NOTICE: This is the author’s version of a work accepted for publication by Dyna. Changes resulting from the publishing process, including peer review, editing, corrections, structural formatting and other quality control mechanisms, may not be reflected in this document. A definitive version was subsequently published in Dyna, Volume 186, pp. 242-250, August 2014.
Attaining multiple dispatch
in widespread object-oriented languages

Francisco Ortin $^a$, Jose Quiroga $^b$, Jose M. Redondo $^c$, Miguel Garcia $^d$

$^a$Ph.D., Computer Science Department, University of Oviedo, Spain, ortin@uniovi.es
$^b$M.Sc., Computer Science Department, University of Oviedo, Spain, quirogajose@uniovi.es
$^c$Ph.D., Computer Science Department, University of Oviedo, Spain, redondojose@uniovi.es
$^d$Ph.D., Computer Science Department, University of Oviedo, Spain, garciarmiguel@uniovi.es

Received: October 23rd, 2014. Received in revised form: May 10th, 2014. Accepted: June 10th, 2014.

Abstract
Multiple dispatch allows determining the actual method to be executed, depending on the dynamic types of its arguments. Although some programming languages provide multiple dispatch, most widespread object-oriented languages lack this feature. Therefore, different implementation techniques are commonly used to obtain multiple dispatch in these languages. We evaluate the existing approaches, presenting a new one based on hybrid dynamic and static typing. A qualitative evaluation is presented, considering factors such as software maintainability and readability, code size, parameter generalization, and compile-time type checking. We also perform a quantitative assessment of runtime performance and memory consumption.

Keywords: Multiple dispatch; multi-method; dynamic binding; reflection; method overload; hybrid typing.

1 Introduction

Object-oriented programming languages provide dynamic binding as a mechanism to implement maintainable code. Dynamic binding is a dispatching technique that postpones until runtime the process of associating a message to a specific method. Therefore, when the toString message is passed to a Java object, the actual toString method called is that implemented by the dynamic type of the object, discovered by the virtual machine at runtime.

Although dynamic binding is a powerful tool, widespread languages such as Java, C# and C++ only support it as a single dispatch mechanism: the actual method to be invoked depends on the dynamic type of a single object. In these languages, multiple-dispatch is simulated by the programmer using specific design patterns, inspecting the dynamic type of objects, or using reflection.

In languages that support multiple-dispatch, a message can be dynamically associated to a specific method based on the runtime type of all its arguments. These multiple-dispatch methods are also called multi-methods [1]. For example, if we want to evaluate binary expressions of different types with different operators, multi-methods allow modularizing each operand-operator-operand combination in a single method. In the example C# code in Figure 1, each Visit method implements a different kind of operation for three concrete types, returning the appropriate value type. As shown in Figure 2, the values and operators implement the Value and Operator interface, respectively. Taking two Value operands and an Operator, a multi-method is able to receive these three parameters and dynamically select the appropriate Visit method to be called. It works like dynamic binding, but with multiple types. In our example, a triple dispatch mechanism is required (the appropriate Visit method to be called is determined by the dynamic type of its three parameters).
Polymorphism can be used to provide a default behavior if one combination of two expressions and one operator is not provided. Since `Value` and `Operator` are the base types of the parameters (Figure 2), the last `Visit` method in Figure 1 will be called by the multiple dispatcher when there is no other suitable `Visit` method with the concrete dynamic types of the arguments passed. An example is evaluating the addition (`AddOp`) of two Boolean (`Bool`) expressions.

In this paper, we analyze the common approaches programmers use to simulate multiple dispatch in those widespread object-oriented languages that only provide single dispatch (e.g., Java, C# and C++). To qualitatively compare the different alternatives, we consider factors such as software maintainability and readability, code size, parameter generalization, and compile-time type checking. A quantitative assessment of runtime performance and memory consumption is also presented. We also present a new approach to obtain multiple dispatch in languages that provide hybrid dynamic and static typing, such as C#, Objective-C, Boo and Cobra. This alternative provides high maintainability and readability, requires reduced code size, allows parameter generalization, and performs significantly better than the reflective approach. On contrast, it requires 31% more memory resources than the rest of alternatives.

The rest of this paper is structured as follows. In Section 2, the common approaches to obtain multi-methods in widespread object-oriented programming languages are presented and qualitatively evaluated. Section 3 presents a new approach for hybrid typing languages, and a comparison with the previously analyzed systems. Section 4 details the runtime performance and memory consumption evaluation. Conclusions and future work are presented in Section 5.

2 Common approaches

2.1 The Visitor design pattern

The Visitor design pattern is a very common approach to obtain multiple dispatch in object-oriented languages than do not implement multi-methods [2]. By using method overloading, each combination of non-abstract types is implemented in a specific `Visit` method (Figure 1). Static type checking is used to modularize each operation in a different method. The compiler solves method overloading by selecting the appropriate implementation depending on the static types of the parameters. Suppose an n-dispatch scenario: a method with n polymorphic parameters, where each parameter should be dynamically dispatched considering its dynamic type (i.e., multiple dynamic binding). In this n-dispatch scenario, the n parameters belong to the $H_1, H_2, \ldots, H_n$ hierarchies, respectively. Under these circumstances, there are potentially $\prod_{i=1}^{n} C_{i}$ `Visit` methods, $C_{i}$ being the number of concrete (non-abstract) classes in the $H_i$ hierarchy.

Using polymorphism, parameters can be generalized in groups of shared behavior (base classes or interfaces). An example of this generalization is the two last addition methods in Figure 1. They generalize the way strings are concatenated with any other `Value`. This feature that allows grouping implementations by means of polymorphism is the parameter generalization criterion mentioned in the previous section.

```java
public class EvaluateExpression {

    // Addition
    Integer Visit(Integer op1, AddOp op, Integer op2) {
        return new Integer(op1.Value + op2.Value);
    }
    Double Visit(Double op1, AddOp op, Integer op2) {
        return new Double(op1.Value + op2.Value);
    }
    Double Visit(Integer op1, AddOp op, Double op2) {
        return new Double(op1.Value + op2.Value);
    }
    Double Visit(Double op1, AddOp op, Double op2) {
        return new Double(op1.Value + op2.Value);
    }
    String Visit(String op1, AddOp op, String op2) {
        return new String(op1.Value + op2.ToString());
    }
    String Visit(String op1, AddOp op, Double op2) {
        return new String(op1.Value + op2.ToString());
    }
    String Visit(String op1, AddOp op, Integer op2) {
        return new String(op1.Value + op2.ToString());
    }

    // EqualsTo
    Bool Visit(Integer op1, EqualToOp op, Integer op2) {
        return new Bool(op1.Value == op2.Value);
    }
    Bool Visit(Double op1, EqualToOp op, Integer op2) {
        return new Bool((int)op1.Value == op2.Value);
    }
    Bool Visit(Integer op1, EqualToOp op, Double op2) {
        return new Bool(op1.Value == ((int)op2.Value));
    }
    Bool Visit(Double op1, EqualToOp op, Double op2) {
        return new Bool(op1.Value == op2.Value);
    }
    Bool Visit(Bool op1, EqualToOp op, Bool op2) {
        return new Bool(op1.Value == op2.Value);
    }
    Bool Visit(String op1, EqualToOp op, String op2) {
        return new Bool(op1.Value.Equals(op2.Value));
    }
    Bool Visit(String op1, EqualToOp op, Double op2) {
        return new Bool(op1.Value.Equals(op2.Value));
    }
    Bool Visit(String op1, EqualToOp op, Integer op2) {
        return new Bool(op1.Value.Equals(op2.Value));
    }

    // And
    Bool Visit(Bool op1, AndOp op, Bool op2) {
        return new Bool(op1.Value && op2.Value);
    }

    // The rest of combinations
    Expression Visit(Value op1, Operator op, Value op2) {
        return null;
    }
}
```

Figure 1. Modularizing each operand and operator type combination.
As shown in Figure 2, the Visitor pattern places the Visit methods in another class (or hierarchy) to avoid mixing the tree structures to be visited (Value and Operator) with the traversal algorithms (Visitor) [3]. The (single) dispatching mechanism used to select the correct Visit method is dynamic binding [2]. A polymorphic (virtual) method must be declared in the tree hierarchy, because that is the hierarchy the specific parameter types of the Visit methods belong to. In Figure 2, the Accept method in Value provides the multiple dispatch. When overriding this method in a concrete Value class, the type of this will be non-abstract, and hence the specific dynamic type of the first parameter of Visit will be known. Therefore, by using dynamic binding, the type of the first parameter is discovered. This process has to be repeated for every parameter of the Visit method. In our example (Figure 2), the type of the second operand is discovered with the Accept2 method in Operator, and Accept3 in Value discovers the type of the third parameter before calling the appropriate Visit method.

In this approach, the number of AcceptX method implementations grows geometrically relative to the dispatch dimensions (i.e., the n in n-dispatch, or the number of the Visit parameters). Namely, for $H_1, H_2,... H_n$, hierarchies of the corresponding n parameters in Visit, the number of Accept methods are $1 + \sum_{i=1}^{n-1} \prod_{j=i}^{n} C_j$. Therefore, the code size grows geometrically with the number of parameters in the multi-method. Additionally, declaring the signature of each single AcceptX method is error-prone and reduces its readability.

Adding a new concrete class to the tree hierarchy requires adding more AcceptX methods to the implementation (see the formula in the previous paragraph). This feature reduces the maintainability of this approach, causing the so-called expression problem [4]. This problem is produced when the addition of a new type to a type hierarchy involves changes in other classes.

The Visitor approach provides different advantages. First, the static type error detection provided by the compiler. Second, this approach provides the best runtime performance (see Section 4). Finally, parameter generalization, as mentioned, is also supported. A summary of the pros and cons of all the approaches is presented in Table 1, after analyzing all the alternatives.

2.2 Runtime type inspection

In the previous approach, the dispatcher is implemented by reducing multiple-dispatch to multiple cases of single dispatch. Its high dependence on the number of concrete classes makes it error-prone and reduces its maintainability. This second approach implements a dispatcher by consulting the dynamic type of each parameter in order to solve the specific Visit method to be called. This type inspection could be performed by either using an is type of operator (e.g., is in C# or instanceof in Java) or asking the type of an object at runtime (e.g., GetType in C# or getClass in Java). Figure 3 shows an example implementation in C# using the is operator. Notice that this single Accept method is part of the EvaluateExpression class in Figure 1 (it does not need to be added to the tree hierarchy).
public class EvaluateExpression {
  // * Selects the appropriate Visit method in Figure 1
  public Value Accept(Value op1, Operator op, Value op2) {
    if (op is AndOp) {
      if (op1 is Bool) {
        if (op2 is Bool) return Visit((Bool)op1, (AndOp)op, (Bool)op2);
        else if (op2 is String) return Visit((String)op1, (AndOp)op, (String)op2);
        else if (op2 is Double) return Visit((Double)op1, (AndOp)op, (Double)op2);
        else if (op2 is Integer) return Visit((Integer)op1, (AndOp)op, (Integer)op2);
      }
      else if (op1 is String) { ... } else if (op1 is Double) { ... } else if (op1 is Integer) { ... }
      else if (op is EqualToOp) { ... }
      else if (op is AddOp) { ... }
      Debug.Assert(false, String.Format("No implementation for op1={0}, op={1} and op2={2}", op1, op, op2));
      return null;
    }
  }
}

Figure 3. Multiple dispatch implementation using runtime type inspection with the is operator (ellipsis is used to obviate repeating code).

public class EvaluateExpression {
  // * Selects the appropriate Visit method in Figure 1
  public Value Accept(Value op1, Operator op, Value op2) {
    MethodInfo method = this.GetType().GetMethod("Visit", BindingFlags.NonPublic | BindingFlags.Instance, null, new Type[] { op1.GetType(), op.GetType(), op2.GetType() }, null);
    if (method == null) {
      Debug.Assert(false, String.Format("No implementation for op1={0}, op={1} and op2={2}", op1, op, op2));
      return null;
    }
    return (Value)method.Invoke(this, new object[] { op1, op, op2 });
  }
}

Figure 4. Multiple dispatch implementation using reflection.

Figure 3 shows the low readability of this approach for our triple dispatch example with seven concrete classes. The maintainability of the code is also low, because the dispatcher implementation is highly coupled with the number of both the parameters of the Visit method and the concrete classes in the tree hierarchy. At the same time, the code size of the dispatcher grows with the number of parameters and concrete classes.

The is operator approach makes extensive use of type casts. Since cast expressions perform type checks at runtime, this approximation loses the robustness of full compile-time type checking. The GetType method also has this limitation together with the use of strings for class names, which may cause runtime errors when the class name is not written correctly. Parameter generalization is provided by means of polymorphism. As discussed in Section 4, the runtime performance of these two approaches is not as good as that of the previous alternative.

2.3 Reflection

The objective of the reflection approach is to implement a dispatcher that does not depend on the number of concrete classes in the tree hierarchy. For this purpose, not only the types of the parameters but also the methods to be invoked are discovered at runtime. The mechanism used to obtain this objective is reflection, one of the main techniques used in meta-programming [5]. Reflection is the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions [6]. Using reflection, the self-representation of programs can be dynamically consulted and, sometimes, modified [7]. As shown in Figure 5, the dynamic type of an object can be obtained using reflection (GetType). It is also possible to retrieve the specific Visit method implemented by its dynamic type (GetMethod), passing the dynamic types of the parameters. It also provides the runtime invocation of dynamically discovered methods (Invoke).

The code size of this approach does not grow with the number of concrete classes. Moreover, the addition of another parameter does involve important changes in the code. Consequently, as shown in Table 1, this approach is more maintainable than the previous ones. Although the reflective Accept method in Figure 4 may be somewhat atypical at first, we think its readability is certainly higher than the one in Figure 3.

The first drawback of this approach is that no static type checking is performed. If Accept invokes a nonexistent Visit method, an exception is thrown at runtime, but no compilation error is produced. Another limitation is that parameter generalization is not provided because reflection only looks for one specific Visit method. If an implementation with the exact signature specified does not exist, no other polymorphic implementation is searched (e.g., the last Visit method in Figure 1 is never called). Finally, this approach has showed the worst runtime performance in our evaluation (Section 4).
A hybrid typing approach

Hybrid static and dynamic typing (henceforth referred to simply as hybrid typing) languages provide both typing approaches in the very same programming language. Programmers may use one alternative or the other depending on their interests, following the static typing where possible, dynamic typing when needed principle [8]. In the case of multiple dispatch, we have used static typing to modularize the implementation of each operand and operator type combination (Visit methods in Figure 1). We propose the use of dynamic typing to implement multiple dispatchers that dynamically discover the suitable Visit method to be invoked.

In a hybrid typing language, its static typing rules are also applied at runtime when dynamic typing is selected. This means that, for instance, method overload is postponed until runtime, but the resolution algorithm stays the same [9]. We have used this feature to implement a multiple dispatcher that discovers the correct Visit method to be invoked at runtime, using the overload resolution mechanism provided by the language. At the same time, parameter generalization by means of polymorphism is also achieved.

Figure 5 shows an example of multiple dispatch implementation (Accept method) in C#. With dynamic, the programmer indicates that dynamic typing is preferred, postponing the overload resolution until runtime. The first maintainability benefit is that the dispatcher does not depend on the number of concrete classes in the tree hierarchy (the expression problem [4]). Besides, another dispatching dimension can be provided by simply declaring one more parameter, and passing it as a new argument to Visit. The dispatcher consists in a single invocation to the overloaded Visit method, indicating which parameters require dynamic binding (multiple dispatching) with a cast to dynamic. If the programmer wants to avoid dynamic binding for a specific parameter, this cast to dynamic will not be used. This simplicity makes the code highly readable and reduces its size considerably (Table 1). At the same time, since the overload resolution mechanism is preserved, parameter generalization by means of polymorphism is also provided (i.e., polymorphic methods like the two last addition implementations for strings in Figure 1).

In C#, static type checking is disabled when the dynamic type is used, lacking the compile-time detection of type errors. However, there are research works on hybrid typing languages, such as the StaDyn programming language [10], which provide static type checking when the dynamic type is used. When this feature is not supported, the best approach is to use static types to declare the Accept parameters using polymorphism (restricting their types to Value and Operator, as shown in Figure 5). At the same time, exception handling is another mechanism that can be used to make the code more robust – notice that parameter generalization reduces the number of possible exceptions to be thrown, compared to the reflection approach.

Finally, this approach shows a runtime performance between the statically typed implementation and the reflective one (see Section 4). Hybrid typing languages, including C#, commonly implement a dynamic cache to improve runtime performance of dynamically typed code [11]. This technique provides a significant runtime performance improvement compared to reflection [12].

Table 1. Qualitative evaluation of the approaches.

<table>
<thead>
<tr>
<th>Visitor Pattern</th>
<th>Maintainability</th>
<th>Readability</th>
<th>Code Size</th>
<th>Generalization</th>
<th>Compile-time checking</th>
<th>Runtime Performance</th>
<th>Memory Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>is Operator</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GetType Method</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reflection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Hybrid Typing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

4 Evaluation

In this section, we measure execution time and memory consumption of the five different approaches analyzed. Detailed information is presented to justify the performance and memory assessment in the two last columns of Table 1.

4.1 Methodology

In order to compare the performance of the proposed approaches, we have developed a set of synthetic micro-benchmarks. These benchmarks measure the influence of the following variables on runtime performance and memory consumption:

- Dispatch dimensions. We have measured programs executing single, double and triple dispatch methods. These dispatch dimensions represent the number of
parameters passed to the `Accept` method shown in Figures 3, 4 and 5.

- Number of concrete classes. This variable is the number of concrete classes of each parameter of the `Accept` method. For each one, we define from 1 to 5 possible derived concrete classes. Therefore, the implemented dispatchers will have to select the correct `Visit` method out of up to 125 different implementations (5^3).
- Invocations. Each program is called an increasing number of times to analyze their performance in long-running scenarios (e.g., server applications).
- Approach. The same application is implemented using the static typing, runtime type inspection (is and `GetType` alternatives), reflection, and hybrid typing approaches.

Each program implements a collection of `Visit` methods that simply increment a counter field. The idea is to measure the execution time of each dispatch technique, avoiding additional significant computation –we have previously evaluated a more realistic application in [13].

Regarding the data analysis, we have followed the methodology proposed in [14] to evaluate the runtime performance of applications, including those executed on virtual machines that provide JIT compilation. We have followed a two-step methodology:

1. We measure the elapsed execution time of running multiple times the same program. This results in p (we have taken p = 30) measurements \( x_i \) with \( 1 \leq i \leq p \).
2. The confidence interval for a given confidence level (95%) is computed to eliminate measurement errors that may introduce a bias in the evaluation. The confidence interval is computed using the Student’s t-distribution because we took \( p = 30 \) [15]. Therefore, we compute the confidence interval \([c_1, c_2]\) as:

\[
c_1 = \bar{x} - t_{1-a/2, p-1} \frac{s}{\sqrt{p}} \quad c_2 = \bar{x} + t_{1-a/2, p-1} \frac{s}{\sqrt{p}}
\]

where \( \bar{x} \) is the arithmetic mean of the \( x_i \) measurements, \( \alpha = 0.05(95\%) \), \( s \) is the standard deviation of the \( x_i \) measurements, and \( t_{1-a/2, p-1} \) is defined such that a random variable \( T \) that follows the Student’s \( t \)-distribution with \( p - 1 \) degrees of freedom, obeys

\[
Pr[T \leq t_{1-a/2, p-1}] = 1 - \alpha/2.
\]

The memory consumption has been measured following the same methodology to determine the memory used by the whole process. All the tests were carried out on a lightly loaded 3.4 GHz Intel Core i7 2600 system with 16 GB of RAM running an updated 64-bit version of Windows 8 Professional.

### 4.2 Runtime performance

Figure 6 shows the execution time of single, double and triple dispatch, when each parameter of the multi-method has five concrete derived types. Each `Visit` method is executed at least once. To analyze the influence of the number of invocations on the execution time, we invoke multi-methods in loops from 1 to 100,000 iterations. Figure 6 shows the average execution time for a 95% confidence level, with an error interval lower than 2%.

As can be seen in Figure 6, all the approaches have a linear influence of the number of iterations on execution time. However, the dispatch dimension (i.e., the number of multi-method parameters) of the analyzed approaches shows a different influence. For single dispatch, the hybrid typing approach is 19% and 2,787% faster than `GetType` and reflection, respectively, but requires 157% and 876% more execution time that `is` and static typing. For double dispatch, the runtime performance of the hybrid approach improves in comparison with the rest of alternatives (Figure 6). For triple dispatch, the hybrid static and dynamic typing alternative is the second fastest one, performing 1.4, 2.5 and 265 times better than `is`, `GetType` and reflection, respectively (static typing is 2.7 times faster than hybrid typing in this scenario).

Figure 7 shows execution time, when the number of concrete classes that implement each multi-method parameter increases (for 100,000 fixed iterations). For each parameter, we increment (from 1 to 5) the number of its derived concrete classes. In the case of triple dispatch and five different concrete classes, the multiple dispatcher has to select the correct `Visit` method out of 125 (5^3) different implementations.

As show in Figure 7, the relative performance of the hybrid approach improves as the number of concrete classes increases. For single dispatch, hybrid typing requires 213% more execution time than `GetType` for one concrete type of the single parameter; however, the hybrid approach is 19% faster than `GetType` for five different concrete types. For double dispatch, the hybrid approach improves its relative performance, being faster than `GetType` for any number of classes. When the dimension of the dispatch is triple, the relative runtime performance of the hybrid approach also improves as the number of concrete classes increases. With five different types for any of the three parameters, the hybrid approach is the second fastest one, being 40% faster than `is` and 265 times faster than reflection (static typing is 2.7 times faster than hybrid typing).

### 4.3 Memory consumption

We have measured memory consumption, analyzing all the variables mentioned in the Section 4.1. There is no influence of the number of iterations, the dimensions of dispatch, or the number of concrete classes, in the memory consumed by the benchmark.

The memory required by the approaches but hybrid typing are similar (the difference is 1%, lower than the 2% error interval). However, the hybrid approach involves an average increase of 31% compared with the rest of approaches. This difference is due to the use of the Dynamic Language Runtime (DLR) [16]. The DLR is a new layer over the CLR to provide a set of services to facilitate the implementation of dynamic languages. The DLR implements a runtime cache to optimize runtime performance of
dynamically typed operations, performing better than reflection (as shown in Figures 6 and 7) [13]. However, this runtime performance improvement also requires additional memory resources.

5 Related work

In this section, we describe the existing languages and frameworks that provide multiple dispatch [17]. CLOS [18] and Clojure [19] are examples of dynamically typed
languages that include multi-methods in their semantics. Clojure has recently created a port for .NET that makes use of the DLR [20]. These approaches are fully dynamic, detecting all the type errors at runtime.

Xtend is a Java extension that provides statically typed multiple dispatch [21]. Method resolution and binding in Xtend are done at compile time as in Java. Dylan [22], Cecil [1] and, recently, Groovy 2 [23] are programming languages that provide both dynamic and static typing. Although these three languages support dynamically typed multi-methods, multiple dispatch can also be achieved with the hybrid typing approach proposed in this article.

Many different approaches exist to provide multiple dispatch to the Java platform. One of the first works is Runabout, a library to support two-argument dispatch (i.e., double dispatch) for Java [24]. Runabout is based on improving a previous reflective implementation of the Visitor pattern called Walkabout [25]. The appropriate method implementation is found via reflection, but method invocation is performed by generating Java bytecode at runtime performing better than Walkabout.

Dynamic Dispatcher is a double-dispatch framework for Java [26]. Three different dispatch methods are provided, combining the use of reflection and dynamic code generation. It provides the generalization of multi-method parameters by means of polymorphism.

Sprintab is another double-dispatch alternative for Java, provided as a library [27]. Sprintab uses a naming convention to identify multi-methods. Multi-methods implement a runtime type inspection dispatch (the GetType approach). The dispatch object implements a cache to efficiently obtain the different method implementations at runtime, avoiding the use of reflection.

MultiJava is a backward-compatible extension of Java that supports any dispatch dimension (not just double dispatch) [28]. Given a set of multi-method implementations, the MultiJava compiler produces a single Java dispatch method containing the bodies of the set of multi-method implementations. The multi-method implements the runtime type inspection approach, using the instanceof Java operator (is in C#).

The Java Multi-Method Framework (JMMF) uses reflection to provide multiple dispatch for Java [29]. Multi-methods can be defined in any class and with any name. JMMF is provided as a library; it proposes neither language extensions nor virtual machine modifications.

PolyD is aimed at providing a flexible multiple dispatch technique for Java [30]. PolyD generates Java bytecodes dynamically, and allows the user to define customized dispatching policies. Three standard dispatching policies are available: multiple dispatching (cached GetType runtime type inspection), overloading (static method overload) and a ‘non-subsumptive’ policy (only calls a method if the classes of the arguments match exactly those of the method parameters; i.e. no parameter generalization).

6 Conclusions

Different alternatives are nowadays used to achieve multiple dispatch in widespread languages that do not provide multi-methods. A qualitative evaluate has shown the pros and cons of each approach.

A new alternative has been described for hybrid typing languages. Their benefits are high readability and maintainability, loose coupling with the number of concrete classes and the dispatch dimensions, and parameter generalization. The main limitation is no compile-time type error detection. Its runtime performance is analogous to the runtime type inspection approaches. The average execution time of all the measured hybrid programs took 3.9 times more execution time the Visitor design pattern, being 36.6 times faster than reflection. The proposed approach has consumed 31% more memory resources than the rest of alternatives.

Since the main limitation of the hybrid typing approach is its lack of compile-time error detection, we are currently working on defining and implementing a hybrid language that provides compile-time type checking [10]. That language, called StaDyn, is an extension of C# that performs type inference over dynamic references. This C# extension may eventually detect some type errors of the hybrid typing approach at compile-time [31]. Another future work will be analyzing the suitability of implementing multi-methods in Java using the new invokeldynamic opcode [32].

All the programs used in the evaluation of runtime performance and memory consumption, and the detailed measurement data are freely available at http://www.reflection.uniovi.es/stadyn/download/2013/dyna.zip

Acknowledgements

This work has been partially funded by Microsoft Research and the Department of Science and Innovation (Spain) under the National Program for Research, Development and Innovation: project TIN2011-25978.

References


[20] ClojureCLR. A port of Clojure to the CLR, part of the Clojure project [May 2014]. Available at: https://github.com/clojure/clojure-clr.


Francisco Ortin. 1973, is an Associate Professor of the Computer Science Department at the University of Oviedo, Spain. He is the head of the Computational Reflection research group (www.reflection.uniovi.es). He received his BSc in Computer Science in 1994, and his MSc in Computer Engineering in 1996. In 2002, he was awarded his PhD entitled A Flexible Programming Computational System developed over a Non-Restrictive Reflective Abstract Machine. He has been the principal investigator of different research projects funded by Microsoft Research and the Spanish Department of Science and Innovation. His main research interests include dynamic languages, type systems, aspect-oriented programming, computational reflection, and runtime adaptable applications. ORCID: 0000-0003-1199-8649

Jose Quiroga. 1982, is a Research Assistant of the Computer Science Department at the University of Oviedo. He was awarded his BSc degree in Computer Science in 2004 and, in 2009, an MSc in Computer Engineering. He worked in the Research and Development department of the CTIC Foundation until 2012, when he became a Research Assistant financed by the Spanish Department of Competitiveness and Productivity. As a PhD student, he is doing research on the optimization of dynamically typed programming languages, performing compile-time inference of type information. ORCID: 0000-0002-1646-4796

José M. Redondo. 1978, is an Assistant Professor of the Computer Science Department at the University of Oviedo, Spain. He received his BSc in Computer Science in 2000, and his MSc in Computer Engineering in 2002. In 2007, he was awarded his PhD entitled Improving the performance of structural reflection using JIT compilation techniques. He has participated in different research projects funded by Microsoft Research and the Spanish Department of Science and Innovation. His main research interests include virtual machines, JIT compilation, computational reflection, and runtime adaptable applications. ORCID: 0000-0002-0939-0186

Miguel García. 1979, is a Postdoctoral Research Assistant of the Computer Science Department at the University of Oviedo. He received his BSc degree in Computer Science in 2005. In 2008, he was awarded an MSc in Web Virtual machine, and an MSc in Software Engineering Research in 2010. In 2013, he presented his PhD dissertation entitled Improving the Runtime Performance and Robustness of Hybrid Static and Dynamic Typing Languages. His research interests include compiler construction, programming languages design, and aspect-oriented software development. ORCID: 0000-0002-3150-2826