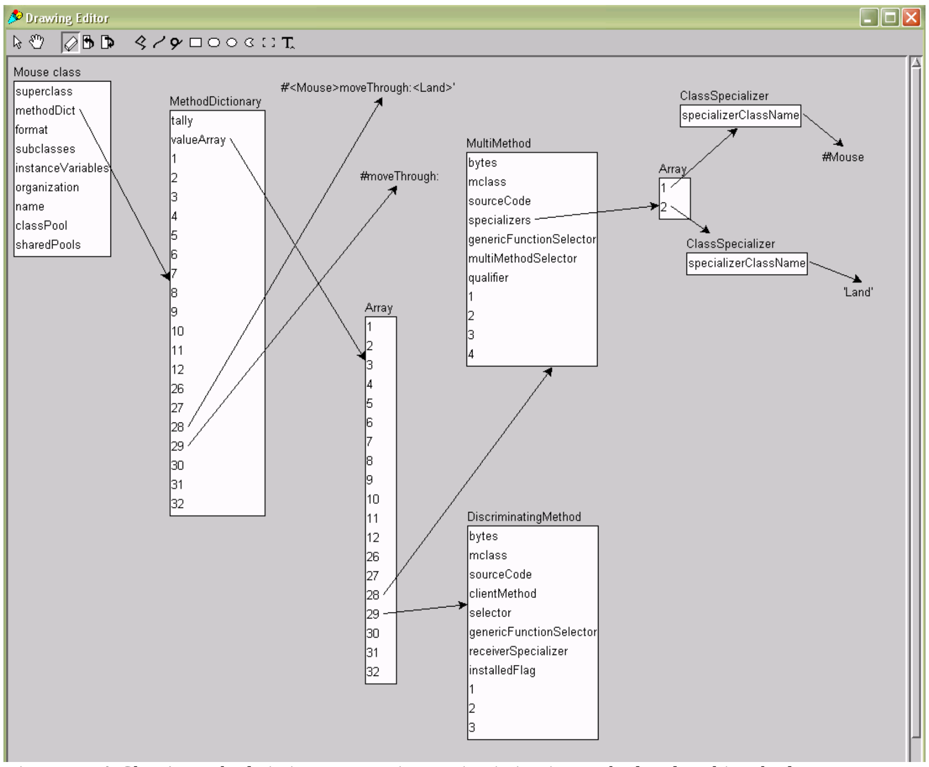


Figure 3 – A Class's MethodDictionary mapping a DiscriminatingMethod and Multimethods

The relationship between `MultiMethods`, `GenericMessages`, and `DiscriminatingMethods` is as follows. There is one `GenericMessage` for every message name in the system that has at least one specialized method body. Every method body defined with a specializer is represented by a `MultiMethod` object (and its associated `Specializers`) — all of these are known by their `GenericMessage`, and can be chosen by its `MethodCombination`. Finally, every class that can understand this multimethod name (i.e. that has at least one `MultiMethod` body defined) will have a `DiscriminatingMethod` stored under that name that is again linked to its `GenericMessage`.

Invoking Multimethods
All these metaobjects collaborate to implement the dispatch whenever a program sends a message to an object that implements that message using multimethods. First, a `DiscriminatingMethod` is used to gain control. The `DiscriminatingMethod` then forwards the message send and its argument values to the `GenericMessage` object for all multimethods with that name. Next, the `GenericMessage` iterates across each candidate `MultiMethod` looking for all the applicable `MultiMethods`, that is, all `MultiMethods` whose `Specializers` match the actual arguments of the message send.

The `GenericMessage` then sorts the applicable methods in order of applicability, and passes the list to the `GenericMessage's` `MethodCombination` object. The `MethodCombination` then selects and executes the body of the chosen `MultiMethod`. This result is then returned (via the `GenericMessage`) to the `DiscriminatingMethod`, and thus is returned (as normal) to the caller of the multimethod.

As with CLOS, these objects are designed to allow the caching of partial results.

4. Examples

In this section, we present some examples to show how multimethods could be used to support the design of Smallltalk programs.

Eliminating Class tests: Smalltalk methods often use explicit tests on the classes of their arguments. For example, the method to add a visual component to a component part in the VisualWorks interface framework behaves differently if the argument is a `BorderedWrapper`. This is implemented using an explicit class test:

```
ComponentPart>>
  add: aVisualComponent borderedIn: aLayoutObject

^(aVisualComponent isKindOf: BorderedWrapper)
  ifTrue: [aVisualComponent layout: aLayoutObject.
      self addWrapper: aVisualComponent]
  ifFalse: [self addWrapper:
   (self borderedWrapperClass on: aVisualComponent
               in: aLayoutObject)]
```

With multimethods, this could be refactored to two multimethods, one handling `BorderWrapper` arguments, and another the rest:

```
ComponentPart>>
  add: aVisualComponent <BorderWrapper>
  borderedIn: aLayoutObject

  aVisualComponent layout: aLayoutObject.
  ^self addWrapper: aVisualComponent.

ComponentPart>>
  add: aVisualComponent <Object>
  borderedIn: aLayoutObject

^self addWrapper:
  (self borderedWrapperClass on: aVisualComponent
               in: aLayoutObject)
```

Visitor: The following example, drawn from [Brant 1998], illustrates the impact of multimethods on the Visitor pattern [Gamma 1995]. First, consider a typical Smalltalk implementation of Visitor:

```
ParseNode>>acceptVistor: aVisitor
  ^self subclassResponsibility

VariableNode>>acceptVistor: aVisitor
  ^aVisitor visitWithVariableNode: self

ConstantNode>>acceptVistor: aVisitor
  ^aVisitor visitWithConstantNode: self

OptimizingVisitor>>visitWithConstantNode: aNode
  ^aNode value optimized

OptimizingVisitor>>visitWithVariableNode: aNode
  ^aNode lookupIn: self symbolTable
```

When `MultiMethods` are available, however, the double-dispatching methods in the `ParseNodes` disappear, since the type information does not need to be hand-encoded in the selectors of the calls to the `Visitor` objects. Instead, the `Visitor` correctly dispatches sends of `visitWithNode` to the correct `MultiMethod`. Thus, adding a `Visitor` no longer requires changing the `ParseNode` classes.

```
OptimizingVisitor>>visitWithNode: aNode <ConstantNode>
         ^self value optimized

OptimizingVisitor>>
         visitWithNode: aNode <VariableNode>
         ^aNode lookupIn: self symbolTable
```

5. Implementation

We have experimented with a number of different implementations for multimethods. The first and simplest scheme is just to execute the Smalltalk code for the metaobjects directly. While acceptable for simple examples, such a strategy proved unacceptably slow, and so we therefore experimented with a number of different optimizations, some of which can execute code using multimethod as quickly as handwritten Smalltalk code for multiple dispatching. This section describes these implementations, and then presents our performance results.

Metaobjects: Our initial, unoptimized implementation simply executed the Smalltalk code in the metaobjects to dispatch multimethods: A method wrapper is used to gain control (adding about an order of magnitude to the dispatch process), the generic message iterates across each multimethod body and its specialzers, the resulting list is sorted, and so on. Even before performance testing, it seemed obvious that this approach would be too slow to be practical. Fortunately, there is quite a bit that can be done to speed things up.

Dictionary: Our first optimization uses a Smalltalk Dictionary to map from arrays of specializers to target methods. It is, in effect, a simple implementation of the hashtable scheme discussed by Kiczales and des Rivieres in [Kiczales 1991]. Our scheme relies on the fact that it would be applied in a `DiscriminatingMethod`, and left out the first argument: the other argument classes are cached in a table so that the applicable multimethod body can be found directly.

Case: Our second optimization was to directly test the classes of each argument and calls the appropriate method. The idea is that the decision tree itself is inlined as in a case statement. We wrote this version by hand, but code to implement these case tree dispatchers could be synthesized automatically, should this approach prove practical.

Multidispatch: Our third optimization is a generalization of the double dispatch scheme described by Ingalls [Ingalls 1986]. Instead of merely redispatching once, redispatchers are generated so that each argument gets a chance to dispatch. Hence, triple dispatch is performed for three argument multimethods, quadruple dispatch for four, octuple dispatch for eight, etc. At each step, identified class/type information is “mangled” into the selectors, that is, we automatically generate the same code that a programmer would write to implement multiple dispatch in Smalltalk . Since this approach takes advantage of the highly optimized dispatching code in the Visual Works Virtual Machine, we expected its performance to be quite good. The main problem with multiway dispatch is that a large number of methods may be generated:

$$\begin{aligned} |D| = {} & |S_1| \\ & + |S_2| \times |S_1| \\ & + |S_3| \times |S_2| \times |S_1| \ldots \\ & + |S_n| \times \ldots \times |S_2| \times |S_1| \end{aligned}$$

or, alternately,

$$|D| = \sum_{i=1}^{n} \prod_{j=1}^{i} |S_j|$$

where $|S_j|$ denotes the cardinality of the set of specializers for the indicated argument of a particular generic message and |D| is the cardinality of the set of dispatching methods which must be generated.

Our multidispatch code generates all the required multidispatch methods automatically. They are all placed in a special Smalltalk protocol category: 'multidispatch methods'. These methods are named using the compound selector syntax inherited from VisualWorks 2.0 that has its roots in the Borning and Ingalls multiple inheritance design [Borning & Ingalls 1982]. These selectors allow periods to be included as part of the message selector name. The original selector is placed at the beginning of each multidispatch selector, with dots replacing the colons. Argument specifiers are indicated using 'arg1', followed by the specializer for the argument, if one was recognized by a previous multidispatch method.

Our implementation generates an additional, final dispatch to the initial multimethod receiver so that the target multimethod body can be executed as written. In one sense, this final dispatch is a concession to the low-level asymmetry inherent in our Smalltalk implementation. In effect, this final ricochet closes the multidispatched circle. This introduces an additional factor of two into the last term in the formula above. Given this, the number of methods we generate becomes:

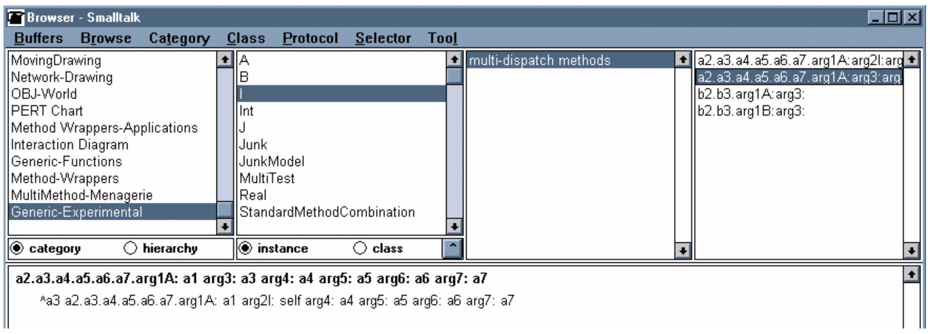
$$\begin{aligned} |D| = |S_1| & \\ & + |S_2| \times |S_1| \\ & + |S_3| \times |S_2| \times |S_1| \ldots \\ & + |S_n| \times \ldots \times |S_2| \times |S_1| \times 2 \end{aligned}$$

Note that the class of the recipient of this final dispatch will be pre-determined by the time this call is made, hence, virtual machines that employ inline caching mechanisms will incur minimal overhead for all but the initial call to such methods.

We can reduce the number of methods we have to generate by shuffling the order in which the arguments are dispatched. Since each S_j introduced is a factor in every subsequent term of the formula above, dispatching from the lowest cardinality specializer up to the highest will minimize the number generated methods. Of course, Smalltalk forces us to start with the first argument, S_1, instead of whichever we wish. Since leftmost factors are repeated more often, reordering multiway dispatch so that the smaller factors are the ones that recur minimizes the number of methods that must be generated.

Two additional optimizations are possible. Were the target method's body merged with the final set of dispatching methods, the final "× 2" factor in the final term of the equation above could be elided. Also, only arguments that are actually *specialized* need be redispatched. That is to say, if the cardinality of the set of specializers is one (that is, the argument is <u>not</u> specialized), then it can be bypassed. To put it another way, <u>only</u> arguments for which more than one specializer is present need be treated as members of the set of specializers.

While the formula above might suggest to some readers that in the worst case there is cause for concern that this generalized multiway dispatch scheme might entail the generation of an unacceptably large number of methods, we believe that the potential for practical problems with this approach is rather low. Indeed, Kiczales and Rodriguez [Kiczales 1990] observed that only four percent of the generic functions in a typical CLOS application specialized more than a single argument. We expect that multimethods with large numbers of specializers on multiple arguments to be rare birds indeed.



```
a2.a3.a4.a5.a6.a7.arg1A: a1 arg3: a3 arg4: a4 arg5: a5 arg6: a6 arg7: a7
    ^a3 a2.a3.a4.a5.a6.a7.arg1A: a1 arg2I: self arg4: a4 arg5: a5 arg6: a6 arg7: a7
```

Figure 4 – Generated Multidispatch Example

Performance

Table 1 compares the performance of the various implementation techniques with the cost of performing an Ingalls-style double dispatch. This is shown in **row 1**. The cost for the simplest standard Smalltalk single-dispatched method call that returns the called object is shown in **row 2.** Returning self in Smalltalk is a special case both in the bytecode and the compiler, however it is only five times faster than a double dispatch that must do significantly more work. Each row in the table shows the results of a single implementation (the number in parenthesis in the leftmost column is the number of arguments the multimethod dispatched upon). We make multiple runs of each benchmark, timing 1,000,000 (multi)method sends, and report the minimum and maximum invocation times for each run.

Row 3 shows the performance of the straight-ahead Smalltalk implementation of multimethods, giving the overhead when a target method dispatching on two method arguments simply returns itself. That is, we take about 600 microseconds to do nothing. R**ow 4**, with full method combination support calling an overridden multimethod is extremely slow. This sort of dismal performance is not unheard of when reflective facilities are used. For instance [Palsberg 1998] found 300:1 performance decreases for their reflective implementations of variants on the Visitor pattern. **Row 5** gives the performance of a simple extension to this scheme, where the final multimethod body lookup is cached, giving a fivefold increase in performance but still being slow relative to a hand coded implementation.

Rows 6 and **7** give the performance of the Dictionary and case-statement lookups respectively, dispatching on three specialized arguments. Again, these optimizations provide another order of magnitude but are still twenty times as slow as the basic multiple dispatch.

Rows 8 and **9** finally show the performance of the generated multidispatch implementation, **row 8** again dispatching on three arguments and **row 9** on seven. Here at last is an implementation that performs at roughly the same speed as the standard Smalltalk system, because our generated code is effectively the same as the code an experienced Smalltalk programmer would write to implement multiple dispatch. This is the implementation we have adopted in our system.

Dispatch Type	nanosec min	nanosec. max	Ratio
1. Multidispatch (2 args)	521	524	1.00
2. Tare (^self) (1 arg)	90	120	0.20
3. Metaobjects (^self) (2 args)	597,000	624,000	1168
4. Metaobjects (super) (2 args)	679,000	750,000	1367
5. Metaobjects cached (2 args)	117,000	125,000	231
6. Dictionary (3 args)	13227	13335	25
7. Case (inline) (3 args)	10654	10764	20
8. Multidispatch (3 args)	633	779	1.35
9. Multidispatch (7 args)	1200	1221	2.32

Table I -- ***Performance Results***
200MHz Pentium Pro
1,000,000 calls/multiple runs

There are a variety of trade-offs that must be considered among these approaches. The "pure" Smalltalk solution is relatively easy to use, but performs so poorly that it is little more than a toy. It is a testament to the power of reflection that the range of strategies for improving this performance can be addressed at all from within the Smalltalk programming environment itself. Still, these are not without their costs. The multidispatch approaches can litter the method dictionaries with dispatching methods. These, in turn, beg for improved browsing attention.

The final performance frontier is the virtual machine itself. While possible, this would require a way of controlling the dispatch process "up-front" [Foote 1989], and would greatly reduce portability. Given that performance of our multidispatch scheme is as quick as standard Smalltalk, we consider that the complexity of changing the virtual machine is not justified by the potential increase in dispatching performance.

6. Discussion

In this section we address a number of issues regarding the provision of multimethods in Smalltalk.

Access to Variables: although multimethods are dispatched on multiple arguments, they remain encapsulated within a single class as in standard Smalltalk and can only access instance variables belonging to `self`. It is, of course, possible to generate accessor methods so that multimethods could access instance variables belonging to all arguments. This is, in essence, the approach take by CLOS. Given that we aimed to retain as much of Smalltalk's object model as possible (and Smalltalk's strong variable encapsulation is an important part of that model) we elected not to change this part of the language. As in standard Smalltalk, programmers can always choose to provide instance variable accessor methods if they are needed by particular multimethods. Indeed, such accessors, together with a judicious choice of `MethodCombination` objects, allow multimethods to be programmed in a symmetric style.

Class-based dispatch: as in standard Smalltalk, our multimethods are dispatched primarily based upon the classes of arguments. Smalltalk has an *implicit* notion of object type (or *protocol*), based on the messages implemented by a class, so two classes can implement the same interface even if they are completely unrelated by inheritance. We considered providing specializers that would somehow select methods based on an argument's *interface* or *signature*, but this would require an *explicit* notion of an object's type signature, which standard Smalltalk does not support (although extensions to do so have long been proposed [Borning 1987, Lalonde 1986]). One advantage of class-based dispatch, given that Smalltalk supports only single inheritance, is that class-based selectors will never be ambiguous, as is possible with multiple inheritance or multiple interfaces.

Our implementation does support instance-based `EqualSpecializers` as well. We have not as yet made a detailed assessment of either their impact on performance, or of their overall utility.

Portability and Compatibility: we have taken care to maximize the portability of our multimethod design across different Smalltalk implementations. Our syntax is designed so that it is completely backwards compatible with existing Smalltalk syntax and to impose no overhead on programmers if multimethods are not used. Similarly, our design requires no changes to Smalltalk virtual machines and adds no performance penalty if multimethods are unused. The largest portability difficulties are with individual Smalltalk compilers and browsers, as these differ the most between different language implementations. A final aspect of portability relates to the compiled code for optimized implementations. Because the generated multidispatch code does not depend on any other part of the system, it can be compatible with Smalltalk systems without the remainder of the multimethod system.

Method Qualifiers — Extending Method Combinations: A great advantage of building multimethods by extending Smalltalk's existing metaobjects is that our implementation can itself be extended by specializing our metaobjects. We have implemented a range of extended method combination schemes, modeling those of CLOS and Aspect/J, to illustrate this extensibility.

Our extended method combination scheme allows `MultiMethods` to be given `Qualifiers`. `Qualifiers` are symbols such as `#Before`, `#After`, or `#Around`. These qualifiers indicate to `MethodCombination` objects the role these methods are to play, and how they should be executed. There are no a priori limitations on these qualifiers; they can be any symbols that the `MethodCombination` objects can recognize. As in CLOS and Aspect/J, we provide some stock `MethodCombination` objects that implement before, after, and around methods that execute before, after, or before-and-after other methods in response to a single message send [Brant 1998, Kiczales 2001]. We also provide a `MethodCombination` that emulates the Beta [Kristensen 1990] convention of executing methods from innermost to outermost. We also provide a `SimpleMethodCombination` object that executes its applicable method list in the order in which it is passed to the `MethodCombination` object.

These extended method combination metaobjects interpret the qualifiers during multimethod dispatch. The main change is that more multimethod bodies can match a particular method send, because a qualified method can execute in addition to other methods that also match the arguments of a message send. For example, as in CLOS, all before (or after) multimethods will execute before (or after) the one unqualified message chosen by the base multimethod dispatch.

Our current implementation of extended method combination is experimental. In particular, qualifiers must be assigned programmatically to multimethods as we have not yet provided syntactic or browser support.

Our design, as well as the designs of Smalltalk and CLOS, for that matter, is distinguished from more recent work based upon Java derivatives [Boyand 1977, Kiczales 2001] in that given that each is built out of objects, programmers can extend these objects themselves to construct any mechanism they want. It is a testimony to the designers of Smalltalk and CLOS [Gabriel 1991, Bobrow 1993] that principled architectural extensions, rather than inflexible, immutable preprocessor artifice, can be employed to achieve this flexibility.

Language design, it has been said, is not about what you put in, but about what you leave out. A system built of simple, extensible building blocks allows the designer to evade such painful triage decisions. The real lesson to be gleaned from the metalevel architectures of Smalltalk and CLOS is that if you provide a solid set of building blocks, programmers can construct the features they really need themselves, their way. This might be thought of as an application of the “end-to-end principle” [Saltzer 1981] to programming language design.

Programming languages versus idioms and patterns: Finally, our work raises the philosophical question of when programming idioms or design patterns should be incorporated into programming languages. From a pragmatic perspective, it is unnecessary to add multimethods into Smalltalk because multiple dispatch can be programmed quite effectively using idioms such as double dispatch [Ingalls 1986] or the Visitor pattern [Gamma 1995]. Our most efficient implementation merely matches the performance of these hand-coded idioms — some of our more basic implementations perform significantly worse — so efficiency is not a reason for adopting this extension.

Indeed, our harmonious melding of the CLOS MOP atop Smalltalk's kernel objects (the Smalltalk "MOP", if you will) suggests that neither single dispatch nor multi-dispatch is more fundamental that the other. Instead, they can be seen as complements or duals of each other. Single dispatch can be seen as merely a predominant, albeit prosaic special case of generalize multiway dispatch. Alternately, our results show that you can curry your way to multiple dispatch in any polymorphic, single dispatch language, one argument at a time.

In general, we consider that an idiom — such as double dispatch — should be incorporated into a language when it becomes very widely used, when a hand coded implementation is hard to write or to modify, when it can be implemented routinely, and at least as efficiently as handwritten code. The Composite and Proxy patterns, for example, may be widely used, but their implementations vary greatly, while implementing the Template Method pattern is so straightforward that it requires no additional support. On the other hand, the Iterator pattern is also widely used, but its implementations are amenable to standardization, and so we find Iterators incorporated into CLU, and now Java 1.5 and C#.

Multimethods are particularly valuable as Mediators. Since, for instance, a binary multimethod can be seen as belonging to either both or neither of a pair of class it specializes, it can contain glue that ties them together, while leaving each of its specializing classes untouched. The promise of clean separation of concerns, however admirable, is honored, alas, in many systems mainly in the breach [Foote 2000]. Multimethods are ideal in cases where mutual concerns arise among design elements that had heretofore been cleanly separated. Multimethods can help when concerns converge.

We believe that multiple dispatch is sufficiently often used; sufficiently routine; sufficiently arduous to hand code; and that our (and others) implementations are sufficiently efficient for it to be worthwhile to include into object-oriented programming languages.

7. Related Work

Multiple dispatch in dynamic languages was first supported in the LISP based object-oriented systems LOOPS and NewFlavours [Bobrow 1983; 1986]. As an amalgam of these systems, the Common Lisp Object System incorporated and popularized multiple dispatch based on generic functions [Bobrow 1988a , Keene 1989]. CLOS also incorporated a range of method combinations, although more recently these have also been adopted by aspect-oriented languages, particularly Aspect/J [Kiczales 2001]. Dan Ingalls described the now standard double-dispatch idiom in Smalltalk in what must be the OOPSLA paper with the all-time highest possible power-to-weight ratio [Ingalls 1986]. All these systems had the great advantage of dynamic typing, so were able to avoid many of the issues that arise in statically typed languages.

The first statically typed programming language with object-oriented multiple dispatch was the functional language Kea [Mugridge 1991]. While a range of statically typed languages provide overloading, (Ada, C++, Haskell, Java) satisfactory designs for incorporating dynamically dispatched multimethods into statically typed languages proved rather more difficult to develop. Craig Chamber's Cecil language [Chambers 1992] provided a model where multimethods were encapsulated within multiple classes to the extent that the multimethods were specialized on those classes. Further developments of Cecil demonstrated that statically typed multimethods could be integrated into practical languages and module systems with separate compilation. Cecil-style multimethods have also been incorporated into Java [Clifton 2000], and have the advantages of a solid formal foundation [Bruce 1995, Castagna 1995]. Bjorn Freeman-Benson has also proposed extending Self with Multimethods [Chambers 1992]. Rather than providing multiple dispatch by extending message sends Leavens and Millstein have proposed extending Java to dispatch on tuples of objects [Leavens 1998].

Closer to the design in this paper are Boyland and Castangna's Parasitic Multimethods. These provide a type-safe, modular extension to Java by dispatching certain methods (marked with a 'parasitic' modifier) according to the types of all their arguments [Boyland 1997]. As with our system, the parasitic design treats multimethods differently from normal ("host") messages, and then the distinguished receiver argument differently from the other arguments of a message. Multimethods are contained within their receiver's class and may access only those variables that are members of that class. Boyland and Castagna note that much of the complexity of their system comes from their goal of not changing Java's existing overloading rules, and recommend that future languages support only dynamic dispatch — ironically perhaps, the resulting language would be quite similar in expressiveness to Smalltalk with Multimethods.

The Visitor pattern is one of the main contexts within which double-dispatch is generally applied [Gamma 1995]; as with many patterns, Visitor has spawned a mini-industry of research on efficient implementation [Palsberg 1998, Grothoff 2003] that sometimes go as far as raising the specter of a Visitor-oriented programming "paradigm" [Palsberg 2004]. Similarly, incorporating features from Beta into more mainline (or at least less syntactically eccentric) object-oriented languages has also been of interest of late, with most work focusing on the Beta type system [Thorup 1997] although "inner-style" method combination has recently been adapted to a Java-like language design [Goldberg 2004].

Our work also draws on a long history of language experimentation, particular in Smalltalk. The dot-notation for extended selectors was originally proposed for multiple inheritance [Borning 1982] but has been used to navigate part hierarchies [Blake 1987]. More recent work on Array-based programming [Mougin 2003] employs somewhat similar techniques to extend Smalltalk, although without providing an extensible meta-model. Scharli et al. [2004] describe a composable encapsulation scheme for Smalltalk that is implemented using method interception techniques. This encapsulation model could be extended relatively straightforwardly to our multimethods, and would have the advantage that multimethods could thereby access private features of their argument objects.

This work draws upon many techniques developed over many years for meta-level programming [Smith 1983, Maes 1987a,b], both in Smalltalk and other languages. Our `DiscriminatingMethods` are derivations of `MethodWrappers` [Brant 1998], and the notion of extending method dispatch by meta-level means goes back at leat to CLOS and LOOPS [Bobrow 1983, Kiczales 1991]. Coda [McAffer 1995] provides an extended Smalltalk meta-object system that has been used to distribute applications across large scale multiprocessors. MetaclassTalk provides a more complete CLOS-style metaobject system for Smalltalk, again implemented with MethodWrappers, that has been used to implement various aspect-oriented programming constructs [Rivard 1997].

Finally, Bracha and Ungar [Bracha 2004] have classified the features of reflective systems into *introspection* (self-examination of a program's own structure); *self-modification* (self explanatory); *executing* dynamically generated code (ditto); and *intercession* (self-modification of a language's semantics from within). According to their taxonomy, Smalltalk scores highly on all categories except intercession. The dispatching metalevel we present in this paper can be seen either as a strong argument that Smalltalk does, in fact, provide powerful intercession facilities, or, more humbly, that straightforward, portable extensions can add these facilities to Smalltalk.

8. Conclusion

Though Smalltalk does not support multimethods, they can be built by programmers who understand Smalltalk's reflective facilities. There are several ways to go about this, and they differ dramatically in terms of power and efficiency. Taken together, they demonstrate the power of building programming languages out of objects, and opening these objects to programmers, and teach some interesting lessons.

One is that syntax matters. To build multimethods, we needed to be able to modify the compiler to support argument specializers.

A second lesson, however, is that when programs are objects, there are other mechanisms besides syntax that an environment can use to change a program. Our browsers support multimethods because methods have a uniform interface to these tools that allows our multimethod syntax to be readily displayed. Furthermore, since multimethods are objects, their attributes are subject to direct manipulation by these tools.

A third lesson is that runtime changes to objects that define how an object is executed are an extremely powerful lever. Using method wrappers to change the way methods act on-the-fly provides dramatic evidence of this.

A fourth is that there is a place for synthesized code, or code written by programs rather than programmers, in reflective systems. Generative programming [Czarnecki 2000] approaches have their place. Our efficient multiway dispatch code made use of this, allowing our reflective implementation to perform as well as hand-written code, without any changes to the Smalltalk virtual machine.

Our experience makes a powerful case for building languages out of objects. By doing so, we allow to the very objects from which programs are made be a vehicle for the language's own evolution, rather than an obstacle to it, as is too often the case.

To conclude, we have designed and implemented efficient multimethod support for Smalltalk. Our multimethods provide a very clean solution: programmers can define them using a simple extended syntax, their semantics are quite straightforward, they interoperate well with Smalltalk’s metaobjects, they impose no syntactic or runtime overhead when they are not used, and they are as efficient to execute as comparable hand-written code using sequences of single dispatches.

References

[Benoit 1986] Ch. Benoit, Yves Caseau, Ch. Pherivong, Knowledge Representation and Communication Mechanisms in Lore. *ECAI 1986*, 215-224

[Blake & Cook 1987]. D. Blake and S. Cook. On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk. In *ECOOP Proceedings* 1988, 41-50.

[Bobrow 1983] Daniel G. Bobrow. *The LOOPS Manual.* Xerox Parc, 1983

[Bobrow 1986] Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. CommonLoops: Merging Lisp and Object-Oriented Programming. In *OOPSLA Proceedings*.1986.

[Bobrow 1988a] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices*, Volume 23, September 1988

[Bobrow 1988b] Daniel G. Bobrow and Gregor Kiczales. The Common Lisp Object System Metaobject Kernel -- A Status Report. In Proceedings of the 1988 Conference on Lisp and Functional Programming, 1988.

[Bobrow 1993] Daniel G. Bobrow, Richard P. Gabriel, Jon L. White, CLOS in Context: The Shape of the Design Space, in Object-Oriented Programming: The CLOS Perspective, Andreas Paepcke, editor, MIT Press, 1993, http://www.dreamsongs.com/NewFiles/clos-book.pdf

[Borning & O’Shea, 1987] Alan Borning and Tim O'Shea. Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language. In ECOOP Proceedings,1987, 3-12.

[Borning & Ingalls 1982] A. H. Borning and D. H. H. Ingalls. A Type Declaration and Inference System for Smalltalk. In *POPL Proceedings*, 1982, 133-141.

[Boyland & Castagna 1997] John Boyland and Giuseppe Castagna Parasitic Methods: An Implementation of Multi Methods for Java. In *OOPSLA Proceedings* 1997.

[Bracha 2004] Gilad Bracha, David Ungar: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In OOPSLA Proceedings, 2004. 331-344

[Brant 1998] John Brant, Brian Foote, Don Roberts and Ralph Johnson. Wrappers to the Rescue. In *ECOOP Proceedings*, 1998.

[Bruce 1995] Kim Bruce , Luca Cardelli , Giuseppe Castagna , Gary T. Leavens , Benjamin Pierce, On binary methods, *Theory and Practice of Object Systems*, v.1 n.3, p.221-242, Fall 1995

[Caseau 1986] Yves Caseau, An Overview of Lore. *IEEE Software* 3(1): 72-73

[Caseau 1989] Yves Caseau, A Model for a Reflective Object-Oriented Language, SIGPLAN Notices 24(4), 22-24

[Castagna 1995] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431--447, May 1995

[Chambers 1992] Craig Chambers. Object-Oriented Multimethods in Cecil. In *ECOOP* Proceedings, 1992

[Clifton 2000] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of OOPSLA 2000*, 130-145.

[Czarnecki 2000] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000

[Deutsch 1984] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. *In Proceedings of the Tenth Annual ACM Symposiumon Principles of Programming Languages,* 1983, 297-302

[Feinberg 1996] Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Washington. *The Dylan Programming Book.* Addison-Wesley Longman, 1996

[Foote & Johnson 1989] Brian Foote and Ralph E. Johnson. Reflective Facilities in Smalltalk-80. In *OOPSLA '89 Proceedings,* 1989, 327-335

[Foote & Yoder 1998] Metadata. In Proceedings of the *Fifith Conference on Pattern Languages of Programs (PLoP '98)* Monticello, Illinois, August 1998. Technical Report #WUCS-98025 (PLoP '98/EuroPLoP '98) Dept. of Computer Science, Washington University September 1998

[Foote 2000] Brian Foote and Joseph W. Yoder, Big Ball of Mud, in *Patterns Languages of Program Design 4* (PLoPD4), Neil Harrison, et al., Addison-Wesley, 2000

[Gabriel 1991] Richard P. Gabriel, Jon L. White, Daniel G. Bobrow, CLOS: Integrating Object-Oriented and Functional Programming, *Communications of the ACM*, Volume 34, 1991

[Gamma 1995] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addision-Wesley, 1995.

[Grothoff 2003] C. Grothoff. Walkabout revisited: The runabout. In *ECOOP Proceedings*, 2003.

[Goldberg 1976] Adele Goldberg and Alan Kay, editors, with the Learning Research Group. *Smalltalk-72 Instruction Manual*. Xerox Palo Alto Research Center

[Goldberg 1983] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983

[Goldberg 1984] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA, 1984

[Goldberg 2004] David S. Goldberg, Robert Bruce Findler, Matthew Flatt. Super and inner: together at last! In *OOPSLA Proceedings* 2004, 116-129

[Hebel 1990] Kurt J. Hebel and Ralph E. Johnson. Arithmetic and Double Dispatching in Smalltalk-80. In *Journal of Object-Oriented Programming*, V2 N6 March/April 1990, 40-44

[Ingalls 1978] Daniel H. H. Ingalls. The Smalltalk-76 Programming System: Design and Implementation. In *5th ACM Symposium on POPL,* 1978, 9-15

[Ingalls 1986] D.H.H. Ingalls. A simple technique for handling multiple polymorphism*. In Proceedings of OOPSLA '86,* 1986.

[Johnson 1988b] Ralph E. Johnson, Justin O. Graver, and Laurance W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA '88 Proceedings*, 1988, 18-26

[Kiczales & Rodriguez 1990] Gregor Kiczales and Luis Rodriguez. Efficient Method Dispatch in PCL. In Proceedings of *the ACM Conference on Lisp and Functional Programming*, 1990, 99-105.

[Kiczales 1991] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow.*The Art of the*

Metaobject Protocol. MIT Press, 1991

[Kiczales 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP Proceedngs* 2001.

[Keene 1989] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Introduction to CLOS.* Addison-Wesley, 1989

[Krasner 1983] Glenn Krasner, editor. *Smalltalk 80: Bits of History, Words of Advice.* Addison-Wesley, Reading, MA 1983

[Kristensen 1990] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Language.* 1990

[LaLonde 1986] Wilf R. LaLonde, Dave A. Thomas and John R. Pugh. *An Exemplar Based Smalltalk.* OOPSLA '86 Proceedings . Portland, OR, October 4-8 1977 pages 322-330

[Leavens and Millstein] Multiple Dispatch as Dispatch on Tuples. In *OOPSLA Proceedings*, 1998, 274-287.

[McAffer 1995[Jeff McAffer. Meta-level Programming with CodA. In *ECOOP Proceedings* 1995, 190-214.

[Maes 1987a] Pattie Maes. *Computational Reflection.* Artificial Intelligence Laboratory. Vrije Universiteit Brussel. Technical Report 87-2, 1987

[Maes 1987b] Pattie Maes. Concepts and Experiments in Computational Reflection*. In OOPSLA '87 Proceedings*. 1987, 147-155.

[Moon 1986] David Moon, Object-Oriented Programming with Flavors, In *OOPSLA '86 Proceedings,* 1986 1-8

[Mougin 2003] Philippe Mougin, Stéphane Ducasse: OOPAL: integrating array programming in object-oriented programming. *In OOPSLA Proceedings*, 2003, 65-77

[Mugridge 1991] Warwick Mugridge, John Hamer, John Hosking. Multi-Methods in a StaticallyTyped Programming Language. In *ECOOP Proceedings*, 1991, 147-155

[Paepcke 1993] Andreas Paepcke (editor), Object-Oriented Programming: The CLOS Perspective, MIT Press, 1993

[Palsberg 1998] Jens Palsberg, C. Barry Jay, James Noble. Experiments with Generic Visitors. In *the Proceedings of theWorkshop on Generic Programming*, Marstrand, Sweden, 1998.

[Palsberg 2004] Jens Palsberg and J Van Drunen. Visitor oriented programming. In the Workshop for Foundations of Object-Oriented Programming (FOOL), 2004.

[Rivard 1997] Fred Rivard. *Evolution du comportement des objets dans les langages a classes reflexifs*. PhD thesis, Ecole des Mines de Nantes, France, June 1997

[Saltzer 1981] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design, *Second International Conference on Distributed Computing Systems* (April, 1981) pages 509-512.

[Scharli 2004] Nathanael Schärli, Andrew P. Black, Stéphane Ducasse: Object-oriented encapsulation for dynamically typed languages. In *OOPSLA Proceedings.* 2004, 130-149

[Smith 1983] Brian Cantwell Smith. Reflection and Semantics in Lisp. In *POPL Proceedings*, 1984, 23-35

[Stefik 1986a] Mark Stefik and Daniel G. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine* 6(4): 40-62, 1986

[Stroustrup 1986] Bjarne Stroustrup. *The C++ Programming Language,* Addison-Wesley, Reading, MA, 1986

[Thorup 1997] Thorup, K. K. Genericity in Java with virtual types. In *ECOOP Proceedings* 1997, 444-471.

[Ungar 1987] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA '87 Proceedings.* 1987, 227-242.