A Simple Dispatch Technique for Pure Java Multi-Methods

Rémi Forax, Etienne Duris, and Gilles Roussel

Université de Marne-la-Vallée, Institut Gaspard Monge
5, boulevard Descartes - 77454 Marne-la-Vallée Cedex 2, France
{forax, duris, roussel}@univ-mlv.fr

Abstract. In Java, method dispatch is done at runtime, by late-binding, with respect to the dynamic type of the only receiver object. Allowing late-binding on dynamic type of all arguments is usually considered desirable to simplify many programming designs and is known as multi-polymorphism. To achieve this feature, several recent research works attempt to provide Java with multi-methods. In contrast to these works that either extends the language or modify the virtual machine, our approach proposes a pure Java framework that intensively uses the reflection mechanism of the language.
Presenting this approach, this paper focuses on a new simple and efficient multi-method dispatch technique, implemented in the Java Multi-Method Framework (JMMF) optional package. We also discuss visibility and inheritance implications for this implementation of multi-methods.

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, 6, 4, 16, 18, 7, 9].

This paper presents a simple implementation of multi-methods in an optional package that does not extends the core language nor modifies the Java Virtual Machine (JVM) semantics. This *Java Multi-Method Framework* (JMMF) package is a *pure* Java API that intensively uses the reflection mechanism of the language. This choice and the fact that the class hierarchy is dynamically extensible require a fully dynamic implementation. 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. In this paper, we focus on a new simple and efficient algorithm to find this most specific method, that is more efficient than [10] and surprisingly simple.

Among the advantages of multi-methods [5, 15] we are more concerned with their ability to simplify the specification of algorithms outside the objects they deal with [12, 17, 11]. 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. Then, section 6 explores multi-method inheritance issues in this framework. Before conclusion, section 7 presents some special issues concerning multi-methods and section 8 presents some related works.

Most properties of algorithms presented in this paper are formally proved in section 4,5 and 6, but proofs may be forgotten by convinced readers.

2 Multi-method use-cases

In order to give an intuitive idea of the JMMF, we illustrate the process of constructing and using a multi-method through a simple example.

2.1 Component-based approach

Consider the following XML representation of this paper (`paper.xml`):

```
<paper>
 <title>A Simple Dispatch Technique for Pure Java Multi-Methods</title>
 <abstract>
  In Java, method dispatch at runtime, by late binding ...
 </abstract>
```

```
  <section name="Introduction">
   <paragraph>
    Component-based software development ...
   </paragraph>
  </section>
  ...
</paper>
```

In this context, we want to count the number of `Element` in the XML document. To do this, we specify a Java program that manipulates XML documents with Document Object Model (`DOM`) data-structures.

DOM is a standard set of interfaces (cf. figure 1) defined by the `W3C` that allow different vendor implementations of XML parsers. DOM is used as an interface to the proprietary data structures and API. In our example we use the `Xerces Parser` from the `Apache Project`.

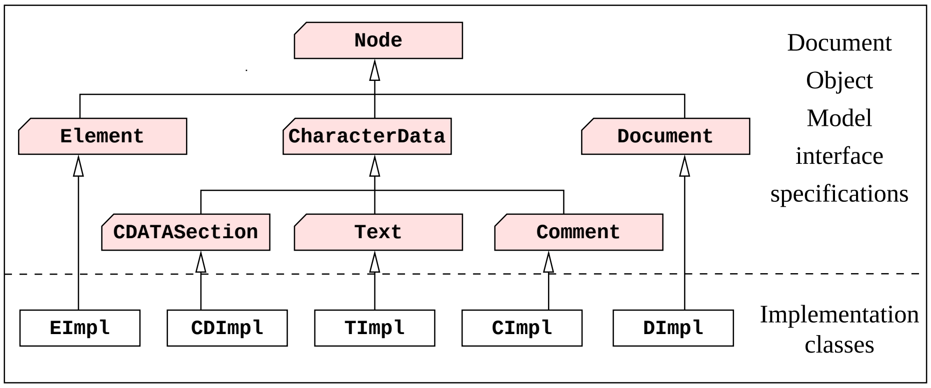

Fig. 1. Document Object Model and a specific implementation

Since, in the DOM API, no method counting the number of elements is available and since we want to respect the component-based approach of DOM, we can not change interfaces and we do not want to directly access or extend implementation classes.

Thanks to the JMMF multi-method framework, the following class `DOMElementCount` allows the algorithm to be specified in a single class, apart from the DOM hierarchy.

```
import org.w3c.dom.*;

class DOMElementCount {
  MultiMethod mm = MultiMethod.create(getClass(), "traverse", 1);
  int traverse(Element element) {
```

```
    int count=1;
    if (element.hasChildNodes()) {
      NodeList children=element.getChildNodes();
      for(int i=0; i<children.getLength(); i++)
        count+=call(children.item(i));
    }
    return count;
  }
  int traverse(CharacterData data) { return 0; }
  int traverse(Document doc) { return call(doc.getDocumentElement()); }
  int call(Node node) {
    try {
    // call the most specific method traverse according to node type
      return ((Integer)mm.invoke(this, new Object[]{node})).intValue();
    } catch(Exception ex) { return 0; }
  }
}
```

In this example, the static method `create` constructs an instance of class `MultiMethod` that stands for the set of classical Java methods hosted by the class `DOMElementCount`, having the name `traverse` and exactly one parameter.

Then, an invocation of method `call` with an expression statically typed `Node` as argument, the instance `mm` of class `MultiMethod`, field of `DOMElementCount`, will perform the dispatch of method `traverse` with respect to the dynamic concrete type of the `Node` argument.

```
  import org.apache.xerces.parsers.DOMParser;
  ...
  DOMParser parser = new DOMParser();
  parser.parse("paper.xml");
  org.w3c.dom.Node root=parser.getDocument();

  DOMElementCount counter=new DOMElementCount();
  System.out.println(counter.call(root));
```

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 `traverse` according to the actual type of the argument and, if any, invokes it.

2.2 Some design issues

In the previous example, the method resolution for `traverse` is carried out by a call to the method `call` that we name the *invocation method*. It is also possible to give the name `traverse` 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 `Node`. 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 `Node`.

Restricting the type of the `call` parameter is the only way to support static type checking in our framework. All other type checks are performed dynamically, in particular in this example the fact that `traverse` returns an `int` is only verified at run-time by the cast into `Integer`[1].

3 Overview of method resolution for multi-method

Basically, the algorithm we propose for multi-method dispatching consists in two main steps. The first one is processed at the multi-method *creation time*, that is when the JVM processes the method `create` of the class `MultiMethod`. This step performs reflection-based analysis and computes several data-structures that will be used anywhere a dispatch is necessary for an invocation of this multi-method. The second step of our algorithm is processed at *invocation time*, that is when the JVM processes the method `invoke` of an instance of `MultiMethod`. Based on the data structure 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 one, possibly requiring a disambiguation process.

These two stages are described in details in the next two sections.

4 Data structure construction at creation time

4.1 Syntactically applicable methods

In the first part of this paper (section 4, 5 and 7), we consider as general case the multi-method constructed by:

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

where `hostClass` is a concrete class in which all methods of name `methodName` with `n` arguments are public and non static. We also suppose that none of the considered methods is overrides other method. These restrictive choices are made in a first time, in order to simplify explanations, but we will deal with cases that relax these constraints in sections 7 and 6.

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 are matching with the required method. In the classical Java method resolution ([13] § 15.12.2.1), the notion of *applicable*

[1] As the `invoke` method of `java.lang.reflect.Method` do.

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 but it is the case at first sight, since we've assumed that all methods are public. We will deal with access modifier in relation with inheritance in section 7. 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.

The set of method signatures that are considered by the multi-method is formally represented by a set $\mathcal{M}$, that is sometimes called the *behavior* of the multi-method. We do not take into account the return types and thrown exception since Java does not use these information to determine the method to invoke.

Definition 1 (Methods and parameter types).
Given a multi-method defined by a host class, a name and a number of arguments, the set of methods it represents is the set of methods with the right name and the right number of parameter and that are accessible in the host class. This set is represented as follows:

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

where, as in the rest of this paper, the notation $T_{i,j}$ is used to generically identify the type of the i-th parameter of the method m_j.

This set is arbitrarily indexed by integers from 1 to p. These indexes uniquely identify each method (signature) since we've assumed that, at first time, no overriding between methods arises. As a corollary, note that if m_j and m_k have exactly the same signature then j equals k.

In the later, for sake of simplicity, we will identify a method with its signature, since we have supposed a one to one association. From the implementation point of view, we only assume that a table allows us to associate each signature in $\mathcal{M}$ (a given m_j, $j \in [1..p]$) to the corresponding method object of class `java.lang.reflect.Method`. In fact, this association could be done when looking for accessible method in the host class, and it will be used, at invocation time, to actually invoke the chosen method (cf. section 5.4).

4.2 Subtyping relations and parameter type hierarchy

In order to sort methods with each other, even if a total order could not exists, and to determine, from a n-uple of values of given types, which methods could be invoked with this values as argument, we need to consider, and organize each other, a set of types that includes the type of each parameter of each syntactically applicable method declaration, together with all their supertypes in the type hierarchy they belong to.

Subtyping relations are one of the main features of object oriented languages, because they allow any subtype T' of type T to be used in place of T by an *assignation conversion*. This subtyping relation we will use is the reflexive transitive closure of the non reflexive relation of *direct subtype*, noted $T' <_1 T$ if T' is a direct subtype of T.

In Java, there are three cases where a value of type T' could be assigned to a variable of type T ([13], § 5.1) and, when any method is invoked in Java, one of these assignation conversions (in this case called *invocation conversion*) is necessarily applied to each argument:

1. *identity conversion* in cases where $T = T'$. This implies that both $T \leq T'$ and $T' \leq T$;
2. *widening primitive conversion*, sometimes called implicit primitive cast, as when a value of type `short` could be assigned to a variable of type `int`. The relations for primitive types are the following: `byte` $<_1$ `short` $<_1$ `int`, `char` $<_1$ `int` and `int` $<_1$ `long` $<_1$ `float` $<_1$ `double` (cf. figure 2);
3. *widening reference conversion* is provided by several contexts:
   - explicit inheritance: if T' `extends` T, then $T' <_1 T$, T and T' being classes or interfaces;
   - implicit inheritance: any type T', that does neither explicitly extends nor implements any other one, implicitly extends the class `Object` and thus $T' <_1$ `Object`, T' being such a class or interface;
   - interface implementation: if T' `implements` T, then $T' <_1 T$, T' being a class and T an interface;
   - some other cases, specific to Java, provide types with relation $<_1$, e.g., with arrays, as shown in figure 2.

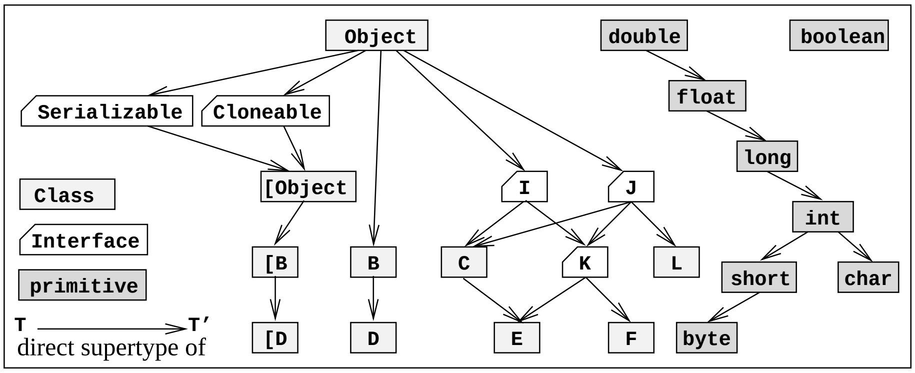

Fig. 2. Classes, interfaces and primitive types hierarchies

Definition 2 (Subtyping).
We will say that a type T' is a subtype of a type T, noted $T' \leq T$, if and only if

there exists an assignation conversion from type T' to type T.

$$T' \leq T \Leftrightarrow (T' = T) \vee (T' <_n T)$$

where $<_n$ represents n successive applications of $<_1$ with $n \geq 1$.

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 provides a method `getSupertypes(`T`)` that returns the set of all *direct* supertypes of T that is, with respect to our notations, $\{T' \mid T <_1 T'\}$.

More precisely, this method of the JMMF uses a `TypeModel` object whose default value respect the subtyping semantics of Java, but this behavior can be changed.

4.3 Directed acyclic graph

To 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 $T \rightarrow T'$ means that T "*is a direct supertype of*" T', that is $T' <_1 T$. 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 for $\mathcal{G}$ and we classically note $T \rightarrow^* T'$ if $T' \leq T$.

Definition 3 (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 to $\mathcal{M}$ 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 $T' <_1 T$.

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 distinguish between *extends*, *implements* or other subtyping (assignation conversion) relations.

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

4.4 Annotate the DAG

Pursuing our aim of being able to compare methods each other from their n-uples of declared types (signatures), we now annotate each type considered in the

DAG by its ability to be an acceptable argument type for methods of the multi-method. This annotation must be done for each method and at each parameter position. This annotation could be 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 i-th argument of method m_j.

Definition 4 (Type Annotation).
The annotation $\mathcal{A}_{T_{i,j}}[r][c]$ of type $T_{i,j}$ at position r of method m_c is set to 1 if and only if one of the following is true:

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

In all other case, $\mathcal{A}_{T_{i,j}}[r][c] = 0$.

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

Proof. *There are two implications to prove:*

1. *First, suppose that T is acceptable as r-th argument of method m_c. Since m_c could be invoked with an argument of type T as r-th argument, there is an invocation conversion from T to the declared parameter type $T_{r,c}$ of m_c. Thus, from section 4.2, this implies a subtyping relation between these types: $T \leq T_{r,c}$. By definition of our DAG and its edges, it follows that $T_{r,c} \rightarrow^* T$. By the first case in the definition 4, we have $\mathcal{A}_{T_{r,c}}[r][c] = 1$ and by second case of this same definition, since $T_{r,c} \rightarrow^* T$, we have $\mathcal{A}_T[r][c] = 1$.*
2. *Now, suppose that $\mathcal{A}_T[r][c] = 1$. By definition 4, two cases are possible:*
   - *either T is the type declared as the r-th parameter of method m_c. Then m_c obviously accepts values of type T as r-th argument;*
   - *or there exists a type $T_{r,c}$ such that $T_{r,c} \rightarrow^* T$ and $\mathcal{A}_{T_{r,c}}[r][c] = 1$. In this case, the edge from $T_{r,c}$ to T implies that $T \leq T_{r,c}$ and thus, a invocation conversion is possible from T to $T_{r,c}$. Then, via this conversion, m_c accepts values of type T as r-th argument.*

□

The second item of definition 4 represents the fact that if a value of type T is acceptable as i-th argument of method m_j, then any value of any subtype T' of T is also acceptable at the same position of the same method. From the DAG point of view, this implies a kind of *propagation* of annotations along its edges toward subtypes.

The constructive algorithm for this annotations successively considers the parameter type at each position of each method declaration. For a given parameter type T at position i of a method m_j, the bit $\mathcal{A}_T[i][j]$ is set. Annotations are recursively *propagated* onto $\mathcal{A}_{T'}[i][j]$ for every subtype T' of T considered in the DAG. This propagation, that follows the edge of the DAG, could simply performs a bit-wise `OR` with previously computed annotations. The fact that our graph is a DAG provides that this propagation terminates.

To illustrate our purpose, we choose a guiding example that exploits a part of 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` declared in `MyHostClass` and having exactly three parameters are defined with the following signatures:

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

The annotated DAG obtained for this multi-method is presented in figure 3 where a bullet figures out the fact that the annotation $\mathcal{A}_T[i][j]$ is set. Dark bullets stands for set annotations (types declared as parameter) and light bullets for propagated ones.

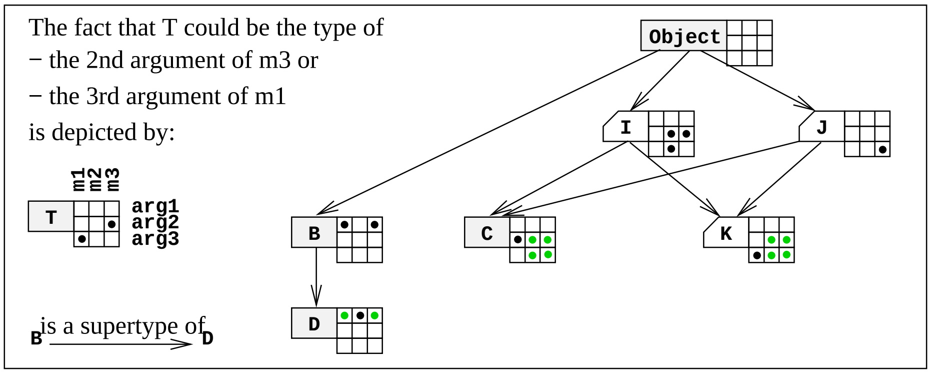

Fig. 3. An example of annotated DAG

4.5 Partial order over methods

From this annotated DAG, we now could establish, when it is possible, relations between method signatures based on their respective n-uples of parameter types. Indeed, from subtyping relation between types (cf. section 4.2), we define the relation of specificity between two methods and their signatures as follows: a method is more specific than another if each parameter type of the former is a subtype of the corresponding parameter type of the latter. More formally:

Definition 5 (Specificity Relation Between Method Signatures). *We say that a method signature* $m_j : (T_{1,j}, ..., T_{n,j})$*, is more specific than a signature* $m_k : T_{1,k}, ..., T_{n,k})$*, and note* $m_j \leq m_k$ *if and only if, for each parameter position,* $T_{i,j}$ *is a subtype of* $T_{i,k}$ *(or they are equal):*

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

We say in this case that m_k *is less specific than* m_j

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

1. The intuition behind the relation "m_j is *more specific* than m_k", formalized later in section 5.2, is that m_k could always be invoked with any n-uple of argument types accepted by m_j.
   For instance, consider the parameter types declared for methods m_1:`(B,C,K)` and m_3:`(B,I,J)`: the fact that `B` $\leq$ `B`, `C` $\leq$ `I` and `K` $\leq$ `J` provides that m_1 is more specific than m_3. Actually, m_3 could accept as argument any triple of types that is acceptable for m_1 (cf. figure 4-a).
2. Next, it is possible with definition 5 that both $m_j \leq m_k$ and $m_k \leq m_j$; this case only arises when, for all position i, $T_{j,i} = T_{i,k}$. This obviously implies that $m_j = m_k$ since we have assumed that two methods with the same signature are necessarily equals (cf. section 4.1).
3. Last thing to note is that it is not always possible to order each other two method signatures with respect to the *specificity* relation, i.e., some m_j could be neither more specific nor less specific than some m_k. These methods are said *not comparable* and provides us with a partial order on the set of methods $\mathcal{M}$. Two main reasons could yield to not comparable methods.
   - On the one hand, the declared parameter types $T_{i,j}$ and $T_{i,k}$ at a given position i could be not comparable, i.e., both $T_{i,j} \nleq T_{i,k}$ and $T_{i,k} \nleq T_{i,j}$. Types `I` and `J` of our hierarchy example leads to such a situation as third parameter types of methods m_2 and m_3 that are therefore not comparable (cf. figure 4-b).
   - On the other hand, for a given position i_1, one could have $T_{i_1,j} \leq T_{i_1,k}$ and for another position i_2, $T_{i_2,k} \leq T_{i_2,j}$.
     Such an example is given by methods m_1:`(B,C,K)` and m_2:`(D,I,I)` of our example since, at first position, `D` $\leq$ `B` and at second position, `C` $\leq$ `I`. Then m_1 and m_2 are not comparable (cf. figure 4-c).

The key structure we are using to represent these relations between methods is again a bit matrix, with p rows and p columns; such a single matrix allows us to store all method relations for a given multi-method.

Let $\mathcal{PO}$ be this matrix of partial order between methods. The bit $\mathcal{PO}[r][c]$ at the r-th row and the c-th column is set if $m_r \leq m_c$. In other words, the method m_c could accept as argument any n-uple of value that are acceptable by the method m_r.

Definition 6 (Partial order over methods). *Given two indexes of methods,* r *and* c *in* $[1..p]$*,* $\mathcal{PO}[r][c]$ *is set to 1 if and only if* $m_r \leq m_c$*. It is set to 0 in any other cases.*

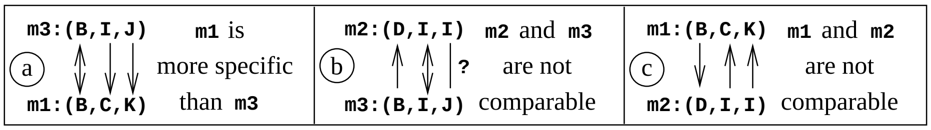

Fig. 4. Specificity relation defines only partial order

A row $\mathcal{PO}[r][*]$ is a bit array whose values at 1 identify the set (the indexes) of methods that are less 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. Furthermore, we can equally use the set-theoretical operations (union/intersection) and bit-wise operations (`OR`/`AND`).

Definition 7 (Less Specific Methods).
Given a method m_j, the set of methods $\{m_k \mid \mathcal{PO}[j][k] = 1\}$ is the set of methods that are less specific than m_j. We will also note this set $\mathcal{PO}[j]$.

Definition 8 (More Specific Methods).
Given a method m_j, the set of methods $\{m_k \mid \mathcal{PO}^t[j][k] = 1\}$ is the set of methods that are more specific than m_j. We will also note this set $\mathcal{PO}^t[j]$.
In this notation, $\mathcal{PO}^t$ is the transposed matrix of $\mathcal{PO}$, formally defined by:

$$\forall j, k \in [1..p], \mathcal{PO}^t[j][k] = \mathcal{PO}[k][j]$$

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

Theorem 2 ($\mathcal{PO}$ as annotation conjunction).
If m_j is a method of declared signature $(T_{1,j}, ..., T_{n,j})$, then for all $k \in [1..p]$ $\mathcal{PO}[j][k] = 1$ if and only if $\texttt{AND}_{i\in[1..n]}\mathcal{A}_{T_{i,j}}[i][k] = 1$

Proof. There are two implications to prove:

1. *Suppose that, for a given k, $\mathcal{PO}[j][k] = 1$. This means, from definition 6, that $m_j \leq m_k$. From the definition of this specificity relation, it follows that $\forall\ i \in [1..n],\ T_{i,j} \leq T_{i,k}$. By our DAG's construction this implies that, $\forall\ i \in [1..n],\ T_{i,k} \to^* T_{i,j}$. But with our generic notation and from the first case of definition 4, we know that $\forall\ i \in [1..n],\ \mathcal{A}_{T_{i,k}}[i][k] = 1$. Then, the second case of definition 4 provides us, since $\forall\ i \in [1..n],\ T_{i,k} \to^* T_{i,j}$, with $\forall\ i \in [1..n],\ \mathcal{A}_{T_{i,j}}[i][k] = 1$. It follows that $\texttt{AND}_{i\in[1..n]}\mathcal{A}_{T_{i,j}}[i][k] = 1$.*
2. *Let us suppose now that, for a given k, $\mathcal{A}_{T_{i,j}}[i][k] = 1$ for all $i \in [1..n]$. For each i, two cases are then possible:*

– *either* $k = j$ *and thus* $T_{i,j} \leq T_{i,k}$,
– *or* $k \neq j$ *and then, since* $\mathcal{A}_{T_{i,j}}[i][k] = 1$, *there exists* $T_{i,k}$ *such that both* $\mathcal{A}_{T_{i,k}}[i][k] = 1$ *and* $T_{i,k} \rightarrow^* T_{i,j}$. *By definition of our DAG edges, this implies that* $T_{i,j} \leq T_{i,k}$.

Since $\forall\ i \in [1..n],\ T_{i,j} \leq T_{i,k}$, *then* $m_j \leq m_k$ *and* $\mathcal{PO}[j][k]$ *must be set to 1.*

□

Corollary 1 (Computation of Less Specific Methods).
Previous theorem provides us with a simple computation of the set of methods that are less specific than a given method m_j*:*

$$\mathcal{PO}[j] = \texttt{AND}_{i \in [1..n]} \mathcal{A}_{T_{i,j}}[i]$$

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

$$\begin{array}{l}
\mathcal{PO}[1] = \mathcal{A}_B[1] \ \texttt{AND}\ \mathcal{A}_C[2] \ \texttt{AND}\ \mathcal{A}_K[3] = [\bullet|\circ|\bullet]\texttt{AND}[\bullet|\bullet|\bullet]\texttt{AND}[\bullet|\bullet|\bullet] = [\bullet|\circ|\bullet] \\
\mathcal{PO}[2] = \mathcal{A}_D[1] \ \texttt{AND}\ \mathcal{A}_I[2] \ \texttt{AND}\ \mathcal{A}_I[3] \ = [\bullet|\bullet|\bullet]\texttt{AND}[\circ|\bullet|\bullet]\texttt{AND}[\circ|\bullet|\circ] = [\circ|\bullet|\circ] \\
\mathcal{PO}[3] = \mathcal{A}_B[1] \ \texttt{AND}\ \mathcal{A}_I[2] \ \texttt{AND}\ \mathcal{A}_J[3] \ = [\bullet|\circ|\bullet]\texttt{AND}[\circ|\bullet|\bullet]\texttt{AND}[\circ|\circ|\bullet] = [\circ|\circ|\bullet]
\end{array}$$

This matrix, built horizontally here through $\mathcal{PO}$'s rows, will be used later, reading it vertically through the other entry $\mathcal{PO}^t$'s rows (i.e., $\mathcal{PO}$ columns), to determine the set of methods that are more specific than a given method m_j (i.e. the set of methods m_k whose bit is set at the index k in $\mathcal{PO}^t[j]$). Figure 5 shows the matrix $\mathcal{PO}$ of our example together with it transposed version $\mathcal{PO}^t$.

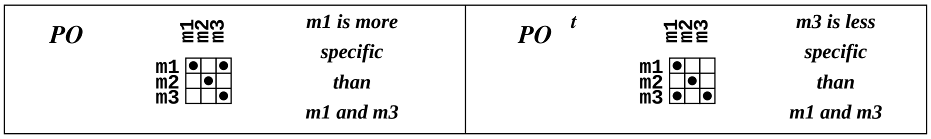

Fig. 5. Partial order between the methods of our example

5 Multi-method dispatch at invocation time

The process we described in the previous section is completely done at creation time. We are now interested in invocation time process, that is, given a n-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 n-uple of types. The first problem that can arise is that one of these types could be unknown in the DAG we built. This type can be unknown at compile-time since in Java types (classes) can be loaded dynamically at run time. Thus, even a static analysis of the type hierarchy at creation time cannot identify such a type.

5.1 Completing DAG with dynamic types

From our running example, let a multi-method invocation be the following:

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

In such a case, the dynamic type `L` of reference `l`[2] 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 both establish relations between `L` and the other types and compute its annotations.

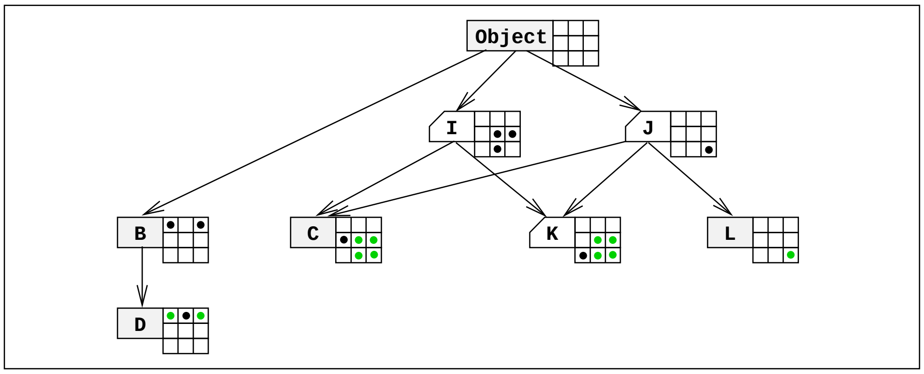

Fig. 6. Our DAG example dynamically completed by newly discovered type `L`

The algorithms performed at invocation time to complete the DAG $\mathcal{G}$ are of the same nature as those performed at creation time. A newly discovered type T is first added as a vertex of the DAG, together with all its supertypes that not yet appear in $\mathcal{G}$. The annotations of all the newly added types are recursively deduced (by bitwise `OR` propagation) from those of their direct supertypes, as illustrated by figure 6. In fact, the initialization from existing type annotations of the new type annotations can be performed recursively at the same time as the DAG completion.

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

[2] We will generically note by a lowercase character such as `t` a variable that contains an object typed by the corresponding uppercase character, such as `T`.

5.2 Semantically applicable methods

Type annotations computed either at creation time or invocation time tell us if a single given type is acceptable for an argument at a given position of a given method. We now want to deduce, for a n-uple $u = (T_1, ..., T_n)$ of argument types of a given invocation site, if it corresponds not any, one and only one, or multiple acceptable methods in $\mathcal{M}$. Indeed, we are looking for the set of *semantically applicable* methods for which invocation conversions would be able to accept u as argument types. This set, noted $\mathcal{SA}_u$ and defined below, could be simply computed from T_i's annotations.

Definition 9 (Semantically Applicable Methods).
Given a n-uple $u = (T_1, ..., T_n)$ of types in $\mathcal{V}(\mathcal{G})$, $\mathcal{SA}_u[j]$ is set to 1 if and only if u is a n-uple of argument types acceptable for m_j.

Theorem 3 ($\mathcal{SA}$ as Annotation Conjunction).
Let $u = (T_1, ..., T_n)$ where each $T_i \in \mathcal{V}(\mathcal{G})$, then

$$\mathcal{SA}_u = \mathtt{AND}_{i \in [1..n]} \mathcal{A}_{T_i}[i]$$

Proof. The proof is obvious by definitions 4, 9 and theorem 1.
□

The number of bit set to 1 in $\mathcal{SA}_u$ give us important information, summarized below in three cases:

1. if zero bit is set to 1 then there are no semantically applicable method;
2. if one bit at index i is set to 1 then there is only one semantically applicable method which profile is m_i;
3. if more than one bit is set to 1 then there are multiple semantically applicable methods and we do not have enough information here to decide what will happen. Some *disambiguation* processing is needed.

Let us go back to our example of multi-method `myMM` (cf. section 4.4 and figure 3) to exhibit three illustrating examples for these cases.

Let u_1 be the n-uple (B, C, D). Theorem 3 leads to:

$$\mathcal{SA}_{u_1} = \mathcal{A}_B[1] \;\mathtt{AND}\; \mathcal{A}_C[2] \;\mathtt{AND}\; \mathcal{A}_D[3] = [\bullet, \circ, \bullet]\mathtt{AND}[\bullet, \bullet, \bullet]\mathtt{AND}[\circ, \circ, \circ] = [\circ, \circ, \circ]$$

There is no semantically applicable method for (B, C, D) argument type tuple.

If $u_2 = (D, C, L)$, the same principle gives $\mathcal{SA}_{u_2} = [\circ, \circ, \bullet]$. The only semantically applicable method for (D, C, L) argument type tuple is the method with signature $m_3 : mymethod(B, I, J)$.

Let now u_3 be the n-uple (D, C, C). It follows that $\mathcal{SA}_{u_3} = [\circ, \bullet, \bullet]$. There is two semantically applicable methods for (D, C, C) argument type tuple: the method with signature $m_2 : mymethod(DI, I)$ and the one with signature $m_3 : mymethod(B, I, J)$.

5.3 Disambiguation process from method's partial order

The disambiguation process presented in this section is (only) performed when the number of semantically applicable methods, for a given n-uple u of argument types, 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 10 (Most Specific Syntactically Applicable Method).
Given a n-uple $u = (T_1, ..., T_n)$ of types in $\mathcal{V}(\mathcal{G})$, we define

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

Theorem 4 (Set of methods associated to $\mathcal{MSA}$).
Given a n-uple $u = (T_1, ..., T_n)$, the set of methods $\{m_j | \mathcal{MSA}_u[j] = 1\}$ is either empty or a singleton.

Proof. Let us suppose, by contradiction, that there exists j and k such that $j \neq k$, $\mathcal{MSA}_u[j] = 1$ and $\mathcal{MSA}_u[k] = 1$. Since, in construction of $\mathcal{MSA}_u$, a bitwise `AND` *is applied with $\mathcal{SA}_u$, then $\mathcal{SA}_u[j] = 1$ and $\mathcal{SA}_u[k] = 1$. This implies that $\mathcal{PO}^t[j]$ and $\mathcal{PO}^t[k]$ appear in $\texttt{AND}_{\{l \ | \ \mathcal{SA}_u[l]=1\}} \ \mathcal{PO}^t[l]$ and since we have supposed that $\mathcal{MSA}_u[j] = \mathcal{MSA}_u[k] = 1$, $\mathcal{PO}^t[j][j] = \mathcal{PO}^t[j][k] = 1$ and $\mathcal{PO}^t[k][j] = \mathcal{PO}^t[k][k] = 1$. By definition of $\mathcal{PO}$, this means that both $m_k \leq m_j$ and $m_j \leq m_k$. This is possible if and only if $j = k$, that contradicts our hypothesis.*
□

Theorem 5 (Most Specific Method).
Given a n-uple $u = (T_1, ..., T_n)$, a method m_k is the most specific semantically applicable method for u if and only if $\mathcal{MSA}_u[j] = 1$.

Proof. There are two implications to prove:

1. *Suppose that m_k is the most specific method for u. Then, it is semantically applicable, i.e., $\mathcal{SA}_u[k] = 1$. Furthermore, for all other method m_j such that $\mathcal{SA}_u[j] = 1$, $m_k \leq m_j$ by hypothesis. Then, for all these j, $\mathcal{PO}^t[j][k] = 1$, and then $\mathcal{MSA}_u[j] = 1$.*
2. *Suppose now that $\mathcal{MSA}_u[j] = 1$. This implies, in the one hand, that m_j is semantically applicable (since $\mathcal{SA}_u[j] = 1$) and, on the other hand, that for all other semantically applicable methods m_k, $\mathcal{PO}^t[k][j] = 1$. Then $m_j \leq m_k$ for all such k.*

□

Corollary 2 (Existence of a most specific method).
There is no most specific method if and only if $\forall j \in [1..p]$, $\mathcal{MSA}_u[j] = 0$.

In order to illustrate these situations, consider the n-uple $u_3 = (D, C, C)$ for which we deduce, at section 5.2 and from $\mathcal{SA}_{u_3} = [\circ, \bullet, \bullet]$, that there were an ambiguity. Now, from definition 10 and from the matrix $\mathcal{PO}^t$ at figure 5, we could compute $\mathcal{MSA}_{u_3}$:

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

Then, from corollary 2, there exists no most specific semantically applicable method for u_3. This is not surprising since, as illustrated in figure 4, the only semantically applicable methods for u, are m_2 and m_3 and they are not comparable.

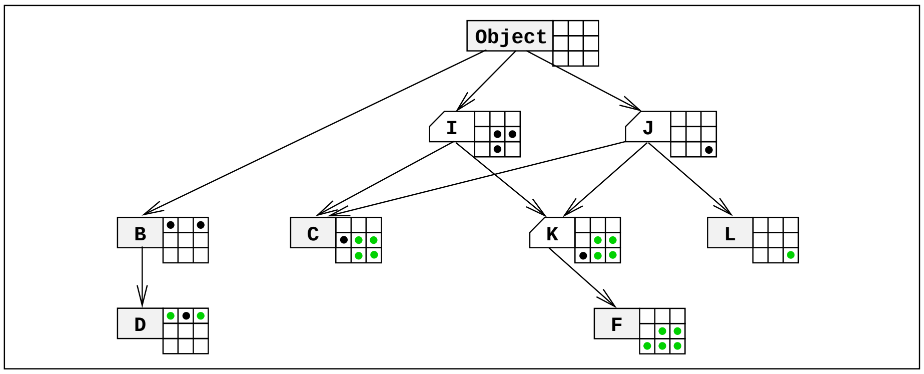

Fig. 7. The DAG example completed by newly discovered type `F`

Let us consider another example of invocation with the n-uple of types $u_4 = (B, C, F)$ where the type F is dynamically loaded at invocation time from a class `F` that implements `K`. The DAG is then completed as illustrated in figure 7 and we could compute, from theorem 3, that $\mathcal{SA}_{u_4} = [\bullet, \circ, \bullet]$. In this case of ambiguity, we then compute $\mathcal{MSA}_{u_4}$:

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

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

5.4 Implementation

Algorithms and data-structures described by now permits to determine, given a n-uple of argument types u, the index of the unique most specific method signature in $\mathcal{M}$ if it exists.

This process may fail at two stages:

- after computation of $\mathcal{SA}_u$ (section 5.2), if it only contains 0 at each index. In this case, where not any method signature is semantically applicable with respect to types of u, our implementation throws a `NoSuchMethodException` exception;
- after computation of $\mathcal{MSA}_u$ (section 5.3), if it only contains 0 at each index. In this case, where it is not possible to decide which method signature to use among several semantically applicable ones, our implementation throws a `MultipleMethodException` exception.

Similarly, this process may complete at two stages:

- after computation of $\mathcal{SA}_u$, if there is a unique index k for which $\mathcal{SA}_u[k] = 1$;
- after computation of $\mathcal{MSA}_u$, if there is a unique index k for which k for which $\mathcal{MS}_u = 1$.

In last two cases, given the index k of the most precise signature, the corresponding method has to be invoked. In the JMMF implementation, this is done with the Java reflection API and from an instance of class `Method` whose method `invoke()` is called with the provided arguments of u. To do this, we just need to retrieve the instance of `Method` corresponding to the signature m_k, as we presumed in section 4.1.

With this aim in view, when `create(hostClass,methodName,n)` is called at creation time, the set $\mathcal{M}$ is computed using the reflective method `getMethods()` on `hostClass`. This method returns a array of `Method` instances which is filtered, according to `methodName` and `n`, to produce an array that only contains syntactically applicable methods. Actually, this last array can be used to implement the set $\mathcal{M}$ since `getMethods()` implementation ensures that each signature of the returned `Method` instances is unique, in particular with method overriding. Moreover, with this implementation, retrieving the `Method` instance given the index of the most precise signature only consists in an array access.

6 Inheritance and multi-methods

Providing multi-methods in a language that supports inheritance and subtyping, we may hope that multi-methods also support these features. To this end, the JMMF implementation accepts a multi-method call with a target object that is an instance of a subtype `SubHostClass` of the class `HostClass` specified at creation time[3].

Since the class `SubHostClass` may contain other syntactically applicable methods than `HostClass`, data-structures computed for `HostClass` are not valid. Then, at a first sight, one could process at invocation time all data-structures

[3] If the class object passed as argument is not a subclass of `hostClass` an exception is thrown.

constructed at creation time for `HostClass`. 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 semantically applicable method signatures $\mathcal{M}$, all computations can be performed on a larger set $\mathcal{M}'$. Thus, signatures coming from `HostClass` and from `SubHostClass` could coexists in a same data-structure.

Definition 11 (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 $\mathcal{MS}'_u$.

Moreover, if $\mathcal{M} \subseteq \mathcal{M}'$, for each signature index j such that $m_j \in \mathcal{M}$ we note $j_{\mathcal{M}'}$ the index of the same signature in $\mathcal{M}'$ such that $m_{j_{\mathcal{M}'}} \in \mathcal{M}'$ and $m_j = m_{j_{\mathcal{M}'}}$.

Definition 12 (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, $\forall m_k \in \mathcal{M}'$, $\mathcal{M}ask_{\mathcal{M}}[k] = 1$ if and only if it exists $m_j \in \mathcal{M}$ such that $k = j_{\mathcal{M}'}$.

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 the method signatures in $\mathcal{M}'$ 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.

Lemma 1 (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. $\mathcal{A}_T[i][j] = 1$ *if and only if* $\mathcal{A}'_T[i][j_{\mathcal{M}'}]$ `AND` $\mathcal{M}ask_{\mathcal{M}}[j_{\mathcal{M}'}] = 1$;
2. $\mathcal{PO}[j][k] = 1$ *if and only if* $\mathcal{PO}'[j_{\mathcal{M}'}][k_{\mathcal{M}'}]$ `AND` $\mathcal{M}ask_{\mathcal{M}}[j_{\mathcal{M}'}] = 1$.
3. $\mathcal{SA}_u[j] = 1$ *if and only if* $\mathcal{SA}'_u[j_{\mathcal{M}'}]$ `AND` $\mathcal{M}ask_{\mathcal{M}}[j_{\mathcal{M}'}] = 1$

Proof. *There are three equivalences to prove:*

1.
   - *If $\mathcal{A}_T[i][j] = 1$ then either $T = T_{i,j}$ or T is a subtype of $T_{i,j}$. Since $T_{i,j}$ is the i-th parameter type of method m_j and $m_j = m_{j_{\mathcal{M}'}}$, $T_{i,j} = T_{i,j_{\mathcal{M}'}}$. Thus, $T = T_{i,j_{\mathcal{M}'}}$ or T is a subtype of $T_{i,j_{\mathcal{M}'}}$ which implies that $\mathcal{A}'_T[i][j_{\mathcal{M}'}] = 1$. By definition, $m_j \in \mathcal{M}$, that proves the implication.*
   - *If $\mathcal{A}'_T[i][k]$ `AND` $\mathcal{M}ask_{\mathcal{M}}[k] = 1$ then it exists $m_j \in \mathcal{M}$ such that $k = j_{\mathcal{M}'}$ and $\mathcal{A}'_T[i][j_{\mathcal{M}'}] = 1$. The rest of the proof is equivalent to previous implication.*
2.
   - *If $\mathcal{PO}[j][k] = 1$ then m_j is more specific than m_k. Since $m_j = m_{j_{\mathcal{M}'}}$ and $m_k = m_{k_{\mathcal{M}'}}$, $m_{j_{\mathcal{M}'}}$ is more specific than $m_{k_{\mathcal{M}'}}$ and thus $\mathcal{PO}'[j_{\mathcal{M}'}][k_{\mathcal{M}'}] = 1$. By definition, $m_j \in \mathcal{M}$, that proves the implication.*
   - *If $\mathcal{PO}'[j_{\mathcal{M}'}][k_{\mathcal{M}'}]$ `AND` $\mathcal{M}ask_{\mathcal{M}}[j_{\mathcal{M}'}] = 1$ then it exists $m_j \in \mathcal{M}$ such that $k = j_{\mathcal{M}'}$ and $\mathcal{PO}'[j_{\mathcal{M}'}][k_{\mathcal{M}'}] = 1$. The rest of the proof is equivalent to previous implication.*

3. – *If $\mathcal{SA}_u[j] = 1$, by definition, $\forall i \in [1..n], \mathcal{A}_{T_i}[i][j] = 1$. From previous equivalence it follows that $\forall i \in [1..n], \mathcal{A}'_{T_i}[i][j_{\mathcal{M}'}]$ AND $\mathcal{M}ask_{\mathcal{M}}[j_{\mathcal{M}'}] = 1$ and since, by hypothesis, $m_j \in \mathcal{M}$, it folows that $\mathcal{M}ask_{\mathcal{M}}[j_{\mathcal{M}'}] = 1$ and $\forall i \in [1..n], \mathcal{A}'_{T_i}[i][j_{\mathcal{M}'}] = 1$. This proves the implicition.*
   – *Other implication is comparable.*

□

Definition 13 (Semantically applicable methods for included set). *Let $\mathcal{M}$ and $\mathcal{M}'$ be two sets of method signatures such that $\mathcal{M} \subseteq \mathcal{M}'$. Given a n-uple u of argument type, we define semantically applicable methods of $\mathcal{M}$ for u 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 both semantically applicable for the n-uple u and considered in $\mathcal{M}$. As we did in section 5.3 with theorem 5, we now want to know how to refine this set if several such semantically applicable exist. In order to disambiguate them, we have to take care that in the set $\mathcal{PO}'^t$, we must take only into account methods indexes of $\mathcal{SA}'_{u,\mathcal{M}}$ (and not all $\mathcal{SA}'_u$).

Theorem 6 (Most specific method computation for included set).

$$\mathcal{MSA}_u[j] = 1 \textit{ if and only if } \mathcal{SA}'_{u,\mathcal{M}}[j_{\mathcal{M}'}] \texttt{ AND } (\texttt{AND}_{\{l \ | \ \mathcal{SA}'_{u,\mathcal{M}}[l]\}} \mathcal{PO}'^t[l]) = 1$$

Proof. *The proof is obvious from previous lemmas.*

□

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 computations (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 previous assigned indexes are not modified. In the implementation, figure 8 illustrates such a case. Newly computed values are shown in gray. This figure also represents the mask used for each class and the table of `Method` instances.

7 Special issues

7.1 Invocation semantics

The semantics of our disambiguation process is slightly different from those performed at compile-time by java compilers (JLS § 15.12.2.2). Indeed, the JLS disambiguation takes into account the type T (class or interface) in which the method is declared. This type T could be any super-type of the receiver declared type. This basically means that the receiver is considered as an hidden

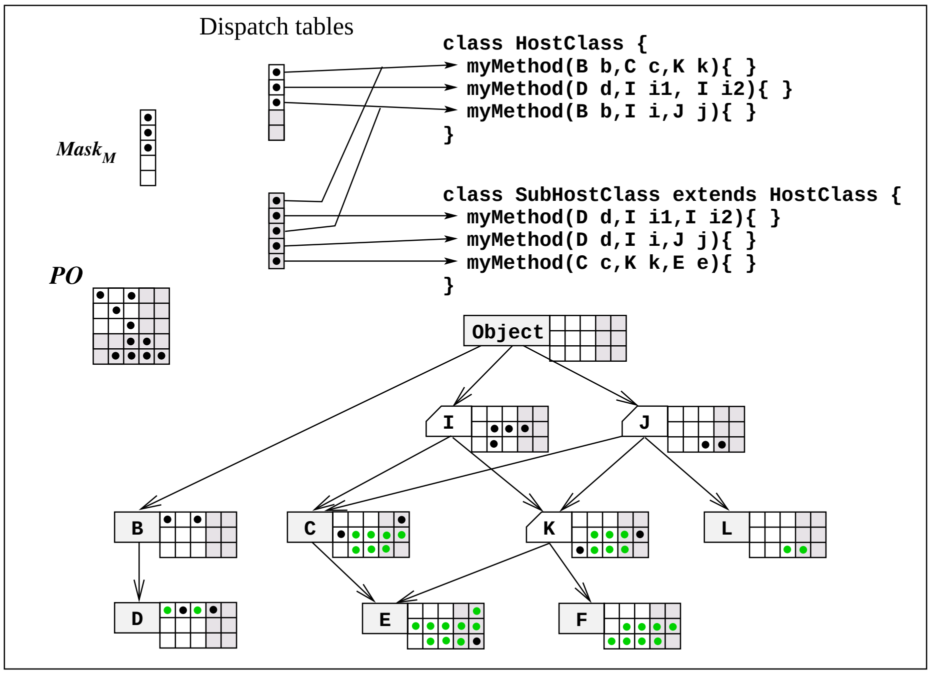

Fig. 8. Example of data-structure involved in inheritance

parameter, i.e., a method declared in the class T_0 with signature $m(T_1, ..., T_n)$ is actually viewed, for the JLS disambiguation process, as a method of signature $m(T_0, T_1, ..., T_n)$.

Since our disambiguation process does not takes into account the type of the method declaring class, this improve performance by lowering of one the number of arguments involved and thus improve performance. But this is not the reason for this choice and furthermore, we think that the semantics provided by taking this declaring class type into account leads to undesirable behavior. Let us examine why, with two examples.

Inheritance from two not comparable declaring interfaces

```
class C {}
class D extends C {}
interface I { void m(C c); }
interface J { void m(D d); }
interface K1 extends I, J {}
interface K2 extends I, J { void m(D d); }
class Test {
 static void test(K1 k1, K2 k2, D d) {
```

```
  k1.m(d); // Reference to m is ambiguous.
  k2.m(d); // Correct
 }
}
```

In this example, invocation `k1.m(d)` is considered ambiguous since methods `m(C c)` and `m(D d)` are declared in two interfaces `I` and `J` that are not comparable. However, any dynamic type of an actual receiver will necessary have both methods definition.

Nonetheless, invocation `k2.m(d)` is not ambiguous, even if interfaces `K2` and `K1` implies the same semantics on any concrete class `Impl` that implements them:

- this allow any reference to instance of class `Impl` to be assigned to variables of type `K1` (resp. `K2`), `I` and `J`;
- they impose that `Impl` implements both methods `m(D d)` and `m(C c)`.

This example shows that method declaration overriding has some surprising semantics.

Covariant declaring types with contravariant parameter types.

```
class C {}
class D extends C {}
class A {
  void m(D d) {}
  void m(C c) {}
}
class B extends A { void m(C c) {} }
class Test2 {
 static void test(D d, B b) {
  b.m(d);  // Reference to m is ambiguous.
  A a=b;
  a.m(d);  // correct
 }
}
```

Even the contravariant method overloading classical example, presented below, is neither satisfactory.

In this example, invocation `b.m(d)` is ambiguous since it both matches method `m(C c)` declared in `B` and method `m(D d)` declared in `A`. Indeed, as JLS disambiguation process takes into account the class of the method declaration, it tries to sort the type couples `(A,D)` and `(B,C)`, but they are not comparable since $B < A$ and $D < C$.

However, it is obvious that in class `B`, method `m(D d)` will always be available, then there is no reason not to choose it as more the specific. Furthermore, subtyping `b` into an object of type `A` typechecks at compile time and produces, at runtime, the “expected” behavior of the previous invocation.

We claim that this behavior, as those illustrated in the previous example, are not satisfactory.

Receiver's declared type as additional parameter type. One justification presented for taking into account the class of the receiver in method disambiguation is the fact that, if two methods override each other, then the overriding method (declared in a subclass) automatically appears to the disambiguation process as more specific than the overidden one. But a method signature used at runtime does not take into account the class in which it is declared (both overriding and overidden methods produce the same signature). It is therefore useless to differenciate these two methods at the typing level.

One other reason to take into account the receiver declared type is that it is not equivalent to inherit from a method than to define it in a class. For instance, since private methods of a given class could only be invoked in this class, Java compilers use the type of the class declaring a private method to forbids its invocation in a subclass. But even this solution is useful to performs a visibility-check at compile time, this does not rely on type checking.

Finally, the main drawbacks of not taking into account the declaration class is to yield different semantics to method reusability between inheritance and parameter subtyping, but we think this fits well with class languages. Let us explain this distinction. If a method `m` needs to access two references of type `A` and `C`, there are two ways to define it: either as a method with one parameter of type `C` in class `A` or as a method with a two parameters of type `A` and `C`, declared anywhere. Whereas in the former case, `m` must be non static, in the latter case it could be static. Furthermore, suppose (for visibility reasons), that the static method is declared in `A`; we obtain the following code.

```
class A {
  void m(C c) {}
  static void m(A a, C c) {}
}
```

Now, if `m` is considered in a multi-dispatch invocation, both solutions are equivalent. Nevertheless, if these methods are reused through a subtype `B` extending `A`, the inherited static method will not behave like the non-static one in a disambiguation process as those presented in this paper. In fact, with our disambiguation, the non static method will be considered as belonging to `B` with a signature `(C)`, whereas the static method be always considered belonging to `A` with a signature `(A,C)` (since it is declared static). With a disambiguation that would takes declaring class into account, the signature of the non static method `m` would have been considered as `(A,C)`, and then not considered in a multi-dispatch through `B`.

With all these advantages and drawbacks in mind, we consider that the type of the receiver object should not be taken into account by the disambiguation process. It comes to the same thing as considering that all accessible method of a class are declared in the class, and to view the class as a "blackbox" with respect to method declaration. This could be view as a kind of type "reification".

7.2 Method restrictions

In section 4.1 we have supposed that all methods were public, concrete and non-static. In fact, there are no restriction concerning abastract methods since they behave like concrete ones for our algorithms.

Concerning static methods, there is no reason not to consider them in the set of syntactically applicable methods. The only special behaviour 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. To do so, we introdce another mask for static method and process disambiguation in the same way than inheritance.

If the method is not public, then our implementation must verify that the class containing the invocation site of the mutli-method has the correct visibility and has to throw an exception in case of visibility violation. Moreover, for private and default methods, extra verification should be performed in case of inheritance, as suggested by the following example with classical (single) dispatch. For all these reason and for the multi-methods use case we address, our implementation only deals with public methods by now.

```
class A {
 private void m1(){
   System.out.println("private");
 }
 public void m2(){
   this.m1(); // Method invocation we want to emulate
 }
}
class B extends A {
 public void m1(){
   System.out.println("public");
 }
 public void test(B b){
   b.m2(); // Prints private since m1 is binded at compile time
   b.m1(); // Prints public is private m1 is not accessible here
 }
}
```

7.3 The case of `null`

By now, we did not considered the possibility to perform an invocation of a multi-method with the `null` reference as argument. However, in `myMM.invoke(target, new Object[]{d,c,l})`, the target reference and any argument (except if a primitive type is expected) could be the `null` reference. A `null` argument at the target position corresponds to the invocation of a static method, as seen in the previous section. For `null` argument in all other positions, we deal with `null` as a value of a special type that is considered as a subtype of any object reference type. Thus, the DAG is completed with this special type which annotation is directly constructed from methods profile. Then at invocation time, the disambiguation process is exactly the same as the one of classical argument types.

8 Related works

There exists a lot of works on multi-methods [1, 8, 6, 3, 18], but 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 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 simple and very efficient translation into `instanceof` statements.

Clifton, Leavens, Chambers and Millstein [7] proposed a conservative extensions to Java to support multi-method dispatch, called MultiJava. MutliJava introduces syntactic modifications to provide 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 methods implementations. This approach is completely opposed to our, indeed, we do not perform any static type-checking but we allow maximum flexibility in implementation.

Holst, Szafron, Leontiev and Pang [9] have 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 work and similarly to our, this implementation proposes loose type checking (with warnings) for multi-methods; exceptions are thrown at run-time in case of type-checking error. One of the proposed implementations is based on the SRP technique [14] to provide 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 to add "virtual" methods and to sort methods to ensure that in case of ambiguity the most specific method always exists and is the one with smallest index. Our implementation does no have these requirements but imposes extra bit-wise `AND` on $\mathcal{PO}$ rows to perform disambiguation. Moreover, as shown in section 6, since no extra methods nor order is needed on methods, annotations and matrix $\mathcal{PO}$

can be shared in case of inheritance. Implementation of these two techniques in a common framework is still required to quantify their respective advantages.

9 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 keyword to the original language definition. With respect to other research works on multi-method [1, 3, 4, 7, 9] that address typing issues, ours focus on the simplicity of design, use and implementation efficiency rather than on static type checking.

Indeed, the new simple multi-method dispatch algorithm presented in this paper provides a dispatch in $\mathcal{O}(n+p)$ where n is the number of parameter and p the number of methods in the multi-method. Moreover, this algorithm provides large reuse of data-structures.

Nevertheless, the most important issue of this work is that it provides the programmer with an easy way to design and maintain algorithm specification. In particular, multi-methods simply allow algorithm on recursive data-structures [11], such as trees, to be externally specified. For instance, JMMF is intensively used in the project SmartTools [2] 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 *OOPSLA'91 proceedings*, ACM SIGPLAN, pages 113–128, Phoenix Arizona, October 1991.
2. 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.
3. François Bourdoncle and Stephan Merz. Type-checking higher-order polymorphic multi-methods. In *POPL'97 proceedings*, ACM SIGPLAN-SIGACT, pages 302–315, Paris, France, January 1997.
4. J. Boyland and G. Castagna. Parasitic methods: An implementation of multi-methods for Java. In *OOPSLA'97*, number 32–10 in SIGPLAN Notices, pages 66–76, New York, October 1997. ACM Press.
5. 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.
6. Craig Chambers. Object-oriented multi-methods in Cecil. In *ECOOP'92 proceedings*, Utrecht, The Netherlands, July 1992.
7. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch. In *OOPSLA'00 proceedings*, ACM SIGPLAN Notices, Minneapolis, USA, October 2000.
8. Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An overview. In *ECOOP'87 Proceedings*, pages 151–170, Paris, France, June 1987.

9. Christopher Dutchyn, Paul Lu, Duane Szafron, Steve Bromling, and Wade Holst. Multi-dispatch in the java virtual machine design and implementation. In *COOTS'01 proceedings*, San Antonio, USA, January 2001.
10. Rémi Forax, Etienne Duris, and Gilles Roussel. Java multi-method framework. In *TOOLS Pacific'00 Proceedings*, Sidney, Australia, November 2000. IEEE Computer.
11. Rémi Forax and Gilles Roussel. Recursive types and pattern-matching in Java. In *GCSE'99 proceedings*, number 1799 in LNCS, Erfurt, Germany, September 1999.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
13. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification – Second Edition*. Addison-Wesley, 2000.
14. Wade Holst, Duane Szafron, Yuri Leontiev, and Candy Pang. multi-method dispatch using single-receiver projections. Technical Report 98-03, Departement of Computer Science, University of Alberta, Edmonton, Alberta, Canada, 1998.
15. Gregor Kiczales, Jim Des Rivieres, and Daniel Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.
16. Maxim Kizub. Kiev language specification. An extension of Java language that inherits Pizza features and provides multi-methods (`http://forestro.com/kiev/`), July 1998.
17. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
18. Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP'99 proceedings*, number 1628 in LNCS, pages 279–303, Lisbon, Portugal, June 1999.