Multiple dispatch in reflective runtime environment

Rajeev Kumar*, Vikram Agrawal

Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, West Bengal 721 302, India

Received 13 July 2005

Abstract

Message dispatch in object-oriented programming (OOP) involves target method lookup in dispatch table/tree. Reflective environment builds dispatch data-structure at runtime as types can be added at runtime. Hence, algorithms for reflective environments require dynamic data structure for dispatch. In this paper, we propose a tree-based algorithm for multiple dispatch in reflective runtime environment. New classes can be added to the system at runtime. Proposed algorithm performs lookup in time proportional to \( \log(n) \) times the polymorphic arguments, where \( n \) is number of classes in a system. Proposed algorithm uses type-safe approach for multimethod lookup resolving ambiguities. We compare performance of the proposed algorithm with the dispatch mechanism in commonly used virtual/reflexive systems, e.g., Java and Microsoft’s Common Language Runtime (MS-CLR), in respect of efficiency and type-safety.

© 2005 Elsevier Ltd. All rights reserved.

Keywords: Object-oriented programming; Message dispatch; Dynamic dispatch; Multimethods; Reflection; Virtual table; Java; Microsoft’s common language runtime

1. Introduction

Message dispatch [1] is a central feature of object-oriented programming (OOP) languages. Depending on the number of arguments participating (polymorphic arguments) in a message dispatch for method lookup, object-oriented languages can be classified into single dispatch and multiple dispatch languages. The methods with more than one polymorphic argument are called multimethods. Multimethods manifest themselves in diverse applications. For example, in addition of two numbers, the method invoked depends on both operands/arguments; addition of two integers is different from addition of a floating point number and an integer. However, many languages (e.g., C++, C# and Java) do not directly support multimethods.

The main disadvantage of multimethods is inefficiency. A dispatch table for multimethods requires a large amount of space and time for target method lookup. However, many algorithms [2–7] have been proposed for multiple dispatch for non-reflective environment. Preprocessors [8–10] have also been proposed for simulation of multimethods in single dispatch languages. These algorithms require information about all classes for building dispatch structures using some heuristics. Algorithms by Chambers [4], Dujardin [11] and Naik [12] use optimized dispatch tree for multiple dispatches. These trees are built at compile time for non-reflective environments, and therefore, addition of new types at runtime
was not required. Emerging programming environments are virtual execution-based systems supporting reflection [13], e.g., Java & Microsoft’s common language runtime (MS-CLR). Since reflective environments allow addition of new types at runtime, message dispatch requires suitable data structure(s) in which types can be added at runtime.

In this paper, we propose an algorithm for message dispatch in reflective environments. The main advantages of the proposed algorithm are: (i) addition of new types at runtime is possible, (ii) algorithm is more efficient than the existing linear search structure-based algorithm in a reflective environment and, more importantly, (iii) algorithm supports type-safe approach; type safety is an important feature of OOP. The proposed algorithm is a tree-based search at each argument level. It requires \( \log(n) \) times the number of polymorphic arguments time for multimethod lookup, where \( n \) is the number of types in the system. The proposed algorithm is a type safe approach to multimethod lookup resolving ambiguity if more than one applicable methods are implemented. We compare proposed algorithm with the existing algorithms implemented in Java and MS-CLR in respect to efficiency and type safety.

The remainder of this paper is organized as follows. In Section 2, we briefly review techniques of multimethods and reflection. Existing dynamic dispatch algorithms for non-reflective and reflective environments are discussed in Section 3. We describe the proposed tree-based algorithm in Section 4. Results are presented in Section 5. Finally, we conclude in Section 6.

2. Preliminaries

2.1. Notations

In this subsection, we explain notations used throughout in this paper. For this, we consider a class \( \text{Shape} \) and the corresponding hierarchy as shown in Fig. 1. The corresponding method names associated with each class are shown in the figure. Method names are abbreviated as \( i\langle X_1\rangle\langle X_2\rangle \), where \( i \) denotes method name ‘intersect’ and \( X_i \) denotes argument type. For example, ‘\( i\langle R\rangle\langle R\rangle \)’ represents method ‘intersect \((\text{Rectangle},\text{Rectangle})\)’. Methods are encapsulated in the class corresponding to their first argument. Class \( \text{Shape} \) is an abstract class, hence, this is not considered while building the dispatch tree.

2.2. Multimethods

In object-oriented languages, every task is performed by sending a message to an object called receiver. Receiver bears responsibility to perform action by invoking an appropriate method to serve the request. The (target) method invoked depends on the runtime type of the receiver. Other arguments, if any, do not participate in method lookup. Such message dispatch, where only one argument participates in method lookup is called single dispatch and languages supporting single dispatch are called single dispatch languages. For example, C++, Java & C# are single dispatch languages.
Method lookup on more than one arguments is required for both system and application programs. For example, addition of two integers is different from addition of two floating point numbers or addition of an integer and a floating point number. Thus, for adding two numbers target method depends on runtime types of both operands or arguments. Dispatching on more than one argument is essential for many other applications too. Message dispatch on more than one argument is called \textit{multiple dispatch}. Languages supporting multiple dispatch are called multiple dispatch languages, and methods with more than one polymorphic arguments are called \textit{multimethods}. For example, CLOS [14] and Cecil [3] are multiple dispatch languages.

Most object-oriented languages are single dispatch languages, therefore, multimethods can be simulated in these languages [10,15]. For example, preprocessor like Cmm [8] replaces keywords with message dispatch routine in a program to realize multimethods. In object-oriented multiple dispatch language, the compiler adds dispatch routine in the program. Listing 1 is an example of a program fragment to intersect two shapes in a multiple dispatch language.

abstract class Shape
private members:
   center : Point
end class

class Rectangle parent Shape
private members:
   upper_left, lower_right : Point
end class

class Circle parent Shape
private members:
   radius : real
end class

// Multimethods
void intersect (r1 : Rectangle, r2 : Rectangle){
   /* Code to intersect rectangle with a rectangle */
}
void intersect (r1 : Rectangle, c2 : Circle){
   /* Code to intersect circle with a rectangle */
}
void intersect (c1 : Circle, r2 : Rectangle){
   /* Code to intersect rectangle with a circle */
}
void intersect (c1 : Circle, c2 : Circle){
   /* Code to intersect circle with a circle */
}

Code Listing 1. Intersecting two shapes in a multiple dispatch language.

Each method \texttt{intersect()} implements the code for intersecting two shapes as specified by the arguments. In a multiple dispatch language, a programmer will call \texttt{intersect(s1, s2)} (\texttt{s1} and \texttt{s2} are shapes) and appropriate target method will be invoked depending on the instances held by shapes \texttt{s1} and \texttt{s2}. See Kumar et al. [10] for details.

Dispatch or virtual tables in traditional languages like C++ [1] are built at compile time as all classes and their structures are available at compile time. However, in programming environment supporting reflection, new classes can be added at runtime, thus, necessitating support at runtime.

2.3. \textit{Reflection}

A \textit{reflective computational system} [13,16] is one in which otherwise implicit aspects of the system’s structure and behavior are available for explicit inspection and manipulation. A reflective system is able to reason about itself in terms of its structure and computation. Thus, a reflective system needs embedding of itself within it. The reflective environment maintains structure and the computation of a program and allows its inspection and manipulation by the program itself. In other words, programs are allowed to change structure and computation of itself at runtime. For example, definition and instantiation of new data types at runtime that may not be known at compile time, may be needed.
Reflection is a behavior where computation is performed about the computation itself in a casually connected way. The casual connection [17] between two systems describes a link between them such that change in one leads to a corresponding change in another. For example, changing semantic in reflective environment will lead to changed computation in the application program.

In reflective environment, data about program (called, metadata) are maintained by an external agency. In object-oriented languages, metadata can be represented using metaobjects [18]; metaobjects are instances of metaclass (a class for defining other classes). Every entity in the program has corresponding metaobject containing information about it. The representation of program entities in the form of metaobjects is called reification [19]. The external agencies are interpreters or virtual machines executing the program. These external agencies are called reflective processor program (RPP) [16].

Types of reflection, depending on the type of accesses allowed, are:

1. **Introspection**: Program can access its structure. Data types are available for inspection only. Information about types and structure of types can be obtained at runtime.
   
   *Example*: In Java,
   ```java
   (object).getClass().getName();
   ```
   – Get name of a class at runtime.

2. **Structural**: Structural reflection allows a program to alter its own structure in terms of memory layout. RPP or interpreter has complete access to memory layout of a program in the underlying levels [20].
   
   *Example*: In Java,
   ```java
   Class c = Class.forName((name));
   ```
   – Add a new class at runtime.

3. **Computational**: Computation reflection is the activity performed by a computation system when doing computation about its own computations [17]. The computational or behavioral reflection allows program to change its own interpretation [21]. As computational reflection is not available in Java, therefore, we have taken an example from a reflective system called nitrO [22] which supports computational reflection.
   
   *Example*: In nitrO,
   ```java
   assignment.opt[0].actions.append(SemanticAction((code)));
   ```
   – Appending a new action to assignment operation in the language semantics.

### 2.4. Message dispatch in reflective environments

Reflective environments allow addition of new types during runtime. New classes can be added using reflection API. For example, in Java, a new class can be added at runtime by `Class.forName((classname))` API call. Thus, a data structure for message dispatch [1] requires updating at runtime. Most of the message dispatch algorithms [4–7] use data structures in the form of compact or compressed dispatch tables. Updating in many such dispatch structures is not straightforward. Hence, such algorithms cannot be used, in their original form, in reflective environments. Multiple dispatch is required in reflective environments irrespective of the support extended by a language syntax. New data types and methods can be added at runtime in reflective environments. These types and methods are referred by name or a unique id. Thus, reflection API is provided by the system to invoke such methods referred by name, called named method invocation (NMI). However, NMI needs method lookup on runtime type of all arguments i.e., NMI simulates multimethods [10].

For example, we consider Shape class hierarchy (Fig. 1). Multimethod for intersect can be realized in Java using NMI as shown in Listing 2. Methods intersect implement code to intersect different shapes. Invocation of NMI in turn invokes appropriate target method depending on runtime type of the polymorphic arguments. Thus, in NMI API target method depends on the runtime type of the arguments.

```java
public Shape intersect (Rectangle r1, Ellipse e2){
  /* Code to intersect Rectangle r1 and Ellipse e2 */
}
public Shape intersect (Rectangle r1, Rectangle r2){
  /* Code to intersect Rectangle r1 and Rectangle r2 */
}
```
public Shape intersect (Rectangle r1, Circle c2) {
    /* Code to intersect Rectangle r1 and Circle c2 */
}

public Shape intersect (Square sq1, Ellipse e2) {
    /* Code to intersect Square sq1 and Ellipse e2 */
}

... ...

/* Method lookup for shapes s1 and s2 */
try{
    Target_Method = s1.getClass().getMethod("intersectMM",
        new Class[]{s1.getClass(),s2.getClass()});
}catch(NoSuchMethodException e){
    /* Code to handle method lookup fail exception */
}

/* Named method invocation (multimethod call) */
try{
    Target_Method.invoke(s1, new Object[]{s1,s2});
}catch(IllegalArgumentException e){
    /* Code to handle illegal access exception */
}catch(InvocationTargetException e){
    /* Code to handle error in invoking target method */
}


3. Multiple dispatch: a review

3.1. Static-tree-based algorithms in non-reflective environment

Tree-based techniques for multiple dispatch [4,6,11,12] build a dispatch tree or a directed acyclic graph (DAG) at compile time. Every node of a dispatch tree acts as a multiway switch. Depending on the types of arguments, a path is followed in the dispatch tree to obtain target method as shown in Fig. 2. These trees are built by reordering arguments or compressing it depending on some heuristics [4,11,12]. These optimizations give an efficient lookup, but prohibit updating at runtime. Hence, these algorithms cannot be used in reflective environments with optimization. Also, these algorithms do not give details for performing multiway switching at each node.

For example, Fig. 2 is a static dispatch tree for Shape hierarchy. Information about all classes is needed at compile time for building the static dispatch tree. Each edge is labelled with a type and a node with an index of argument on which decision about the path will be taken. At root node decision will be taken on the type of the first argument, hence this is labelled with ‘1’. A path from root to leaf gives a combination of types of arguments with leaf as target method for the combination. Thus, target methods are added for all combinations of types. Method lookup is performed by following a path representing a given combination of arguments’ runtime types. Starting from the root node, edge labelled with type of argument corresponding to label (index) of the node is followed to get the next node. For target method of intersect(Square, Ellipse), at root node edge labelled with ‘Sq’ is followed to get the next node. At next node second argument is considered and edge labelled with ‘E’ is followed. Thus, this path gives target method i_R_E.

3.2. Multiple dispatch in reflective environment

3.2.1. Virtual table in Java

Virtual table is a known technique for single dispatch in non-reflective programming environments, for example, C++ [1]. A virtual table is associated with each class and every instance of that class contains a reference to the virtual table. Virtual table contains addresses of all the member methods of the associated class. This technique is extended and implemented in reflective environments of Java [23] and MS-CLR [24].
Fig. 2. An example of tree-based multiple dispatching.

Fig. 3. An example of a virtual table.

Addresses of all methods are stored in a virtual table. Method lookup for NMI is performed by strict type checking; no polymorphism is considered for the multiple lookup. If more than one method exists with same arguments, method with the most specific return type is selected as a target method. Thus, in Java 'message-not-understood' errors are raised even if some applicable method with supertype argument(s) is present.

3.2.2. Virtual table in MS-CLR

In reflective environment of MS-CLR, virtual tables are built at runtime. When a new class is added to the reflective system, a corresponding virtual table is also built using virtual table of the corresponding superclass. First, all methods implemented by the class are added to the virtual table followed by inherited methods from the virtual table of the superclass. Thus, virtual table technique gives a dynamic data structure for message dispatch in reflective environments.
Since, a method can be referred by its name, method information is stored in the virtual table hashed by its name, as shown in Fig. 3. For single dispatch, target method is to be found by a simple lookup in the virtual table by hashed method name.

Fig. 3 is an example of a virtual table. All methods with same name are stored into the same bucket in an unordered list. All methods \( A \) are stored into one single bucket. Method lookup for \( A(a, b) \) requires first obtaining the list from the hash table followed by searching the most specific method in the list. New methods can be added to the hash table by adding elements in the corresponding list.

**Algorithm for multiple dispatch.** All methods corresponding to a name (or a message) are stored in a single bucket of a virtual table. Hence, a method lookup for multiple dispatch involves finding the most specific method from the list of these candidate methods. A linear search is performed over the list to find the target method. Method lookup in reflective environment of MS-CLR using linear technique is given in Algorithm 1.

### Algorithm 1. Multimethod lookup using linear search

1. **Input:** \( msgname \) - Message Name, \( arg_1 \ldots arg_n \) - arguments passed
2. Get bucket of methods from virtual table for message name \( msgname \).
3. **for each** method \( m \) in bucket do
   4. **for each** argument \( a \) in method \( m \) do
      5. if \( arg_i \) is assignable to \( a \) then
         6. next argument
      7. else
         8. next method
      9. end if
   10. end for
11. Add method \( m \) to \( match \)
12. end for
13. if \( match \) has no method then
14. print ‘Message not understood’
15. end if
16. if \( match \) has only one method then
17. Invoke method
18. end if
19. if \( match \) has more than one method then
20. Find_most_specific_method from \( match \)
21. end if

### Algorithm 2. Check whether type \( T_1 \) is assignable to type \( T_2 \)

1. **Input:** Types \( T_1 \) and \( T_2 \)
2. if \( T_1 = null \) or \( T_2 = null \) then
3. return false
4. else
5. if \( T_1 = T_2 \) then
6. return true
7. else
8. return \((parentof T_1) \) is assignable to \( T_2 \)
9. end if
10. end if

First step of a method lookup is finding all methods conforming with arguments passed in method invocation. A method signature conforms with arguments passed with the message if and only if number of arguments are same and
each argument of the method signature is a supertype of the corresponding argument sent with the message. These conforming methods are stored in a match list. Any of these methods can be invoked. If match contains only one method then that method is invoked. In case of more than one method in match, the most specific method is invoked. Algorithm 3 finds the most specific method. Most specific method is found by comparing methods with respect to the position of their arguments in the class hierarchy. If an argument of a method occupies a lower position in the class hierarchy than the corresponding argument of another method, former method is more specific. Most specific method is invoked from the match list. If there are more than one most specific method, then an ambiguous-message exception is raised.

Algorithm 3. Find_most_specific_method from match list

1. Input: match list of all matching methods
2. ambig ← false
3. minmethod ← First method in match
4. for each method m in match do
5.   if minmethod > m then
6.     ambig ← false
7.     minmethod ← m
8.   end if
9.   if minmethod = m then
10.  ambig ← true
11. end if
12. end for
13. if ambig = true then
14.  print ‘Ambiguous Message Dispatch’
15. else
16.  Invoke minmethod
17. end if

This algorithm is inefficient for larger class hierarchies as this uses linear search. Also operations isassignableto and Find_most_specific_method require traversal of the class hierarchy.

4. Proposed tree-based approach for reflection

4.1. Dispatch tree

The proposed data structure is a dispatch tree consisting of multiple balanced binary search tree (BSTs). A balanced BST is implemented for each polymorphic argument. These BSTs are called argument level subtrees (ALS). Each node in an argument level subtree points to an argument level subtree of the next polymorphic argument. Nodes in ALSs of the last polymorphic arguments point to the applicable target method. Thus, method lookup includes finding an appropriate node in subtrees for the instance held by the passed argument and advancing to next level subtree. Therefore, ALSs act as multiway switches for tree-based dispatch. Each node of Fig. 1 is a ALS in the proposed technique.

A node of a dispatch tree or an ALS can be defined as follows:

```plaintext
struct ALS {
  String strClassName; // Class name for comparison or ordering
  // Children - represented by solid lines in figures
  PtrToALS LeftChild; // Left child
  PtrToALS RightChild; // Right child
  // Pointer to next ALS or target method
```

This algorithm is inefficient for larger class hierarchies as this uses linear search. Also operations isassignableto and Find_most_specific_method require traversal of the class hierarchy.
// Represented by dashed line in Figures.
union {
    PtrToALS NextALS;
    Method TargetMethod;
} Data;

Code Listing 3. Node of an ALS.

In the above structure, LeftChild and RightChild represent left and right children, respectively, of the node in an ALS. These pointers are used to search an ALS for the runtime type of the polymorphic argument. When the required node is found in the ALS, field Data of the structure is used to obtain next argument level subtree or the applicable target method.

Consider an example of class Shape in a reflective system (class hierarchy is given in Fig. 1). Fig. 5 shows an example of a dispatch tree for multimethod intersect(Shape, Shape). Bold lines represent path traversed to find target method for intersect(Circle, Rectangle) (similarly, in all the subsequent figures, bold lines represent the traversed path to find the target method(s)).

In the proposed dispatch tree, new classes and methods can be added by adding new nodes to the existing argument level subtrees and new subtrees, if required. For addition of a new method, a path is created for the new target method by adding new nodes and edges. All possible paths may not be available at initialization time. Initially paths are added only for the argument types of all target methods, but not for their subtypes. Such paths are added at runtime if a message is dispatched with subclasses and a target method with superclass arguments is being invoked due to polymorphism. Adding such paths, when required, increases efficiency of the lookup for all subsequent message dispatches with the same arguments. Algorithm 4 illustrates steps for adding a new type in the reflective environment.

Algorithm 4. Adding a new class to dispatch data structures

1. Input: Class c
2. if isLoaded(c) then
3. return
4. end if
5. loadClass(parentOf c)
6. for each method m in class c do
7. dispatch_tree ← getTree(m)
8. if dispatch_tree = null then
9. dispatch_tree ← newTree(m)
10. end if
11. ALS ← dispatch_tree // ALS - argument level subtree
12. for each argument a in method m do
13. loadClass(a)
14. tmp ← Search argument level subtree ALS for type a
15. if tmp = null then
16. tmp ← createNewBST()
17. CreateEntry in subtree ALS for type a with data tmp
18. end if
19. ALS ← tmp
20. end for
21. CreateEntry in subtree ALS for type a with method m
22. end for
For example, Fig. 4 represents creation of the dispatch tree for two classes Rectangle and Circle. Tree is created in the following steps:

**Step 1:** Class Rectangle starts loading in empty environment. Entry for method \( i\_R\_R \) is added. An argument level subtree ‘ALS1-1’ is created with an node \( R \) in it. ‘ALS1-1’ corresponds to root node of Fig. 2. Reference in Data field of node \( R \) in ‘ALS1-1’ gives reference to next argument level subtree for the second argument.

**Step 2:** In this step, argument level subtree ‘ALS2-1’ is added to the dispatch tree. A node \( R \) is added to ‘ALS2-1’ with reference to target method \( i\_R\_R \) in Data field. Reference from node \( R \) of ‘ALS1-1’ corresponds to an edge labelled \( R \) from the root node in Fig. 2.

**Step 3:** For adding method \( i\_R\_C \), a reference to ‘ALS2-1’ is obtained from node \( R \) of ‘ALS1-1’. A new node \( C \) is added to ‘ALS2-1’ with reference to target method \( i\_R\_C \) in Data field.

**Step 4:** Class Circle is loaded from this step. A new node \( C \) is created in ‘ALS1-1’ for the new first argument type Circle.

**Step 5:** A new argument level subtree ‘ALS2-2’ is created. Reference from node \( C \) of ‘ALS1-1’ points to ‘ALS2-2’. A node \( C \) is created with reference to target method \( i\_C\_C \).

**Step 6:** For adding method \( i\_C\_R \), a reference of ‘ALS2-2’ is obtained from node \( C \) of ‘ALS1-1’. A new node \( R \) is added to ‘ALS2-2’ with reference to target method \( i\_C\_R \).
4.2. Message dispatch algorithm

Multimethod lookup is trivial with a dispatch tree as built in the previous subsection. For each polymorphic argument, in turn, a search is performed in an argument level subtree to get the reference to a new argument level subtree, and to find a path for the target method in the dispatch tree. Nodes of the subtree corresponding to the last polymorphic argument give reference to target methods. For NULL target method message-not-understood exception is raised. If a path is not present in the tree then the target method for supertypes of the arguments is searched by back-tracking. A new path is created for the arguments passed with message. If a target method for a superclass is found, then this is added as target method of the new path else NULL is added to the target method of the new path to represent message-not-understood clause. Algorithm 5 illustrates the steps to method-lookup using a dispatch tree. Steps for creating a new entry in dispatch tree is given by Algorithm 6.

Algorithm 5. Multimethod lookup using Tree-Based Search

1. Input: msgname - Message Name, arg1...argn - passed arguments
2. Get root of dispatch tree for msgname
3. next ← root
4. for each parameter argi do
5. next ← Search argument level subtree next for type argi
6. if next = null then
7. print 'Message not understood'
8. end if
9. if next = 'entry not found' then
10. next ← CreateEntry in subtree next for type argi
11. end if
12. end for
13. Invoke method in next

Algorithm 6. CreateEntry in argument level subtree

1. Input: tree - argument level subtree, a - argument type
2. if a = null then
3. return null
4. end if
5. next ← Search argument level subtree tree for parentof a
6. if next = null then
7. return null
8. end if
9. if next = 'entry not found' then
10. next ← CreateEntry in subtree next for parentof a
11. end if
12. Create new branch in tree for type a and target next
13. return next

In Fig. 5, i_C_R is a target method for intersect(Circle, Rectangle). Method-lookup includes the following steps:

1. Get reference to the dispatch tree for intersect message.
2. Search argument level subtree ‘ALS1-1’ for first argument type Circle and get reference to next subtree i.e., argument level subtree ‘ALS2-2’ from its Data field.
Shape is an abstract class, hence not included in dispatch tree.

Fig. 5. An example of a dispatch tree.

Fig. 6. New path added to the dispatch tree.
3. Search argument level subtree ‘ALS2-2’ for runtime type of the second argument Rectangle to get reference to target method \(_i\_R\_C\).

Paths for all possible combinations of types are not added in the dispatch tree while loading classes. Such paths are created on demand at runtime. For example, consider \texttt{intersect(Rectangle, Square)} for dispatch tree in Fig. 5. Path for method \texttt{intersect(Rectangle, Square)} is not present in the dispatch tree and the most specific target method is \(_i\_R\_R\). Hence, a new path is added when \texttt{intersect(Rectangle, Square)} is invoked for the first time in the program as shown in Fig. 6. For method lookup, reference of ‘ALS2-3’ is obtained from node \(R\) of ‘ALS1-1’. As ‘ALS2-3’ does not contain node \(S\), method for superclass of Square i.e., Rectangle is searched. Hence, method \(_i\_R\_R\) is the target method for \texttt{intersect(Rectangle, Square)}. A new node \(S\) is added in ‘ALS2-3’ with reference to method \(_i\_R\_R\). Node and edges in gray color signify the node and links added on demand.

All subsequent message dispatches with Rectangle and Square will find a path to target method with increasing lookup efficiency.

5. Experimental results

We evaluate performance of the algorithms by varying various parameters, specific to inheritance and polymorphism, namely, height and width of the class hierarchy, number of polymorphic arguments and number of multimethod calls. For simulation, we considered approximately 350 programs of varying dimensions. Number of classes in a program is equal to height \(\times\) width of the class hierarchy.

\textit{Height of class hierarchy}. Height of class hierarchy has significant effect on performance. Number of candidate methods increases with height of the class hierarchy. Also, time required for execution of \texttt{isassignableto} operation increases with height of class hierarchy. Thus, linear search algorithm shows a polynomial increase in lookup time with increase in height of class hierarchy, as shown in Fig. 7. Whereas, tree-based algorithm shows logarithmic increase in lookup time with increase in height. Tree-based algorithm takes \(\log(n)\) time for each polymorphic argument, where \(n\) is the number of classes in the program. Increase in height of class hierarchy leads to increase in number of classes.

\textit{Width of class hierarchy}. Increase in the width of a class hierarchy does not show significant effect on lookup time in either cases. Adding a new branch does not affect number of candidate methods for linear search. For tree-based approach, number of classes gets increased but practically it leads to an increase in number of argument level subtrees. Increase in width does not increase number of nodes in an argument level subtree significantly. Hence, lookup time almost remains the same. The nature of plots found in simulation is linear for both cases (Fig. 8). Execution time
increases linearly due to the time required to add new classes in the system. Number of classes in the system increases with the width of class hierarchy. Also, time required for addition of class is lesser in tree-based approach.

*Number of polymorphic arguments.* Increase in number of polymorphic arguments leads to an exponential increase in number of candidate methods. Lookup time in linear search depends on number of candidate methods. Hence, lookup time increases exponentially with increase in number of polymorphic arguments in multimethods. In tree-based approach, lookup time depends linearly on number of arguments. This is empirically observed and shown in Fig. 9.

*Number of multimethod calls.* In both cases, execution time increases linearly with number of multimethod calls as shown in Fig. 10. Lookup time for individual multiple dispatch is independent of the number of dispatches.
5.1. Polymorphism and type safety

Proposed algorithm uses polymorphism for method lookup. If a method is not found for a given combination of argument types then methods with the corresponding supertype(s) is invoked. Static-tree-based algorithms also implement polymorphic lookup. Multiple lookup in MS-CLR is polymorphic, however, multiple lookup in Java is strictly-type checked.

Let \( C \) denote a class and \( \preceq \) denotes inheritance relation and partial ordering between classes such that \( C_1 \preceq C_2 \) represents \( C_1 \) is a subclass of \( C_2 \). \( \preceq \) relation is a reflexive and transitive relation. A multimethod can be defined as \( \text{msg}(C_1, C_2, \ldots, C_n) \), where \( \text{msg} \) is name of message and \( C_1, \ldots, C_n \) are polymorphic arguments [12]. Partial ordering \( \preceq_{\text{mmeth}} \) between multimethods is defined as follows:

\[
m_1(C_{11}, C_{12}, \ldots, C_{1n}) \preceq_{\text{mmeth}} m_2(C_{21}, C_{22}, \ldots, C_{2n})
\]

iff \( m_1 \) and \( m_2 \) have same name and

\[
C_{1i} \preceq C_{2i} \forall i
\]

or

\[
C_{1j} = C_{2j} \text{ and } C_{1i} \preceq C_{2i} \forall j < i \land 1 \leq i < n
\]

Polymorphism. In the proposed algorithm, for a multimethod \( m(C_1, \ldots, C_n) \) all methods with same arguments or supertypes of arguments are the candidate methods. Most specific method is invoked from candidate methods. All methods \( n(A_1, \ldots, A_n) \) such that \( n \preceq m \) are candidate methods. Hence, methods are not strictly typed checked for multimethod lookup.

Type safety: Multimethod lookup by the proposed method is type safe as arguments of all candidate methods can accept passed arguments. When more than one most specific methods are present in the system for any multimethod call, existing algorithms raise exception of \textit{ambiguous-call}. This ambiguity is resolved by the proposed algorithm by adding an additional condition \( C_{1j} = C_{2j} \) and \( C_{1i} \preceq C_{2i} \forall j < i \land 1 \leq i < n \) to relation \( \preceq_{\text{mmeth}} \). Resolving ambiguity by adding this condition allows users to execute programs type-safely without any additional method required by other algorithms for resolving ambiguity. It also allows to predict the target method invoked at runtime which is not possible if a method is chosen arbitrarily. The following example demonstrates resolution of ambiguity in the proposed algorithm.
Consider an example with two class hierarchies included in Fig. 11, one encapsulating the concept of different shapes (as shown in previous examples), and the other encapsulating the concept of graphics libraries which implement different algorithms to intersect two shapes.

The following code represents a program fragment which intersects shapes using a graphics library, in some assumed multiple dispatch language.

```java
abstract class GraphLib
    private methods:
    intersect (r1 : Rectangle, e2 : Ellipse);
    intersect (r1 : Rectangle, r2 : Rectangle);
    intersect (r1 : Rectangle, c2 : Circle);
    intersect (sq1 : Square, e2 : Ellipse);
    intersect (sq1 : Square, sq2 : Square);
    intersect (e1 : Ellipse, r2 : Rectangle);
    intersect (e1 : Ellipse, e2 : Ellipse);
    intersect (c1 : Circle, r2 : Rectangle);
    intersect (c1 : Circle, c2 : Circle);
    intersect (c1 : Circle, sq2 : Square);
end class

class MyGL1 parent GraphLib
    private methods:
    // ... other methods defined here...
    intersect (r1 : Rectangle, r2 : Rectangle) {
        // Code to intersect rectangle with rectangle
        // using algorithm AlgRR1
    }
    intersect (r1 : Rectangle, c2 : Circle) {
        // Code to intersect rectangle with circle
        // using algorithm AlgRC1
    }
    // ... other methods ...
end class
```

Fig. 11. Class hierarchies for graphics libraries and shapes.
Consider the specific multimethod call intersect(MyGL1, Square, Circle). Method lookups for the given multimethod call in Java, MS-CLR and the proposed algorithm, respectively, are as follows:

**Java**: Java performs type strict multimethod lookup. As method intersect(Square, Circle) does not exist in class MyGL1, given multimethod call is message-not-understood error in Java.

**MS-CLR**: In MS-CLR, Algorithm 1 raises ambiguous-call exception with two most specific methods $m_1 \equiv \text{intersect(MyGL1, Rectangle, Circle)}$ and $m_2 \equiv \text{intersect(MyGL1, Square, Ellipse)}$ for the given multimethod invocation.

In above two multimethods, second argument makes multimethod $m_1$ more specific than $m_2$. But for the third argument $m_2$ is more specific than $m_1$. Since both conditions cannot hold simultaneously, in Algorithm 3 both methods are considered at the same level ($m_1 = m_2$). Two most specific methods exist for invocation. Thus, in MS-CLR, the given multimethod call is an ambiguous call.

**Proposed algorithm**: In proposed Algorithm 5, $\leq_{\text{mmeth}}$ defines partial ordering with condition $C_{1j} = C_{2j}$ and $C_{1i} \preceq C_{2i}$ $\forall j < i \land 1 \leq i < n$. This condition gives $m_2 \preceq m_1$, as MyGL1 $= \text{MyGL1}$ and Square $\preceq$ Rectangle. Hence, ambiguity is resolved by considering a method with more specific argument at a lower index position. Thus, target method for the given multimethod call by the proposed algorithm is intersect(MyGL1, Square, Ellipse).

The comparison of the number of method-not-understood errors in Java, MS-CLR and the proposed algorithm is shown in Fig. 12. Such occurrences are maximal in Java because of its strict-typedness, lesser in MS-CLR, and none by the proposed algorithm for the polymorphic data considered.
Table 1
Comparison between different multiple dispatch algorithms

<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>MS-CLR</th>
<th>Java</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatch table built at</td>
<td>Compile time</td>
<td>Runtime</td>
<td>Runtime</td>
<td>Runtime</td>
</tr>
<tr>
<td>Used in reflective environment</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Polymorphism in multiple dispatch</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Efficient</td>
<td>Inefficient</td>
<td>Inefficient</td>
<td>Efficient</td>
</tr>
</tbody>
</table>

The comparison between virtual table and the proposed algorithm based on parameters is enumerated below:

1. **Time for adding new class**: In virtual table technique, methods defined in the class is simply added to table, but search for existence of a method in the virtual table increases linearly with number of entries. Tree-based algorithm requires \(\log(n)\) time for each polymorphic argument.

2. **Time for multimethod lookup**: In virtual table technique, methods are compared and searched sequentially. Also, operation of finding the most specific method is linear in time. Tree-based approach requires \(\log(n)\) time for each polymorphic argument for multimethod lookup. However, for small hierarchies, the difference is negligible.

3. **Memory space**: Virtual table requires relatively lesser space in comparison to the tree-based technique. In linear search approach, lookup time depends on the number of candidate methods and height of the class hierarchy. Whereas, in tree-based approach lookup depends on number of classes and number of polymorphic arguments. Tree-based search performs better than virtual table-based linear search for multimethods in reflective environments. For very small hierarchies with very less number of multimethods, virtual method techniques give better result than tree-based search. But for large hierarchies and more multimethods, proposed tree-based algorithm performs superior to the existing techniques. Even with a hierarchy of height of 5, number of multimethods equals to 20, and with 200 multimethod calls in a program, tree-based algorithm takes approximately 10% lesser time than linear search. This factor increases with increase in number of multimethods in the system.

Static-tree-based approaches build tree at compile time and provide a better lookup efficiency. But, addition of classes at runtime is inefficient in such trees. Proposed algorithm allows addition of new classes and methods at runtime. Hence, proposed algorithm is more suitable for reflective environments.

We summarize in Table 1 the features of multiple dispatch algorithms.

6. **Conclusions**

Multiple dispatch in reflective environments requires dispatch tables/trees that can be updated at runtime. Most of the proposed algorithms for multiple dispatch build dispatch tree at runtime using some heuristic. Hence, creation and updating of such trees are expensive at runtime. Also, virtual table implemented in reflective environments of Java and C# provide inefficient linear lookup. In this paper, we proposed a tree-based algorithm for efficient multiple dispatch in reflective environments. We discussed techniques of creation and updating of the dispatch tree at runtime. The multiple dispatch by proposed algorithm is type-safe as ambiguity is resolved to find the most specific method. Proposed algorithm is compared with the existing algorithms in terms of efficiency and type safety. The algorithm uses balanced binary search tree without any heuristic. Heuristics can be applied to make multiple dispatches in reflective environment still faster; this is an area of active research. Multiple dispatch using suitable data structures that can be updated and searched more efficiently at runtime is another area of future research.

**Acknowledgements**

Authors thankfully acknowledge part of the implementation work done by Anshuman Mishra. A part of this work is supported from a Microsoft project grant.
References

[8] Cmm. C++ with multimethods. ⟨http://www.op59.net/cmm⟩.