Java Multi-Method Framework

Rémi Forax, Etienne Duris and Gilles Roussel
Institut Gaspard Monge - Université de Marne-la-Vallée
5 bd Descartes - 77454 Marne-la-Valle Cedex 2 - France
`Firstname.Lastname@univ-mlv.fr`

Abstract

In Java, method resolution is done at runtime, by late-binding, with respect to the dynamic type of the target object. Some object-oriented languages such as CLOS propose, in addition, late-binding according to dynamic types of arguments. This feature is known as multi-polymorphism and usually achieved by multi-methods. In this paper, we propose a pure Java framework that provides multi-methods, without extending the base Java language nor modifying its semantics but intensively using the reflection mechanism of the language. This paper focus on the algorithms and the data structures involved in the method resolution strategy we have implemented in an optional package called Java Multi Method Framework.

Résumé

En Java, la résolution de méthode est effectuée à l'exécution, par liaison tardive, en fonction du type dynamique de l'objet cible. Certains langages orientés objet comme CLOS proposent de plus la liaison tardive en fonction du type dynamique des arguments. Cette caractéristique est conue sous le nom de multi-polymorphisme et est en général obtenue grâce aux multi-méthodes. Dans cet article, nous proposons un environnement pur Java fournissant des multi-méthodes, sans étendre le langage ni modifier sa sémantique mais en utilisant le mécanisme de réflexion de Java. Cet article se concentre sur les algorithmes et les structures de données impliqués dans la stratégie de résolution de méthode que nous avons implémentée sous la forme d'un paquetage optionnel nommé Java Multi Method Framework.

1: Introduction

Component-based software development is now recognized as one of the quicker and cheaper way to produce maintainable applications. This kind of development is strongly linked to object-oriented concepts and especially to encapsulation and locality. Indeed, objects provide a simple and modular access to component functionalities, that corresponds to methods in most object-oriented languages. Moreover, modularity and encapsulation impose that objects contain implementation of the functionalities under their own responsibility. This allows to interchange components that share the same interface with different implementations, facilitating reusability. Then, given a method call on an object, *late binding* mechanism of object-oriented languages dynamically provides a direct access to the right implementation, with respect to the component the object belongs to.

In Java, late binding only concerns the target object (receiver) of a method call, and not its arguments. This is generally sufficient for typical operations whose semantics is related to object state. Nevertheless, for operations that depend on the kind of component or on the relations between objects, late binding on all arguments is sometimes more suitable. This feature is known as *multi-polymorphism* and could be achieved with *multi-methods* [8, 7, 5, 13, 15]. This paper presents a optional package providing Java with multi-methods without extending the core language nor modifying the Java Virtual Machine (JVM) semantics. This *Java Multi-Method Framework* (JMMF) package is a *pure* Java API that uses the reflection mechanism of the language. This choice and the fact that the class hierarchy is dynamically extensible imply a fully dynamic implementation, which differs from other works dealing with multi-polymorphism in Java [5, 13]. In this framework, a multi-method stands as an object representing a set of methods that have same name and same number of arguments. For a given context, a target object and a n-uple of actual parameter types, our *method resolution* provides the corresponding *most specific* method (by default, consistent with standard compile-time method resolution).

Among the advantages of multi-methods [6, 12] we are more concerned with their ability to simplify the specification of algorithms outside the objects they deal with [10, 14, 9]. More precisely, in component based software development, where functionalities have to be added to provided components accessible through interfaces, multi-methods allow to respect an object-oriented style. Indeed, they preserve locality since a method can be specified for each specific parameter type, and provide encapsulation since all these methods are specified in a same class. Encapsulation could also be achieved by successive tests (for instance using an `instanceof` operator) but this solution is not object-oriented and does not preserve the locality of the specification (there is not, for each element kind, one particular method).

After illustrating our framework with an example in section 2, section 3 intuitively presents the whole method resolution for multi-methods. Its two main stages are more precisely described in section 4 and section 5. Before conclusion, section 6 presents some benchmarks of our JMMF implementation.

2: Multi-method use-cases

In order to give an intuitive idea of our multi-method framework, we first illustrate the process of constructing and using a multi-method through a simple example. Next we discuss some design issues related to the use of multi-methods.

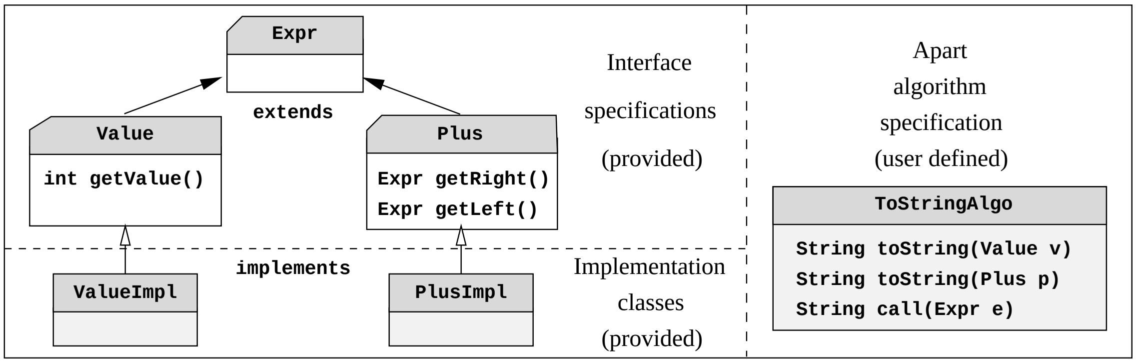

Figure 1. Algorithm user-specified apart from provided interfaces and classes

2.1: Component-based approach

Consider the mini-expression syntax case with integer *values* and a single operation, *plus*. In a component-based object-oriented conception, assume that a third party provides us two classes, `ValueImpl` and `PlusImpl` that respectively implement interfaces `Value` and `Plus`, both inheriting an empty interface `Expr`, as illustrated in figure 1.

In this context, we want to display a string representation of any expression of type `Expr`. Since no method performs this algorithm and due to our component-based approach, we want to respect the two following restrictions: do not change any given interface and do not change nor modify any provided implementation classes. Thanks to our multi-method framework, the following class `ToStringAlgo` allows to specify the algorithm in a single class apart from the existing hierarchy.

```
class ToStringAlgo {
  MultiMethod mm = MultiMethod.create(getClass(), "toString", 1);
  String toString(Value v) { return Integer.toString(v.getValue()); }
  String toString(Plus p) { return call(p.getLeft()) + "+" + call(p.getRight()); }
  String call(Expr e) {
    try { // call the most specific method toString according to the actual type of e
      return (String) mm.invoke(this, new Object[]{e});
    } catch(Exception ex) { return null; }
  }
}
```

In this example, the static method `create` constructs an instance of the class `MultiMethod` that stands for the set of classical Java methods hosted by the class `ToStringAlgo`, having the name `toString` and one parameter. Thus, given an argument expression statically typed `Expr`, an instance of the class `ToStringAlgo` could be used to delegate, to the multi-method mechanism, the resolution of which method `toString` had to be called with respect to the dynamic type of the argument (and recursively on the sub-expressions).

```
  Expr e = new PlusImpl(new ValueImpl(1), new ValueImpl(2));
  System.out.println(new ToStringAlgo().call(e));
```

When the method `invoke` is called, transmitting the argument as an `Object` array, the method resolution mechanism for multi-methods looks for the most specific method `toString` according to the actual type of the argument and, if any, calls it.

2.2: Some design issues

In the previous example, the method resolution for `toString` is carried out by a call to the method `call` that we name the *invocation method*. It is also possible to give the name `toString` to this invocation method, but this requires to pay particular attention. Indeed, there could be a clash between static compile-time method resolution and dynamic multi-method one. To avoid this problem, the argument can be casted into the parameter type of the invocation method to be sure that this method will be chosen by the compiler.

We could also note that the parameter of the method `call` is declared of type `Expr`. A parameter of type `Object` can be used instead, in order to relax static type-checking, allowing to add other methods to this multi-method (for instance by inheritance), even if they are not declared with a parameter subtype of `Expr`.

3: Overview of method resolution for multi-method

The classical method resolution mechanism in Java successively uses compile time and runtime information to select the most specific method. First, the signature (name and declared types of target object and parameters) is statically determined; next, the actual type of the target object (only known at runtime) enables the final selection. As said before, since our framework is pure Java, the method resolution for multi-methods does not involve other compile time mechanisms and hence is fully dynamic.

In order to intuitively expose the problem, consider a call to the invocation method with a single n-uple of arguments whose actual types are dynamically known as being $(T_1, ..., T_n)$. The resolution could then be sketched as follows. Given each position i, find the set S_i of types that are both statically declared as i-th parameter of a method and supertypes of the argument type T_i (cf. section 4.3). With these sets for all positions, find the set M of methods of which signatures appear in $S_1 \times ... \times S_n$. If this set M is empty, type-checking has failed and an exception will be thrown. If a single method matches, this is the one to be invoked. If several methods match and if one is *more specific* than all the others (cf. section 4.7), it could be invoked. Otherwise an exception will be thrown.

Since several different n-uples of arguments could be considered, this costly approach is suboptimal and could be improved by splitting method resolution for multi-method in two stages: *creation time* that corresponds to the execution of the `create` method and *invocation time* corresponding to the execution of, for instance, `mm.invoke(target, argArray)`. At creation time, several computation of the previous algorithm are factorized. For all combination of types in $S_1 \times ... \times S_n$, we compute the corresponding applicable methods and if there exists or not a single one. Then, a data-structure is initialized to cache and speed-up the type hierarchy traversals at invocation time. Furthermore, in conflict cases between several methods, finding if one is more specific than all others could also being factorized. Finally, the data-structure could be completed at invocation time when, using a n-uple of dynamic argument types, we obtain, if it exists, the n-uple of parameter types it *behaves* like. In better cases, creation time computations directly give the most specific method.

4: Pre-computation for method resolution

In this section, we first build the set of *syntactically applicable* methods that gathers the set of classical Java methods on which we will first focus, because they allow us to determine n-uples of types interesting to consider as possible invocation signatures. The chosen structure representing relations between these types also guides and supports the computation of some *annotations*, at creation time. With these annotations, we determine the set of *semantically applicable* methods, that restricts the set of *syntactically applicable* methods to those that are correctly typed with respect to a given n-uple. If several methods are still candidates, a *disambiguation* process could find, if any, the most specific method corresponding to this n-uple of declared types.

4.1: Syntactically applicable methods

As general case in the rest of the paper, we consider the multi-method constructed by:

```
MultiMethod mm = MultiMethod.create(hostClass, "methodName", n);
```

Since we are looking for the set of methods hosted by class `hostClass`, declared with the name `methodName` and with exactly n parameters, we must add to methods declared in `hostClass` those inherited from superclasses and superinterfaces. We call it the set of *syntactically applicable* methods since only the name and the number of parameters match with the required method. In the classical Java method resolution ([11], § 15.12.2.1), the notion of *applicable* methods is used. It means, in addition to our *syntactically applicable* notion, that the type of each argument can be converted to the type of the corresponding parameter and we will need to take care, further, of this – semantic – information. We also need to insure that methods have visibility rights. This is what Java Language Specification (JLS) [11] calls *accessible* methods and it depends on the access modifier of each method. At this step, each syntactically applicable and accessible method only needs type information to become a full candidate to invocation with respect to n given argument types.

4.2: n-uples of argument types

Suppose the previous step has selected p syntactically applicable and accessible methods that we note $\mathcal{M} = \{m_1, ..., m_p\}$. We are interested in their n-uples of parameter types, i.e., each method signature[1]:

$$\mathcal{M} = \{m_1 : \texttt{methodName}(T_{1,1}, ..., T_{1,n}),\ m_2 : ...,\ m_p : \texttt{methodName}(T_{p,1}, ..., T_{p,n})\}$$

For the multi-method, this means that the first argument of an invocation could be of any type in the set $\{T_{1,1}, ..., T_{p,1}\}$, and so forth with each argument until the n-th argument of a type in the set $\{T_{1,n}, ..., T_{p,n}\}$. For each argument position $j \in [1..n]$, we define $\mathcal{T}(j)$ as the set of distinct possible types for the j-th argument of a multi-method invocation:

$$\forall j \in [1..n], \mathcal{T}(j) = \{T_{i,j} \mid i \in [1..p]\}$$

We also define the set $\mathcal{N}$ of all distinct n-uples of *declared types*, build from these sets, and representing possible n-uples of argument types in a multi-method invocation:

$$\mathcal{N} = \{(T_1, ..., T_n) \mid \forall j \in [1..n], T_j \in \mathcal{T}(j)\}$$

We focus our efforts on this set of n-uples, since it allows us to factorize information computations for static (declared) types. We now investigate, for each n-uple of types in $\mathcal{N}$, if there exists not any, one, or several corresponding methods $m_k \in \mathcal{M}$, whose parameter types could *match* with these n-uples.

4.3: Subtyping relations

To determine if a method could be called with such a given n-uple, we have to verify that the value of the i-th element of the n-uple could be converted into the type of the i-th method parameter, and that for all $i \in [1..n]$.

Subtyping relations[2] are of the main features of object oriented languages. They allow any subtype T' of T to be used in place of T. In Java, there are three cases where a value of type T' could be assigned to a variable of type T ([11], § 5.1) and, when any method is invoked in Java, one of these conversions is necessarily applied to each argument: *identity conversions* in cases where $T = T'$; *widening primitive conversions*, for instance, a value of type `short` could be assigned to a variable of type `int`; *widening reference conversions* provided by (explicit or implicit) inheritance, interface implementation and some other cases specific to Java (e.g., with arrays, as shown in figure 2).

Since we will have to compare types to each other and in order to ease the programmer to deal with subtyping relations, the JMMF package we propose provides a method `getSupertypes(`T`)` that returns the set of all *direct* supertypes of T.

4.4: Directed acyclic graph

To store and deal with these relations between types, we use a graph $\mathcal{G}$ where vertices represent types and edges represent subtyping relations. First, this graph $\mathcal{G}$ is oriented such that an edge from T to T' means that T "*is a supertype of*" T', in

[1]We do not take return types into account because they are not involved in the Java method resolution.
[2]We do not make any distinction here between the notions of *subtyping* and *assignation*.

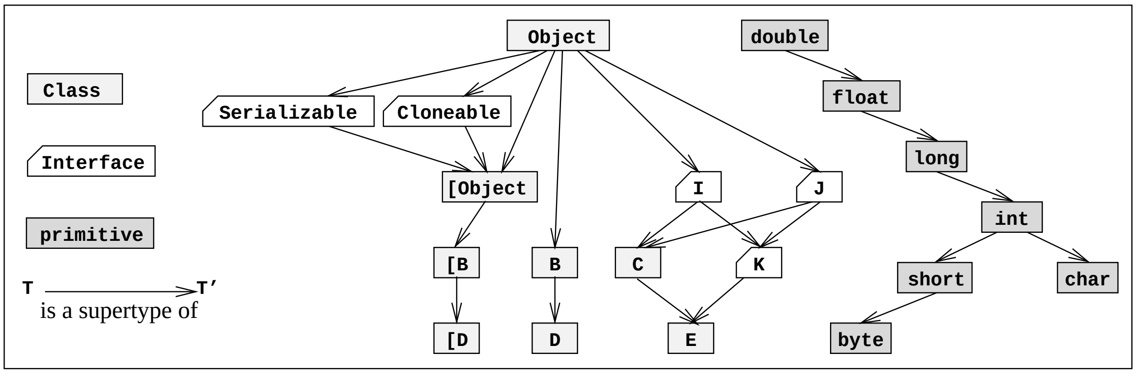

Figure 2. Classes, interfaces and primitive types hierarchies

the sense previously described of subtyping. We will note $T \to T'$ this relation when $T \neq T'$. Next, from the essence of subtyping, this graph is acyclic. It is not a tree because of the multiple supertyping allowed by Java features like, for instance, the ability of an interface to extend more than one interface. Thus, we use a structure of directed acyclic graph (DAG) for $\mathcal{G}$ and we classically note $T \to^* T'$ if there is a path by $\to$ from T to T' or if $T = T'$.

Figure 2 gives a hierarchy example of Java classes, interfaces and primitive types, which is very close to our expected DAG $\mathcal{G}$. Note that, as our DAG will, this figure does not distinguishes between *extends*, *implements* or other subtyping (assignation) relations.

Given a multi-method, the corresponding DAG is constructed by adding recursively all the types that appear as a parameter of its methods, together with all their supertypes, obtained by the `getSupertypes` method, until reaching the fix point. Termination is insured by the types `Object` and `double` that are the roots of the subtyping hierarchies.

4.5: Annotate the DAG

Pursuing our aim of associating methods (signatures) to n-uples of types, we now want to annotate each vertex of a multi-method DAG by its ability to be an acceptable argument type for methods represented by the multi-method. This annotation must be done for each method and at each parameter position. Given a type T, we then compute the set of methods m_i, noted $\mathcal{A}(T, j)$ for which T is able to stand for the type of the j-th argument in a call to the method m_i. Once build, this set is added to $\mathcal{A}(T', j)$ for every subtype T' of T. The fact that our graph is a DAG provides that this propagation terminates.

To illustrate our purpose, we choose a guiding example that exploit the (particularly intricate) type hierarchy of figure 2, and consider the multi-method defined as follows:

```
MultiMethod myMM=MultiMethod.create(MyHostClass.class,"myMethod",3);
```

where accessible and syntactically applicable methods `myMethod` hosted in `MyHost-`

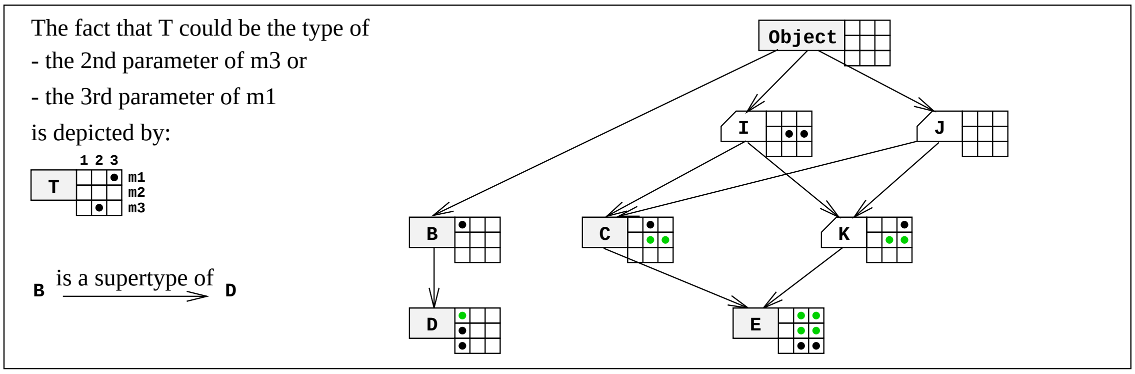

Figure 3. An example of annotated DAG

`Class` and having exactly three parameters are defined with the following signatures:

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

The annotated DAG obtained for this multi-method is presented in figure 3. A vertex annotation is figured as a matrix of p rows and n columns where (i, j) contains a bullet[3] in order to represent the fact that the corresponding vertex is able to be the type of the j-th argument in the method m_i. In other words, $\mathcal{A}(T, j)$ is the j-th column of the matrix.

4.6: Semantically applicable methods

These annotations tell us if a single given type is acceptable at a given position of a given method. We now want to deduce, for a given n-uple of types in $\mathcal{N}$, if it corresponds not any, one and only one, or multiple acceptable methods. Thus, for a given n-uple u in $\mathcal{N}$, we note $SemApp(u)$ the set of methods for which an invocation conversion would be able to accept u as argument types. We say that methods in $SemApp(u)$ are *semantically applicable* for u, and define them as follows:

$$\forall u = (T_1, ..., T_n) \in \mathcal{N}, SemApp(u) = \cap_{j \in [1..n]} \mathcal{A}(T_j, j)$$

The cardinality of such a set gives us important information, summarize in three cases:

$SemApp(u)$	Card	Meaning	At invocation time
$\emptyset$	0	Not any method matches	`NoSuchMethodException`
$\{m_k\}$	1	Only one method matches	Invocation of m_k
$\{m_{k_1}, ..., m_{k_q}\}$	$q > 1$	Multiple methods match	`MultipleMethodException` or, if disambiguation succeed, invocation of the most specific method

To illustrate these different situations, we go back to our example of multi-method `myMM` (cf. section 4.5 and figure 3) and consider all combination of n-uples in $\mathcal{N}$, i.e., with either `B` or `D` as first argument type, `C`, `I` or `E` as second, and `K`, `I` or `E` as the

[3]Dark bullets stands for set annotations (types declared as parameter) and light for propagated ones.

With `m1 : (B,C,K)` `m2 : (D,I,I)` and `m3 : (D,E,E)`	`B C K` $\downarrow\ \uparrow\ \uparrow$ `D I I` — `m1` and `m2` are not comparable	`B C K` $\downarrow\ \downarrow\ \downarrow$ `D E E` — `m3` is more precise than `m1`	`D I I` $\ \ \downarrow\ \downarrow$ `D E E` — `m3` is more precise than `m2`

Figure 4. Using partial orders to resolve ambiguities

third argument type. In this case, $\mathcal{N}$ contains 18 distinct n-uples but, in the worst general case, n^p could exist.

We now exhibit some illustrating examples. Let u_1 be the n-uple (B, C, I). It follows:

$$SemApp(u_1) = \mathcal{A}(B, 1) \cap \mathcal{A}(C, 2) \cap \mathcal{A}(I, 3) = \{m_1\} \cap \{m_1, m_2\} \cap \{m_2\} = \emptyset$$

Calling `invoke` on `myMM` with an object array `b,c,i` of types `B`, `C` and `I`, e.g., `myMM.invoke (target, new Object[]{b,c,i})` will throw a `NoSuchMethodException` exception.

If $u_2 = (D, C, I)$, the same principle gives $SemApp(u_2) = \{m_2\}$. Thus, `myMM.invoke(target, new Object[]{d,c,i})` implies the invocation of m_2 which is the most specific method according to these argument types.

Let now u_3 be the n-uple (D, C, K). It follows that $SemApp(u_3) = \{m_1, m_2\}$. Here, the execution of `myMM.invoke(target, new Object[]{d,c,k})` could either lead to the invocation of m_1 or the invocation of m_2 or else throw a `MultipleMethodsException`. In fact, we do not have enough information to decide here what will happen and say there is an *ambiguity*. More precisely, there are different cases of ambiguity but in the case of u_3, we cannot say that m_1 is more specific than m_2 nor that m_2 is more specific than m_1.

4.7: Disambiguation process

As in the classical method resolution, when at least two methods are semantically applicable, it is sometimes possible to elect one because it is more specific than all the others. For instance, if $u_4 = (D, E, E)$, $SemApp(u_4) = \{m_1, m_2, m_3\}$ and the ambiguity concerns three methods. Nevertheless, we can resolve here the ambiguity and argue that m_3 is more specific than m_1 and m_2, and actually elect m_3: we call this process the *disambiguation*.

Intuitively, we see that u_4 is exactly the n-uple declared as m_3 parameter types, but this is not the right (only) justification. In fact, m_3 is more specific than m_1 because any parameter type of m_3 is more precise than (or at least as precise as) the corresponding parameter type of m_1. This notion of precision comes from the subtyping relations and, for any subtype T' of a given type T, we said that T' is *more precise* than T.

More formally, we say that a method m_k is *more precise* than another method m_l, and note $m_k \prec m_l$, if any parameter of m_k is a subtype[4] of the corresponding parameter of m_l:

[4] Either a subtype or this type itself.

$$m_k : (T_{k,1}, ..., T_{k,n}) \prec m_l : (T_{l,1}, ..., T_{l,n}) \Leftrightarrow \forall j \in [1..n], T_{l,j} \rightarrow^* T_{k,j}$$

If we cannot assert that $m_k \prec m_l$ nor $m_l \prec m_k$, we say that m_k and m_l are *not comparable*. Indeed, this relation possibly induces several partial orders among all syntactically applicable and visible methods (cf. figure 4 for methods m_1, m_2 and m_3 of our example).

4.8: Summary and implementation issues

At this point, for a multi-method definition, we dispose of a process that defines a function *Resolve* which provides, given a n-uple of $\mathcal{N}$, one of the following diagnostic: *no* method fits, *one* method is the most specific method or *multiple* methods fit without a most specific one. In our current implementation, computations corresponding to *Resolve* are completely precomputed at creation time, and tabulated (cached) in order to be immediately available at invocation time (see section 5.2). Another implementation choice consists in computing *Resolve* "on the fly". The former case allows the annotation structures to be freed (and also clarify the forthcoming presentation of dynamic computations) while the latter computes, from the annotation structures, the values of *Resolve* when they are required. Nevertheless, deciding which (implementation) solution is more efficient is an open problem that we will not discuss here.

5: Multi-method invocation process

Information provided by *Resolve* is not sufficient to resolve all invocations of the multi-method. Indeed, the `invoke` method could be called with any n-uple of argument types that are not necessarily considered in $\mathcal{N}$. We have to be able to associate any invocation n-uple of argument types to a n-uple in $\mathcal{N}$, especially for types that are only known at invocation time[5] and that not yet appear in the DAG; this implies some dynamic part in the process.

In fact, we only need to associate, to each encountered argument type, the type in the DAG it *behaves* like. This can be done without taking care of compatibility between types in the n-uple of arguments, since this information (type combination) is provided by *Resolve*.

To guide the whole process, we reuse the DAG of the previous section but, since we have the function *Resolve*, we could waste all previous annotations $\mathcal{A}$. In place of them, we supplement the basic DAG with two kinds of information: subtyping relations for any newly discovered types (vertices and edges) and *behavioral annotations* for any type at any position. We call *behavioral annotation* of a type T at the position i, noted $\mathcal{B}(T, i)$, the *closest* type of T in $\mathcal{T}(i)$, to be used instead of T. For a given vertex T and a given position i, the behavioral annotation $\mathcal{B}(T, i)$ can take

[5] Recall that some of these *dynamic* types could even not been loaded before the invocation time.

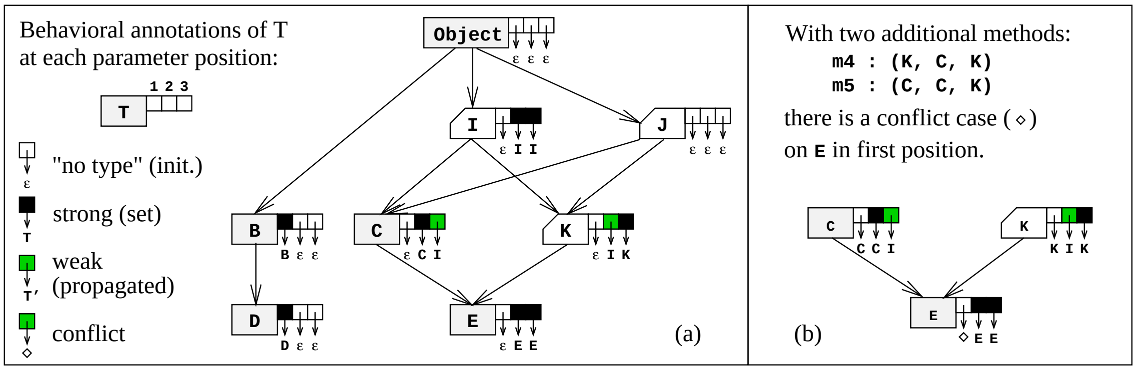

Figure 5. Behavioral annotations of the DAG

three kinds of values: the initial value ε meaning "no type", a type T' in $\mathcal{T}(i)$ or the value $\Diamond$ that represents an insolvable conflict.

Despite unavoidable "dynamic" part of the process, the major part of the behavioral DAG (i.e., the behavioral annotation of the basic DAG) could be computed at creation time. At invocation time, there will only remain to supplement the DAG with new types dynamically encountered and to behaviorally annotate them.

5.1: Creation time annotations

In the behavioral annotation of the basic DAG, we distinguish two kinds of annotations: *strong* and *weak*. A *strong* annotation $\mathcal{B}(T, j) = T$ is set when T is the type of the j-th parameter in the declaration of a given m_i. This means that T at the position j will always play the role of (behave as) itself. Now, any subtype T' of this T could possibly play the role of (behave as) T at the position j in an invocation of the same method m_i. This is why we propagate the *weak* annotation $\mathcal{B}(T', j) = T$.

Our algorithm implements several other rules that we intuitively describe here. First, any type propagated as annotation on a vertex prevails over ε on this vertex. Next, when an annotation b_1 is propagated on a vertex v that already has an annotation b_2, the most specific between b_1 and b_2 prevails and must be annotated on v but, in particular, a strong annotation prevails over a weak one. If an annotation changes on v, its propagation must be pursued on subtypes of v; else, the propagation is stopped along this particular subtyping branch in the DAG. In the case where two weak annotations b_1 and b_2 are not comparable, there is a *conflict*: the $\Diamond$ must be annoted on v, and propagated on v's subtypes. This $\Diamond$ annotation prevails over any weak annotation.

Figure 5 (a) shows our DAG example annotated by this algorithm. This example does not introduce any conflict case but, to illustrate it, suppose that $\mathcal{M}$ contains two supplementary methods, $m_4 : (K, C, K)$ and $m_5 : (C, C, K)$; then, the corresponding part of the annotated DAG would have been those presented in figure 5 (b).

We are now able to determine, for a given n-uple arbitrarily chosen among vertices of the creation time DAG, to which n-uple in $\mathcal{N}$ it corresponds, if any. But, in an

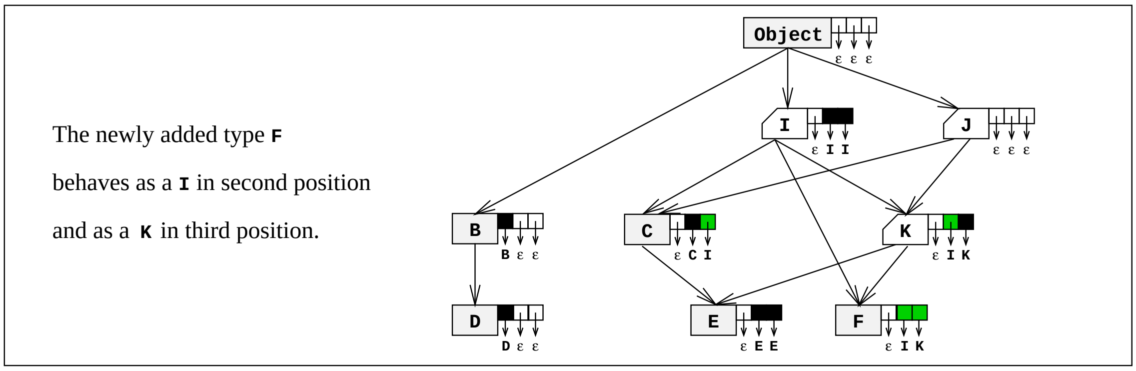

Figure 6. Dynamically supplement behavioral DAG

argument type n-uple of a multi-method invocation, if we encounter a type that does not yet appear in the graph, we also have to be able to associate it to a n-uple in $\mathcal{N}$, if any exists.

5.2: From dynamic to declared types

Let us now consider the statement `mm.invoke(target, new Object[]{d,c,f})` where `f` is of a new type `F`, a class implementing both interfaces `I` and `K`. We have to supplement the initial DAG with F and determine its behavioral annotations from those of its supertypes. To add a new vertex to the DAG we reuse the creation time algorithm described in section 4.4. To propagate behavioral annotations to the new vertex, rules are similar to those of the section 5.1. In fact, at this stage, these two algorithms are merged together: whereas the former recursively supplements the DAG with new vertices, bottom-up in the class hierarchy, the latter propagates behavioral annotations in the reverse order, top-down. At the termination condition, i.e., when a existing vertex in the DAG is found, instead of propagating its behavioral annotations to each of its subtypes, it suffices to only propagate them along the newly created edge.

Figure 6 shows information computed at invocation time for this example. First, a vertex is added for the type F with two edges from I and K. Then, the annotation propagation provides that F behaves as ε in first position and as I in second position. For the third position, between the two propagated annotations I and K, K prevails because, in our example, it is a subtype of I.

At this point, we could deduce that from the point of view of method resolution, invoking the multi-method with argument types (D, C, F) is equivalent to invoke it with types (D, C, K). Finally, function *Resolve* provides us the most specific method, if any, corresponding to this n-uple of $\mathcal{N}$ (in this case, a `MultipleMethodException` will be thrown).

Thus, thanks to the behavioral DAG and the function *Resolve*, and given any n-uple of dynamic argument types of a multi-method invocation, we are able to determine information allowing either the corresponding most specific method to be called or the right exception to be thrown.

6: Implementation benchmarks

In order to prove the usability of our implementation framework JMMF[6], we present here some execution times. The table below shows, for two examples, the creation time and invocation time costs with our multi-method framework, and the cost of the "equivalent" `instanceof`-based hand-written program. The first execution times come from our running example of multi-method `myMM` (cf. figure 6), with argument types in `B`, `C`, `D`, `E` and `F`. Second example simply chooses, given three arguments, between three methods of signatures `(T1,T1,T1)`, `(T2,T2,T2)` and `(T3,T3,T3)` (these types do not have any subtyping relation). Given times are average results over one million calls (with random arguments for which a most specific method exists) of programs compiled and run with the JDK1.3 on a 32Mb PC Celeron 300Mhz under Windows 98.

Implementation	`myMM`'s example	`T1`, `T2`, `T3` example
Creation of the multi-method	1.02733 ms	0.85500 ms
1 multi-method-based invocation	0.00973 ms	0.00988 ms
1 `instanceof`-based invocation	0.00227 ms	0.00511 ms

As first observation, the time required to create a multi-method is about hundred times longer than to invoke it. Thus, it is only worthwhile to create a multi-method if one expects to call it often. Nevertheless, all the work done at multi-method creation time should be considered as compile-time computations, and hence, must be compared to the complex design or the error prone evolution and maintenance of the `instanceof`-based version.

Secondly, the times required by one multi-method invocation are equivalent for both examples but, depending on the example, two or five times slower than one `instanceof`-based invocation. We explain the relative velocity of the `myMM`'s `instanceof`-based invocation by the fact that, in this example, all parameter types at a given position could be totally ordered (this is not the case in the second example). This particularity allows to simplify the nested if-then-else statements in the `instanceof` hand-written program.

Finally, multi-method invocation times could still be improved by implementation tuning (by avoiding creations of some temporary objects and using compression techniques [2, 16]).

7: Conclusion

This paper presents a Java framework that provides the programmer with multi-methods. The main feature of our implementation is to be a customizable *pure* Java

[6] This framework and some examples are available at `http://igm.univ-mlv.fr/~forax/works/jmmf`.

optional package. It does not involve any JVM patch nor extra keyword to the original language definition. Our approach intensively uses the Java reflection mechanism rather than dynamically producing parts of code (generation of `instanceof` tests). The relative efficiency and the usability of JMMF validate our choice of using reflection. With respect to other research works on multi-method [1, 4, 5, 15] that address typing issues, ours focus on implementation efficiency and on the simplicity of design and use rather than on static type checking.

The most important issue of this framework is that it provides the programmer with an easy way to design and maintain algorithm specification. In particular, multi-methods simply allow to externally specify algorithm on recursive data-structures [9] such as trees. For instance, JMMF is intensively used in the project SmartTools [3] that aims at providing generic tools for compiler constructions and programming environments.

References

[1] Rakesh Agrawal, Linda DeMichiel, and Bruce Lindsay. Static type-checking of multi-methods. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, ACM SIGPLAN, pages 113–128, Phoenix Arizona, October 1991.

[2] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'94)*, ACM SIGPLAN Notices, pages 244–258, October 1994.

[3] Isabelle Attali, Franck Chalaux, Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. SmartTools. Cooperative project for Interactive Generic Tools (`http://www-sop.inria.fr/oasis/SmartTools/`), June 2000.

[4] François Bourdoncle and Stephan Merz. Type-checking higher-order polymorphic multi-methods. In *Proceedings of Principles Of Programming Languages (POPL'97)*, ACM SIGPLAN-SIGACT, pages 302–315, Paris, France, January 1997.

[5] John Boyland and Giuseppe Castagna. Parasitic methods: An implementation of multi-methods for Java. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, number 32–10 in SIGPLAN Notices, pages 66–76, New York, October 1997. ACM Press.

[6] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[7] Craig Chambers. Object-oriented multi-methods in Cecil. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'92)*, Utrecht, The Netherlands, July 1992.

[8] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An overview. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'87)*, pages 151–170, Paris, France, June 1987.

[9] Rémi Forax and Gilles Roussel. Recursive types and pattern-matching in Java. In *Proceedings of International Symposium on Generative and Component-based Software Engineering (GCSE'99)*, number 1799 in LNCS, Erfurt, Germany, September 1999. LNCS.

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

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification – Second Edition*. Addison-Wesley, 2000.

[12] Gregor Kiczales, Jim Des Rivieres, and Daniel Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

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

[14] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[15] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 99)*, number 1628 in LNCS, pages 279–303, Lisbon, Portugal, June 1999.

[16] Candy Pang, Wade Holst, Yuri Leontiev, and Duane Szafron. Multi-method dispatch using multiple row displacement. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'99)*, pages 304–328, Lisbon, Portugal, June 1999.