INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Efficient Implementation of Multi-Methods through static analysis

Volker Turau* Weimin Chen†

TR-95-053

September 1995

Abstract

Some of the benefits of object-oriented programming such as extensibility and reusability are fundamentally based on inheritance and late binding. Dynamic dispatching is not only time consuming but it also prevents the usage of some optimization techniques such as inlining or interprocedural analysis. The situation is even more severe for languages supporting multi-methods, where dispatching is not only performed based on the type of the receiver, but also based on the types of the arguments. The most efficient way to perform dynamic dispatching is to avoid it as often as possible, without restricting the use of multi-methods. In this paper it is shown how this goal can be achieved through static analysis. We present a technique which discards all method calls which can be statically bound. Furthermore, even if a method cannot be statically bound, we derive information which will at run time speed up the dispatching process considerably.

*on leave from: FH Wiesbaden, Fachbereich Informatik, Kurt-Schumacher-Ring 3, 65197 Wiesbaden, Germany, turau@informatik.fh-wiesbaden.de

†ICSI - GMD - Integrated Publication and Information Systems Institute, Dolivostr.15 64293 Darmstadt, Germany, chen@darmstadt.gmd.de

1 Introduction

Some of the benefits of object-oriented programming such as extensibility and reusability are fundamentally based on inheritance and late binding. The price for these techniques is sometimes a significant performance overhead. Dynamic dispatching is not only time consuming but it also prevents the usage of some optimization techniques such as inlining or interprocedural analysis. The situation is even more severe for languages supporting multi-methods, where dispatching is not only performed based on the type of the receiver, but also based on the types of the arguments. Even so there are techniques available to reduce the complexity of multi-method dispatch, they still have the above mentioned disadvantages [1, 2]. It is therefore worth seeking for mechanisms to avoid dynamic dispatching for multi-methods as often as possible without restricting the use of them. This will also be of benefit for languages not supporting multi-methods within the framework of the language, because the technique of specialization used to optimize object-oriented languages in general may lead to the need of multi-methods at an internal level [5].

One of the goals of optimization techniques for multi-methods should be the recognition of all cases where methods can be statically bound. In this paper it is shown how this goal can be achieved for statically typed languages. We present a technique which discards all method calls which can be statically bound. Furthermore, even if a method cannot be statically bound, we derive information which will at run time speed up the dispatching process considerably. This information consists either of a small set of methods among the dispatched method will be selected or of a subset of the arguments, which at run time decide which method to be dispatched. In any case the process of selecting the appropriate method is speeded up.

Research into optimization of dispatching can be divided into two directions: Techniques to determine when a method call can be statically bound and efficient algorithms for performing method selection at run time. While the first direction tries to minimize the number of dynamic dispatches, the second tries to perform the dispatching with small time and space requirements.

Static class analysis tries to narrow down the range of classes of objects stored in a variable. If the class of the receiver of a single dispatched method call or the classes of the arguments of a multi-method call can be determined uniquely, then the method call can be realized by a static function call. In case the receiver's class can be narrowed down to a small set, then the method call can be realized by *type-case* expressions, where each case results in a static function call [6].

Profile-guided receiver class prediction receives its information about classes of receivers by monitoring the execution of a program. This way statistics are gathered and based on these *type-case* expressions are inserted into the code, given preference to classes which occurred more often. Then the program is recompiled [7, 9].

Research into the second direction can be classified into dynamic and static techniques. Dynamic techniques are based on caches and their organization [10, 8]. Static techniques

usually aim at finding data structures which are both time and space efficient. Compressing dispatch tables is one possibility [1].

The above mentioned techniques are available for single and multi dispatched methods, even so things are much more complicated in the second case. In a previous paper we have presented a technique for dispatching based on automata which is suitable for multi-methods [2, 11]. The used lookup automaton realizes dispatch without needing a lot of time and space resources. The space requirements are in no case larger then those reported in [1].

The outline of this paper is as follows. In section 2 we present the overall approach. In section 3 we briefly review the concept of the lookup automaton. In section 4 the techniques to detect invocations which can be bound statically is presented and in section 5 the dynamic dispatching is described.

2 Overview of the compilation process

In the following we describe the overall approach for handling multi-methods at compile time. The compiler constructs for each multi-method a lookup automaton (LUA). This construction is described in the following section. During the translation of the actual code the following steps are performed when a multi-method invocation $m(t_1, \ldots, t_n)$ is encountered.

1. The static types T_i of the parameters t_i are determined. The run time types must be subtypes of these types.
2. Using the LUA for the multi-method m the compiler determines whether there are methods applicable for this invocation. It is also possible to detect ambiguities at this moment.
3. A check is accomplished, whether this call can be bound statically.
4. If the method can not be bound at this time, the compiler determines a start state q_j in the LUA at which simulation starts at run time and the corresponding argument number i.
5. Based on the result of the last step the following alternatives are available:
   (a) A special LUA to be used at run time for that invocation is build.
   (b) A type-case expression is generated to perform method selection at run time.
   (c) The general LUA for method m is used at run time.

The first two steps have been covered in [2]. The aim of this paper is to present an efficient algorithm for the remaining steps. Especially we demonstrate how steps 3 and 4 can be performed using the LUA. Finally we demonstrate how the last step can be realized. The notation of this paper follows [2].

3 The lookup automaton LUA

Let $(\mathcal{T}, \preceq)$ be a type hierarchy possibly with multiple inheritance, in which the types are ordered either globally or by a local type ordering. Furthermore let $\mathcal{M}$ be a set of multi-methods with name m and arity n. For clarity of exposition, the methods in $\mathcal{M}$ are called $m_1, \ldots, m_l$. A method $m_i(T_1^i, \ldots, T_n^i)$ is *applicable* for a method invocation $m(T_1, \ldots, T_n)$, iff

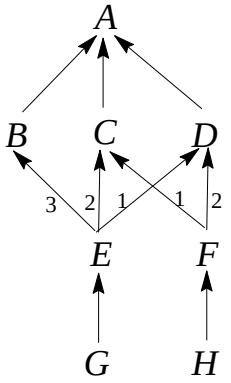
$$T_k \preceq T_k^i, \quad \text{for all } \; k = 1, \ldots n.$$

Let $m_i(T_1^i, \ldots, T_n^i)$ and $m_j(T_1^j, \ldots, T_n^j)$ be two applicable methods for an invocation

$$m(T_1, \ldots, T_n).$$

Then m_i is more *specific* then m_j for this call, if the following holds: let k be the first position in which the types of the formal arguments of m_i and m_j differ, then T_k^i precedes T_k^j (with respect to T_k for local type ordering). Using this definition the applicable types for an invocation are totally ordered and a most specific method exists, if an applicable method exists at all. The task of the dispatcher is to select the most specific method for each invocation or to signal an error in case no method is applicable.

There are other alternative definitions which order the applicable methods only partially. When there are several methods applicable an error *message ambiguous* is signaled [4]. Our approach can also be used for this definition [3]. Figure 1 shows an example we will use in this paper. The type hierarchy consists of 8 types which are ordered using a local type ordering. There are three multi-methods m_1, m_2 and m_3.



$m_1(A, B, B, B)$
$m_2(C, C, B, B)$
$m_3(C, D, A, F)$

Figure 1: A type hierarchy and a set of multi-methods.

A LUA is a deterministic finite automaton $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a set of input symbols, δ is a state transition function, q_0 is the start state and $F \subset Q$ is the set of accepting states. In our case the input symbols are the types of $\mathcal{T}$ and each accepting state is labeled with a particular multi-method. If the dispatcher detects a method call $m(T_1, \ldots, T_n)$, where the T_i are the actual types at run time, then dispatching is performed as follows: simulation of the LUA starts at state q_0 and the argument types are read from left to right. If a state has no transition for a given type, then an error is

signaled, otherwise an accepting state q is reached and the label of q is the most specific method for this invocation.

Simulation starts at the start state with the first argument type. Let $\delta(q, \cdot)$ be the set of labels of the transitions at state q, i.e. the types which do not lead to an error. Let $LUB(T, q)$ be the least upper bound of type T in $\delta(q, \cdot)$. If the simulation is at state q and the next argument type is T, then simulation follows transition $LUB(T, q)$. Figure 2 shows the LUA for the above example. Let $m(E, H, C, F)$ be a method invocation, where E, H, C, F are the run time types. Then $LUB(E, q_0) = C$, so we proceed to state q_2. Then $LUB(H, q_2) = F$, hence we proceed to state q_6. Continuing the procedure we arrive at the accepting state q_{13}. This is labeled with method m_3, hence this method will be dispatched. Note that the construction guarantees, that the LUB's are either unique or do not exist. In the latter case a type error has occurred.

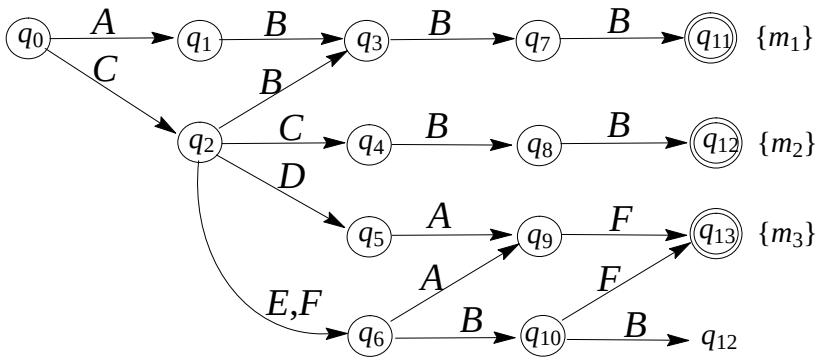


Figure 2: An example LUA.

The LUA can be used for statically and dynamically typed languages. It has several advantages. In [2] it is proven, that the number of states of the LUA is minimal under certain conditions. It can be implemented with standard automaton techniques or by a multi-dimensional array. This is done by collecting the types at each level of the automaton into sets and the entry of the corresponding cell will be the method to be dispatched. In our example it will be a $2 \times 5 \times 2 \times 2$ array. In a 2-dimensional array for each type and position the corresponding set number is recorded. This way dispatching consists of a few array accesses. The size of the array is in no case larger then those constructed in [1].

4 Binding methods statically

In statically typed languages single dispatching can be performed rather efficiently (for example through virtual function tables in C++). Furthermore in some cases it is possible to eliminate dynamic dispatching by binding methods statically. These cases can be found through static analysis. This can also be done for multi-methods as we show in this section.

For statically checking types of method calls, two cases must be distinguished:

1. The compiler accepts method calls, for which there exist at least one method, which is applicable for some subtypes of the arguments.

2. The compiler accepts only method calls, for which there is an applicable method for all subtype combinations of the arguments.

In the first case, there may still be errors at run time (i.e. no method applicable). In the second case there can be no run time errors. For our example an invocation $m(A, A, A, A)$ would be accepted in the first case but not in the second case. Our technique can handle both cases, but as we will see dispatching is easier to handle in the second case.

To understand our approach consider again our example. Suppose that at compile time, it is known that the types of the arguments will be subtypes of C, E, E, E respectively. Then it is straight forward to check with the help of the LUA, that m_2 will always be dispatched. This can be done by checking that each argument (independently of its run time type) selects a unique transition at each state reached. Hence in that case m_2 can be bound statically. More generally, the following observation can be made: A method call $m(T_1, \ldots, T_k)$ can be bound statically iff for all types $T'_1, \ldots, T'_k$ with $T'_i \preceq T_i$ the corresponding invocations lead to the dispatch of the same method. This leads to the following definition.

Definition. A method invocation is called *statically safe*, if all legal run-time arguments will lead to the dispatching of the same method implementation.

In the above example all possible invocations take the same path in the LUA. But there are cases in which an invocation is statically safe, but there is no unique path in the LUA. For example if the types of the formal arguments will be subtypes of C, D, B, F, then consider the actual argument types C, G, B, F and E, D, B, F. In both cases the method m_3 will be dispatched, but during the LUA simulation different paths are chosen.

Our goal was to detect all statically safe method invocations. The LUA can be utilized to perform this task. The principle idea is very simple: to test whether a method invocation $m(T_1, \ldots, T_n)$ is statically safe, simulate the LUA for all possible invocations $m(T'_1, \ldots, T'_n)$, with $T'_i \preceq T_i$ for all $i = 1, \ldots, n$. If they all lead to the same method, then the invocation is clearly statically safe. A brute force approach is to inefficient, since the number of subtype combinations is growing exponentially. We can do better if we exploit the information encoded in the LUA.

The above described procedure corresponds basically to the simulation of the LUA as a nondeterministic finite automaton. If all cases result in the same method, then the invocation is statically safe. There are techniques available for transforming a nondeterministic finite automaton into a deterministic one. But this is to costly at this point. Note that the LUA has more the structure of an acyclic graph, after n transitions an accepting state is always reached. Therefore basic search techniques are more appropriate.

There are basically two approaches: depth first and breadth first. The first possibility has the advantage, that the search can be interrupted as soon as two different methods are encountered. Since the LUA is not a tree, precautions have to be taken, so that no states are unnecessarily processed twice.

Using the breadth first search each level is calculated individually. This way it is easy to guarantee that each state is processed only once. But the search has to proceed to the last level, to decide whether a unique method is found. At first sight this seems to be a possible overhead, but even in the case that the method is not statically safe, dispatching can be made easier.

During the breadth first search we record at each level the number of states. Consider the case in which the states in the last level are not labeled with the same method (i.e. the method is not statically safe). Let k be the last level, where there is only one state q. Then all dispatch paths will pass through this state q. Therefore it is obvious, that the first k arguments are not relevant for dispatching. If $k = n$ then the call is statically safe. If $k < n$ then simulation can start at run time at the unique state q using the argument number $k + 1$. Only if $k = 0$ nothing can be saved.

As an example consider the method invocation $m(C, E, A, A)$. Breadth first search shows, that the call might result in the dispatching of m_2 or m_3. Hence the call is not statically safe. But we note during the breadth first search, that all paths pass through state q_6. The run time types of the first two arguments are not relevant in this case and simulating the LUA can start with the third argument starting at state q_6.

More formally, let Q_0 to Q_n be the sets of states at level $0, \ldots, n$ respectively, which are encountered during the simulation of the LUA as a nondeterministic finite automaton for the method invocation $m(T_1, \ldots, T_n)$. Then

$$Q_0 = \{q_0\} \quad \text{and} \quad Q_i = \bigcup_{q \in Q_{i-1}} \{\delta(q, T) \mid T \preceq T_i\} \quad \text{for } i = 1, \ldots, n.$$

Some calculations can be saved by introducing the set

$$\omega(q, T) = \{\delta(q, T') \mid \exists\, T'' \preceq T \text{ s.t. } T' = LUB(T'', q)\}$$

Then

$$Q_i = \bigcup_{q \in Q_{i-1}} \omega(q, T_i) \quad \text{for } i = 1, \ldots, n.$$

The task is to calculate $\omega(q, T)$. Let $T_q = LUB(T, q)$, $T'' \preceq T$ and $T' = LUB(T'', q)$. Then since $\delta(q, \cdot)$ is glb-closed, we have $T' \preceq T_q$. Hence

$$\omega(q, T) \subseteq \{\delta(q, T') \mid T' \preceq T_q\}.$$

If $T \in \delta(q, \cdot)$, then $T_q = T$ and we have equality in the above formula. So consider the case $T \notin \delta(q, \cdot)$. Then $\omega(q, T)$ can be a proper subset of the second set. In this case it is necessary to check the subtypes of T to see which one does contribute to $\omega(q, T)$.

In order to calculate Q_i, it is necessary to take the union of the $\omega(q, T_i)$ where q ranges over Q_{i-1}. To calculate this union efficiently, a bit-array L_q is build for each relevant state q at level $i - 1$ to represent $\omega(q, T_i)$. The length of L_q is equal to the number of states in the LUA of that level. Then $L_q[j] = 1$ iff the j-th state of that level is in $\omega(q, T_i)$. Now

calculating the Q_i can be performed with binary OR-operations. Note that within each level the arrays have the same length, but they may be of different length at different levels.

Consider the case that a call can not be bound statically. Let max be the largest index such that $|Q_{max}| = 1$, i.e. $|Q_i| > 1$ for all $i > max$. Then $max < n$ and simulating the LUA at run time can start at state q, where $Q_{max} = \{q\}$, with argument number $max + 1$. Hence, if a method call cannot be bound statically, the compiler generates a pair (q, i), indicating that at run time the simulation starts with the i-th argument at state q.

For the above example there are 751 legal dispatch cases. Of those 580 (77%) can be bound statically. For 45 (5%) (for example (C, C, B, B)) dispatching can start with the second argument. Only the remaining 126 cases (for example (A, B, B, B)) the full dynamic dispatching is necessary.

5 Dynamic Dispatch

Even in the case of a statically typed languages not all method invocations can be bound statically. But the standard lookup automaton can be further compressed.

Let us consider first the case where the compiler accepts only method calls, for which there is an applicable method for all subtype combinations of the arguments. In this case some of the states can be pruned. In particular let q be a state from which all paths lead to the same accepting state q_a. Then the state q can be removed and all the transitions going into this state are redirected into q_a. The reason is, that we know that the type checking guarantees, that at these states no errors can occur and therefore the unique transition is chosen. In Figure 2 for example the states q_1, q_3 and q_7 can be removed.

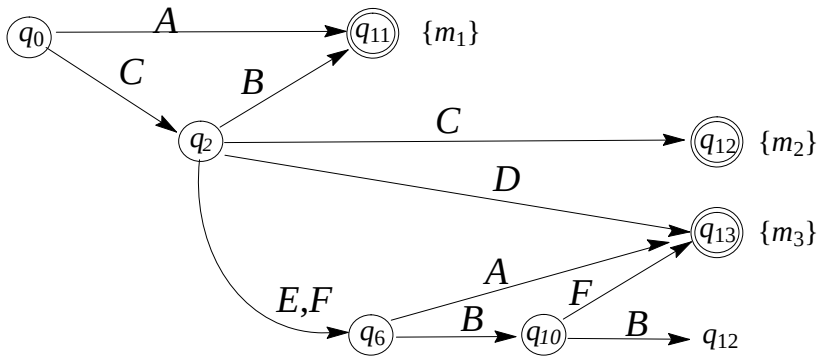


Figure 3: An example LUA.

Removing states is best done from right to left. This way a considerable number of states can be removed. To illustrate this we consider again our example from the first section. Figure 3 shows the pruned LUA for the above example. From the 11 non accepting states 7 could be removed. This compression is also reflected in the implementation based on arrays (a $2 \times 4 \times 2 \times 2$ arrays is sufficient).

If the dynamic LUA is small enough, it can be coded into the source code. In this case the dispatch mechanism is in the code. The same compression can also be used for the first option of type checking. But in this case the run time system must check, whether the run time types of the arguments conform to the argument types of the selected method.

Now consider again the case when a method call can not be bound statically, but simulating the LUA needs not to start at state q_0. In this case there are 2 possibilities:

- Build a special LUA.
- Include the code for dispatching in the compiled code.

Consider again our example and the method call $m(C, E, A, A)$. Then $Q_2 = \{q_6\}$ and $|Q_3| = |Q_4| = 2$. Hence dispatching can start with the third argument at state q_6.

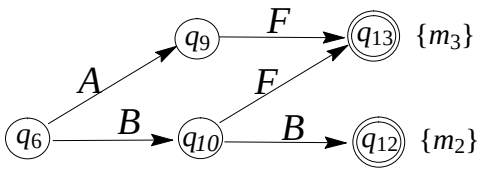


Figure 4: LUA with start state q_6.

The first possibility leads to the LUA shown in Figure 4. Note that pruning can be used for such a LUA.

The code for this LUA is shown in Figure 5 at the left. Here T_i denotes the run time type of i-th argument. At the right of this figure the array implementation is depicted. The top array holds the relation between the types and the sets and the bottom array holds the actual method addresses.

```
if T3 ⪯ B then
    case T4 subtype of
        F:  call m3
        B:  call m2
    else
        ERROR
else
    if T4 ⪯ F then
        call m3
    else
        ERROR
```

	A	B	C	D	E	F	G	H
3	1	2	1	1	2	1	2	1
4	0	2	0	0	2	1	2	1

	1	2
1	m_3	0
2	m_3	m_2

Figure 5: Realization of the LUA of Figure 4.

The code can directly be derived from the corresponding automaton in the following way: at each state q calculate for each transition with label T the set

$$C_T = \{T' \in \mathcal{T} \mid T = LUB(T', q)\}.$$

Note that these sets are disjoint. The different transitions are then realized by type-case expressions based on these sets. Note that $C_T \subseteq SUB(T)$. In case equality holds, the type-case expression can be expressed with the help of the subtype relation. This is done in the above example. Note the invocation $m(C, E, A, A)$ is not legal, if the second option for type checking is used. In general the code for the second option can be derived from from the code fpr the first option by noting, that we can always move one case into the *else*-part.

Generating this code naturally increases the code of an application program, but it also allows other optimization techniques such as interprocedural analysis to be applied.

6 Other related work

The only other work on static analysis for multi-methods that we are aware of is the work of Dean et. al. [6]. Their approach consists of calculating for each multi-method m the so called *applies-to tuples*. This set consists of all tuples of types, for which m will be dispatched. For each method invocation, they calculate a set which consists of all tuples of subtypes of the types of the invocation. Then they calculate the intersection of this set with each applies-to tuple. A method can be statically bound, if all but one intersection are empty. The problem with this approach is, that the calculation of the applies-to tuples and the intersections can be very expensive. The main reason is that the applies-to tuples are given as unions of other sets of tuples. Therefore it is necessary to form intersections of unions. In case a method cannot be bound statically, they do not exploit partial result as we do.

7 Conclusion

In this paper we have presented a new technique to improve the efficiency of dispatching multi-methods. All cases in which a method invocation can be statically bound are detected. Furthermore, even if a method cannot be statically bound, we derive information which will at run time speed up the dispatching process considerably. The algorithms are based on a lookup automaton which we have presented previously. Even in the case of cross product patterns, for which the LUA needs a large amount of space, many methods can be bound statically [2, 1]. The next step is to implement the algorithm and to test real examples to measure the effects and to have means to decide in which cases it is better to realize the LUA by type-case expressions.

References

[1] Amiel, E., Gruber, O. and Simon E. *Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables.* Proc. Conf. OOPSLA, 1994.

[2] Chen, W. and Turau V. *Multiple Dispatching Based On Automata.* J. Theory and Practice of Object Systems 1 (1), 1995.

[3] Chen, W. and Aberer, K. *Efficient Multiple Dispatching Using Nested Transition-Arrays.* Arbeitspapiere der GMD No. 906, Sankt Augustin, March 1995.

[4] Chambers, C. *Object-oriented multi-methods in Cecil.* Proc. ECOOP 1991.

[5] Dean, J., Chambers, C. and Grove, D. *Selective Specialization for Object-Oriented Languages.* Proc. Conf. PLDI, 1995.

[6] Dean, J., Grove, D. and Chambers, C. *Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis.* Proc. ECOOP 1995.

[7] Grove, D., Dean, J., Garrett C. and Chambers, C. *Profile-Guided Receiver Class Prediction.* Proc. Conf. OOPSLA, 1995.

[8] Hölzle, U., Chambers, C. and Ungar, D. *Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches.* Proc. ECOOP 1991.

[9] Hölzle, U. and Ungar, D. *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback.* Proc. Conf. PLDI, 1994.

[10] Kiczales G. and Rodriguez, L. *Efficient Method Dispatch in PCL.* Proc. Conf. on Lisp and Functional Programming, 1990.

[11] Turau, V. and Chen, W. *GLB-Closures in Directed Acyclic Graphs and their Applications.* Proc. Workshop on Graph Theoretic Algorithms in Computer Science, 1994.