PolyD: A Flexible Dispatching Framework

Antonio Cunei
cunei@cs.purdue.edu

Jan Vitek
jv@cs.purdue.edu

Department of Computer Science
Purdue University
250 N. University Street
West Lafayette, IN 47907-2066

ABSTRACT

The standard dispatching mechanisms built into programming languages are sometimes inadequate to the needs of the programmer. In the case of Java, the need for more flexibility has led to the development of a number of tools, including visitors and multi-method extensions, that each add some particular functionality, but lack the generality necessary to support user-defined dispatching mechanisms. In this paper we advocate a more modular approach to dispatching, and we present a tool, PolyD, that allows the programmer to design custom dispatching strategies and to parametrize many aspects of the dispatching process. PolyD exhibits excellent performance and compares well against existing tools.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.3.3 [**Software Engineering**]: Language Constructs and Features—*Patterns*

General Terms

Languages

Keywords

Dispatching, Multimethods, Visitor Pattern, Java

1. INTRODUCTION

Object-oriented programming revolves around organizing data and code as distinct objects that communicate using messages. When a message is received by one object, the object responds in its own way, by using one of the available methods. The message is "dispatched", meaning that one of the methods is selected and invoked as a result of the arrival of the message. The details of the dispatching process, however, and in particular of the method selection, can vary considerably.

If, for instance, compile-time information is used exclusively in order to select the correct method, the only run-time action required will be the method invocation, a situation referred to as static resolution. Much more frequently, however, dynamic dispatching is used instead, meaning that the actual run-time class of one or more objects is used in the selection process. Most typed languages, including Java, rely on a combination of static resolution and *single dispatching*, in which the run-time selection of the method is based on the class of a single, distinguished, argument. The remaining arguments of the message are only inspected statically in order to resolve overloaded methods. Other languages implement a *multiple dispatching* mechanism, in which the run-time class of multiple objects is considered in order to select the most appropriate method [4, 19, 12]. Multiple dispatching, together with its generalization to predicate dispatching, has been studied in the context of extensions to Java [25, 8, 13, 16, 35] and from the point of view of static type-checking [14, 1, 7]. Multiple dispatching is an elegant tool that can help to solve programming problems that would otherwise require more complex workarounds. For instance, the classic Visitor pattern is a convoluted substitute for a straightforward double dispatching [29].

The goal of PolyD is to provide a *modular framework for user-defined dispatching mechanisms for Java*. We argue that when programmers are faced with the need for a dispatching mechanism distinct from what is provided by the language, a tool like PolyD provides them with a more efficient, flexible, and easy to use solution than hand-coded mechanisms. By emphasizing the separation and the modularity between the message send and the actual dispatching mechanism, PolyD allows the programmer to select different dispatching mechanisms in different parts of the code, to modify their choice of dispatcher during development or even at run time, and to customize various aspect of the standard dispatchers provided. In order to be practical, the design of PolyD is constrained by the following requirements. The framework should be *non-intrusive*. This means that changes to the syntax of Java or the tool-chain (*ie* source compiler, debugger, bytecode format) are not acceptable. The solution should be *portable* in the sense that it should run on any implementation of Java and thus virtual machine changes are ruled out. Dispatchers should be *modular* allowing users to redefine either the implementation of the client methods, target methods, or of the dispatcher, independently without requiring recompilation. The framework should be sufficiently *flexible* to allow user to modify the semantics of the standard dispatchers bundled with the tool. This paper presents the following main contributions:

- The idea of a modular approach to dispatching, enabling a greater flexibility in the choice of the dispatching mechanisms.
- The design and implementation of the PolyD dispatching

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

framework, which allows new dispatchers to be defined and used in the standard Java execution environment with minimal effort.

- A multi-method dispatcher for Java that has good performance and scales better than other tools when increasing the number of methods involved. An original handling of `null` values is also described.
- A tool that can be used as a teaching aid to show how different dispatching mechanisms cause the same code to behave differently.
- Examples of applications for which PolyD is useful.
- A performance evaluation of PolyD, comparing it against other tools.

2. MOTIVATING EXAMPLE

To see a practical example of the impact of the dispatching strategy on the code organization, consider the following, concrete example. The Ovm project, developed at Purdue University, is an open source framework for building language runtimes that includes an ahead-of-time compiler and tools for manipulating bytecode [41]. Ovm transforms incoming bytecode to an intermediate representation called OvmIR that is the common input of all execution modes (interpret/compile). We will first give some background on the issues involved in writing bytecode (or OvmIR) manipulation tools, then we discuss issues specific to dispatching.

2.1 A bytecode manipulation framework

In Ovm each intermediate representation instruction corresponds to a subclass of the `Instruction` class. Analyses written for the framework use the flyweight pattern [28] to avoid the need for multiple instances of the same instruction class. An `InstructionBuffer` class maintains the current program counter and interprets constants referenced from the bytecode stream. This technique allows instruction objects to retrieve and interpret their immediate operands without any state of their own. For example, the concrete instruction `GETFIELD` subclasses the abstract class `FieldAccess`. `FieldAccess` provides a method `getSelector()` to return information about the name and type of the field being accessed. The state required by the method is encapsulated in the instruction buffer argument. This design follows the flyweight pattern, allowing the `InstructionSet` class to hold a single instance of each concrete instruction.

The semantic specification of every instruction is provided in the constructor without parameters of the respective class. Each instruction implements the methods `size()` (to return its size in bytes), and `getOpcode()`. Further behavior is added by appropriate subclasses. Thus, manipulations of the OvmIR can rely on the type checker to prevent some errors, for instance trying to use a constant pool index as a jump target. The core of the instruction hierarchy used within Ovm is shown in Fig. 1. Such hierarchy reflects properties of the Java bytecode instruction set and is based on pragmatic considerations, not on any systematic analysis of features of all conceivable instructions of stack machines.

It turns out to be impossible to model all the features of bytecode using single inheritance. For example, `FlowChange` is an interface which is implemented by the concrete instruction `RET` (return) that inherits from `LocalAccess`. Another example is the `Throwing` interface which is implemented by all instructions that can throw exceptions.

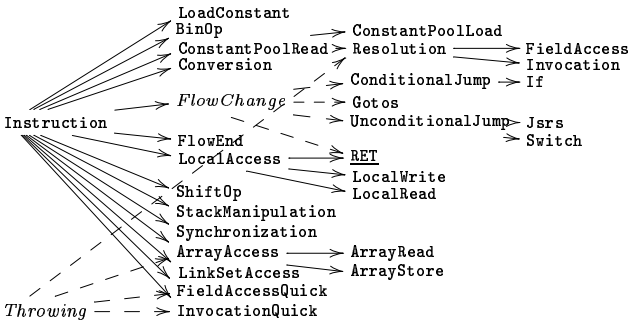


Figure 1: The instruction hierarchy. OvmIR consists of 237 concrete instructions (`RET` is the only one shown here). Italics indicate interfaces, all others are abstract classes.

2.2 Dispatching over instructions

The software architecture of Ovm has evolved over time. Originally, the instruction objects used dedicated methods to perform bytecode manipulation. The use of dedicated methods had the disadvantage that every additional analysis or processing step required changes to each instruction class. Thus, each of the instructions was extended with an `accept()` method and various analyses were written as visitors operating on the instructions.

In order to make code factoring easier, the instructions were arranged in the hierarchy of Fig. 1 to factor out commonalities between the instructions. The visitors that implement the various analyses are able to take advantage of the instruction hierarchy; the visit methods can be refactored using the hierarchical visitor pattern [39]. For example, our access modifier inference tool does not need to distinguish between Java's four field access operations (`GETFIELD`, `PUTFIELD`, `GETSTATIC`, `PUTSTATIC`). Using the hierarchical visitor pattern, we only need to implement a visit method for the abstract `FieldAccess` instruction class. In order to make the hierarchical visitor pattern work, a helper method that redirects calls from `visit(PUTSTATIC i)` to `visit(FieldAccess i)` is required. For an instruction, like `GETFIELD`, the parent class of all hierarchical visitors would have the following method:

```
void visit(GETFIELD i) {
   visit((FieldAccess) i);
}
```

The sole purpose of this method is to redispatch calls that are not implemented by the specific visitor to the parent type. This is a useful technique, as it allows us to factor out implementations applicable to a group of instructions. Writing this indirection code, while conceptually trivial, turns out to be cumbersome. Each time the instruction hierarchy evolves, the base-classes of the visitors need to be rewritten. With over 200 instruction classes it is difficult to track changes in the hierarchy. The use of the visitor pattern requires that every analysis supplies visit methods for *all* instructions. Thus, every change in the hierarchy of the instruction set requires updates to several visitors.

The problem was solved by replacing the use of visitors with the Walkabout pattern [42]. The Walkabout declares visit methods just like visitors, but instead of doing double-dispatch with `accept()` methods in the instruction objects, the appropriate visit methods are found by reflection. In Ovm, hundreds of `accept()` methods were removed from the instruction objects and hundreds of `visit()` methods that were either abstract (visitor interface), empty (default

base class) or indirecting to other visit methods (hierarchical visitor) became obsolete. The plain Walkabout was subsequently replaced with the Runabout [29], an alternate implementation of the same pattern that relies on dynamic bytecode generation in order to achieve higher performance.

2.3 The meaning of "most appropriate"

While the Runabout considerably simplified the structure of Ovm, a number of more subtle issues arose, originated by the algorithm chosen to select a suitable visit method in the various circumstances. Consider the following example, taken from the Ovm code that adds write barriers:

```
void visit(PUTFIELD i) { ...
 if (...)
  ...replaceInstruction().
        addPUTFIELD_WITH_BARRIER_REF();
}
void visit(PUTFIELD_WITH_BARRIER_REF i) { }
void visit(PUTSTATIC_WITH_BARRIER_REF i) { }
void visit(AASTORE_WITH_BARRIER i) { }
```

The code replaces, when appropriate, some occurrences of the `PUTFIELD` bytecode. However, some empty visit methods are necessary in order to restrict the manipulation to instances of `PUTFIELD`, while excluding instance of its subclass `PUTFIELD_WITH_BARRIER_REF`. The same applies to other instructions. While the hierarchy used makes sense in many other parts of the code, in this particular case we would rather have the opportunity to use a "non subsumptive" visitor, in order to make the code clearer and more maintainable, even if the hierarchy changes and new subclasses are added. The "most appropriate" method for the developer, in this case, would not be the one selected by the Runabout.

Other problems related to the method selection arise because of the complex hierarchy, which includes subclasses and subinterfaces, used to organize the set of instructions. The Runabout arbitrarily gives preference to `visit` methods encountered following the chain of superclasses over `visit` methods found through superinterfaces. Such a resolution algorithm caused several headaches during the development of Ovm, since it is difficult to track exactly which visit method will be called in all circumstances.

Even more tricky is the question of visitor-style dispatching when inheritance is used to organize visitors themselves. Consider the following example, taken from an Ovm bug report:

```
class C { }
class D extends C { }
class BR extends VisitorTool {
 void visit(D _) { }
}
class DR extends BR {
 void visit(C _) { }
}
```

What is the semantics that should be applied if a message `visit(new D())` is sent to an instance of `DR`? The role of the visitor tool is to discover the dynamic type of the parameter, and to invoke the "most appropriate" method, but there are actually various choices, of which the tool selects just one. For instance, the developer might design the various subclasses as successive refinements that together describe the visitor. In that case `visit(D)` in `BR` should be applied, being the most specific according to the argument type. On the other hand, the developer might regard every new subclass as an entirely new layer of implementation; in that case the `visit(C)` in `DR` should be preferred, as it is able to deal with a `new D()` and is more appropriate in the sense that it is defined in a more specific subclass. Some developers might be interested in distinguishing between the two cases according to the static type of the argument, or other factors.

In other words, developers require a degree of flexibility in the selection of the "most appropriate" method that is not granted by current tools. The lack of flexibility in the choice of the dispatching policy impacts adversely on the usability of the tool, on the ability of developers to give the desired structure to their code, and ultimately on code maintainability.

3. TOWARDS A MODULAR APPROACH TO DISPATCHING

The lack of flexibility in the existing tools is what prompted our research for a more versatile solution, and the subsequent development of PolyD. Rather than developing multiple tools, each suitable for individual situations, we explore how different approaches can be integrated within a single framework, and how the dispatching mechanism can be modularized in order to achieve the necessary degree of flexibility. We argue that such an approach is not limited to our current implementation, but it is in fact a general technique, applicable to object-oriented languages, that offers more powerful dispatching features to programmers.

3.1 Modular dispatching mechanisms: a general approach

Having multiple, but independent, dispatching mechanisms has the potential to be inefficient in many ways. First of all, if different calling formats or conventions are used it becomes difficult to replace one dispatching strategy with another at a later stage. Second, every dispatcher is implemented from scratch, despite the fact that parts of their inner workings are likely to be similar. Finally, the programmer has no direct way to customize relevant aspects of the dispatching process.

In order to make the dispatching procedure more general, we will instead use a modular approach, so that individual components can be replaced and customized. By abstracting and separating common elements, it becomes easier to add custom dispatching mechanisms, to alter their operation, to replace them in a modular fashion, and to obtain new functionalities operating directly at the dispatching level. We will use the term "dispatcher" to refer to any component that implements a specific dispatching mechanism, be it a part of the built-in language runtime or an additional API. In order to describe our modular approach to dispatching, we will progress in stages by decomposing dispatchers into their basic components.

3.1.1 Selection and invocation

Let us consider the structure of a generic dispatcher. The fundamental steps that every dispatcher, regardless of its exact operation, needs to perform at runtime are the selection of one of the available methods, according to the supplied arguments, and the invocation of the method selected. Figure 2 reflects this first subdivision in the internal structure, and the flow of data at runtime during a message dispatching.

In the most general case the dynamic selection of a method can potentially use not just the class to which the arguments belong, but their values as well. For instance, one of the message arguments could contain an index used to select from a set of available methods. Considering the abstract structure of dispatchers, we can make no assumptions about the specific method selection strategy in use. Conceivably, the choice of the method could depend on ex-

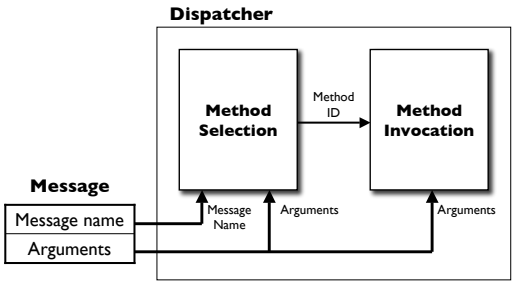


Figure 2: Basic structure of a dispatcher

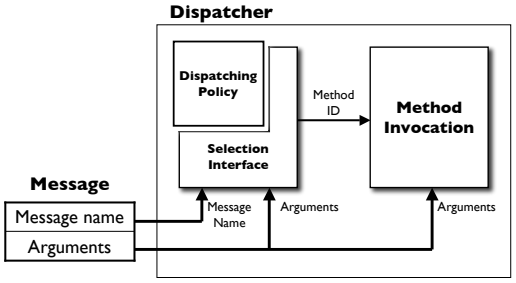


Figure 3: Separating the policy from caching and invocation

ternal factors or on the execution history. Different methods could be selected for successive identical messages with identical arguments. The selection could even be completely random.

3.1.2 Modularizing the selection module

While, in general, it is not possible to anticipate the way in which the selection module will choose one of the available methods, the dynamic dispatching strategies that are normally used in programming languages rely solely on the classes[1] of the arguments, and always select the same method given the set of applicable methods and the list of argument classes. Such a deterministic behavior, together with the computational cost of finding the most appropriate methods, leads naturally to a further modularization in the dispatcher structure, reflecting the introduction of a caching mechanism and a further confinement of the method selection policy.

The method selection module can be divided into two separate parts. The "dispatching policy" component will contain the core logic of the method choice, while the selection interface will deal with the implementation aspects, including caching. The general structure of a dispatcher assumes therefore the form shown in Figure 3, while the internal structure of the caching method selection module is shown in Figure 4.

An aspect worth emphasizing is that, by concentrating on the efficient implementation of the cache and the method invocation, all the dispatcher implemented according to this structure will share a high dispatching efficiency after the initial warm-up stage. Subtle implementation details such as cache synchronization issues, present in multithreaded code, are also removed from the core dispatching logic and can be dealt with just once for the whole system.

[1]We use the term "class" although "type" would be more appropriate, even more so since some arguments could be primitive values rather than objects. This use of "class" reflects the terminology used in the reflective API of Java, in which even primitives have a `Class` descriptor.

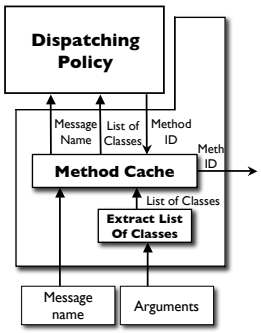


Figure 4: Caching the method selection

The mechanism enables developers to add new dispatching policies without putting an excessive emphasis on the optimization of their code, while focusing instead on other aspects such as correctness and code maintainability. The range of dispatchers made available by PolyD all share a high dispatching efficiency thanks to the use of the modular approach described here.

3.2 Building dispatchers

Having introduced a first modular structure for dispatchers, we can consider how different modules can co-exist, and the effects on the code that uses the resulting dispatchers.

The separation between method selection and method invocation suggests a first approach to the modular composition of dispatchers. In the simplest case, the invocation consists in little more than jumping to the method code, but it might also involve code loading or other operations. One implementation of the invocation module can be shared among multiple dispatchers. If that invocation module is modified or optimized, all of the dispatchers will benefit from the new implementation.

As a first step, we can replace distinct dispatchers, shown in Figure 5, with the structure shown in Figure 6.

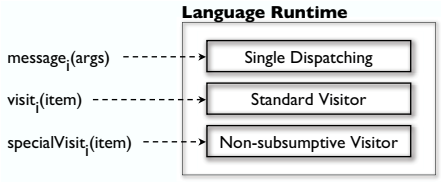


Figure 5: Separate dispatching subsystems

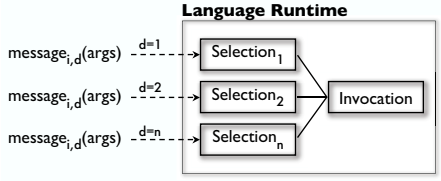


Figure 6: A more homogeneous structure

A single call format. In order to make the use of different dispatchers more symmetrical, different call formats are replaced by a single format, in which the chosen dispatching mechanism is involved in the message sending operation, potentially as a runtime argument. By using a single and consistent mechanism for using the different dispatchers in the client code, it becomes possible to replace dispatchers during development, or even dynamically at runtime. That can be of interest considering that, in the same way in which classes can be loaded dynamically, new dispatching policies suitable for specific class hierarchies can be loaded dynamically as well. The new dispatchers can then be used without any need for client code recompilation.

In a sense, moving from a fixed dispatcher for each call to a parametric call site, in which the dispatcher is a dynamic factor, is similar to the gain in flexibility obtained by moving from strict overloading to true dynamic dispatching: we gain the ability to decide dynamically not just that we want the "most appropriate" method for those arguments, but also what "most appropriate" means at that moment.

Having the ability to observe the behavior of the very same code using different dispatchers is also interesting from an educational point of view. Except for simple examples, determining the effects that overloading, or multiple dispatching, or different techniques have on more complex programs is not trivial. As we shall see later, PolyD allows students to actually run the same code using different mechanisms, and to compare critically the results.

Multiple invocation policies. While the basic invocation module only needs to call the selected method, there is no reason why multiple invocation policies should not be implemented. For instance, one invocation policy could keep track of the methods calls in order to profile the program, a different one could log all the arguments and the return values for debugging purposes, and yet another one could apply transparently security checks.

By creating custom invocation policies, or adapting existing ones, new features can be introduced at the level of the method call without having to deal with the complexity of the rest of the dispatcher. Having the ability to alter the invocation stage, in other words, allows the programmer to address cross-cutting concerns that are typical of aspect-oriented programming, without requiring a static code weaving. The new situation is reflected in Figure 7.

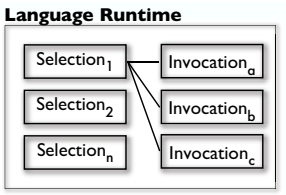


Figure 7: Modular composition of dispatchers

3.3 Method selection

An aspect that is not immediately obvious in Figure 4 is the determination of the set of available methods from which the dispatching policy draws the most appropriate element. The dynamic content of the message arguments is only part of the information used by a dispatcher to locate the correct method. The information available statically at each call site, namely the "static class" that each argument appears to have at that point in the code, also plays a factor. In the case of pure overloading, for instance, the static information is the only information used used to perform the selection. The diagram in Figure 8 shows the two-stage selection process commonly used in use in object-oriented languages.

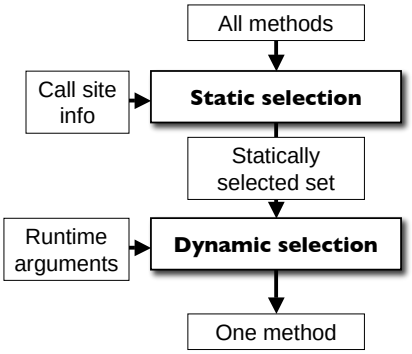


Figure 8: Static and dynamic method selection

Such a structure is reorganized in our case as shown in Figure 9. The specific selection algorithms are encapsulated in the dispatching policy component, while the remaining implementation aspects, including obtaining information for the call site and storing the statically preselected set, are demanded to a common framework. This code organization is well reflected by the simple Dispatching Policy API of PolyD, described later in Section 4.13.

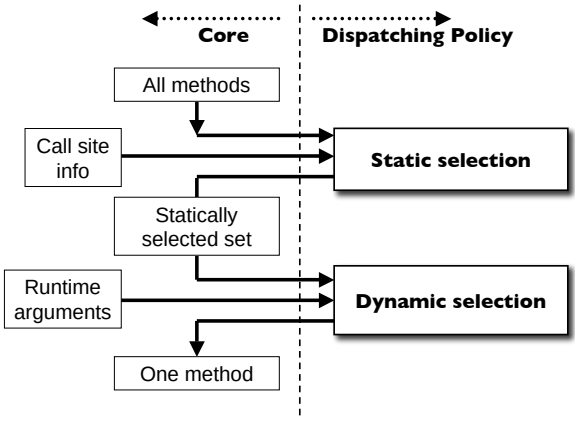


Figure 9: Separating selection from implementation

3.4 More open choices

The dispatchers built into programming languages rely on a series of design choices, which cannot be altered by the programmer. We intend to allow the progrmmer to customize some of those choices, allowing a greater degree of flexibility. Among the most obvious choices are the handling of inheritance (whether to allow multiple code inheritance, for instance), and the treatment of ambiguities. About the latter, ambiguities can be rejected, or precedence lists can be used to resolve them (as in the case of CLOS/C++).

There are also other, possibly less obvious facets of the dispatching process that are also usually fixed. However, there is no intrinsic reason to deny to developers the ability to alter those aspects in the way that they see fit. We will now discuss briefly two notable

aspects, and in Section 4 we will show how they can be controlled in our framework, PolyD.

3.4.1 Null arguments

One area in which the design of the dispatching mechanism involves elements of arbitrary choice is the handling of `null` arguments. In the standard Java language, `null` belongs to a special nameless type (there is no associated keyword). While it is not possible to associate `null` with any class, `null` is always a legal argument when an object is expected, so the dispatcher must be ready to deal with that special case. Since there is no run-time class to base a method selection on, the implementation can adopt different strategies.

A possible choice is considering all `null` argument invalid, causing the dispatching code to fail, for instance throwing an exception. That behavior is similar to the way in which Java natively operates while dispatching on a single receiver. The rationale is that it is not legal to send a message to `null` since the value does not represent a valid object. In the case of multimethods, however, the position is less tenable. If all arguments to all calls are involved while performing the dispatching, `null` values would be disallowed as arguments altogether, which is an unreasonable restriction.

A different approach, with several sub-cases, consists in specifying a way to obtain a default behavior if one or more of the arguments are `null`. One possible choice is to select the method with the most specific applicable combination of classes (if unique). This strategy, used by Java when resolving overloading, is linked to the idea that the null type is a subtype of every other type in the system. For example, consider the classes `A`, `B`, `C`, and `D`, with `D` subclass of `C`, `C` of `B`, and `B` of `A`, and the following definitions:

```
static void alpha(A x) { result("A"); }
static void alpha(B x) { result("B"); }
static void alpha(D x) { result("D"); }

alpha(null);
--> Result is "D"
```

The most specific method, according to the resolution mechanisms used by Java, was chosen. Let's suppose that class `D2` is also a subclass of `C`, adding another method:

```
static void alpha(D2 x) { result("D2"); }

alpha(null);
Over.java:10: reference to alpha is ambiguous, both
method alpha(D) in Over and method alpha(D2) in Over
match
 alpha(null);
 ^1 error
```

In this case the compiler is unable to find a single most specific method, and compilation aborts. That only happens when Java statically knows that the argument is `null`. While conceptually correct, this strategy is not always the best choice on a practical level. In Java `null` is not a real object, and it is not possible to write methods that accept specifically `null` arguments. The presence of `null` indicates instead a special situation that should be handled appropriately, and a different strategy might be desirable. One option is specifying a particular default methods that should be used. An alternative is looking for the most general method applicable for a certain combination of classes. Other strategies can also be devised, depending on the specific circumstances.

3.4.2 Missing methods

Some dispatching policies cannot determine statically whether a message will always find a valid matching method. For example, a "non-subsumptive" dispatcher must examine the exact dynamic type of the arguments to determine whether any method can be applied. Similarly, other strategies may need a special way to handle messages with no matching methods. There are different possible alternatives: ignoring the message, throwing an exception, or taking some user-defined action. The exact definition of the handler is an additional customizable element in the definition of a dispatcher.

We have until now discussed the ways in which dispatchers can be made more modular and customizable. It is now to put those ideas into practice by creating a concrete modular dispatching infrastructure. The next section describes the implementation and the features of PolyD, our flexible dispatching framework.

4. POLYD

PolyD is a pure Java tool that, using dynamic bytecode generation, allows the user to define customized dispatching policies, altering many aspects of message dispatching that are usually predetermined and that cannot be easily changed.

In other words, using PolyD it is possible to define a set of methods and invoke them according to user-defined criteria. For example, PolyD can be used to implement a visitor-like mechanism, or general multiple dispatching, or more unusual dispatching policies. PolyD makes it possible to personalize many aspects of the dispatching process: the handling of null arguments, of missing methods, of ambiguities, of the method invocation, and so on. In that respect, PolyD is a more general and flexible solution to the "expression problem" than other tools.

As an added advantage, PolyD only uses standard Java, and does not require special syntax, special bytecode, preprocessors, or custom virtual machines. The performance of the tool is well-suited to real-life applications: for unary methods (methods with a single argument) PolyD has performance similar, or even better, than the Runabout and the Sprintabout; for other arities we find that PolyD scales better than MultiJava and JMMF (more details on these tools are available in Section 7).

Let's see an introductory example.

4.1 QuickStart

Let us assume that we want to describe the effect of a `Person` dancing in a given `Place`. Using PolyD we shall define a suitable dispatcher that selects the appropriate method. In this example we will use multiple dispatching, although different policies could be used. Java 1.5 annotations are used to denote PolyD specifications, called "tags" in the text.

PolyD cleanly separates the interface that specifies the messages from the implementing methods. This is the interface definition:

```
@ PolyD
@ DispatchingPolicy (MultiDisp.class)
interface Dance {
  void dance(Person p,Place q);
}
```

The `@PolyD` tag is used as a marker to introduce a PolyD interface. The `@DispatchingPolicy` tag informs the framework that all specified methods will use that dispatching policy. We can now write the implementing methods.

```
class Impl {
 void dance(Dancer p, Stage q) {
  printComment("Dance is an expression of art!");
 }
 void dance(Person p, Stage q) {
  printComment("What is that guy doing on the stage?");
```

```
 }
 void dance(Person p, Place q) {
  printComment("That person is dancing. Strange.");
 }
}
```

The methods describe three possible responses to various combinations of arguments. The dispatcher can now be used:

```
Person joe     = new Person();
Place  office  = new Place();
Person nureyev = new Dancer();
Place  bolshoi = new Stage();

Dance d = PolyD.build(Dance.class,new Impl());

d.dance(joe,bolshoi);
d.dance(nureyev,bolshoi);
d.dance(nureyev,office);



> What is that guy doing on the stage?
> Dance is an expression of art!
> That person is dancing. Strange.
```

Note that in all three cases the arguments to `dance()` are statically a `Person` and a `Place`, but nothing more specific. Since multiple dispatching is used in this example, however, dynamically the most appropriate methods are chosen. We could try, in contrast, to use overloading in the same example:

```
@ PolyD
@ DispatchingPolicy (Overloading.class)
interface Dance2 extends Dance {}
Dance d = PolyD.build(Dance2.class,new Impl());
```

The result is now:

```
> That person is dancing. Strange.
> That person is dancing. Strange.
> That person is dancing. Strange.
```

The same client code can be used with both dispatchers, obtaining a different behavior according to the different policy.

4.2 More features

PolyD allows dispatching on an arbitrary number of receivers, return values are supported as are primitive types. Interfaces can define any number of prototypes, each using different dispatchers. Multiple bodies can be specified with a single interface, enhancing the opportunities for code reuse and suggesting a “mixin-like” approach:

```
Dance d = PolyD.build(Dance.class,body1,body2,body3);
```

All of the methods available in the various bodies will be combined to satisfy the prototypes declared in the interface. It is also possible to share one body among multiple dispatchers, and to call the same group of methods using different names.

The standard dispatching policies available are multiple dispatching, overloading, and a ”non-subsumptive” policy that only calls a method if the classes of the arguments match exactly those of the method parameters. The policy that implements multiple dispatching policy also offers a form of symmetric multiple inheritance, treating equally interfaces and classes (MultiJava, in contrast, only deals with subclasses). It is possible to define personalized dispatching policies using a simple API, described later. Similarly,

```
@PolyD
@DispatchingPolicy(MultiDisp.class)
@InvocationPolicy(DebuggingInvocation.class)
public interface Dance {
  Person test(long i,@IfNull(Place.class) Place p);

  @Preload({
   @Seq({Person.class,Office.class}),
   @Seq({Worker.class,Workplace.class})
  })
  void dance(Person p,Place q);

  int four(String a,char c,Object o,Place p);

  @Name("dance")
  @DispatchingPolicy(NonSubsump.class)
  @OnMissing(Missing.IGNORE)
  void nonSubsumpDance(Person a,Place b);

  @Name("dance")
  void danceSuper(@As(Person.class) Person a,Place b);

  @Name("visit")
  void visitAsPlace(@As(Place.class) Place p);
}
```

Figure 10: A more complex PolyD interface

is is possible to define custom invocation policies, which define the operations that should be performed when a method is called. For instance the policy can log all method calls, or inspect the arguments on-the-fly for debugging purposes or security checks, or gather statistics.

It is possible to specify exactly what should happen if null arguments are encountered (an aspect that is rarely customizable in programming languages), restrict the interpretation of the dynamic class of arguments in order to implement variations on the "super" concept, and define custom handlers for messages that cannot be satisfied by any methods according to the dispatching policy in use. Included is also a Runabout emulation layer, that allows existing programs that use that tool to take advantage of the new features without sacrificing compatibility. Figure 10 shows a more complex example of a PolyD interface. All of the features shown in that example are explained in detail in the following sections.

4.3 Building a Dispatcher

The construction of a dispatcher takes place using `PolyD.build()`. As previously mentioned, it is possible to use multiple bodies for every interface, as follows:

```
Interf d=PolyD.build(Interf.class,a,b,c);
```

where `a`, `b`, and `c` are instances of three different classes. The methods of all those classes are combined to build the final dispatcher. Only the methods specified in the interface are used to build the dispatcher; other methods in the bodies are treated as support methods. The classes used for the bodies are not required to implement the interface.

A method specified in the interface, with a certain name and arity, causes all public methods with same name and arity to be included in the custom dispatcher. Multiple methods with same name and arity can be specified in the interface. For instance:

```
@ PolyD
@ DispatchingPolicy (MultiDisp.class)
interface Several {
 void test(A a, X x);
 void test(X x, B b);
}
```

The method `test()` will be usable on arguments that respect the shown combinations, but not others. The methods used in the interface can be of arbitrary arity, and use and return any values, including primitives.

4.4 @DispatchingPolicy

The tag `@DispatchingPolicy` is mandatory, and specifies the way in which one method is selected when the dispatcher is used. The tag can be added to the interface or to individual prototypes. The tag on individual prototypes overrides the specification for the whole interface. If all prototypes are individually tagged, the tag on the interface is optional. Methods with the same name can be resolved, if desired, in different ways. For example:

```
@ PolyD
interface Several {
 @ DispatchingPolicy(Overloading.class)
   void test(A a);
 @ DispatchingPolicy(MultiDisp.class)
   void test(C b);
}
```

If `C` and `A` are not related, the behavior is obvious. If, just to make a convoluted example, `C` is a subclass of `A`, the policy in use depends on which of the two prototypes is chosen by Java, according to its own overloading mechanism. If the argument is statically known to be at least a `C`, multiple dispatching will be used, otherwise overloading is used instead. The standard dispatching policies available are listed in the following subsections. Section 4.13 explains how to create custom policies.

ovm.polyd.policy. MultiDisp

The policy implements a fully symmetric multiple dispatching, considering inheritance through subclasses and subinterfaces as equivalent. Consequently, the policy also implements a form of multiple inheritance simply by specifying different methods that accept instances of different classes or interfaces that have a common descendant.

If a dispatcher contains a method which uses this policy, and the call `PolyD.build()` is completed successfully, all subsequent method invocations will always find a matching method. In other words, there will never be a `MissingMethodException`. The policy is also able to detect most ambiguities at dispatcher building time, although some might be undetectable at this stage because of Java's dynamic loading. For example, a class and an interface that appear unrelated might acquire a common descendant at any point in time, generating a late ambiguity. The conflict, however, can be discovered during the first dispatching that involves the new class. If the list of combinations of arguments that are to be used with a certain method is known in advance, then all ambiguities can be detected at dispatcher building time (see Section 4.11). While our default implementation of multidispatching makes sure that only a single method will ever be eligible for a certain combination of arguments, other policies can be easily created to use instead precedence lists or any other resolution technique.

ovm.polyd.policy. Overloading

The policy mimics the usual static resolution adopted by Java on the list of arguments. All ambiguities are detected statically (partly by Java itself) and, if the call `PolyD.build()` completes successfully, no `MissingMethodException` will ever be thrown.

ovm.polyd.policy. NonSubsump

The resolution rule used by the `NonSubsump` policy uses the method implementation whose list of parameter classes matches *exactly* the list of classes of supplied arguments. So, if we have a method defined on "`FieldAccess`", for example, the method will be called on instances of `FieldAccess` but not on instances of its subclasses. If a call is made and the policy cannot find an implementation that matches exactly, the call is ignored. Such behavior can altered if desired, as explained in the next sections.

ovm.polyd. MissingMethodException

Some dispatching policies might be unable to establish at dispatcher building time that all subsequent method calls will be successful. If a dispatching policy is unable to find a suitable method for a given combination of arguments, a `MissingMethodException`, although every policy can define a custom response.

4.5 @OnMissing

The tag `@OnMissing` is used to describe the preferred handling of those messages that have no matching method according to the policy in use. If a tag is specified for the whole interface, and a different one is specified for a single prototype, the local one overrides the global selection. The options accepted are the following:

ovm.polyd.Const.Missing. STANDARD

The handling of the error situation is demanded to the `onMissing()` method contained in the selected dispatching policy. It is possible to customize this aspect by creating a subclass of the desired policy and overriding `onMissing()`. This option is the default.

ovm.polyd.Const.Missing. IGNORE

When a message does not match any method, the call is ignored. If the method is supposed to return a value of some sort, a dummy result is returned instead (zero, or null, or false).

ovm.polyd.Const.Missing. WARN

As above, but a warning message is additionally printed on the standard error stream.

ovm.polyd.Const.Missing. FAIL

The request for a message that does not match any available method (according to the selected dispatching policy) causes an exception `MissingMethodException` to be thrown.

ovm.polyd.Const.Missing. ABORT

If a matching method cannot be found, execution aborts.

4.6 @InvocationPolicy

In PolyD it is possible to specify additional actions that should be executed when a method is called. A special invocation policy can be attached to the whole interface or to individual prototypes. If no invocation policy is specified, the method is simply called; this is the default policy and also the faster mechanism. The use of a custom invocation policy imposes some overhead during dispatching.

ovm.polyd.policy. PlainInvocation

Only supplied as an example, the `PlainInvocation` policy performs a simple invocation, and returns the result supplied by the called method. This policy can be used as a template to create customized invocation policies.

ovm.polyd.policy. DebuggingInvocation

By adding this invocation policy to any interface or prototype, all method calls will be logged to the standard output stream, together with their arguments and their return values.

4.7 @IfNull

The value `null` has no class associated to it, and it it not possible to extract the list of classes of the arguments necessary to determine which method is the more appropriate one according to the given policy. When one or more arguments can be `null`, PolyD offers different ways to specify the default behavior. The tag `@IfNull` can be added to an argument of a prototype in order to specify the class that should be used when that argument is `null`. For example:

```
void test(long i,@IfNull(Place.class)Place p);
```

The class used in the `@IfNull` must be equal or an arbitrary subclass of the corresponding argument. Independent `@IfNull` tags can be added to the various parameters. The use of this construct adds just a marginal overhead to the dispatching cost, and it is the most convenient solution when good efficiency is desired. If a more general mechanism is desired instead, it is possible to use the `remapNull()` facility, described at the end of Subsection 4.13.1.

4.8 @As

In PolyD it is possible to override the dynamic interpretation of the class of arguments using the `@As` tag. For example:

```
void dance(@As(Person.class) Person a,Place b);
```

The class specified by the tag can be identical or any superclass of the class of the parameter, as long as compatible methods exist in the bodies supplied to build the dispatcher. The main application of the `@As` tag is the implementation of a generalized form of "super", that can be applied also in case of multiple inheritance or multiple dispatching. In that sense, specifying an `@As` class is similar to the qualified super form available in C++. In PolyD, however, the class can be any ancestor of the class of the parameter, and not necessarily a direct superclass. Multiple `@As` tags can be specified for the different parameters. While this construct is sufficient to replicate the functionality of "super" for individual prototypes, it might be more practical to define a more general "super" mechanism. That can be achieved by defining a custom dispatching policy that selects, depending on the context, the correct method.

4.9 @Name

It can be useful to bind together prototypes and methods even if their names differ. That effect can be obtained using the @Name tag, as in the following example:

```
void dance(Person a,Place b);
...
@Name("dance")
@DispatchingPolicy(NonSubsump.class)
void nonSubsumpDance(Person a,Place b);
void dance(Person a,Place b);
...
@Name("dance")
void danceSuper(@As(Person)Person a,Place b);
```

The implementations of `dance()` will also be accessible through the name danceSuper(), but in that case the first argument will always be interpreted as a Person. The tag `@Name` can be applied to prototypes in the interface as well as to individual methods directly in the bodies.

4.10 @Self

The tag `@Self` can be applied to variables, declared in the implementing bodies, in order to allow a method to call another method using the same dispatcher that was used to reach the current method in the first place. For example:

```
class Body {
  @Self Interf self;

  void m(B x) {
   self.m2(x);
  }
}
```

In the above example, the variable `self` will be automatically initialized when a dispatcher is created using the interface `Interf`, as in:

```
PolyD.build(Interf.class,new Body());
```

There can be multiple variables tagged with @Self, and they may refer to different interfaces. If a single body is shared among multiple dispatchers (which use distinct interfaces), each variable will be initialized when the dispatcher corresponding to that interface is built. For instance:

```
class Body {
  @Self OverloadingInterface over;
  @Self MultidispInterface multi;

  void m(B x) {
   over.m2(x);
   multi.m2(x);
  }
}
Body b=new Body();
y=PolyD.build(OverloadingInterface.class,b);
z=PolyD.build(MultidispInterface.class,b);
y.m(...)
```

4.11 @Preload

Each dispatching policy is free to decide how much checking should be done statically (at dispatcher building time) or rather dynamically. For example, the standard policy `MultiDisp` performs an extensive static checking making sure that, if the dispatcher is built successfully, all successive method invocations will always find a matching method. The `MultiDisp` policy also tries to discover as many ambiguities in the method definitions as possible.

Java, however, is founded on dynamic class loading, and that implies that new classes may introduce new ambiguities or conflicts at a later stage, according to the rules of each particular dispatching policy. Consequently, some of the checks could require a lazy approach, done after the main dispatcher construction. Such additional checks are only required for messages involving new classes,

and the results are still saved by the caching subsystem, so the overhead is limited. If the classes that will be used to dispatch messages are known in advance, and it is preferable to force an early detection of potential error conditions, the tag `@Preload` can be used to force the same checks in an eager fashion. This is an example of `@Preload` in action:

```
@Preload({
  @Seq({Person.class,Office.class}),
  @Seq({Worker.class,Workplace.class}),
  @Seq({Dancer.class,Place.class}),
  @Seq({Dancer.class,Office.class})
})
void dance(Person p,Place q);
```

The specified combinations of classes will be checked and preloaded eagerly in the method cache of the dispatcher.

4.12 @Raw

The `@Raw` tag can be used to pass arguments directly from the call site to the method selector. Such arguments will be removed from the argument list prior to the actual method call. Only objects or integers can be used as raw arguments.

Raw arguments can be used, for example, to identify specific call sites, even if the regular method arguments are the same; that information can be used to implement general forms of "super". For instance, let us have a class C subclass of B, and B subclass of A.

```
class X {
 void visit(C a) {
  ...
  next.visit(X.class,C.class,a);
 }
 void visit(A a) {
  ...
  next.visit(X.class,A.class,a);
 }
}
class Y {
 void visit(B a) {
  ...
  next.visit(Y.class,B.class,a);
 }
}
```

In this example, the first two arguments to the visit message identify the specific call site, so that the method selector can decide which one is the appropriate "next" method to call. The raw arguments are used by the selector and removed, while the third argument is passed on to the following visit method.

Another possible application of raw arguments is marking the remaining arguments in order to modify their interpretation. For instance, an enum class can define multiple states, and each raw specification can modify the following argument:

```
d.message(WHITE,a,RED,b,RED,c,WHITE,d);
```

the raw arguments are separated and passed to the method selection. Once the correct method is selected, taking into account the given argument modifiers, the proper `message(a,b,c,d)` will be called.

4.13 Custom Policies

It is possible, in PolyD, to define personalized dispatching and invocation policies. The API is rather simple and the standard policies can be used as templates to develop new ones. This section can be used as a general reference about the main aspects involved.

4.13.1 Dispatching Policies

Each dispatching policy defines different aspects of the method selection and dispatching. The following are the main calls involved in the policy definition.

compatibleSet. The routine `compatibleSet` performs a static preselection, finding in the supplied set of methods those that can are applicable for a given call site, for this dispatching policy. This routine can also be used to perform a consistency check on the set of supplied methods, discovering duplicate methods, violation in covariance rules, ambiguities, conflicts, and so on. If the selection performed by the dispatching policy is entirely dynamic, `compatibleSet` can just return the whole array of method given as argument, without performing any preselection. In this case it is not necessary to override, in the user-defined policy, the default implementation.

The list of classes corresponds to the classes that can be determined statically for a given call site. However, such list is not necessarily the actual list of specific static types of the arguments, but it depends on what Java can discriminate according to the list of prototypes in the interface used to build the dispatcher. An example is required to make this aspect clear. Let's assume that we have a class `A`, its subclass `B`, and a subclass of the latter `C`.

```
interface I {
 void m(A,B);
 void m(B,C);
}
...
 d.m(c,c);
 d.m(b,c);
 d.m(b,b);
 d.m(a,b);
```

Even if we know statically that `c` is of class `C`, `b` of `B`, and `a` of `A`, the four call sites will only be discriminated according to what Java knows according to the interface. The first two calls will be determined statically to be `(B,C)`, the last two `(A,B)`. It is important to keep this aspect present when implementing a policy that has a component of static resolution.

bestMatch. The dynamic counterpart of the previous call is the method `bestMatch`, which determines in the preselected set the one and only method that is more appropriate for the list of classes supplied, representing the actual dynamic classes of the arguments supplied by the message. The function should return the index in the array of methods corresponding to the best match for the given classes. If no suitable method is found, `bestMatch` returns -1.

handleMissing. The default behavior in case a suitable method cannot be found is specified by the method OnMissing, defined in each dispatching policy. Such method can be overridden in order to implement the most appropriate handler for the specific case. The routine `handleMissing()` accepts as arguments the list of classes that caused the special handling and the set of applicable methods (which can have various names because of the `@Name` tag).

disableCaching. The standard method caching mechanism offered by PolyD is disabled for this policy if this method returns `true`. That enables the definition of policies that select different methods at different times for the same combination of argument classes. It can also be useful for debugging the policy.

4.13.2 *remapNull*

A general mechanism used to handle `null` arguments is offered by the `remapNull()` routine. When a `null` argument is encountered during dispatching, and no `@IfNull` clause is specified for the corresponding parameter, control is transferred to the `remapNull()` routine associated with the dispatching policy in use. The routine `remapNull()` can be used, for instance, to find the most general method that applies, or the most specific one, or to take whatever action the programmer sees fit. The routine is supplied the unexpected sequence of argument classes, including nulls, and the list of methods that are applicable to that call site. The result, if the remapping is successful, is a new combination of classes, appropriate to the context, that will be used to resume the dispatching process. If no custom behavior is defined, a `NullPointerException` is thrown by default.

4.13.3 *Invocation Policy*

The structure of an invocation policy is rather simple. A single method `invoke()` needs to be defined:

```
public Object invoke(Object obj,Method m,Object[] args)
```

The method should perform all the additional operations required by this policy and call the supplied method of the given object using the given arguments. If some of the arguments are primitives, they are wrapped and unwrapped following the conventions used by the `invoke()` method of the reflective Java API.

4.14 Compatibility with pre-1.5 Java

By default PolyD uses Java annotations, and generates bytecode compatible with Java 1.5. However, a pre-1.5 version of PolyD is also automatically generated from the main source tree. All of the features described thus far are therefore also available to older Java VMs, using a specific API. The calls are necessarily more verbose, but they are conceptually no more complicated than using the annotations that we have seen so far. In the following examples some casts will be omitted for the sake of clarity.

4.14.1 *Descriptors*

The construction of dispatchers using the pre-1.5 compatible API relies on Descriptors, which are used to accumulate the kind of information that PolyD usually obtains exploring the annotations on interfaces and bodies. A new descriptor is created using:

```
Descriptor d1=new Descriptor(Interf.class,
      new Class[] {BodyA.class,BodyB.class});
```

This descriptor will be used to construct a dispatcher that uses the interface `Interf` and two bodies of class `BodyA` and `BodyB`. The dispatching policy can be added to the descriptor using:

```
d1.setDispatching(MultiDisp.class);
```

Similarly, the interface-wide handler for missing methods and invocations policy are set using calls similar to the following:

```
d1.setInvocation(DebuggingInvocation.class);
d1.setMissingHandling(Missing.Ignore);
```

The properties of each individual method can be set by extracting reflectively the method reference, and then using the appropriate calls as follows:

```
mt=Dance.class.getMethod("visit",new Class[]{Place.class});
d3.setMethodDispatching(mt,MultiDisp.class);
d3.setMethodName(mt,"lxn");
d3.setMissingHandling(mt,Missing.Ignore);
d3.setMethodPreload(mt2,new Class[][]
  {{String.class,Object.class},{Object.class,Place.class}});
d3.setMethodAsClasses(mt2,new Class[]{Place.class,null});
d3.setMethodNullDefaults(mt3,new
Class[]{null,Object.class,null});
```

The calls are self-explanatory, and mirror the annotations already described. In the case of the calls `setMethodAsClasses()` and `setmethodNullDefaults()`, each position in the array is `null` at the position in which no default is specified. The last property that can be specified is the use of `@Self` variables:

```
f=Body.class.getField("self");
d3.setSelfField(f);
```

Once the descriptor is ready, it is possible to proceed with the creation of the actual dispatcher, using the facilities described in the following subsections.

4.14.2 *Factories*

The more direct way to create a dispatcher is the use of a factory. The creation of factories, and the creation of new dispatchers, is illustrated by the following example:

```
Factory fact=d5.register();
Interf disp1=f1.getDispatcher1(bod1);
Interf disp2=f1.getDispatcher1(bod2);
```

This approach minimizes the time required to build a new dispatcher. If the descriptor was created specifying one body, the method `getDispatcher1()` should be used, if two bodies are used then use getDispatcher2() and so on up to `getDispatcher4()`. If more than four bodies are used, the method `getDispatcherN()` will accept an array of bodies. It is the responsibility of the programmer to pass to `getDispatcher()` bodies that are compatible with the list of classes used to build the descriptor. If the requirement is not satisfied, the dispatcher creation will abort.

4.14.3 *Registered Dispatchers*

If, due to the structure of the client, it is not possible or practical to pass around a factory, it is still possible to create dispatchers after the `register()` operation as follows:

```
d5.register();
...
Interf disp1=buildFromDescriptor(Interf.class,bod1);
Interf disp2=buildFromDescriptor(Interf.class,bod2);
```

After a descriptor is registered in the system, the creation of a new dispatcher using buildFromDescriptor() will look for the more recently registered descriptor associated with the supplied interface and classes of bodies, and will use the corresponding implementation to build new dispatchers. The mechanism is functionally equivalent to the use of factories, except for the speed penalty involved in looking up into the internal maps to find the correct implementation.

5. RUNABOUT EMULATION

In order to verify the usability of PolyD as a general tool, we have integrated the framework in existing applications. The first

large application was Ovm, the open source framework for building language runtimes developed at Purdue University. The source of the project comprises about 400,000 lines of code (plus libraries), and many of the internal operations of the framework are organized in a visitor-like fashion. The second application we considered was Kacheck/J, an encapsulation checker for Java which analyzes the use of confined types in programs [30]. In order to integrate PolyD in the existing code base, we have developed a Runabout emulation layer exploiting the ability of PolyD to integrate new dispatching strategies into its structure with minimal effort.

The core method resolution strategy of the Runabout was converted into a custom dispatching policy for PolyD, reproducing faithfully the mode of operation of the original tool. The new policy consists of less than 150 lines of code. In order to ease debugging, two new invocation policies were added in order, respectively, to perform a customized logging and to keep a count of all the creations of new dispatchers and the number of times each of them is used. The two new invocation policies had less than 50 and 75 lines of code respectively. Using these extremely simple modules, the existing dispatching engine of PolyD was very easily converted into a functional replica of the Runabout, while making no modification to the dispatching core. Using this emulation layer, PolyD, has then been integrated in the applications that we mentioned earlier. As discussed later, the resulting performance was comparable to the original, showing even some gain. The additional features of PolyD (the custom invocation policies, for example) are made available to the applications without requiring relevant changes to the client code. The following subsections list the main components included in the emulation layer.

ovm.polyd.legacy. RunaboutBis

`RunaboutBis` is a rather accurate replacement for the Runabout class. It does support the standard `visitAppropriate(Object)` and `visitAppropriate(Object,Class)`, it supports `the method visitDefault()` and it handles primitives. It does not emulate the special call `addExternalVisit()`, but it is an otherwise quite faithful emulation.

ovm.polyd.legacy. RunaboutCore

This emulation is similar to `RunaboutBis` but it offers no support for primitives and is slightly faster. The features are otherwise the same.

ovm.polyd.legacy. RunaboutStat

`RunaboutStat` is based on `RunaboutCore` and similar features. While it runs, it keeps a track of all the dispatchers created and invoked. The final report can be obtained by calling RunaboutStats.printStats().

ovm.polyd.legacy. RunaboutQuick

The RunaboutQuick is a faster implementation that directly uses factories to speed up the creation of new dispatchers. It always ignores missing methods and does not handle primitives. The source of the client code requires minimal modifications to the dispatcher creation to take advantage of this implementation.

ovm.polyd.legacy. RunaboutDisp

This dispatching policy contains a port of the core method selection strategy used by the Runabout. The module is a perfectly standard PolyD dispatching policy, and therefore it can be used independently from the emulation layer, subclassed if desired, and further customized.

6. PERFORMANCE

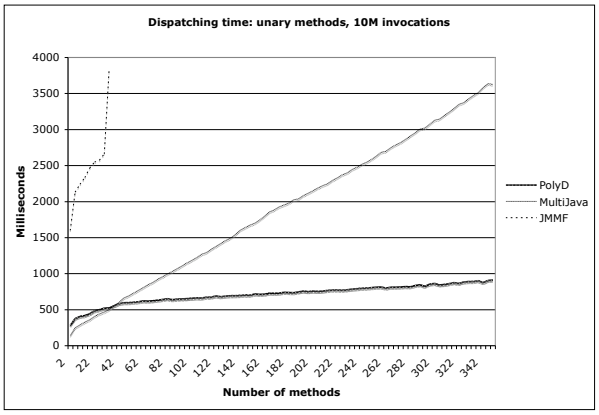
The performance levels offered by PolyD for single-argument methods are competitive with those of related, but less general, tools, as will be shortly shown. In the case of multiple-argument methods, we find that PolyD can offer higher dispatching speed than other multidispatching tools, exhibiting a dramatically better performance when increasing the number of methods. We now briefly describe the implementation mechanism, and subsequently discuss the benchmarking tests.

6.1 Implementation details

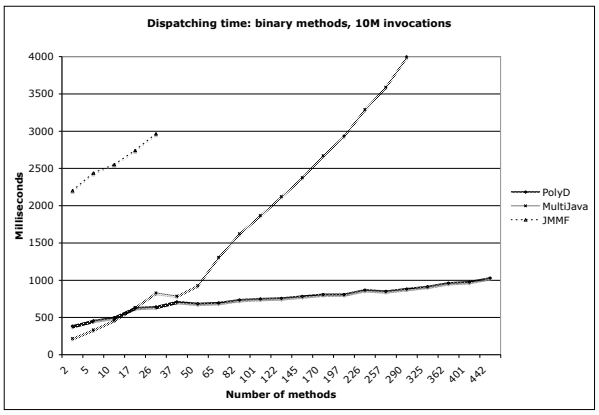
The implementation of PolyD has been tuned to offer performance levels suitable for concrete, realistic uses. PolyD was integrated and tested in real applications, in particular in the Ovm framework, which comprises about 400,000 lines of code and makes intense use of visitor-like dispatching. In order to maximize the dispatching efficiency, PolyD combines a reflective approach with dynamic bytecode generation and caching, getting inspiration by its single-argument predecessor, the Runabout. In the case of PolyD, the construction of the dynamically generated bytecode is considerably more complicated because of the many parametric aspects involved. Our code generator relies on the ASM framework [10].

When a new combination of interface/bodies is encountered for the first time, the annotations that define the behavior of the new dispatcher are parsed and checked. A new synthetic class is then created on-the-fly, so that all of the prototypes defined in the interface are implemented with an adaptor routine that performs the actual dispatching. The compatibility between the prototypes and the list of available bodies is verified using the policy-specific definitions. Each adaptor, dynamically, extracts the classes of the arguments, generates a hash value and looks up in the cache the index of the most appropriate method. The index is then used with a switch table to jump to the correct handler, calling in the end the desired method. More in detail, the list of arguments is extracted, with `@As` annotations overriding the classes of the arguments. Null arguments are checked when indicated by the `@Null` tag. If an unhandled null argument is encountered, `remapNull()` is invoked. The resulting sequence of classes is hashed and a lookup is performed in the local cache, which is shared among all the dispatchers that use the same interface/bodies combination. If no matching entry is found, control is passed to an external routine that updates the cache using the appropriate `bestMatch()`, taking care of synchronization issues, and eventually returns the index of the most appropriate method. If no suitable method is found, the error is handled as specified by the `@OnMissing` tag, possibly calling the `onMissing()` handler. Otherwise the index is used as a selector in a switch statement, reaching the invocation stage. If no special invocation policy is used, the appropriate body among those supplied at dispatcher-construction time is used to call the correct method, taking into account the renaming imposed by the `@Name` tag. If a special invocation policy is required, the corresponding `invoke()` handler is used, wrapping and unwrapping the arguments and the return value as necessary.

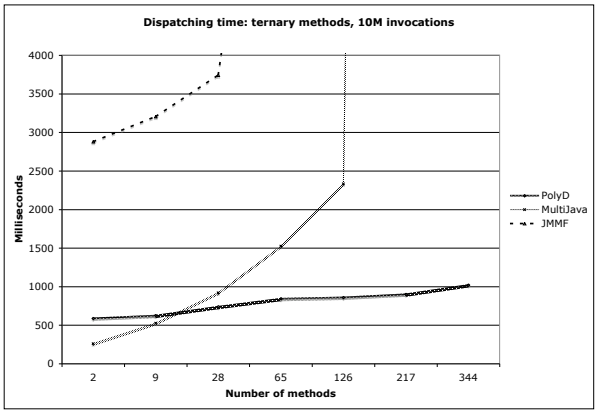
Despite the lengthy description, great effort was spent into minimizing the time required for message dispatching and for the construction of new dispatchers. The dispatching fast path is quite short and consists of just a handful of bytecode instructions. The creation of new dispatchers takes place by simply instantiating the synthetic class and setting up a few fields. A fast dispatcher creation time is particularly important if a large number of dispatchers are generated, as it was the case for the applications that we tested. During the creation of a new virtual machine for a short test program the Ovm compiler creates about 230,000 dispatchers,



(a) Unary methods, flat hierarchy, linear scale



(b) Binary methods, flat hierarchy, linear scale



(c) Ternary methods, flat hierarchy, linear scale

Figure 11: PolyD, MultiJava, JMMF

which are collectively used approximately 5,900,000 times. The Kacheck/J tool, used to parse the rt.jar file from a Sun JDK, creates about 285,000 dispatchers, and uses them about 9,400,000 times.

6.2 Benchmarking

We evaluated the performance of our implementation by adapting two existing applications, using PolyD as the core infrastructure for the visitor-style dispatching. The execution time of the PolyD version was then compared with the Runabout version of the same applications. Additionally, a number of microbenchmarks were run comparing PolyD against Runabout, Sprintabout, Dynamic Dispatcher, MultiJava, the Java Multi-Method Framework (JMMF), and plain visitors (references on these systems are available in Section 7). All benchmarks were run on an AMD Athlon™ XP1900+ at 1600MHz, 1GB of RAM, running Red Hat Linux (kernel 2.4.20), and Sun's JDK 1.5.0 in server mode. All of the timings shown in the tables and the graphs are averages of at least ten runs.

The first application used for the test is Kacheck/J, an encapsulation checker which analyzes the use of confined types in programs in Java programs. The performance of the PolyD version of the application was found to be similar, or slightly better, to the performance of the Runabout version. Figure 12 shows a comparison of the running time required by Kacheck/J to detect encapsulated types in the rt.jar file contained in Sun's 1.4.2-03 JDK. Similar figures were found for the Ovm framework: the numbers in Figure 12 refer to the execution time of the ahead-of-time compiler while processing a test program (with all the libraries), up to the code generation stage.

Execution time (sec.)	*Kacheck client VM*	*Kacheck server VM*	*Ovm server VM*
PolyD	21.707	22.921	104.165
Runabout	22.201	22.966	104.297

Kacheck: 285,656 dispatchers created
9,384,595 invocations

Ovm: 226,620 dispatchers created
5,864,145 invocations

Figure 12: Speed comparison, Kacheck/J and Ovm

Detailed tests were also performed using microbenchmarks, in particular comparing the relative performance of PolyD and other tools when dealing with visitors, or multimethods, defined over a progressively larger hierarchy of classes. PolyD was compared against MultiJava and JMMF when dealing with unary, binary, and ternary methods. The expected result was that PolyD, thanks to its hashing core, should scale well moving towards more complex systems. The experimental results confirm our intuition.

MultiJava, which relies on a chain of "instanceof" tests, has a good efficiency when just a few methods are involved, but the longer sequence of tests causes a severe performance degradation when a higher number of methods are involved. The graphs show the timings obtained using JDK 1.5.0 in server mode using a flat hierarchy, in which a single superclass has a number of direct subclasses, or a deep hierarchy, in which all classes are arranged in a chain. Using a flat hierarchy, for unary methods PolyD achieves a higher dispatching speed than MultiJava when more than about 50 methods are defined. For binary methods, PolyD is faster when dealing with more than about 15 methods. Using a deep hierar-

chy the performance of MultiJava degrades even more rapidly, and PolyD shows better dispatching speed when using as few as 9 methods in the unary case, as shown in Figure 16.

JMMF, which operates reflectively, exhibited much slower dispatching times. We encountered some unexpected exceptions using JMMF for certain method combinations, and that is reflected in the missing samples in the graphs. The benchmarks show that PolyD, in addition to its extensive features set, is also an efficient solution for general multiple dispatching, scaling much better than other tools when dealing with complex systems.

Regarding the visitor-like tools, Figures 14 and 16 compare PolyD against the Runabout, the Sprintabout, and other tools. The Dynamic Dispatcher and JMMF required a considerably longer time to complete the tests than the other tools, and their graph is out of scale. Figure 15 displays the relative performance of PolyD, Runabout, and Sprintabout when a larger number of methods are involved.

The dispatching efficiency of the three tools results overall comparable. That should not be too surprising considering that they all rely on class hashing and on method caching to speed up performance. PolyD exhibits a good level of performances despite having to cater for a much more general system. The experimental measurements confirm that PolyD can be used as a more general, and equally efficient, replacement for existing visitor-like tools.

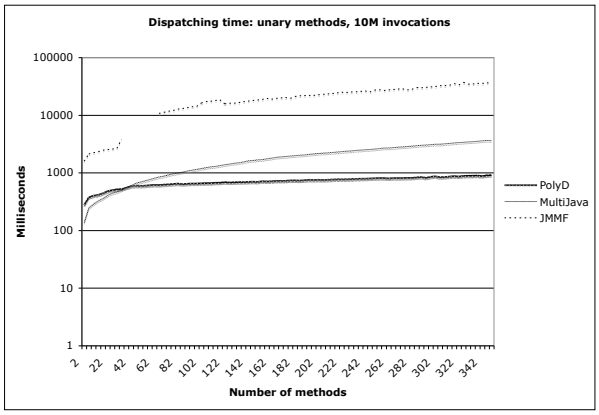
7. RELATED WORK

A number of tools have been developed to extend the native dispatching mechanism offered by Java. A relevant class of tools aim to provide features similar to the classic Visitor pattern [28], but improving on the implementation. Palsberg and Jay were the first to suggest that the use of a reflective approach would obviate the need for `accept()` methods, implementing the Walkabout [42]. Bravenboer and Visser improved on the idea by adding a cache in order to reduce the cost of reflective lookups [9], as these were extremely costly in early implentations of Java. Grothoff further improved efficiency by using runtime bytecode generation; his tool is known as the Runabout [29]. A similar approach, albeit with lower performance, was also used by the Dynamic Dispatcher [11]. The mechanism used by the Runabout was refined by Forax et.al. in the Sprintabout [26]. Their tool, like PolyD, uses a table-based dispatcher. Visitors have been proposed as a native language feature, in the Peripaton language [46].

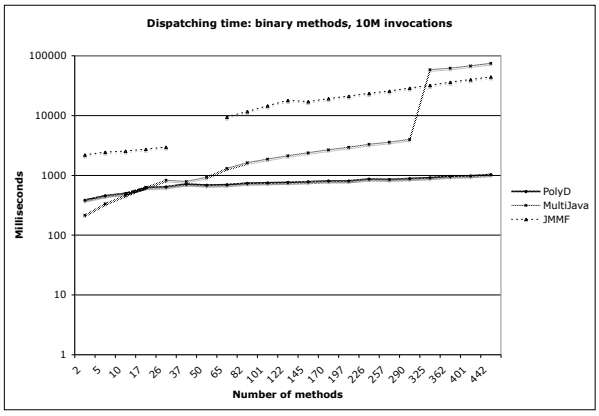
Several projects add multiple dispatching to Java, either by supplying the feature in the form of a library or by requiring special syntax, preprocessors, or virtual machines. MultiJava [16] uses a special syntax and a preprocessor. It also does not dispatch following interfaces and subinterfaces, but is limited to single inheritance. JMMF [25] works solely using reflection, but suffers from low performance. The system described by Dutchyn et al. [20] changes the semantics of Java dispatching, and uses a custom VM.

Other notable Java tools or proposals advocate alternate forms of dispatching, although none of them relies on pure Java: JPred [35] implements a general form of predicate dispatching; Boyland and Castagna use Parasitic Methods [8] to implement a restricted form of multimethods; Tuple [34] implements multidispatching as dispatching on tuples; the Half&Half paper [5] suggests the addition of multimethods and retroactive abstraction to Java; Nice [38] and Kiev [32] are two Java derivatives that include multimethod features. Proposals for adding multidispatching to other languages also exist. For example, Foote, Johnson, and Noble add multimethods to Smalltalk-80 [24].

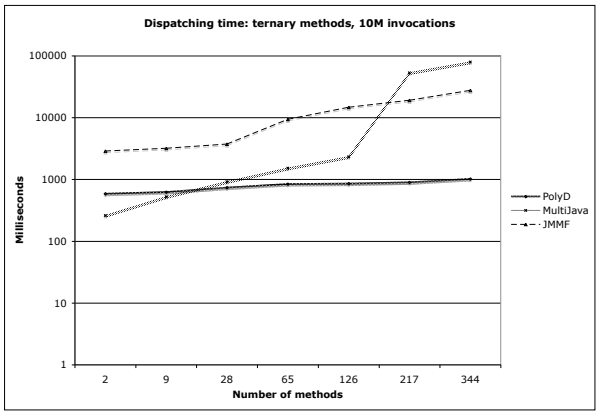
PolyD separates the dispatching concerns from the main program code, and as such it can be seen as a specialized aspect-



(a) Unary methods, flat hierarchy, logarithmic scale

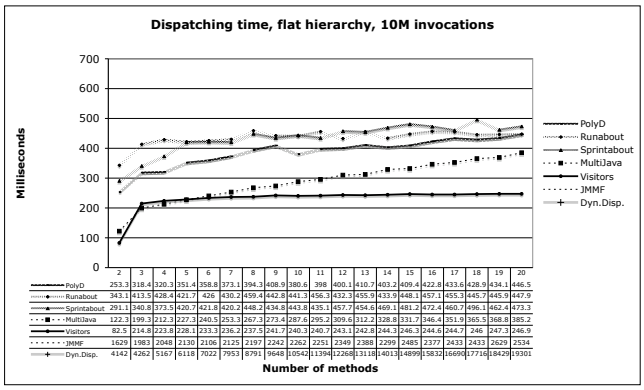


(b) Binary methods, flat hierarchy, logarithmic scale



(c) Ternary methods, flat hierarchy, logarithmic scale

Figure 13: PolyD, MultiJava, JMMF



	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PolyD	253.3	318.4	320.3	351.4	358.8	373.1	394.3	408.9	380.6	398	400.1	410.7	403.2	409.4	422.8	433.6	428.9	434.1	446.5
Runabout	343.1	413.5	428.4	421.7	426	430.2	459.4	442.8	441.3	456.3	432.3	455.9	433.9	448.1	457.1	455.3	445.7	445.9	447.9
Sprintabout	291.1	340.8	373.5	420.7	421.8	420.2	448.2	434.8	443.8	435.1	457.7	454.6	449.1	481.2	472.4	460.7	496.1	462.4	473.3
MultiJava	122.3	199.3	212.3	227.3	240.5	253.3	267.3	273.4	287.6	295.2	309.6	312.2	328.8	331.7	346.4	351.9	365.5	368.8	385.2
Visitors	82.5	214.8	223.8	228.1	233.3	236.2	237.5	241.7	240.3	240.7	243.1	242.8	244.3	246.3	244.6	244.7	246	247.3	246.9
JMMF	1629	1983	2048	2130	2106	2125	2197	2242	2262	2251	2349	2388	2299	2485	2377	2433	2433	2629	2534
Dyn.Disp.	4142	4262	5167	6118	7022	7953	8791	9648	10542	11394	12268	13118	14013	14899	15832	16690	17716	18429	19301

Figure 14: Speed comparison, flat hierarchy

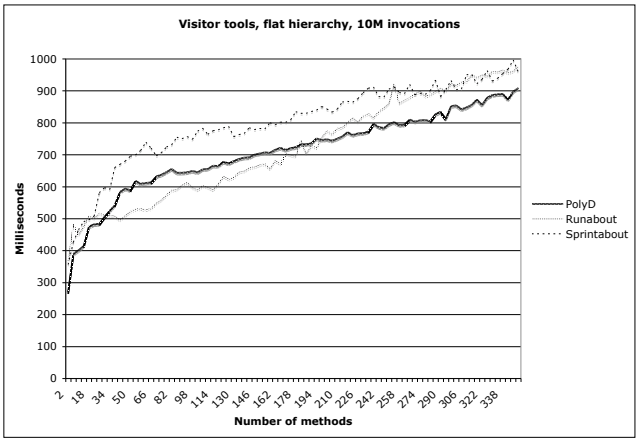


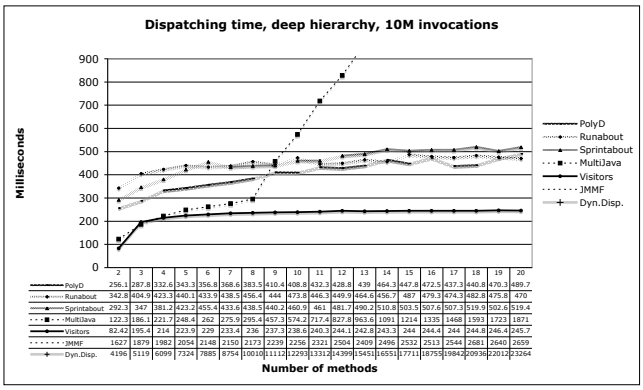
Figure 15: Speed comparison, more methods, flat hierarchy

oriented system. In that perspective, the functionalities offered by PolyD could be made part a general aspect framework as, for instance, XAspects [43]. The Fred/Socrates system [40] brings together elements of predicate dispatching and aspect oriented programming, allowing the programmer to define multiple "branches", corresponding to a given message, selected using boolean predicates but which can also be refined in stages. However, in Fred both the predicate and the body are part of the code that describes each branch. PolyD, conversely, totally segregates the selection logic, allowing for instance the same bodies to be selected through different policies. In that sense, PolyD is more clearly a specialized aspect language in which the dispatching concern is clearly confined, rather than being distributed throughout the code.

Open classes and the extensibility problem, also known as the expression problem, are extensively discussed in literature [47, 33, 45, 41].

8. FUTURE WORK

We are planning several enhancements and further developments for our framework. In terms of improvements to the current code base, the dispatching core could be optimized when a small number of methods are involved and a sequence of tests can replace the method lookup for the specific policy in use. That would allow us to implement, for instance, multiple dispatching more efficiently



	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PolyD	256.1	287.8	332.6	343.3	356.8	368.6	383.5	410.4	408.8	432.3	428.8	439	464.3	447.8	472.5	437.3	440.8	470.3	489.7
Runabout	342.8	404.9	423.3	440.1	433.9	438.5	456.4	444	473.8	446.3	449.9	464.6	456.7	487	479.3	474.3	482.8	475.8	470
Sprintabout	292.3	347	381.2	423.2	455.4	433.6	438.5	440.2	460.9	461	481.7	490.2	510.8	503.5	507.6	507.3	519.9	502.6	519.4
MultiJava	122.3	186.1	221.7	248.4	262	275.9	295.4	457.3	574.2	717.4	827.8	963.6	1091	1214	1335	1468	1593	1723	1871
Visitors	82.42	195.4	214	223.9	229	233.4	236	237.3	238.6	240.3	244.1	242.8	243.3	244	244.4	244	244.8	246.4	245.7
JMMF	1627	1879	1982	2054	2148	2150	2173	2239	2256	2321	2504	2409	2496	2532	2513	2544	2681	2640	2659
Dyn.Disp.	4196	5119	6099	7324	7885	8754	10010	11112	12293	13312	14399	15451	16551	17711	18755	19842	20936	22012	23264

Figure 16: Speed comparison, deep hierarchy

for fewer methods, while retaining the considerable advantages of hashing when building larger systems. Other minor improvements are also planned in order to optimize performance when dealing with final classes, and in other particular cases.

More general developments are also planned. PolyD currently allows the user to select, via a modular policy, the "most appropriate" method according to the classes of the arguments as obtained using the reflective features of Java. We are considering defining a parallel reflective API, enabling the user to define an independent, user-defined pseudo-class hierarchy. That should facilitate an easy prototyping of more unusual forms of inheritance (virtual types, for instance), together with their dispatching mechanisms.

9. CONCLUSIONS

Object-oriented programming languages adopt predefined message dispatch strategies that compromise between performance and flexibility requirements needs of most applications. While pragmatic, this approach leaves programmers with the responsibility of hand coding their own dispatchers when the default mechanisms fall short. A number of tools attempt to ease such task by adding specific dispatching extensions. In this paper, we propose a more general solution based on a modular approach to dispatching. We put our ideas into practice by developing a flexible dispatching framework for Java that enables programmers to define customized dispatching strategies that are better suited for the task at hand.

We show that our implementation, PolyD, is a tool that can replace, offering a more general solution, both visitor-derived and multimethod tools. The performance of PolyD is comparable to, and often exceeds, that of the Runabout and related tools, while at the same time offering more flexibility. When used as a multidispatching package, PolyD scales better than MultiJava and JMMF, showing a dramatic improvement over both tools when the number of methods increases. The PolyD framework is a pure Java solution that uses the standard Java runtime, bytecode, and syntax.

The extensive set of features offered by PolyD encompasses many aspects of the dispatching process that are not customizable in comparable systems, including a custom handling of missing methods, of null arguments, of method selection, and of method invocation. The dispatchers generated by PolyD share a common, efficient dispatching core, which ensures a high dispatching performance. In conclusion, we claim that PolyD is a flexible, general, and efficient framework that can free developers from the limitations of conventional dispatching mechanisms.

PolyD can be freely downloaded from `www.ovmj.org/polyd`.

10. ACKNOWLEDGMENTS

We thank Christian Grothoff, Rémi Forax, Etienne Duris, James Noble, Jeremy Manson, and James Baker for contributing to this work with discussions, documentation, and ideas. We also thank the anonymous reviewers for their useful and insightful comments.

This work was partially supported by NSF Grants HDCCSR-0341304 and CAREER-0093282.

11. REFERENCES

[1] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay. Static type checking of multi-methods. In A. Paepcke, editor, *OOPSLA '91 Conference Proceedings*, volume 26(11) of *ACM SIGPLAN Notices*, pages 113–128, New York, NY, Nov. 1991. ACM.

[2] The AspectJ home page. `http://http://eclipse.org/aspectj`.

[3] J. Baker and W. C. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, volume 37(5) of *SIGPLAN Notices*, pages 270–281. ACM, May 2002.

[4] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A monotonic superclass linerization for Dylan. In *Proceedings OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31 of *ACM SIGPLAN Notices*, pages 69–82. ACM, Oct. 1996.

[5] G. Baumgartner, M. Jansche, and K. Läufer. Half & Half: Multiple dispatch and retroactive abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Department of Computer Science, The Ohio State University, Mar. 2002.

[6] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification. *SIGPLAN Not.*, 23(SI):1–142, 1988.

[7] F. Bourdoncle and S. Merz. Type-checking higher-order polymorphic multi-methods. In *Conference Record of POPL '97: the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 302–315, New York, NY, 1997. ACM.

[8] J. Boyland and G. Castagna. Parasitic methods: an implementation of multi-methods for Java. In *Conference Proceedings of OOPSLA '97*, volume 32(10) of *ACM SIGPLAN Notices*, pages 66–76, New York, NY, Oct. 1997. ACM.

[9] M. Bravenboer and E. Visser. Guiding visitors: Separating navigation from computation. Technical Report UU-CS-2001-42, Institute of Information and Computing Sciences, Utrecht University, 2001.

[10] É. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, Grenoble, France, Nov. 2002.

[11] F. Büttner, O. Radfelder, A. Lindow, and M. Gogolla. Digging into the visitor pattern. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004), Banff, Alberta, Canada*, pages 135–141, June 2004.

[12] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, 1992.

[13] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 238–255, New York, NY, November 1999. ACM.

[14] C. Chambers and G. T. Leavens. Typechecking and modules for multi-methods. In *OOPSLA '94 Conference Proceedings*, volume 29(10) of *ACM SIGPLAN Notices*, pages 1–15, Oct. 1994.

[15] C. Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Nov. 2001. Available from `www.multijava.org`.

[16] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, New York, NY, Oct. 2000. ACM.

[17] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Design rationale, compiler implementation, and user experience. Technical Report 04-01, Iowa State University, Dept. of Computer Science, Jan. 2004. Submitted for publication.

[18] M. Cutumisu. MCI-Java: A modified Java virtual machine approach to multiple code inheritance. In *Virtual Machine Research and Technology Symposium*, pages 13–28. USENIX, 2004.

[19] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 151–170. Springer-Verlag, June 1987. Lecture Notes in Computer Science, Volume 276.

[20] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, and W. Holst. Multi-Dispatch in the Java Virtual Machine: Design and Implementation. In *Proceedings of 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, pages 77–92, 2001.

[21] M. D. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98: 12th European Conference on Object-Oriented Programming, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211, New York, NY, 1998. Springer-Verlag.

[22] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 94–104, New York, NY, June 1999. ACM.

[23] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 241–269. Springer, 1999.

[24] B. Foote, R. E. Johnson, and J. Noble. Efficient multimethods in Smalltalk-80. In *Proceedings of the*

European Conference on Object-Oriented Programming, Glasgow, Scotland, July 2005.

[25] R. Forax, E. Duris, and G. Roussel. Java Multi-Method Framework. In *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS '00), Sydney, Australia*, Los Alamitos, California, Nov. 2000. IEEE Computer Society Press.

[26] R. Forax, E. Duris, and G. Roussel. Reflection-based implementation of Java extensions: The double-dispatch use-case. In *Proceedings of the 2005 ACM symposium on applied computing*, New York, 2005.

[27] R. Forax and G. Roussel. Recursive types and pattern-matching in Java. In *GCSE*, volume 1799 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 1999.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[29] C. Grothoff. Walkabout revisited: The Runabout. In L. Cardelli, editor, *Proceedings of ECOOP '03*, volume 2743 of *LNCS*, pages 103–125. Springer-Verlag, July 2003.

[30] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *Proceedings of OOPSLA '01*, pages 241–255. ACM Press, Nov 2001.

[31] W. Holst and D. Szafron. A general framework for inheritance management and method dispatch in object-oriented languages. In *Proceedings of ECOOP '97: European Conference on Object-Oriented Programming*, pages 276–301, 1997.

[32] The Kiev language home page. `http://kiev.forestro.com`.

[33] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP'98*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113, Brussels, Belgium, July 1998. Springer-Verlag.

[34] G. T. Leavens and T. D. Millstein. Multiple dispatch as dispatch on tuples. In *Proceedings of OOPSLA '98*, volume 33(10) of *ACM SIGPLAN Notices*, pages 374–387. ACM, Oct. 1998.

[35] T. Millstein. Practical predicate dispatch. In *Proceedings of OOPSLA '04*, volume 39(11) of *ACM SIGPLAN Notices*, pages 345–364. ACM, Oct. 2004.

[36] T. Millstein, M. Reay, and C. Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 38(11) of *ACM SIGPLAN Notices*, pages 224–240, New York, NY, Nov. 2003. ACM.

[37] T. D. Millstein and C. Chambers. Modular statically typed multimethods. *Journal of Information and Computation*, 175(1):76–118, 2002.

[38] The Nice language home page. `http://nice.sourceforge.net`.

[39] M. E. Nordberg III. Variations on the visitor pattern. In *Proceedings of The Joint Pattern Languages of Programs (PLoP)*. Addison-Wesley, Sept. 1996.

[40] D. Orleans. Incremental programming with extensible decisions. In *First International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, 2002. ACM Press.

[41] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *Proceedings of the ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators, (IVME'03)*, San Diego, California, June 2003.

[42] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC, Vienna, Austria*, pages 9–15. IEEE, Aug. 1998.

[43] M. Shonle, K. J. Lieberherr, and A. Shah. XAspects: An Extensible System for Domain Specific Aspect Languages. pages 28–37, Anaheim, California, 2003. ACM Press. Special Track on Domain-Driven Development.

[44] B. Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring 1987 European Unix Users Group Conference*, Helsinki, 1987.

[45] M. Torgersen. The expression problem revisited: Four new solutions using generics. In *ECOOP '04 - Object-Oriented Programming European Conference*, volume 3086, pages 123–143. Springer-Verlag, 2004.

[46] T. VanDrunen and J. Palsberg. Visitor-oriented programming. In *Proceedings of FOOL-11, the 11th ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, Venice, Italy, January 2004.

[47] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. Technical Report IC/2004/33, École Polytechnique Fédérale de Lausanne, 2004.