A Reflective Implementation of Java Multi-Methods

Rémi Forax, Étienne Duris, Gilles Roussel

A Reflective Implementation of Java Multi-Methods

Rémi Forax, Etienne Duris and Gilles Roussel

***Abstract*— In Java, method implementations are chosen at runtime by late-binding with respect to the runtime class of just the receiver argument. However, in order to simplify many programming designs, late-binding with respect to the dynamic type of *all* arguments is sometimes desirable. This behavior, usually provided by multi-methods, is known as multi-polymorphism.**

This paper presents a new multi-method implementation based on the standard Java reflection mechanism. Provided as a package, it does not require any language extension nor any virtual machine modification. The design issues of this reflective implementation are presented together with a new and simple multi-method dispatch algorithm that efficiently supports class loading at runtime. This implementation provides a practicable and fully portable multi-method solution.

***Index Terms*— Java, Polymorphism, Reflection, Dynamic Class Loading.**

I. Introduction

In order to ease reusability and maintenance, software engineering widely uses object-oriented language features. Among them, late-binding and polymorphism are probably the most important, because they provide a simple way to dynamically choose implementations (behavior) according to the context. However, in most object-oriented languages, a single object is taken into account by this context. For instance, in Java, late-binding only concerns the target object (receiver) of the method call. This is generally sufficient for typical operations whose semantics depends on the kind of single object but, when the context dynamically depends on the kind of several objects, late-binding on all arguments is sometimes more suitable. This feature, known as *multi-polymorphism*, is usually achieved by *multi-methods*.

Multi-methods have already been largely studied [1]–[9] and their usefulness has been established in several application fields, such as binary methods [10]. Among the advantages of multi-methods, we are more concerned with their ability to simplify the specification of algorithms outside the classes they are dealing with [11]–[13]. Moreover, we believe that this feature is particularly valuable in the context of component-based applications, for their development, maintenance and reusability.

Application fields of multi-methods are an important issue but the question of their implementation in Java is also problematic. Indeed, whatever the target language, most existing implementations for multi-methods assume that all possible argument types are known at compile time. However, this restriction is not fully compliant with Java's philosophy, where any new type could be dynamically discovered (loaded) at runtime. In this paper, we propose a multi-method implementation, the Java Multi-Method Framework (JMMF), together with a new algorithm that fulfills this requirement.

Beyond the adequacy of our approach with dynamic class loading, we have chosen to provide users with multi-methods through a pure Java library, using the Core Reflection API. First, this approach is relatively simple and practical to implement compared to implementations based on language or JVM extensions. Next, JMMF allows standard Java environments to use multi-methods without any customization.

Reflection-based implementations are known to yield performance drawbacks. Nevertheless, we think that such an approach can be made practicable if implementation algorithms are carefully tuned to perform as many computations as possible at creation time to reduce invocation time overhead, but without inducing impracticable space usage. We claim that this approach, consisting in finding a good compromise between space, creation time and invocation time, is a general design concern related to reflection-based implementations [14], [15].

The rest of the paper is organized as follows. First, section II explains how multi-methods could be useful in the design and implementation of algorithms, especially in component-based applications; it gives multi-method use-case examples using JMMF. Then, section III discusses design issues in the implementation of such a feature, arguing the worth of a reflection-based approach provided that space and time overhead are finely tuned. Next, section IV dives into the description of the algorithms developed for JMMF. Its performances are discussed in section V, before situating it with respect to related works in section VI and finally concluding.

II. Why Multi-Methods?

After showing how multi-polymorphism could ease the design and maintenance of algorithms for existing components, this section outlines the use of our multi-method solution, JMMF, by giving examples that illustrate its capabilities.

A. Algorithm design with components

Object filtering against their class is an essential feature of most object-oriented languages. It is usually provided by a late-binding mechanism that allows method implementations (behavior) to be chosen dynamically according to the type of the receiving object. This dispatch greatly simplifies code reusability since programmers do not have to implement complex selection mechanism nor to modify the code when new classes implementing the method are introduced.

```
void f(I i) { i.m(); }
```

If a programmer wants to reuse such an implementation (`f`), objects (classes) just have to implement the method `m`

Authors are with Institut Gaspard-Monge, Université de Marne-la-Vallée, 5 bd Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée, France. E-mail: forax@univ-mlv.fr, duris@univ-mlv.fr, roussel@univ-mlv.fr. Corresponding author is Rémi Forax. Phone: 33.1.60.95.75.55, Fax: 33.1.60.95.75.57.

(the interface `I`). However, nowadays, most applications are built using components provided by off-the-shelf libraries. By component, we mean any reusable library or package, such as specific window toolkits or Java standard libraries. Programmers usually do not have access to their implementation classes and thus cannot simply add methods to component objects. Classically, there are two techniques to work around this difficulty: class inheritance and delegation.

1) Single dispatch limitations: Inheritance is not always applicable. Indeed, in addition to the obvious case of final classes, components sometimes do not give access to object creation and programmers only manipulate objects through well-defined public interfaces of the component. In these cases, they could obtain already-constructed objects of these types (interfaces) through factory methods that hide the concrete class of created objects; the programmer never calls the `new` instruction.

It is always possible to use delegation by defining new wrapping classes for each data type. This leads to data structure duplication and to dedicated type filtering. For large data structures or intricate sub-typing relations, this approach may become resource demanding, but also burdensome to implement and maintain.

Thus, none of these approaches is fully satisfactory to allow reusability using late-binding on component objects.

2) Double dispatch lack of generality: One way to avoid this problem is to implement method behaviors outside the component object, separating the data (`o`) from the behavior (`b.visit()`) as the visitor design pattern [11] does.

```
void f(Object o, Behavior b) { b.visit(o); }
```

This design pattern simulates late-binding on the argument of a `visit(o)` method using classical late-binding on an `o.accept(Visitor v)` method provided by the component object. However, this design pattern has several limitations.

First, it can only be used with components engineered to accept visitors, i.e., implementing an `accept()` method. Second, the filtering provided by the visitor design pattern is restricted to one argument and cannot be generalized if filtering on several arguments is required. Finally, the behavior is strongly tied to the component type hierarchy: it must implement a special `Visitor` interface which, for full generality, must include one `visit()` method for each accessible type of the component. This has two drawbacks. The programmer must implement every `visit()` method even if some of them could have been captured by a widening reference conversion of the argument type. Moreover, any extension of the type hierarchy requires the `Visitor` interface to be extended; then, all existing visitor implementations have to be modified since they must implement the new visitor interface to be accepted by components.

3) Multi dispatch: These requirements and limitations show that a general argument filtering mechanism, independent of the argument type hierarchy, would greatly simplify software engineering and reusability, especially in the context of component-based applications.

In order to perform such a filtering against dynamic types of all arguments, an ad-hoc solution consists in using a cascade of `instanceof` tests. However, this approach is static, tedious to implement and difficult to manage, particularly in the presence of inheritance and dynamic loading which are basic features of Java.

This can also be achieved by introducing late-binding on all method arguments. This feature, known as multi-polymorphism, is usually provided by *multi-methods* and has already been largely studied [1]–[9]. Multi-methods preserve locality since a method can be specified for each specific tuple of argument types. They also provide some kind of encapsulation since all these methods could be specified in the same class (hierarchy). Some implementations of multi-methods based on cascades of `instanceof` tests are available for Java through language extensions and provide strong static type-checking [4], [7]. It is not possible to fairly compare these works with our since we do not provide static type-checking. Nevertheless, language extensions constraint their portability whereas static type-checking requires all possible method signatures to be known at compile time.

B. Application domains

Natural domains of application for multi-methods are implementations that handle large data structures with complex sub-typing relations such as parsers [12], [13] (i.e. JAXP, J2EE) and window toolkits (i.e. Swing). However, the application field of multi-methods is not restricted to these domains.

Only considering J2SE packages, many examples can be found. For instance, the design of callback filtering `handle(Callback[])` in `javax.security.auth.callback.CallbackHandler` or those of collection copying `copy(Collection, Collection)` could be simplified with multi-methods. This observation, not specific to Java, has already been noticed for other languages. Studies on binary methods [10] give many examples of multi-method use-cases with two arguments. Chambers also pointed out multi-method use-cases in the design of the Cecil compiler [16].

As a single example, consider the following code part of the method `deepEquals(Object[] a1,Object[] a2)` of class `java.lang.Arrays` in JDK 1.5 that deeply compares its two arguments.

```
public static boolean deepEquals(Object[] a1,Object[] a2){
  ...
  for (int i=0; i<length; i++) {
    Object e1=a1[i]; Object e2=a2[i];
    ...
    boolean eq;
    if (e1 instanceof Object[] && e2 instanceof Object[])
      eq=deepEquals((Object[])e1,(Object[])e2);
    else if (e1 instanceof byte[] && e2 instanceof byte[])
      eq=equals((byte[])e1,(byte[])e2);
    ... // similar tests for all primitive type arrays...
    else eq=e1.equals(e2);
    if (!eq) return false;
  ...
```

This is a typical example of implementation that extends functionalities of an existing component without modifying it. Indeed, rather than modifying implementation classes for arrays, the `deepEquals()` method specifies the algorithm outside these legacy classes (here of the old J2SDK) and has to take into account multiple combinations of argument types. This produces pieces of code that are intricate and difficult to maintain or modify.

C. JMMF package use-case

We propose a new implementation for multi-method dispatch, especially developed for Java and fully portable. It allows dynamic class loading with a minimum overhead compared with existing algorithms that were conceived for languages where all types are known at compile time. Since the dispatch of multi-method in Java needs dynamic type information, and in order to leave the language and the virtual machine unchanged, our implementation provides multi-methods as a pure Java package using the reflection mechanism: the programmer only needs to add a JAR file in its classpath and to use its functionalities to create and invoke multi-methods. In this package, called the *Java Multi-Method Framework* (JMMF), a multi-method stands as an object representing a set of methods that have the same name and the same number of parameters. For a given context, a target object and a tuple of actual argument types, our *method resolution* provides the corresponding *most specific* method (see section IV-A.5).

To give an intuitive idea of the JMMF package and its use-cases, we illustrate the processes of constructing, using and extending a multi-method through an evolving example. This example comes from a real design concern encountered in the development of a parser generator, in which multi-methods simplify the specification of the semantics associated with grammar rules.

Suppose that you want to define the syntax and the semantics of an operator + that either corresponds to the addition of numbers or to the concatenation between sequences of characters, like Java does. In our parser generator, the syntax of such an expression looks like the following piece of code:

```
Expr -> Expr "+" Expr {plus($1, $2)}
      | Number
      | CharSequence
```

The semantics of the operator + is associated with the multi-method `plus()`. It varies with respect to the types of its operands which, here, are either the abstract class `Number` or the interface `CharSequence` of the package `java.lang`.

1) Simple semantics: Let us first consider the class `Plus` that defines the semantics for the operator + as it was implemented before J2SDK. It adds its operands if they are both numbers and it concatenates their string representations if the first operand is a character sequence.

```
import fr.umlv.jmmf.reflect.*;
public class Plus {
  public Number plus(Number l,Number r)
  { return new Double(l.doubleValue()+r.doubleValue()); }
  public CharSequence plus(CharSequence l,CharSequence r)
  { return l.toString()+r; }
  public CharSequence plus(CharSequence cs,Number n)
  { return cs.toString()+n; }
  MultiMethod mm=MultiMethod.create(Plus.class,"plus",2);
  public Object plusMM(Object l,Object r) throws Exception
  { return mm.invoke(this,new Object[]{l,r}); }
}
```

In this class, the programmer has to specify a `plus()` method for each pair of types defining the semantics of the operator, as if multi-polymorphism was available at language level. Next, to simulate multi-polymorphism, he has to construct a multi-method instance that stands for all the methods in the class with name `plus` and with exactly two parameters; he does it calling the static method `MultiMethod.create()`. Even if this suffices to call the `invoke()` method on the multi-method object, to enhance the readability, the programmer usually defines a hint method `plusMM()` with two `Object` parameters to perform the polymorphic dispatch between all `plus()` methods with respect to the dynamic type (class) of its arguments. Note that in this case, the method `plusMM()` must be provided to allow static type checking (here, `Object` is a common super-type for parameters). However, this method may have been declared abstract and the bytecode of its implementation could have been generated automatically by some bytecode generator such as ASM [17] and added by a specific class loader, for instance using inheritance.

When the `invoke()` method is called, transmitting the argument as an `Object` array, our method resolution mechanism for multi-methods looks for the most specific method `plus()` according to the actual type of the arguments and, if any, invokes it. When no such method exists, an exception is thrown.

The class `Plus` could then be used to perform evaluations:

```
public class Evaluation {
  public static void eval(Plus p,Object l,Object r)
    throws Exception { System.out.println(p.plusMM(l,r)); }
  public static void main(String[] args) throws Exception
  { Evaluation.example(new Plus()); }
  public static void example(Plus p) throws Exception {
    Object d=new Double(3.14);
    Object cs=new StringBuffer("abc");
    Object i=new Integer(1);  Object b=new Byte((byte)8);
    eval(p, d, d);   // 6.28
    eval(p, cs, cs); // abcabc
    eval(p, cs, d);  // abc3.14
    eval(p, i, i);   // 2.0
    eval(p, d, i);   // 4.140000000000001
    eval(p, i, b);   // 9.0
    eval(p, i, cs);  // throws NoSuchMethodException
  }
}
```

The observed behavior complies with the semantics we chose. Note that the concatenation of a number with a character sequence, in this order, throws an exception.

2) Simple enhancements: In our framework, it is easy to extend the operator's semantics to be able to concatenate a number with a character sequence (as J2SDK currently does). We just have to specify a new method `CharSequence plus(Number n, CharSequence cs)`.

It is also easy to specialize the semantics for particular types of arguments. For instance, in the previous example, the result is wrapped in a `Double` object even if both arguments are integers. This may be unsatisfactory. To implement this special behavior, it suffices to define a specific method `Integer plus(Integer l, Integer r)`.

Even if the class `Plus` belongs to an unmodifiable component, it is possible to achieve these two enhancements by inheritance, as illustrated by the following `BetterPlus` extension class.

```
public class BetterPlus extends Plus {
  public CharSequence plus(Number n,CharSequence cs)
  { return n.toString()+cs; }        // For symmetry...
  Integer plus(Integer l,Integer r)  // For integers...
  { return new Integer(l.intValue()+r.intValue()); }
}
```

To use this new semantics, the developer just has to provide a `BetterPlus` object to reuse the method `example()`, as proposed in following example:

```
public class BetterPlusMain {
  public static void main(String[] args) throws Exception
  { Evaluation.example(new BetterPlus()); }
}
```

Note that the code related to multi-methods is inherited from the class `Plus` and the class `BetterPlus` never directly references the package JMMF. Thus, this extension class may have been developed without any knowledge of multi-methods, in order to dynamically extend an existing parser. For instance, the method `main` of class `BetterPlusMain` could contain the following code, expecting the name of the extension class from the user.

```
Evaluation.example((Plus)(Class.forName(args[0])
                          .newInstance()));
```

This implementation yields the expected results since our dispatch technique dynamically considers all methods `plus()` with two parameters in the class `BetterPlus`. This is due to the first argument of the `invoke()` method, set to `this` in the super class `Plus`. This provides JMMF with incremental definition capabilities and leads to a behavior that complies with the concept of inheritance in object-oriented languages.

The call `plusMM(i,b)` of the previous example still returns a double object (`9.0`). Indeed, since `Byte` is not a sub-type of `Integer`, any call with an argument of type `Byte` does not match with the integer's `plus()` method. A solution that takes into account all combinations of integer types (`Integer`, `Short` and `Byte`) requires the developer to explicitly define all (nine) methods `plus()`. Nevertheless, this solution is more elegant and maintainable than a cascade of `instanceof` tests.

Note that the "symmetric" semantics of the operator `+` could have been achieved by specifying only the two following methods, `Number plus(Number n1, Number n2)` and `CharSequence plus(Object o1, Object o2)`, the latter matching all cases but those of the former[1].

3) Ambiguous argument type: Imagine that the `Expr` grammar rule is extended with binary integers represented as a sequence of characters `'0'` and `'1'`. These binary integers may be implemented by users with a class that both extends `Number` and implements `CharSequence` as follows:

```
class BinaryInteger extends Number implements CharSequence{}
```

From the point of view of the `+` operator's semantics, objects of type `BinaryInteger` are ambiguous. Indeed, since the types `Number` and `CharSequence` are not comparable, nothing allows us to decide if such objects have to be added or concatenated. More generally, anywhere a `BinaryInteger` is used, there is an ambiguity if several methods could accept one of its super-types. This behavior is illustrated by the following piece of code.

```
Object cs = new StringBuffer("abc");
Object bi = new BinaryInteger("101");
Object p = new BetterPlus();
p.plusMM(bi, bi);
  /* throws fr.umlv.jmmf.reflect.MultipleMethodsException:
     plus(java.lang.Number,java.lang.CharSequence)
     plus(java.lang.Number,java.lang.Number)
     plus(java.lang.CharSequence,java.lang.CharSequence)
     plus(java.lang.CharSequence,java.lang.Number) */
```

[1]These solutions are not equivalent (see section IV-A.5). In the `Plus` class, methods are *not comparable* while there, the first method is *more specific* than the second.

```
p.plusMM(cs, bi);
  /* throws fr.umlv.jmmf.reflect.MultipleMethodsException:
     plus(java.lang.CharSequence,java.lang.CharSequence)
     plus(java.lang.CharSequence,java.lang.Number) */
```

Again, it is possible to refine the semantics by implementing a specific method `plus()` for each particular case. Indeed, the exact semantics of the operator could be specified for any combination of argument types, depending on the user's choice. Nevertheless, by taking into account type relations, it suffices to define some *generic* methods, provided that a single method could be unambiguously associated with any pair of argument types. For instance, the following class `BinaryPlus` defines methods in order to add its arguments if both are `BinaryInteger` or if one is `BinaryInteger` and the other is an `Integer`. In all other cases, it concatenates their string representations.

```
import fr.umlv.jmmf.reflect.*;
public class BinaryPlus extends BetterPlus {
  public BinaryInteger plus(BinaryInteger l,BinaryInteger r)
  { return new BinaryInteger(l.intValue()+r.intValue()); }
  public CharSequence plus(BinaryInteger bi,CharSequence cs)
  { return plus(bi.toString(),cs); }
  public CharSequence plus(CharSequence cs,BinaryInteger bi)
  { return plus(cs,bi.toString()); }
  public CharSequence plus(BinaryInteger bi,Number n)
  { return plus(bi.toString(),n); }
  public CharSequence plus(Number n,BinaryInteger bi)
  { return plus(n,bi.toString()); }
  public BinaryInteger plus(BinaryInteger bi,Integer i)
  { return new BinaryInteger(bi.intValue()+i.intValue()); }
  public BinaryInteger plus(Integer i,BinaryInteger bi)
  { return new BinaryInteger(i.intValue()+bi.intValue()); }
}
```

The dispatch algorithm provided by JMMF saves the programmer the bother of developing and maintaining an obfuscated cascade of `instanceof` tests that switches between `plus` implementations with respect to the dynamic type of the arguments.

Attempting to incrementally develop the corresponding `instanceof` solution will allow anyone to grasp the interest of the JMMF solution.

III. Design Choices

There are multiple ways to implement special language features like multi-polymorphism in the Java environment. Among them, approaches based on reflection are certainly the most flexible but also the least employed. In this section, we first compare them with other existing approaches and then we give general guidelines to make them practicable.

The most classical approach to implement Java extensions consists in modifying its syntax and in providing the corresponding translator or compiler [4]. This approach has the advantage of being static and thus, allows many computations to be performed before execution. However, it has the corresponding drawback: it can only use information available at compile time. However, in the context of Java's dynamic class loading, some essential information may only be available at runtime.

A second approach, which is sometimes complementary when runtime information is required, consists in modifying the Java Virtual Machine or its semantics. It provides precise runtime information on the executing application with minimum overhead, but it requires tricky knowledge of the internal

of a specific virtual machine. Furthermore, applications implemented using such a modified virtual machine require a special execution environment. This is more tedious: Java portability is lost.

An alternative approach, chosen to implement JMMF or others [14], [15], consists in using reflection to access JVM's runtime internal structures. It has the advantage of being simple and directly accessible to any Java programmer through the Java Core reflection API. Moreover, its deployment only consists in adding a single JAR file in the "classpath" of a standard Java environment. Unfortunately, this attractive property is obtained to the detriment of performance. Firstly, the reification of JVM internal objects induces some performance overhead compared to a direct access. For instance, a simple reflective method call is 200 times slower than a classical method call and still 10 times slower if the JVM is started is server mode, but this behavior is JRE-dependent (tested with j2sdk1.4.2 for Linux on Pentium 2.4 Ghz with 512 Mb) proposed to reduce this cost using code generation and we hope that this technique will be included in future virtual machine releases. Secondly, since Java does not allow modifications of its internal structures through reflection, some data structures have to be duplicated outside JVM, in the application. For instance, if you want to associate information with classes, you cannot add a field to `java.lang.Class` objects but you have to rely on an external structure such as hashtables. Thus, using reflection may lead to important performance penalties in terms of time and space if implementation is carried out without caution. We believe that a general strategy, already used by other frameworks [14], allows enhancement of reflection-based implementations.

First, to reduce time overhead, as many computations as possible have to be transfered from invocation time (just in time) to creation time. Indeed, durations of many computations remain negligible (see section V) compared to class loading (disk access and verification) and then, are not perceived by users. Nevertheless, these pre-computations imply that some states must be stored in order to be available just in time. Programmers have to find the right balance between invocation time, creation time and space. For instance, in the context of multi-methods, it is possible to determine, at creation time, which method will be called for every "interesting" tuple of available types. This strategy transfers most computations from invocation time to creation time but, in the worst case, it produces large data structures whose size is proportional to the number of "interesting" types to the power of the number of parameters. Thus, the choice of the algorithm is essential to make reflective implementation practicable in time and space.

Second, to reduce space overhead, as many data as possible must be shared. This can be achieved through specific algorithm implementations, such as table rows sharing [18]. However, this sharing should also conform to classical object-oriented design principles. Indeed, it is usually possible to share data between objects of the same class, e.g. through static fields, and between classes if they are related by inheritance. These data structures should support incremental modifications to insure that creation time information is still usable when some new information is discovered at invocation time. Again, it is primordial to drive algorithmic choices by space concerns.

Developers must also take into account multi-threading. Unfortunately, this worry is usually contradictory with time and space performances. Indeed, to maintain data coherence, synchronization is usually required and induces extra duration for method calls. More precisely, a method call is about 10 times slower if synchronized, even without mutual exclusion delay. One way to avoid synchronization consists in relaxing coherence and duplicating data but it leads to space overhead.

All these precepts have driven our algorithmic choices during JMMF implementation, that has been developed for environments where space concerns are not crucial, e.g. workstations. However, the constraints of a specific environment may induce different weighting between space, creation time and execution time. For instance, small devices' lack of memory may lead to implementations that sacrifice time for space.

IV. Implementation of the Java Multi-Method Framework

Basically, the algorithm we propose for multi-method dispatch consists of two main steps. The first one is processed at the multi-method *creation time*, that is when the static method `MultiMethod.create()` is invoked. This step performs reflection-based analysis and computes several data structures that will be used each time a dispatch is necessary for this multi-method. The second step is processed at *invocation time*, that is when the method `MultiMethod.invoke()` is called. Based on the data structures built at the first step, the set of applicable and accessible methods for this call site is refined in order to provide the most specific method, possibly requiring a disambiguation process. This step is comparable to dynamic tests performed at runtime with `instanceof` guards. These two stages are described in detail in the next two sections.

A. Data structure construction at creation time

1) Syntactically applicable methods: Let us consider the multi-method constructed by `MultiMethod.create(hostClass,"methodName",p)`, where `hostClass` is a concrete class in which each accessible method of name `methodName` with `p` parameters is public, non-static and does not override another method in a super-class of `hostClass`. These restrictions are made in order to simplify explanations but they will be relaxed in sections IV-C and IV-D.

Given these hypotheses, we determine the set of methods hosted by class `hostClass`, declared with the name `methodName` and with exactly `p` parameters. Thus, we add to methods declared in `hostClass` those inherited from super-classes and super-interfaces. We call it the set of *syntactically applicable* methods since only the name and the number of parameters are matching with the required method. In the classical Java method resolution ([19] § 15.12.2.1), the notion of *applicable* methods is used. This means, in addition to our *syntactically applicable* notion, that the type of each argument can be converted[2] into the type of the corresponding parameter

[2]These conversions correspond to our sub-typing relation defined in section IV-A.2.

and we will need to take care, further, of this "semantic" information. As in the classical method dispatch algorithm, return types and exceptions thrown by methods are not taken into account.

The set of method signatures associated with the multi-method is formally represented by a set $\mathcal{M}$, that is sometimes called the *behavior* of the multi-method:

$$\mathcal{M} = \{ \quad m_1 : \texttt{methodName}(T_{1,1}, ..., T_{1,p}), \\ \vdots \\ m_n : \texttt{methodName}(T_{n,1}, ..., T_{n,p}) \ \}$$

where $T_{i,j}$ identifies the declared type of the j-th parameter of the method m_i.

This set is arbitrarily indexed by integers i from 1 to n. These indexes uniquely identify each method (signature) since we have first assumed that no method overrides another. As a corollary, note that if m_i and m_k have exactly the same signature then i equals k. Thus, for the sake of simplicity, we identify a method by its signature.

From the implementation point of view, this implies that a table allows us to associate each signature in $\mathcal{M}$ (a given m_i) with the corresponding implementation (an object of class `java.lang.reflect.Method`). This association is done when looking for accessible methods in the host class using the reflexive method `getMethods()` on `hostClass`. This table is used, at invocation time, to actually invoke the chosen method, and could be compared to *vtables* used by classical dispatch techniques [20].

2) Sub-typing relations and parameter type hierarchy: In this paper, we improperly call sub-typing relation, and denote $T' \leq T$, the Liskov's substitution principle [21] allowing any value of type T' to be used in place of a value of type T. We consider this sub-typing relation as the reflexive transitive closure of a non reflexive relation of *direct sub-type*, denoted $T' <_1 T$.

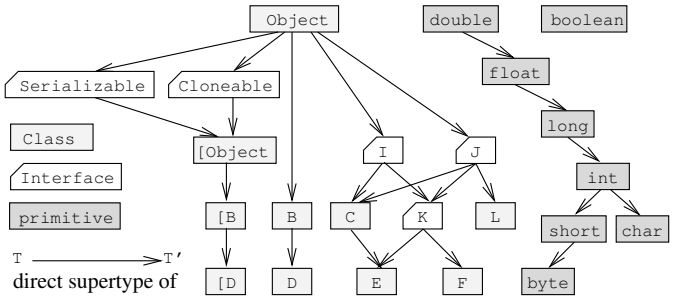


Fig. 1. Classes, interfaces, arrays and primitive types hierarchies

In Java [19], this sub-typing relation is provided by *identity conversion*, *widening primitive conversion*, sometimes called implicit primitive cast (cf. Fig. 1), and *widening reference conversion* provided by several contexts (explicit inheritance, implicit inheritance of class `Object`, interface implementation and some other cases specific to Java – e.g., with arrays as shown in Fig. 1, where `[B` is an array of `B`).

To compare types to each other and deal with sub-typing relations, the JMMF package provides a method `getSupertypes(`T`)` that returns the set of all *direct* super-types of T, i.e., $\{T' \mid T <_1 T'\}$.

3) Directed acyclic graph: To represent these relations between types, we use a graph $\mathcal{G}$ where vertices stand for types and edges for direct sub-typing relations. Thus, $\mathcal{G}$ is oriented and acyclic (from the essence of sub-typing), but it is not a tree because of the multiple super-typing allowed by some Java features[3].

Definition 1 (DAG as type hierarchy): Given a set of methods $\mathcal{M}$ related to a multi-method definition, let $\mathcal{T}_{param} = \{T_{i,j} \mid \exists (i,j) \in [1..n] \times [1..p]\}$ be the set of all types declared as a parameter type of these methods. We consider the type hierarchy associated with $\mathcal{M}$ as a Direct Acyclic Graph (DAG) $\mathcal{G}$ whose set of vertices is $\mathcal{V}(\mathcal{G}) = \bigcup_{T' \in \mathcal{T}_{param}} \{T | T' \leq T\}$ and where there is an edge $T \rightarrow T'$ if and only if $T' <_1 T$. We use the classical notation $T \rightarrow^* T'$ if $T' \leq T$.

Fig. 1 gives examples of Java type hierarchies, involving classes, interfaces, and primitive types, which are very close to our expected DAG $\mathcal{G}$. Note that this figure does not distinguish between *extends*, *implements* or other sub-typing (assignment conversion) relations.

Given a multi-method, the corresponding DAG is constructed by adding each type that appears as a parameter type of an applicable method (in $\mathcal{T}_{param}$), together with all its super-types, recursively obtained by the `getSupertypes()` method, until reaching the fix point[4].

4) Annotate the DAG: In order to compare methods from their p-uples of declared types (signatures), we annotate each type considered in the DAG by its ability to be an acceptable argument type for methods of the multi-method. This annotation is done for each method and at each parameter position. It is represented by a bit matrix $\mathcal{A}_T$ of n rows and p columns, where the value $\mathcal{A}_T[i][j]$ stands for the ability of T to be the type of the j-th argument of method m_i.

Definition 2 (Type annotation): The annotation $\mathcal{A}_{T_{i,j}}[r][c]$ of type $T_{i,j}$ at position c of method m_r is set to 1 if and only if one of the following is true (otherwise $\mathcal{A}_{T_{i,j}}[r][c] = 0$):

- $i = r$ and $j = c$, i.e., $T_{i,j} = T_{r,c}$ is the c-th declared parameter type of method m_r;
- $\exists\ T_{r,c} \in \mathcal{V}(\mathcal{G})$ (c-th parameter type of method m_r) such that $\mathcal{A}_{T_{r,c}}[r][c] = 1$ and $T_{r,c} \rightarrow^* T_{i,j}$.

Lemma 1 (Acceptable argument types): A type $T \in \mathcal{V}(\mathcal{G})$ is acceptable as the c-th argument of method m_r if and only if $\mathcal{A}_T[r][c] = 1$.

If a value of type T is acceptable as the j-th argument of method m_i, then any value of any sub-type T' is also acceptable. Thus, the constructive algorithm for annotations successively considers each parameter type $T_{i,j}$ in $\mathcal{T}_{param}$ and sets to 1 its annotation bit $\mathcal{A}_T[i][j]$. Annotations are then recursively *propagated* onto $\mathcal{A}_{T'}[i][j]$ for every sub-type T' of T considered in the DAG. This propagation, that follows the edge of the DAG, could simply perform a bitwise `OR` with previously computed annotations. The fact that our graph is acyclic implies that the propagation terminates.

[3]For instance, the ability of an interface to extend more than one interface.

[4]Termination is insured by the types `Object`, `double` and `boolean` that are the roots of these hierarchies.

As an example, consider the following multi-method that exploits a part of the (particularly intricate) type hierarchy of Fig. 1:

```
MultiMethod mm=MultiMethod
                .create(HostClass.class,"myMethod",3);
```

where accessible and syntactically applicable methods `myMethod()` declared in `HostClass` with exactly three parameters are defined with the following signatures:

$$\mathcal{M} = \{ \quad m_1 : \texttt{myMethod(B,C,K)}, \\ m_2 : \texttt{myMethod(D,I,I)}, \\ m_3 : \texttt{myMethod(B,I,J)} \ \}$$

The annotated DAG obtained for this multi-method is presented in Fig. 2 where a bullet points out the fact that the annotation $\mathcal{A}_T[i][j]$ is set to 1. Dark bullets stand for set annotations (types declared as parameter) and gray bullets for propagated ones.

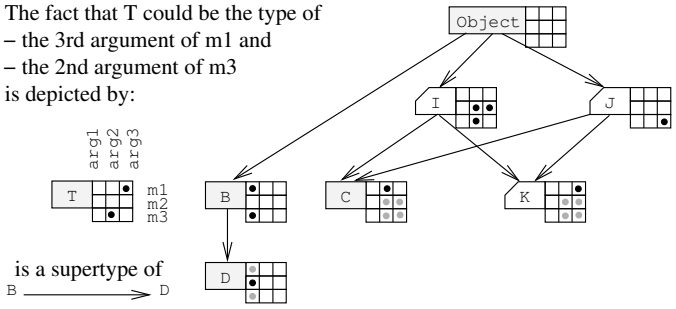


Fig. 2. An example of annotated DAG

5) Partial order over methods: This annotated DAG allows us to establish relations between methods based on their respective p-uples of parameter types. Indeed, from the sub-typing relation between types, we define the relation of *specificity* between two methods as follows:

Definition 3 (Specificity relation between methods): A method $m_i : (T_{i,1}, ..., T_{i,p})$ is more specific than a method $m_k : (T_{k,1}, ..., T_{k,p})$, denoted $m_i \leq m_k$, if and only if, for each parameter position j, $T_{i,j}$ is a sub-type of $T_{k,j}$ (or they are equal):

$$m_i : (T_{i,1}, ..., T_{i,p}) \leq m_k : (T_{k,1}, ..., T_{k,p}) \\ \Leftrightarrow \forall j \in [1..p],\ T_{i,j} \leq T_{k,j}$$

We also say in this case that m_k is less specific than m_i.

At this point, it is worthwhile to note three important things.

1) The intuition behind the relation "m_i is *more specific* than m_k" is that m_k could always be invoked with any p-uple of argument types accepted by m_i. For instance, consider the methods m_1:`(B,C,K)` and m_3:`(B,I,J)`. Since `B` $\leq$ `B`, `C` $\leq$ `I` and `K` $\leq$ `J`, m_1 is more specific than m_3, i.e., m_3 could accept as argument any triple of types acceptable for m_1.
2) Next, it is possible, according to definition 3, that both $m_i \leq m_k$ and $m_k \leq m_i$; this case only arises when, for all position j, $T_{i,j} = T_{k,j}$. This implies that $m_i = m_k$ since we have assumed that two methods with the same signature are necessarily equal (section IV-A.1).
3) Finally, it is not always possible to order two methods with respect to the *specificity* relation, i.e., some m_i could be neither more specific nor less specific than some m_k. These methods are called *not comparable* and provide us with a partial order on the set of methods $\mathcal{M}$. Two main reasons could yield not comparable methods:
   - declared parameter types at a given position are not comparable, e.g. m_2 : `(D,I,I)` and m_3 : `(B,I,J)` are not comparable because `I` $\not\leq$ `J` and `J` $\not\leq$ `I`;
   - parameter sub-typing relations are opposite for two positions, e.g., m_1:`(B,C,K)` and m_2:`(D,I,I)` are not comparable because `D` $\leq$ `B` and `C` $\leq$ `I`.

Given a multi-method, these relations between methods are stored in a bit matrix $\mathcal{PO}$ of n rows and n columns.

Definition 4 (Partial order over methods): Given two methods m_r and m_c, $\mathcal{PO}[r][c]$ is set to 1 if and only if $m_c \leq m_r$, and is set to 0 in all other cases.

A bit set to 1 at $\mathcal{PO}[r][c]$, meaning that $m_c \leq m_r$, implies that method m_r could accept as argument any p-uple of values that is acceptable by the method m_c. A row $\mathcal{PO}[r]$ is then a bit array whose values at 1 identify the set (the indexes) of methods that are more specific than m_r.

Since sets are represented by bit arrays, we will use both notations equally, i.e., a method is in a set if the bit at its index is set to 1. Furthermore, we can equally use the set-theoretical operations (union/intersection) and bitwise operations (`OR`/`AND`).

The algorithm that allows the structure $\mathcal{PO}$ to be computed is very simple. In order to know if a method m_c is more specific than a method m_r, it suffices to verify, for each parameter position j, that the parameter type $T_{c,j}$ is an acceptable argument type for method m_r at this position. Since this information is precisely represented by the annotation $\mathcal{A}_{T_{c,j}}[r][j]$, we could formally define, and then compute, the values of the matrix $\mathcal{PO}$:

Theorem 1 ($\mathcal{PO}$ as annotation conjunction):

$$\mathcal{PO}[r][c] = 1 \Leftrightarrow \texttt{AND}_{j \in [1..p]} \mathcal{A}_{T_{c,j}}[r][j] = 1$$

Corollary 1 (Computation of less specific methods):

$$\mathcal{PO}[*][c] = \texttt{AND}_{j \in [1..p]} \mathcal{A}_{T_{c,j}}[*][j]$$

Considering our multi-method example, the matrix $\mathcal{PO}$ is computed as follows:

$$\begin{aligned} \mathcal{PO}[*][1] &= \mathcal{A}_{\texttt{B}}[*][1] \,\texttt{AND}\, \mathcal{A}_{\texttt{C}}[*][2] \,\texttt{AND}\, \mathcal{A}_{\texttt{K}}[*][3] \\ &= \begin{bmatrix} \bullet \\ \circ \\ \bullet \end{bmatrix} \texttt{AND} \begin{bmatrix} \bullet \\ \bullet \\ \bullet \end{bmatrix} \texttt{AND} \begin{bmatrix} \bullet \\ \bullet \\ \bullet \end{bmatrix} = \begin{bmatrix} \bullet \\ \circ \\ \bullet \end{bmatrix} \\ \mathcal{PO}[*][2] &= \mathcal{A}_{\texttt{D}}[*][1] \,\texttt{AND}\, \mathcal{A}_{\texttt{I}}[*][2] \,\texttt{AND}\, \mathcal{A}_{\texttt{I}}[*][3] \\ &= \begin{bmatrix} \bullet \\ \bullet \\ \bullet \end{bmatrix} \texttt{AND} \begin{bmatrix} \circ \\ \bullet \\ \bullet \end{bmatrix} \texttt{AND} \begin{bmatrix} \circ \\ \bullet \\ \circ \end{bmatrix} = \begin{bmatrix} \circ \\ \bullet \\ \circ \end{bmatrix} \\ \mathcal{PO}[*][3] &= \mathcal{A}_{\texttt{B}}[*][1] \,\texttt{AND}\, \mathcal{A}_{\texttt{I}}[*][2] \,\texttt{AND}\, \mathcal{A}_{\texttt{J}}[*][3] \\ &= \begin{bmatrix} \bullet \\ \circ \\ \bullet \end{bmatrix} \texttt{AND} \begin{bmatrix} \circ \\ \bullet \\ \bullet \end{bmatrix} \texttt{AND} \begin{bmatrix} \circ \\ \circ \\ \bullet \end{bmatrix} = \begin{bmatrix} \circ \\ \circ \\ \bullet \end{bmatrix} \end{aligned}$$

thus,

$$\mathcal{PO} = \begin{bmatrix} \bullet & \circ & \circ \\ \circ & \bullet & \circ \\ \bullet & \circ & \bullet \end{bmatrix} \ \text{ and } \ \mathcal{PO}^t = \begin{bmatrix} \bullet & \circ & \bullet \\ \circ & \bullet & \circ \\ \circ & \circ & \bullet \end{bmatrix}$$

The third row of this matrix (third column of $\mathcal{PO}^t$, the transposed matrix of $\mathcal{PO}$) provides us with the information that m_1 and m_3 are more specific than m_3.

B. Multi-method dispatch at invocation time

The process described in the previous section is completely done at creation time. We now consider the invocation time process: given a p-uple of (dynamic) argument types of the multi-method invocation, our aim is to dispatch the invocation to the most specific method, if any, corresponding to this p-uple of types. A first problem can arise if one of these types does not appear in the DAG we built. This type can be unknown at compile time since, in Java, classes can be loaded dynamically at runtime. Thus, even a static analysis of the type hierarchy at creation time cannot identify such a type.

1) Completing DAG with dynamic types: For instance, consider the following invocation:

```
D d = new D();  C c = new C();
J l = createDynamicInstanceAssignableToJ();
mm.invoke(target,new Object[] {d,c,l});
```

In such a case, the type `L` of `l`[5] is only known at runtime. We suppose here that a class `L`, implementing interface `J`, is dynamically loaded (for instance from the network) by the method `createDynamicInstanceAssignableToJ()`. Thus, we need to complete our DAG $\mathcal{G}$ in order to establish relations between `L` and the other types and also to compute its annotations.

DAG completion at invocation time is performed with the same algorithm as at creation time. A newly discovered type T is first added as a vertex of the DAG, together with all its super-types that do not yet appear in $\mathcal{G}$. Annotations of all newly added types are recursively deduced (by bitwise `OR` propagation) from those of their direct super-types. Actually, both DAG completion and annotation initialization could be performed at the same time.

Note that the data structure of partial order, $\mathcal{PO}$, is not concerned by these modifications since newly discovered types are necessarily different from parameter types.

2) Semantically applicable methods: Type annotations, computed either at creation or invocation time, tell us if a single given type is acceptable as argument at a given position of a given method. For a p-uple $u = (T_1, ..., T_p)$ of argument types of a given invocation site, we are looking for the set (represented by a bit array), denoted $\mathcal{SA}_u$, of *semantically applicable* methods that could accept u as argument types.

Definition 5 (Semantically applicable methods): Let $u = (T_1, ..., T_p)$ be a p-uple of types in $\mathcal{V}(\mathcal{G})$, $\mathcal{SA}_u[i]$ is set to 1 if and only if u is a p-uple of argument types acceptable for m_i.

Theorem 2 ($\mathcal{SA}_u$ as annotation conjunction):

$$\mathcal{SA}_u = \texttt{AND}_{j \in [1..p]} \mathcal{A}_{T_j}[j]$$

The number of bit set to 1 in $\mathcal{SA}_u$ gives us important information, summarized below in three cases and further illustrated by examples u_1, u_2 and u_3:

1) if no bit is set to 1, then there is no semantically applicable method. In this case, our implementation throws an exception of class `NoSuchMethodException`;
2) if only one bit at index i is set to 1, then m_i is the single semantically applicable method. In this case, the method `invoke()` of the object `Method` corresponding to m_i is called;
3) if more than one bit is set to 1, then multiple methods are semantically applicable and we do not yet have enough information to decide what will happen. Some *disambiguation* processing is needed.

Let u_1 = `(B,C,D)` be the p-uple of argument types of a multi-method `mm` invocation. Then, theorem 2 leads to $\mathcal{SA}_{u_1} = \mathcal{A}_{\texttt{B}}[*][1]$ `AND` $\mathcal{A}_{\texttt{C}}[*][2]$ `AND` $\mathcal{A}_{\texttt{D}}[*][3]$ and then $\begin{bmatrix}\bullet\\ \circ\\ \bullet\end{bmatrix}$ `AND` $\begin{bmatrix}\circ\\ \bullet\\ \bullet\end{bmatrix}$ `AND` $\begin{bmatrix}\circ\\ \circ\\ \circ\end{bmatrix} = \begin{bmatrix}\circ\\ \circ\\ \circ\end{bmatrix}$ provides that there is no semantically applicable method for u_1.

If u_2 = `(D,C,L)`, the same principle gives $\mathcal{SA}_{u_2} = \begin{bmatrix}\circ\\ \circ\\ \bullet\end{bmatrix}$. The only semantically applicable method for u_2 is then m_3 : `(B,I,J)`.

Now, let u_3 be `(D,C,C)`. Since $\mathcal{SA}_{u_3} = \begin{bmatrix}\circ\\ \bullet\\ \bullet\end{bmatrix}$, there are two semantically applicable methods for u_3, m_2 : `(D,I,I)` and m_3 : `(B,I,J)`, but we cannot decide which one must be invoked.

3) Disambiguation process from method's partial order: For a given p-uple u of argument types, the disambiguation process presented in this section is only performed when the number of semantically applicable methods is greater than one. In this case, we want to determine if one of these methods is more specific than all the others. In order to get this information, we first compute a bit array $\mathcal{MSA}_u$.

Definition 6 (Most specific semantically applicable):

$$\mathcal{MSA}_u = \mathcal{SA}_u \texttt{ AND } (\texttt{AND}_{\{l \mid \mathcal{SA}_u[l]=1\}} \mathcal{PO}^t[*][l])$$

Theorem 3 (Set of methods associated with $\mathcal{MSA}$): The set of methods $\{m_i | \mathcal{MSA}_u[i] = 1\}$ is either empty or a singleton.

Theorem 4 (Most specific method): A method m_i is the most specific semantically applicable method for u if and only if $\mathcal{MSA}_u[i] = 1$.

Corollary 2 (Existence of a most specific method): Given a p-uple u, a most specific method does not exist if and only if $\forall i \in [1..n],\ \mathcal{MSA}_u[i] = 0$.

In order to illustrate these situations, consider the p-uple u_3 = `(D,C,C)` for which we deduced, in section IV-B.2 and from $\mathcal{SA}_{u_3} = \begin{bmatrix}\circ\\ \bullet\\ \bullet\end{bmatrix}$, that there was an ambiguity. Now, from definition 6 and from the matrix $\mathcal{PO}^t$ computed in section IV-A.5, we could compute $\mathcal{MSA}_{u_3}$:

$$\begin{aligned}\mathcal{MSA}_{u_3} &= \mathcal{SA}_{u_3} \texttt{ AND } (\texttt{AND}_{\{l \mid \mathcal{SA}_{u_3}[l]=1\}} \mathcal{PO}^t[*][l]) \\ &= \begin{bmatrix}\circ\\ \bullet\\ \bullet\end{bmatrix} \texttt{ AND } (\mathcal{PO}^t[*][2] \texttt{ AND } \mathcal{PO}^t[*][3]) \\ &= \begin{bmatrix}\circ\\ \bullet\\ \bullet\end{bmatrix} \texttt{ AND } (\begin{bmatrix}\circ\\ \bullet\\ \circ\end{bmatrix} \texttt{ AND } \begin{bmatrix}\bullet\\ \circ\\ \bullet\end{bmatrix}) = \begin{bmatrix}\circ\\ \bullet\\ \bullet\end{bmatrix} \texttt{ AND } \begin{bmatrix}\circ\\ \circ\\ \circ\end{bmatrix} = \begin{bmatrix}\circ\\ \circ\\ \circ\end{bmatrix}\end{aligned}$$

Then, from corollary 2, there is no most specific method for u_3. This is not surprising since the only semantically applicable methods for u are m_2 and m_3, and they are not comparable. In this case, our implementation throws an exception of class `MultipleMethodsException`.

Let us consider another example of invocation with the p-uple of types u_4 = `(B,C,F)` where `F` is a class that implements `K`. The DAG is then completed (annotations on `F` are exactly those of `K`) and, from theorem 2, $\mathcal{SA}_{u_4} = \begin{bmatrix}\bullet\\ \circ\\ \bullet\end{bmatrix}$.

[5] By convention, we denote by a lowercase character such as `t` a variable that contains an object of type `T`.

Since there is an ambiguity, we compute $\mathcal{MSA}_{u_4}$:

$$\begin{aligned}\mathcal{MSA}_{u_4} &= \mathcal{SA}_{u_4} \texttt{ AND } (\texttt{AND}_{\{l \mid \mathcal{SA}_{u_4}[l]=1\}}\ \mathcal{PO}^t[*][l]) \\ &= \begin{bmatrix}\bullet\\ \circ\\ \bullet\end{bmatrix} \texttt{ AND } (\mathcal{PO}^t[*][1] \texttt{ AND } \mathcal{PO}^t[*][3]) \\ &= \begin{bmatrix}\bullet\\ \circ\\ \bullet\end{bmatrix} \texttt{ AND } (\begin{bmatrix}\bullet\\ \circ\\ \circ\end{bmatrix} \texttt{ AND } \begin{bmatrix}\bullet\\ \circ\\ \bullet\end{bmatrix}) = \begin{bmatrix}\bullet\\ \circ\\ \bullet\end{bmatrix} \texttt{ AND } \begin{bmatrix}\bullet\\ \circ\\ \circ\end{bmatrix} = \begin{bmatrix}\bullet\\ \circ\\ \circ\end{bmatrix}\end{aligned}$$

Thanks to this disambiguation technique, we are now able to determine that the most specific semantically applicable method is m_1.

C. Inheritance and multi-methods

As for other classes, inheritance between multi-methods allows simple code genericity and provides reusability. For instance, remember our introductory example that defines the semantics of the operator +. From the basic multi-method definition in class `Plus`, the semantics could be refined by adding specifications through inheritance, like the class `BetterPlus` does. This could even be done if this class is tardily loaded, after the basic multi-method was created and even invoked. In this case, both semantics could separately coexist, the basic one and the refined one, and they rely on the same multi-method base class. From the implementation point of view, this feature is expected to be provided incrementally without requiring all data structures used by the refined multi-method to be computed from scratch.

The JMMF implementation accepts a multi-method call with a target object instance of a sub-type `SubHostClass` of the class `HostClass` specified at creation time (if the argument type is not a sub-type of `HostClass`, an exception is thrown). Since the class `SubHostClass` may contain additional syntactically applicable methods to those of `HostClass`, data structures computed for `HostClass` are not valid. Then, at first sight, we would have to compute at invocation time all data structures related to the multi-method. Fortunately, an interesting property of our algorithm is that it enables a total data structure sharing between `HostClass` and `SubHostClass`.

To establish this property, we are going to show that given a set of syntactically applicable method signatures $\mathcal{M}$, all computations can be performed on a larger set $\mathcal{M}'$. Thus, method signatures coming from `HostClass` and from `SubHostClass` could coexist in a single data structure.

Definition 7 (Multiple sets notations): Let $\mathcal{M}$ and $\mathcal{M}'$ be two sets of method signatures. We define $\mathcal{A}_T$ and $\mathcal{A}'_T$ the corresponding annotations of type T for, respectively, $\mathcal{M}$ and $\mathcal{M}'$. In the same way we define $\mathcal{PO}'$, $\mathcal{SA}'_u$ and $\mathcal{MSA}'_u$.

Moreover, if $\mathcal{M} \subseteq \mathcal{M}'$, for each signature index i such that $m_i \in \mathcal{M}$, we denote $i_{\mathcal{M}'}$ the index of the same signature in $\mathcal{M}'$ such that $m_{i_{\mathcal{M}'}} \in \mathcal{M}'$ and $m_i = m_{i_{\mathcal{M}'}}$.

Definition 8 (Included set mask): Given two sets of method signatures $\mathcal{M}$ and $\mathcal{M}'$ such that $\mathcal{M} \subseteq \mathcal{M}'$, we define the bit array $\mathcal{M}ask_{\mathcal{M}}$ such that,

$$\forall m_i \in \mathcal{M}',\ \mathcal{M}ask_{\mathcal{M}}[i] = 1 \Leftrightarrow \exists\ m_k \in \mathcal{M} \text{ such that } i = k_{\mathcal{M}'}.$$

Intuitively, the mask $\mathcal{M}ask_{\mathcal{M}}$ is able to hide (by a bit set at 0) in $\mathcal{M}'$ all method signatures that are not considered in $\mathcal{M}$, and to leave visible in $\mathcal{M}'$ the method signatures that are already considered in $\mathcal{M}$, even if they are registered with a different index. This mask will allow us to prove the equivalence between structures computed from $\mathcal{M}$ and structures computed from the part of $\mathcal{M}'$ that corresponds to $\mathcal{M}$, modulo an index permutation.

Theorem 5 (Included set data structures): Given a type T and two sets of method signatures $\mathcal{M}$ and $\mathcal{M}'$ such that $\mathcal{M} \subseteq \mathcal{M}'$:

1) $\mathcal{A}_T[i][j] = 1 \Leftrightarrow \mathcal{A'}_T[i_{\mathcal{M}'}][j] = \mathcal{M}ask_{\mathcal{M}}[i_{\mathcal{M}'}] = 1$;
2) $\mathcal{PO}[i][k] = 1 \Leftrightarrow \mathcal{PO}'[i_{\mathcal{M}'}][k_{\mathcal{M}'}] = \mathcal{M}ask_{\mathcal{M}}[i_{\mathcal{M}'}] = \mathcal{M}ask_{\mathcal{M}}[k_{\mathcal{M}'}] = 1$.
3) $\mathcal{SA}_u[i] = 1 \Leftrightarrow \mathcal{SA}'_u[i_{\mathcal{M}'}] = \mathcal{M}ask_{\mathcal{M}}[i_{\mathcal{M}'}] = 1$.

Definition 9 (Semantically applicable, included set): Let $\mathcal{M}$ and $\mathcal{M}'$ be two sets of method signatures such that $\mathcal{M} \subseteq \mathcal{M}'$. Given a p-uple u of argument types, we define the semantically applicable methods of $\mathcal{M}$ for u (denoted $\mathcal{SA}'_{u,\mathcal{M}}$) from the semantically applicable methods of $\mathcal{M}'$ for u as follows:

$$\mathcal{SA}'_{u,\mathcal{M}} = \mathcal{SA}'_u \texttt{ AND } \mathcal{M}ask_{\mathcal{M}}$$

The bit array $\mathcal{SA}'_{u,\mathcal{M}}$ provides us with a set of methods (indexes) from $\mathcal{M}'$ that are not only semantically applicable for the p-uple u but also considered in $\mathcal{M}$. As we did in section IV-B.3, we now want to know how to refine this set if several such semantically applicable methods exist (ambiguity). In order to disambiguate, we have to take care that, in the set $\mathcal{PO'}^t$, only method indexes of $\mathcal{SA}'_{u,\mathcal{M}}$ have to be considered (and not all $\mathcal{SA}'_u$).

Theorem 6 (Most specific method for included set): Given two sets of method signatures $\mathcal{M} \subseteq \mathcal{M}'$:

$$\mathcal{MSA}_u[i] = 1 \Leftrightarrow \mathcal{SA}'_{u,\mathcal{M}}[i_{\mathcal{M}'}] \texttt{ AND } (\texttt{AND}_{\{l \mid \mathcal{SA}'_{u,\mathcal{M}}[l]\}}\ \mathcal{PO'}^t[i_{\mathcal{M}'}][l]) = 1$$

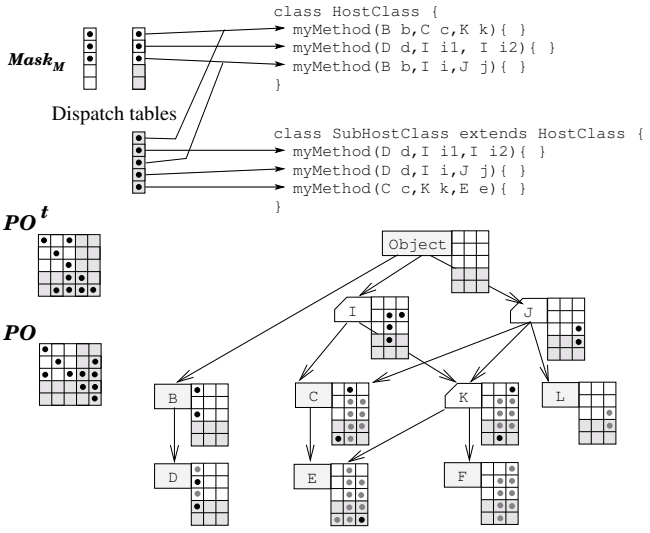


Fig. 3. Example of data structure involved in inheritance

If the set containing both syntactically applicable method signatures from `HostClass` and from `SubHostClass` is constructed without modifying indexes of `HostClass` signatures, then the whole part of the computation (annotations and partial order) that relies on `HostClass` can be reused. More generally, if a new `SubHostClass` is discovered dynamically,

all previous computations can be reused provided that previously assigned indexes are not modified. Fig. 3 illustrates such a case where the parameter type `E` is found in the `SubHostClass` multi-method definition. The values related to the multi-method `SubHostClass` are shown in gray. This figure also represents the mask used for each class and the table of `Method` instances.

D. Method restrictions

In section IV-A.1 we have supposed that all methods were public, concrete and non-static. In this section, we will explore which restrictions could be loosened.

For our algorithms, abstract methods behave like concrete ones, thus there is no restriction on using abstract methods in multi-methods.

Concerning static methods, there is no reason not to consider them in the set of syntactically applicable methods. The only special behavior they induce is when multi-method is called with a `null` target reference. Then, only static methods of the class used at creation time should be selected in the computations. As for hiding some methods in presence of inheritance, a mask allows all non-static methods to be hidden. Then, static methods dispatch is processed exactly as inheritance.

Concerning accessibility modifiers, if the method is not public, then our implementation must check that the class containing the invocation site of the multi-method has the correct access rights and must throw an exception if not. For private and default methods, extra verification should be performed in case of inheritance. By now, our implementation only deals with public methods.

E. The case of `null`

Until now, we did not consider the possibility of invoking a multi-method with a `null` argument. However, in `mm.invoke(target, new Object[]{d,c,l})`, any argument (except if a primitive type is expected) could be `null`.

A `null` argument cannot be considered as simply as a `null` target (static method). Contrarily to the target case where the `null` is typed by the multi-method host class, no information allows us to type a `null` argument. However, it is sufficient to consider `null` as a value of a special type which is a sub-type of all types.

F. Toward Java 1.5

Version 1.5 of Java introduces new language features such as generics, methods with variable arity parameters (*varargs*) and automatic *boxing/unboxing* of primitive types. This will require JMMF to be modified. Generics should not impact our multi-method resolution approach, since method resolution considers them as their *raw type*. Varargs methods require a special treatement. It does not suffice to fix the number of parameters according to the multi-method and to treat it as a regular method. Indeed, to preserve backward compatibility, a varargs method must be less specific than a classical method to be added to the behavior. Automatic boxing and unboxing of primitive types seems more problematic and will surely require more time and space to be spent to find the most specific method.

V. Performance issues

In this section, we present a performance evaluation of the JMMF package version 0.9 which is freely available on the Web at `http://www-igm.univ-mlv.fr/~forax/jmmf/`. First, method invocation performances of the JMMF package are discussed. Then, multi-method creation time is measured and a comparison between several single argument filtering methods is given. Finally, we provide an analysis of the memory space used by multi-methods.

To evaluate these results, one must take into account the fact that our algorithms have been implemented outside the JVM as stated in section III. We must also detail several points about the type hierarchy implementation. Firstly, to allow fast type access, annotations are not retrieved through a hierarchy traversal, but using an auxiliary hashtable. This implementation allows us to consider that annotations are obtained with complexity $\mathcal{O}(1)$. Secondly, in order to minimize space usage, we only keep useful types in the hashtable, other types are discarded.

Moreover, most of the time, our implementation stores bit vectors in integer values. In these cases, bitwise operations on such vectors and their size are in $\mathcal{O}(1)$.

All the tests of this section have been performed on a 2.4GHz Pentium 4 with 512Gb of RAM using SUN j2sdk1.4.2_03 under Linux[6].

A. Invocation performances

To evaluate invocation performance, we compare invocation times of JMMF with an implementation based on typecase guards using `instanceof` which is the usual translation used to introduce multi-polymorphism at language level [4], [7]. We did not compare directly JMMF with the jdk-SRP implementation [8] which customizes an old version[7] of the JVM since its performances are 1.2 times faster but comparable with `instanceof` implementation. We also measure the evolution of dispatch time when the number of methods increases. For each test, we evaluate the invocation time for the JVM in "client" and in "server" mode for ten thousand multi-method calls with randomly chosen arguments. Finally, we compare performance of JMMF with other techniques that only allow late-binding on one argument.

1) Impact of argument types: For the invocation time evaluation, we perform four tests in which the multi-method is composed of four methods with two parameters of two possible types. In the first test, these two types are unrelated interfaces. In the second, they are interfaces with a direct sub-typing relation. In the third one, they are unrelated concrete classes and in the last one they are concrete classes with a direct sub-typing relation. Table I summarizes the results of these tests.

[6]Other versions of the SDK and other systems give comparable results.

[7]It is based on jdk1.2. This proves that evolution of such an implementation is not easy.

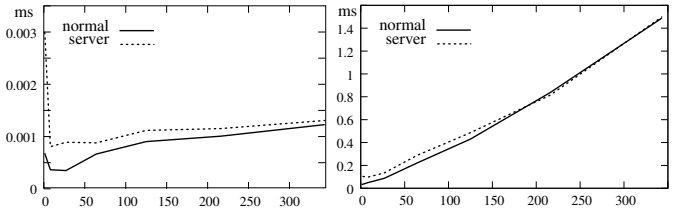
TABLE I
INVOCATION TIME COMPARISON: `INSTANCEOF` TESTS VS. JMMF

	clientVM (μs)		serverVM (μs)	
	Tests	JMMF	Tests	JMMF
unrelated interfaces	0.197	0.293	0.232	0.159
inheritance of interfaces	0.162	0.329	0.196	0.191
unrelated classes	0.128	0.314	0.103	0.163
inheritance of classes	0.125	0.333	0.104	0.201

This table shows that the `instanceof` dispatch is always faster than JMMF, except for two cases. However, performances are always comparable (1.7 times slower) even in the worst cases where JMMF is 2.7 times slower. The same performance ratio is observed with a different number of parameters and with a different number of types.

One can also notice that the invocation times of the tests using `instanceof` vary substantially. First, the number of `instanceof` statements varies with respect to the kind of argument types: some `instanceof` statements can be omitted if types have an inheritance relation or if they are classes rather than interfaces (due to single inheritance). Next, variations of the invocation times are also due to the fact that the `instanceof` test is around 30% faster on classes than on interfaces.

2) Impact of the number of methods: To evaluate the evolution of invocation time with respect to the number of methods, we generate methods with three parameters by increasing the number of parameter types. The results of this test, summarized in Fig. 4(a), show that JMMF invocation time remains usable even in the presence of many methods. The use of three implementations with respect to the number of methods explains the change of slope between 64 and 125 methods.



(a) Invocation time / number of methods (b) Création time / number of methods

Fig. 4. Multi-method invocation time and creation time with respect to the number of methods

3) Impact of primitive types: All previous tests did not use primitive types. This is favorable to the JMMF package. Indeed, since JMMF implementation is based on Java reflection mechanism, primitive types need to be wrapped when used as argument or return types. Wrapping time is comparable to JMMF invocation time and this could significantly penalize performance if many wrappings are required. However, primitive types are typically not used for multi-dispatch but used as return types. In these cases, wrapping does not impact too much on performance.

B. Creation time performance

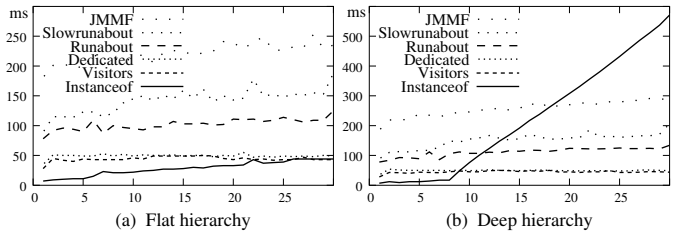
To evaluate multi-method creation time, we have measured multi-method class loading and creation time and we have compared it with the loading time of the corresponding class based on `instanceof`. These tests have been performed on the examples used to evaluate invocation time and they provide the following results: the loading time of the `instanceof` class is 1.24 ms[8], those of the JMMF class is 1.98 ms and the creation of the multi-method takes 0.11 ms. This benchmark shows that multi-method creation time is very small compared to class loading time, whereas `instanceof` class loading time is comparable with the JMMF one, although better. However, since multi-methods share the same implementation classes, this loading is only performed once, when the first multi-method is created.

The last benchmark, illustrated in Fig. 4(b), evaluates the evolution of multi-method creation time when the number of methods increases. For this test, we generate the same classes used to evaluate invocation time. It shows again that the JMMF approach is usable even in the presence of a large number of methods. Multi-method creation time remains of the millisecond order, as does class loading time.

C. Comparison with single argument late-binding methods

In previous tests, we have always used multi-methods with multiple arguments. In this section, we now compare JMMF with techniques restricted to a single argument late-binding and which only accept concrete classes as parameter types. This is clearly the worst use-case for JMMF.

We compare six techniques: *Dedicated* corresponds to an implementation where a dedicated method has been added inside the host class, *Visitors* corresponds to the visitor design pattern, *Instanceof* corresponds to the filtering using `instanceof`, *Runabout* is a reflexive implementation[9] of late-binding on a single argument [22], *Slowrunabout* is a modified version of *Runabout* that uses classical reflection instead of code generation and, finally, *JMMF*. Fig. 5 presents invocation times for one million method calls with an increasing number of methods. Two cases are considered for parameter type classes: unrelated or belonging to a deep hierarchy.



(a) Flat hierarchy (b) Deep hierarchy

Fig. 5. Single dispatch time, with respect to the number of methods

In this figure, one first notices that Dedicated and Visitors implementations are always faster than reflection-based ones,

[8]Note that these measured values strongly depend on hard disk characteristics.

[9]Dispatch is performed using a hashtable containing the method to apply for each argument type and code generation is used to improve reflective method calls.

but their performances are of the same order (twice as fast than *Runabout* and four times faster than `JMMF`). This confirms that the reflection-based approach is usable.

Secondly, the Instanceof implementation is better than all the others with flat hierarchies but its performance dramatically decreases in the presence of a deep inheritance hierarchy, as observed in [22]. However, note that the Instanceof performance decreases as the number of typecases increases. This is observed both in Fig. 5(b) and in Table I. This probably comes from cache-based optimization.

Finally, JMMF is about 2.5 times slower than Runabout. Since Runabout is 1.5 times faster than SlowRunabout, one third of this gap could be filled by generating code for method calls as Runabout does. However, we consider that this feature is not the matter of the library but of the virtual machine. Thus, this would be of benefit to any reflective method call. Moreover, the JVM could perform efficient method sharing, which is not the case for the library.

JMMF penalty also comes from its generality. In JMMF, there are no constraints on the method name nor on the class containing the multi-method implementation. Indeed, there are always at least two arguments in the multi-method, one for the target object (`null` for static methods) and one for the parameter. On the contrary, in Runabout, the name of the visit method is imposed and classes must inherit the `Runabout` class, thus dispatch is simplified. Moreover, we precisely deal with interfaces as parameter types, with static methods and with `null` arguments, whereas Runabout does not treat these cases with full generality.

However, for static methods with one argument JMMF should always be less efficient than Runabout since Runabout only requires one hashtable access where the JMMF version also requires at least one index search in a bit array, a potential disambiguation process and an access to a dispatch table. The forthcoming version of JMMF will modify the treatment of this special case, caching the results of our complete disambiguation process.

Other enhancements could improve special multi-method use cases. However, measures presented in Fig. 5 already show that our general implementation (multiple parameters and interface types support) is practicable.

D. Time analysis

In this section, we provide an informal time complexity analysis of our algorithms.

When all annotations are computed and when bit vectors are represented by integer values, the worst case of dispatch complexity is $\mathcal{O}(p+n)$ where p is the number of parameters and n the number of methods in the multi-method. Indeed, computing $\mathcal{SA}_u$ requires a bitwise `AND` between p annotations of argument types $\mathcal{O}(p)$ and n additional bitwise `AND`'s for ambiguous cases $\mathcal{O}(n)$. However, we do not consider the time to look up annotations.

When annotations have to be computed for a new type, a bitwise `OR` between annotations of all its super-types is required. In the worst case, this process has complexity $\mathcal{O}(c)$ where c is the number of its super-types. However, in practice, all super-types are not visited since the traversal is pruned as soon as an annotated type is found.

Creation time includes computations of annotations for each parameter type and of the matrix $\mathcal{PO}$. In the worst case, for each of the p parameter positions, n different type annotations have to be computed: $\mathcal{O}(p \times n)$. Then, these annotations have to be propagated over all sub-types, leading to the complexity $\mathcal{O}(p^2 \times n^2)$ in the worst case (in practice, the form of the DAG saves many propagations). To compute the n rows of the matrix $\mathcal{PO}$, p bitwise `AND`'s have to be performed. Thus the complexity of this second step is $\mathcal{O}(p \times n)$.

E. Space analysis

In this section we provide a description of the data structures used by our algorithm and we give some measures of JMMF memory usage.

First, all multi-methods share a subset of the type hierarchy which is only used to find super-types. This data structure contains all multi-method parameter types and all argument types of multi-method invocations. It is completed incrementally each time a multi-method is created and when a multi-method is called with a new argument type.

Secondly, as explained in section IV-C, some data structures are common to a "hierarchy" of multi-methods: the matrix $\mathcal{PO}$ and, for each parameter position, a hashtable that associates parameter types and argument class[10] with their annotations $\mathcal{A}_T$.

Finally, for each multi-method receiver class, there is one method dispatch table with possibly several masks, as presented in Fig. 3.

An important part of JMMF memory footprint is due to the systematic loading of parameter types at multi-method creation time, whereas `instanceof`-based implementations only load these types when the test is performed at invocation time. However, for applications like parsers, which potentially use every object class or interface, these two approaches are comparable.

Another important part of the memory usage corresponds to method objects that are stored in the dispatch table. Fig. 6 presents the evolution of the JMMF memory usage (for the multi-method used in section V-A.2) according to the number of methods (and then to the number of parameter types). It also gives the size taken by the dispatch table. It appears that memory usage increases linearly with the number of methods and the number of types that have to be stored. From this linearity and the values obtained for 350 methods, we could estimate the size required by a method as 350 bytes, among which 200 bytes are used for the dispatch table.

More formally, given a multi-method, the number of parameter types present in the type hierarchy is at worst $n \times p$. For each of them, several pieces of information have to be stored: a bit matrix for annotations $\mathcal{A}$ and a bulk of data of constant size. When bit vectors are represented by integer values their size is in $\mathcal{O}(1)$. Then, $\mathcal{A}_T$ is implemented in $\mathcal{O}(p)$ and, for the same reason, $\mathcal{PO}$ is usually in $\mathcal{O}(n)$. Moreover, the size of the dispatch table is in $\mathcal{O}(n)$. Finally, the hierarchy requires a size

[10]The hashtable does not contain intermediate types.

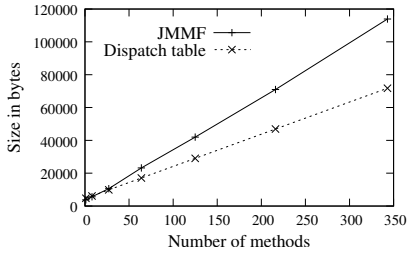


Fig. 6. Memory usage when the number of methods increases

that depends on its depth and on its "degree of super-typing" (average number of super-type for a given type). In Fig. 6, as p is constant, we obtain a complexity of $\mathcal{O}(n)$.

VI. Related works

Many research works have been performed on multi-methods since they were introduced in CLOS [1]. Some of them concern their type checking [3], [6], [23], [24] but others, like ours, focus on implementation. These techniques, like the simple dispatch ones, are divided into different categories: table-based [18], [25]–[27], graph-based [28]–[30] or cache-based [31]. In this section we will focus on works concerning the introduction of multi-methods in Java. We will present these works in chronological order.

Boyland and Castagna [4] first proposed to extend Java with parasitic methods which provide some special form of late-binding on all parameters. This extension is not as general as multi-method but is very attractive since it is comparable to multi-methods in most practical cases. Moreover, it allows strong type checking and multi-method inheritance, preserves modularity and separate compilation, and is conservative (it has no effect on existing Java programs). Parasitic methods are introduced by adding the keyword `parasitic` to the Java syntax. Classes using this extension are translated into standard Java code. Contrarily to the present work, method selection according to the dynamic type of object is not related to the type order but to the textual/inheritance order of parasitic method. This allows a simple and very efficient translation into `instanceof` statements.

Clifton, Leavens, Chambers and Millstein [7] proposed a conservative extension of Java to support multi-method dispatch, called MultiJava. MultiJava introduces syntactic modifications to offer multi-methods. The MultiJava compiler is then in charge of type-checking and of producing the corresponding Java code based on cascaded `instanceof` statements. The main feature of this extension is its ability to perform modular safe type-checking of multi-methods. To do this, they impose strong restrictions on parameter types and method implementations. This approach is completely opposed to ours, since we do not perform any static type-checking but we allow maximum flexibility in implementation.

Dutchyn *et al.* [8] proposed a very efficient implementation of multi-methods for Java, modifying the virtual machine. This extension is conservative, since multi-dispatch is solely applied to methods of classes implementing the interface `MultiDispatchable`. Contrarily to previous works and similarly to ours, this implementation proposes loose type checking (with warnings) for multi-methods; exceptions are thrown at runtime in case of type-checking error. One of the proposed implementations is based on the SRP technique [32] to provide the most specific method. This work was unknown at the moment of the JMMF development, but the technique is comparable to the one presented in this paper. Nevertheless, method disambiguation requires "virtual" methods to be added and methods to be sorted in order to ensure that in case of ambiguity the most specific method always exists and is the one with smallest index. Our implementation does not have these requirements but imposes extra bitwise `AND` on $\mathcal{PO}$ rows to perform disambiguation. Moreover, as shown in section IV-C, since no extra methods nor order are needed on methods, annotations and matrix $\mathcal{PO}$ can be shared in case of inheritance. In order to quantify their respective advantages, implementation of these two techniques in a common framework is still required.

All these works mainly differ from ours since they do not address the implementation problem of dynamic class loading nor the data structure sharing in presence of multi-method inheritance.

A last work from Grothoff [22] has an approach very similar to ours to introduce argument filtering. However, it only provides late-binding on a single argument and does not fully treat parameters with interface types. This work is based on a previous work of Palsberg [33] which proposes a reflective implementation of the visitor design pattern. It is completed with an efficient implementation of reflective method calls using code generation.

VII. Conclusion

This paper presents a Java framework that provides the programmer with multi-methods. Our implementation is a customizable *pure* Java optional package. It does not involve any JVM patch nor extra keywords to the original language definition. Thus, it is fully portable and easy to use: it suffices to add the package in the classpath. Compared to other research works on multi-methods that address typing issues [4], [7], [8], [23], [24], ours focuses on the simplicity of design, use and implementation rather than on static type checking.

Our new simple multi-method dispatch algorithm, presented in this paper together with its implementation, appears to be practicable. Furthermore, it addresses general design concerns that are common to any reflection-based application: it incrementally shares data structures according to object-oriented principles and finds a good balance between space, creation time and invocation time.

This work provides the programmer with an easy way to design and maintain component-based software applications. In particular, multi-methods simply allow algorithms on recursive data structures [13], such as trees, to be specified outside the class defining the data structure. For instance, JMMF is intensively used in the project SmartTools [34] that aims at providing generic tools for compiler constructions and programming environments.

REFERENCES

[1] L. G. DeMichiel and R. P. Gabriel, "The Common Lisp Object System: An overview," in *ECOOP'87 Proceedings*, ser. LNCS. Paris, France: Springer, June 1987, pp. 151–170.

[2] G. Kiczales, J. D. Rivieres, and D. Bobrow, *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[3] C. Chambers, "Object-oriented multi-methods in Cecil," in *ECOOP'92 proceedings*, ser. LNCS. Utrecht, The Netherlands: Springer, July 1992.

[4] J. Boyland and G. Castagna, "Parasitic methods: An implementation of multi-methods for Java," in *OOPSLA'97*, ser. SIGPLAN Notices, no. 32–10. Atlanta, Georgia: ACM Press, Oct. 1997, pp. 66–76.

[5] M. Kizub, "Kiev language specification," July 1998, an extension of Java language that inherits Pizza features and provides multi-methods (`http://forestro.com/kiev/`).

[6] T. Millstein and C. Chambers, "Modular statically typed multimethods," in *ECOOP'99 proceedings*, ser. LNCS, no. 1628, Lisbon, Portugal, June 1999, pp. 279–303.

[7] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein, "MultiJava: Modular open classes and symmetric multiple dispatch," in *OOPSLA'00 proceedings*, ser. ACM SIGPLAN Notices, Minneapolis, USA, Oct. 2000.

[8] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, and W. Holst, "Multi-dispatch in the Java Virtual Machine design and implementation," in *COOTS'01 proceedings*, San Antonio, USA, Jan. 2001.

[9] G. Baumgartner, M. Jansche, and K. Lufer, "Half & half: Multiple dispatch and retroactive abstraction for Java," Dept. of Computer and Information Science, Ohio State University, Tech. Rep. OSU-CISRC-5/01-TR08, Mar. 2002.

[10] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, and B. Pierce, "On binary methods," *Theory and Practice of Object Systems*, vol. 1, no. 3, pp. 221–242, 1996.

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

[12] K. J. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[13] R. Forax and G. Roussel, "Recursive types and pattern-matching in Java," in *GCSE'99 proceedings*, ser. LNCS, no. 1799, Erfurt, Germany, Sept. 1999.

[14] M. Viroli and A. Natali, "Parametric polymorphism in java: an approach to translation based on reflective features," in *Proceedings of OOPSLA'00*, Minneapolis, Minnesota, United States, 2000, pp. 146 – 165.

[15] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin, "JAC: A flexible solution for aspect-oriented programming in java," in *Proceedings of Reflection'01*, ser. LNCS, no. 2192. Kyoto, Japan: Springer-Verlag, Sept. 2001.

[16] C. Chambers, "Object-oriented multi-methods in cecil," in *ECOOP'92 proceedings*, vol. 615. Springer-Verlag, 1992, pp. 33–56.

[17] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems," in *Adaptable and extensible component systems*, Grenoble, France, Nov. 2002.

[18] C. Pang, W. Holst, Y. Leontiev, and D. Szafron, "Multi-method dispatch using multiple row displacement," in *ECOOP'99 proceedings*, ser. LNCS. Lisbon, Portugal: Springer, June 1999, pp. 304–328.

[19] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java™ Language Specification – Second Edition*. Addison-Wesley, 2000.

[20] O.-J. Dahl and B. Myrhaug, "Simula implementation guide," NCC, Tech. Rep. S 47, Mar. 1973.

[21] B. Liskov, "Data abstraction and hierarchy," *SIGPLAN Notices*, vol. 23, no. 5, May 1988.

[22] C. Grothoff, "Walkabout revisited: The runabout," in *ECOOP'03 proceedings*, ser. LNCS. Springer, 2003, pp. 103–125.

[23] R. Agrawal, L. DeMichiel, and B. Lindsay, "Static type-checking of multi-methods," in *OOPSLA'91 proceedings*, ser. ACM SIGPLAN, Phoenix Arizona, Oct. 1991, pp. 113–128.

[24] F. Bourdoncle and S. Merz, "Type-checking higher-order polymorphic multi-methods," in *POPL'97 proceedings*, ser. ACM SIGPLAN-SIGACT, Paris, France, Jan. 1997, pp. 302–315.

[25] E. Amiel, O. Gruber, and E. Simon, "Optimizing multi-method dispatch using compressed dispatch tables," in *OOPSLA'94 proceedings*, ser. ACM SIGPLAN Notices, Portland, Oregon, Oct. 1994, pp. 244–258.

[26] E. A. Eric Dujardin and E. Simon, "Fast algorithms for compressed multi-method dispatch table generation," *TOPLAS*, vol. 20, no. 1, pp. 116–165, Jan. 1998.

[27] Y. Zibin and J. Y. Gil, "Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching," in *Proceedings of OOPSLA'02*. ACM Press, 2002, pp. 142–160.

[28] W. Chen, V. Turau, and W. Klas, "Efficient dynamic look-up strategy for multi-methods," in *ECOOP'94 Proceedings*, ser. LNCS. Springer, 1994.

[29] E. Dujardin, "Efficient dispatch of multimethods in constant time using dispatch trees," INRIA, Rapport de recherche 2892, 1996.

[30] R. Forax, E. Duris, and G. Roussel, "Java multi-method framework," in *TOOLS Pacific'00 Proceedings*. Sidney, Australia: IEEE Computer, Nov. 2000.

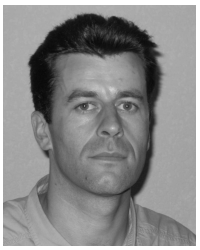
[31] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers, "Vortex an optimizing compiler for object-oriented languages," in *OOPSLA'96 Proceedings*, 1996, pp. 83–110.

[32] W. Holst, D. Szafron, Y. Leontiev, and C. Pang, "Multi-method dispatch using single-receiver projections," Departement of Computer Science, University of Alberta, Edmonton, Alberta, Canada, Tech. Rep. 98-03, 1998.

[33] J. Palsberg and C. B. Jay, "The essence of the visitor pattern," in *COMPSAC'98 proceedings*. IEEE Computer Society, 1998, pp. 9–15.

[34] I. Attali, F. Chalaux, C. Courbis, P. Degenne, A. Fau, and D. Parigot, "SmartTools," June 2000, cooperative project for Interactive Generic Tools (`http://www-sop.inria.fr/oasis/SmartTools/`).

Rémi Forax is Maître de Conférences at University of Marne-la-Vallée since 2003, where he obtained his PhD in 2001 on multi-method implementations in Java. His main research areas concern the use of reflection and of bytecode generation to enhance the Java programming and executing environment. He gives Master's courses at University of Marne-la-Vallée, on object-oriented programming, software engineering and graphical user interfaces. He is co-author of the book Java et Internet.



Etienne Duris is Maître de Conférences at University of Marne-la-Vallée since 2000. He obtained his PhD in 1998 at INRIA and University of Orléans. His research focuses on program transformations and on the use of reflection to extend the expressive power of programming languages. He gives Master's courses at University of Marne-la-Vallée, on object oriented programming and on networks; he also teaches Java at École Polytechnique. He is co-author of the book Java et Internet.

Gilles Roussel is Professor at University of Marne-la-Vallée since 2004. He obtained his PhD in 1994 at INRIA and University of Paris VI, on modularity, genericity and program transformations. His research works cover program transformations, language processing enhancements, genericity and routing. He gives Master and Graduate's courses at University of Marne-la-Vallée, on operating systems and networks; he is also Maître de Conférences à temps partiel at École Polytechnique where he teaches object oriented programming and operating systems. He is co-author of the book Java et Internet.