Efficient Multiple Dispatching Using Nested Transition-Arrays

Weimin Chen, Karl Aberer

GMD-IPSI, Dolivostr. 15, 64293 Darmstadt, Germany
E-mail: {chen, aberer}@darmstadt.gmd.de
Phone: +49-6151-869941
FAX: +49-6151-869966

Abstract. Efficient implementation of multiple dispatching is critical with regard to the success of the multi-methods, which are recognized as a powerful mechanism in object-oriented programming languages. However, the current known time-efficient approaches suffer of poor space efficiency. This paper presents a novel multiple dispatching approach that substantially improves the space efficiency over the known constant time lookup approaches. The approach we propose is applicable to a wide range of languages supporting locally or partially ordered multi-methods, both statically and dynamically typed.

Keywords: Multi-methods, Dispatching, Automaton.
Category: Research paper, Language implementation.

1 Introduction

Multi-methods have been recognized as a powerful mechanism in object-oriented programming languages, by guiding method lookup using the value of all arguments instead of only the receiver. Dispatching on multi-methods is called *multiple dispatching* [10].

There are two different kinds of multi-methods supported by current languages. One is totally ordered multi-methods. The methods are totally ordered by prioritizing argument position, with earlier argument positions completely dominating later argument positions. Perhaps the best known language supporting this kind of multi-methods is CLOS [8]. The advantage of such a kind of multi-methods is the disambiguity of the method specificity. However, the disadvantage is that the method specificity must rely on the argument positions and programmers may not really need that [10].

The other kind of multi-methods takes another view on the method specificity, where all argument positions have an equal priority to determine the method specificity. It is called partially ordered multi-methods, because such a strategy in general just identifies a partial order with respect to the method specificity. Languages supporting partially ordered multi-methods are Cecil [10, 11] and Key [21]

The above two kinds of multi-methods take different philosophies with respect to the method specificity, each of which has its own advantages and disadvantages. However, efficient implementation of multiple dispatching is still critical with regard to the success of using multi-methods. On the one hand, multiple dispatching needs more arguments to guide the method lookup. This makes it slower than dispatching based just on the traditional single dispatched methods (e.g. C++ and Smalltalk). On the other hand, a serious problem in multiple dispatching is the space requirement. Different method combinations can make the lookup structure to explode exponentially. Several previous work have been trying to resolve these problems [5, 12, 17, 20]. Some are time efficient, and some are space efficient,but never both.

The work of this paper is to provide a novel lookup mechanism for multi-methods. Our scheme absorbs and extends the ideas of previous work on multiple dispatching [5, 12]. The lookup is done by means of several nested transition-array accesses. The transition-arrays are constructed from a lookup automaton and are compressed by means of a grouping technique. Analyses and experiments show that our scheme is both *time* and *space* efficient in comparison to previous techniques. Furthermore, the approach we propose is applicable to a wide range of languages supporting locally or partially ordered multi-methods, both statically and dynamically typed.

The organization of this paper is as follows: Section 2 reviews the previous work on multiple dispatching and compares that to our work. Section 3 identifies the model of object-oriented type system and the ambiguity rule for partially ordered multi-methods. The dispatch approach is depicted in Section 4. Section 5 describes the techniques used to construct the lookup structure and Section 6 presents a technique to further compress the lookup structure. Sections 7 and 8 evaluate the time and space efficiency for our techniques. Finally, we conclude our approach in Section 9.

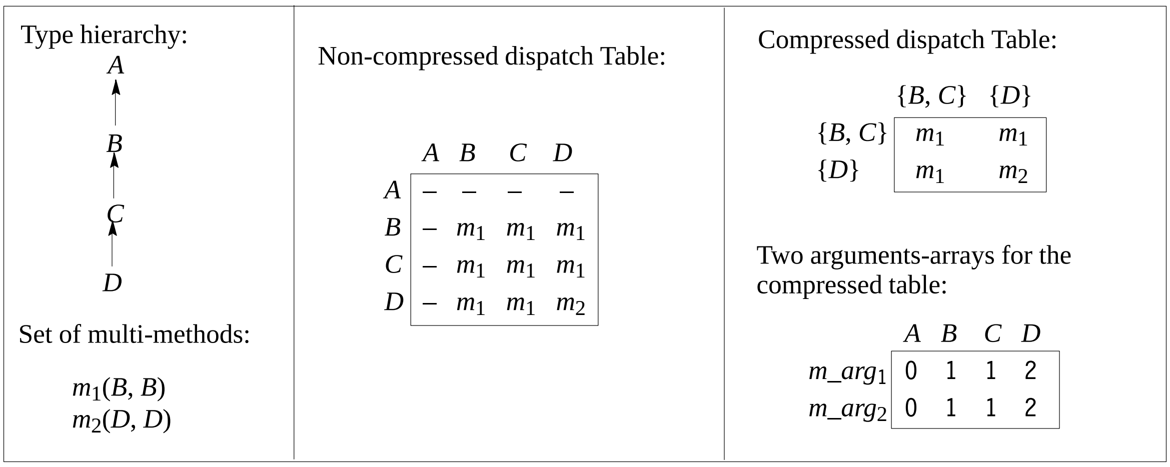
2 Previous Work and Comparison

The first goal of multiple dispatching is time efficiency. Several time efficient approaches have been proposed for multiple dispatching. However, the relevant problem is the large space requirement. For example, in the Flavors system, Moon [20] proposed a lookup structure which is organized as a set of hash tables. With such a scheme, the number of all possible cases is combinatorially explosive. For comparison, Dussud [15] and Kiczales, *et al.* [17] introduced cache techniques for multiple-dispatching running in CLOS. The dynamic cache requires memory only for called methods. At the time of dispatching, the search starts in the cache. However, the problem of combinatorial explosion is not resolved.

Amiel, *et al.* [5] proposed a technique to optimize multi-methods dispatching using compressed dispatch tables to represent all dispatch cases. Initially, all dispatch cases are calculated and represented by tables. For each table, the following compression techniques are employed:

- Eliminating rows or columns containing only null values;
- Grouping identical rows or columns.

In order to access these compressed tables, they suggested to introduce new indices called *argument-array*s: each arguments-array is one-dimensional and indexed by the types. Given a method invocation m, the i^{th} arguments-array of m holds the positions of every type in the i^{th} dimension of the compressed dispatch table of m, i.e. when the type appears as the i^{th} argument. The number 0 indicates that this type cannot appear as the i^{th} argument. Fig. 1 shows an example to illustrate this idea. Thus for a method invocation $m(B, D)$, by means of arguments-arrays m_arg_1 and m_arg_2, the dispatch result can be found in the entry of the compressed table at the $(m\_arg_1[B])^{th}$ row and the $(m\_arg_2[D])^{th}$ column, i.e. $m(B, D)$ is dispatched to m_1. This technique provides


Type hierarchy:

$$\begin{tikzcd} A \\ B \arrow[u] \\ C \arrow[u] \\ D \arrow[u] \end{tikzcd}$$

Set of multi-methods:

$m_1(B, B)$
$m_2(D, D)$

Non-compressed dispatch Table:

	A	B	C	D
A	–	–	–	–
B	–	m_1	m_1	m_1
C	–	m_1	m_1	m_1
D	–	m_1	m_1	m_2

Compressed dispatch Table:

	$\{B, C\}$	$\{D\}$
$\{B, C\}$	m_1	m_1
$\{D\}$	m_1	m_2

Two arguments-arrays for the compressed table:

	A	B	C	D
m_arg_1	0	1	1	2
m_arg_2	0	1	1	2

Fig. 1 An example for compressed dispatch tables

a fast dispatch mechanism; however, the space requirement of the approach can still be very large. The combinatorial explosion problem is still critical with these structures.

In order to reduce the space requirement, in the presence of totally ordered multi-methods (e.g. CLOS and CommonLoops), Chen, *et al.* introduced a lookup automaton structure to implement the method dispatching [12]. With this approach, the space requirement is reduced essentially. The idea is to build an automaton that takes the types of successive arguments as input and whose states record the order of the possible applicable methods, given the type seen so far. In order to reduce the space requirements, the automaton does not list all possible dispatch cases. Consequently, a run-time subtype checking is needed for method lookup. There have been several efficient techniques to speed up the subtype checking (e.g. [1]) and the method lookup can be implemented at constant time. However, the time efficiency of this lookup approach using run-time subtype checking is still lower than the general vector-based approach, like with dispatch tables. On the other hand, the lookup approach described in [12] is just valid for totally ordered multi-methods.

Comparing with the above lookup automaton approach, this paper performs the following improvements:

- Define a new dispatch algebra applicable to both totally and partially ordered multi-methods;
- Based on the dispatch algebra, the lookup automaton lists *all* possible dispatch cases rather than a part, as done in [12]. Consequently, a run-time subtype checking is obsolete, and the time efficiency of the lookup is improved by means of nested transition-array accesses;
- The transition-arrays can be compressed using the grouping technique, which is originally used for dispatching table structure [5].

3 Model

Type Hierarchy and Type Orderings. A type hierarchy, denoted by $\mathcal{T}$, consists of several types for which a subtype (or supertype) relation is defined. In general, the subtype relation forms just a partial ordering and can not resolve the typical problem of multiple-inheritance conflict. Thus, some languages extend the subtype ordering by further ordering the supertypes of a type, e.g. local type ordering (CLOS [8]) and global type ordering (ComonLoops [7]); some languages do not extend that, e.g. Cecil [10, 11] and Key [21]. In any case, we uniformly say that A is *less* than B with respect to C if the following conditions hold:

- A and B are the supertypes of C;
- Either A is a subtype of B, or A precedes B according to the extended order (if defined) among the supertypes of C.

The dispatch techniques presented in this paper are applicable for all type orderings.

Method Applicability. Let

$$\mathcal{M} = \{m_i(T_i^1, ..., T_i^\alpha) \mid 1 \leq i \leq n\}$$

be the set of multi-methods with the same name and the same argument-arity α. It is assumed that the signatures of m_i and m_j are different when $i \neq j$. The method $m_i(T_i^1, ..., T_i^\alpha)$ is *applicable* for a method invocation $m(T^1, ..., T^\alpha)$ if and only if T^k is a subtype of T_i^k, $1 \leq k \leq \alpha$.

Method Specificity. Generally, there may exist more than one method in $\mathcal{M}$ which is applicable to a method invocation. Thus we need to define an order among them. Given methods $m_i(T_i^1, ..., T_i^\alpha)$ and $m_j(T_j^1, ..., T_j^\alpha)$ which are applicable for the invocation $m(T^1, ..., T^\alpha)$, there are two different kinds of multi-methods to identify the specificity between methods m_i and m_j:

- *Totally Ordered Multi-Methods* (TOMM): In a predefined order (such as left-to-right[1]), find the first argument position in which the argument types of m_i and m_j differ, say k. If type T_i^k is less than T_j^k with respect to type T^k, then m_i is more specific than m_j, and vice versa. For example, CLOS supports TOMM;
- *Partially Ordered Multi-Methods* (POMM): m_i is more specific than m_j if and only if, for each argument position k, either T_i^k is less than T_j^k with respect to T^k or T_i^k is equal to T_j^k. This implies that there exists at least one position k such that T_i^k is not equal to T_j^k, because of the assumption that the signatures of m_i and m_j differ. Languages supporting POMM are Cecil [10, 11] and Key [21].

Task of Dispatching. Given a method invocation m, the task of dispatching for TOMM is to calculate the most specific applicable method. For POMM, however, the most specific method may not exist. Uniformly, the task for multiple dispatching is as follows:

(a) Compute the set of applicable methods for the method invocation;
(b) If the set of applicable methods is empty, then the method invocation is not applicable and the error "message not understood" is signalled;
(c) If the set of applicable methods is not empty and this set contains a most specific method, then the most specific method is the dispatch result for the method invocation;

1. Any predefined order can be transformed into left-to-right by exchanging the argument positions during compile time. Hence, without loss of generality, we can assume that the predefined order is left-to-right during dispatching.

	=	<	•	x
=	=	<	•	x
<	<	<	•	x
•	•	•	•	x
x	x	x	x	x

for POMM

	=	<	•	x
=	=	<	•	x
<	<	<	<	x
•	•	•	•	x
x	x	x	x	x

for TOMM

Fig. 2. The multiplication tables of the binary operator ‘&’

(d) If the set of applicable methods is not empty but this set contains no unique most specific method, then the method invocation results in a “message ambiguous” error. This case can happen only for POMM and it is expected that the method ambiguity is resolved by programmers [10].

In a statically typed language, case (b) can be detected at compile-time; in a dynamically typed language, however, case (b) must be detected at run-time in general.

4 Dispatch Algebra

Now we present an algebra to calculate the dispatch result. The overall idea is to identify the method specificity for each pair of multi-methods with regard to a specific position of the argument types. Afterwards, the method specificity for each pair of multi-methods is determined by sequentially going through all positions of the argument types.

Given a type $T \in \mathcal{T}$ and number k, $1 \leq k \leq \alpha$, we define an $n \times n$ matrix

$$\Omega(k, T) = (\omega_{ij})_{n \times n}, \text{ where } \omega_{ij} \in \{\mathrm{x}, \bullet, <, =\}$$

to indicate the specificity for each pair of methods with respect to the kth argument types. The element ω_{ij} in matrix $\Omega(k, T)$ is defined as follows:

- If at least one of T_i^k and T_j^k (the kth argument types of m_i and m_j) is not a supertype of T, let ω_{ij} = ‘x’;
- If both T_i^k and T_j^k are supertypes of T:

  (a) If $T_i^k = T_j^k$, let ω_{ij} = ‘=’;

  (b) If T_i^k is less than T_j^k with respect to type T, let ω_{ij} = ‘<’;

  (c) Let ω_{ij} = ‘•’ in other cases.

Matrix $\Omega(k, T)$ has two properties: (1) If T_i^k is not a supertype of T then all elements in the ith row or the ith column are equal to ‘x’; (2) In the diagonal ω_{ii} is equal to ‘=’ if and only if T_i^k is a supertype of T.

In order to depict the specificity among the methods in $\mathcal{M}$ with respect to *all* positions of argument types, we introduce a binary operator

$$\&: \{\mathrm{x}, \bullet, <, =\} \times \{\mathrm{x}, \bullet, <, =\} \rightarrow \{\mathrm{x}, \bullet, <, =\}.$$

The definition of operator ‘&’ is shown in Fig. 2 and is different for POMM and TOMM.

By means of the operator ‘&’, we now define a binary operation $\wedge$ on matrices $\Omega_1 = \left(\omega_{ij}^1\right)_{n \times n}$ and $\Omega_2 = \left(\omega_{ij}^2\right)_{n \times n}$ as follows:

$$\Omega_1 \wedge \Omega_2 = \left(\omega_{ij}^1 \ \&\ \omega_{ij}^2\right)_{n \times n}, \text{ where } \omega_{ij}^1, \omega_{ij}^2 \in \{\mathtt{x}, \bullet, <, =\}.$$

We now have the following important result for multiple dispatching.

Theorem 1 Given a method invocation $m(T^1, ..., T^\alpha)$, the matrix

$$\Omega(1, T^1) \wedge \Omega(2, T^2) \wedge \ ...\ \wedge \Omega(\alpha, T^\alpha) = (\omega_{ij})_{n \times n} \quad (1)$$

has the following properties:

(a) The set of applicable methods for the invocation $m(T^1, ..., T^\alpha)$ is $\mathcal{A} = \{m_i \in \mathcal{M} \mid \omega_{ii} \neq \text{‘}\mathtt{x}\text{’}\}$;
(b) If $\mathcal{A} = \varnothing$ (i.e. ω_{ij} = ‘x’ for all i and j), the invocation is not understood;
(c) If $\mathcal{A} \neq \varnothing$, consider the set $\mathfrak{D} = \mathcal{A} - \{m_j \in \mathcal{A} \mid \exists i \text{ such that } \omega_{ij} = \text{‘}<\text{’}\}$. If $\mathfrak{D}$ includes only one element (method), then the method is the dispatch result; else the “message ambiguous” error is signalled (this case can happen only for POMM). ∎

The proof of the above theorem is presented in the appendix.

Matrix (1) indicates a way to calculate the dispatch result for a given method invocation. The following dispatch techniques are based on the result of Theorem 1, applicable for both POMM and TOMM.

5 Lookup Structure

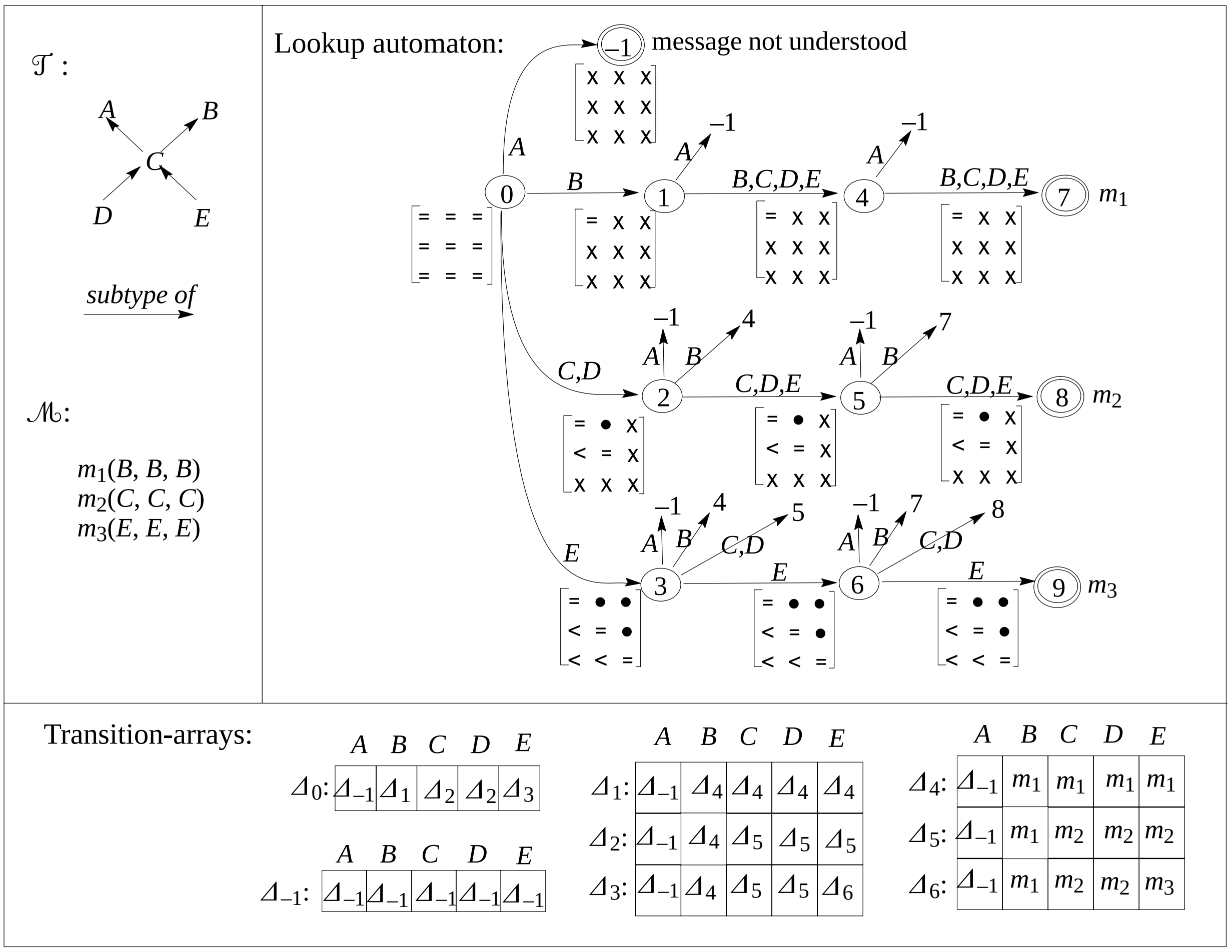
Based on the result of Theorem 1, we introduce a lookup automaton to describe the possible dispatch results. Given a number k, $1 \leq k \leq \alpha$, consider the set of matrices

$$\Omega_k^* = \{\Omega(1, T^1) \wedge \Omega(2, T^2) \wedge \ ...\ \wedge \Omega(k, T^k) \mid (T^1, T^2, ..., T^k) \in \mathcal{T}^k\}.$$

The automaton is constructed with the following characteristics:

- The states of the automaton are grouped into levels from 0 to α. An initial state is at level 0, and final states are at level α;
- There is a one-to-one correspondence between the states at level k ($1 \leq k \leq \alpha$) and the matrices in Ω_k^*. Especially, the initial state (at level 0) corresponds to matrix $(=)_{n \times n}$ (i.e. the matrix $(\omega_{ij})_{n \times n}$ where all ω_{ij} are equal to ‘=’);
- The input symbols for the automaton are the types in $\mathcal{T}$;
- A state, q_i, reaches another state, q_j, on input symbol $T \in \mathcal{T}$ if and only if:
  (1) q_i is at level k, $0 \leq k < \alpha$, and q_j at level $k + 1$;
  (2) $\Omega_i \wedge \Omega(k, T) = \Omega_j$ where matrices Ω_i and Ω_j correspond to q_i and q_j;
- For each final state, the dispatch result is calculated by means of the corresponding matrix and is attached to the state.

By means of this lookup automaton, the dispatch result of a method invocation $m(T^1, ..., T^\alpha)$ can be obtained at the final state which is reached from the initial state on the successive input symbols $T^1, ..., T^\alpha$.



	A	B	C	D	E
$\varDelta_0$:	$\varDelta_{-1}$	$\varDelta_1$	$\varDelta_2$	$\varDelta_2$	$\varDelta_3$
$\varDelta_{-1}$:	$\varDelta_{-1}$	$\varDelta_{-1}$	$\varDelta_{-1}$	$\varDelta_{-1}$	$\varDelta_{-1}$
$\varDelta_1$:	$\varDelta_{-1}$	$\varDelta_4$	$\varDelta_4$	$\varDelta_4$	$\varDelta_4$
$\varDelta_2$:	$\varDelta_{-1}$	$\varDelta_4$	$\varDelta_5$	$\varDelta_5$	$\varDelta_5$
$\varDelta_3$:	$\varDelta_{-1}$	$\varDelta_4$	$\varDelta_5$	$\varDelta_5$	$\varDelta_6$
$\varDelta_4$:	$\varDelta_{-1}$	m_1	m_1	m_1	m_1
$\varDelta_5$:	$\varDelta_{-1}$	m_1	m_2	m_2	m_2
$\varDelta_6$:	$\varDelta_{-1}$	m_1	m_2	m_2	m_3

Fig. 3 An example of type hierarchy 𝒯, a set of multi-methods ℳ (POMM), the corresponding lookup automaton and transition-arrays

Every non-final state corresponding to matrix $(\mathsf{x})_{n\times n}$ (i.e. the matrix $(\omega_{ij})_{n\times n}$ where all ω_{ij} are equal to ‘x’) can only reach a final state corresponding to the same matrix $(\mathsf{x})_{n\times n}$. Thus, the first compression strategy is to merge all states corresponding to matrix $(\mathsf{x})_{n\times n}$ into a final state, say –1, where the dispatch result is “message not understood”.

Fig. 3 shows an example of a lookup automaton. The initial state is 0 and final states are characterized by double circles. In order to avoid drawing cross arrows, the states without circles refer to the corresponding circled states. Note that the matrices are just used at the time of automaton construction and can be deleted after the automaton has been constructed. The detailed algorithm to construct the above automaton is described in the appendix.

Transition-Arrays. Let *succ*(q, T) denote the state reached from the state q on input symbol T, and assume a unique index is attributed to every type in 𝒯, i.e. $\mathcal{T} = \{T_1, T_2, ..., T_u\}$. For each non-final state q there are u state transitions, which can be represented by a transition-array, $\varDelta_q[1 .. u]$, such that $\varDelta_q[j] = succ(q, T_j)$, $1 \leq j \leq u$. For the state –1, a special transition-array $\varDelta_{-1}[1 .. u]$ is introduced such that $\varDelta_{-1}[j] = \varDelta_{-1}$, $1 \leq j \leq u$.

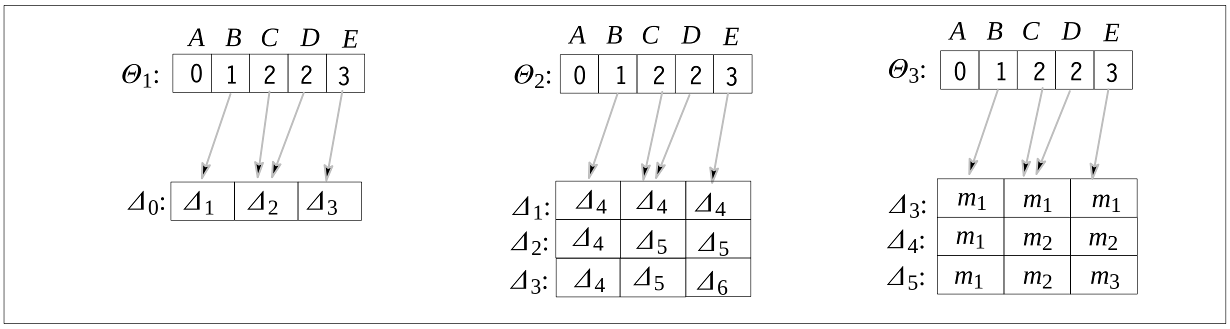

A B C D E
Θ_1: 0 1 2 2 3
Δ_0: Δ_1 Δ_2 Δ_3
Θ_2: 0 1 2 2 3
Δ_1: Δ_4 Δ_4 Δ_4
Δ_2: Δ_4 Δ_5 Δ_5
Δ_3: Δ_4 Δ_5 Δ_6
Θ_3: 0 1 2 2 3
Δ_3: m_1 m_1 m_1
Δ_4: m_1 m_2 m_2
Δ_5: m_1 m_2 m_3

Fig. 4 Grouped transition-arrays

Especially, when $succ(q, T_j)$ is a final state, the corresponding dispatch result is stored in the entry $\Delta_q[j]$. That is, if $succ(q, T_j) = -1$, let $\Delta_i[j] = \Delta_{-1}$; if $succ(q, T_j) \neq -1$, $\Delta_i[j]$ stores the address of the dispatched method (see Fig. 3). This implies that for POMM a predefined method m_{err} must be introduced to deal with message ambiguous error; when this message ambiguous error occurs at a final state, m_{err} is regarded as the dispatched method.

The transition-array Δ_{-1} can be shared by different lookup automata corresponding to different sets $\mathcal{M}_b$.

Dispatch Coding. For every object o, assume $o.typ\_idx$ returns the index of its type. In a statically typed language, the method invocation $m(o_1, o_2, ..., o_\alpha)$ can be translated into the following code:

$meth\_addr = \Delta_0[o_1.typ\_idx][o_2.typ\_idx]...[o_\alpha.typ\_idx]$
call $meth\_addr(o_1, o_2, ..., o_\alpha)$

In a dynamically typed language, however, an additional check is required to ensure that $meth\_addr \neq \Delta_{-1}$; otherwise, the invocation is not understood.

6 Compression of Transition-Arrays

For each level k, $0 \leq k < \alpha$, all transition-arrays (except Δ_{-1}) corresponding to states at level k form a two-dimensional array (i.e. a table), for which the following compression techniques can be used:

- group all identical columns;
- delete the columns that contain only one value—the address of array Δ_{-1}.

Corresponding to this grouping, a new argument-array, $\Theta_k[1\ ..\ u]$, is introduced such that $\Theta_k[i]$ holds the position of the column in the grouped array corresponding to the original column i; if the original column i has been deleted, let $\Theta_k[i] = 0$.

Fig. 4 demonstrates this grouping technique based on the transition-arrays shown in Fig. 3. The transition-array Δ_{-1} does not attend the grouping, because of the assumption that it can be shared by different lookup automata.

Effectiveness of Grouping. The benefit of the grouping depends on the concrete transition-arrays. Assume that an ungrouped table at level k has n_k rows and u columns, so that the table has $u \cdot n_k$ entries. Now assume that the table can be grouped into a new one with n_k rows and p ($p \leq u$) columns. Counting the additional argument-array $\Theta_k[1 .. u]$, the space used in the grouped table and the array Θ_k is $p \cdot n_k + u$. Thus, the grouping is effective if and only if

$$u \cdot n_k > n_k + u. \quad (2)$$

There are two obvious cases where the grouping is not effective: $n_k = 1$ (e.g. when $k = 0$), and $p = u$ (i.e. the table can not be grouped). For example, in Fig. 4 grouping array Δ_0 is not effective but grouping the others is.

Thus, at a given level whether to group the transition-arrays or not depends on its effectiveness, which can be determined by relation (2) at compile-time.

Dispatch Coding. The dispatch code based on the grouped transition-arrays now differs from the code based on the ungrouped arrays. In a statically typed language the method invocation $m(o_1, o_2, ..., o_\alpha)$ can be translated as

$$meth\_addr = \Delta_0[U_1][U_2]...[U_\alpha]$$
$$\text{call } meth\_addr(o_1, o_2, ..., o_\alpha)$$

where U_k is either the code $\Theta_k[o_k.typ\_idx]$ if the transition-arrays at level k have been grouped, or just the code $o_k.typ\_idx$ if not. The concrete codes of all U_k are determined at compile-time.

In a dynamically typed language, a run-time check is necessary to verify that neither $U_k = 0$ nor $meth\_addr = \Delta_{-1}$; otherwise, the invocation is not understood.

7 Time Efficiency

Having presented the techniques for constructing the lookup structure and generating the dispatch code, we evaluate in this section the time efficiency of dispatching using these techniques; we also compare that to the compressed dispatch table structure [5].

The dispatch code based on the transition-arrays is translated into α nested array accesses which take α indirections. Furthermore, by the grouping technique, at most α additional indirections for argument-arrays Θ_k are introduced. Thus, our lookup strategy needs at most 2α and at least α nested one-dimensional array accesses. Comparing with the dispatch table structure, lookup requires one α-dimensional array access plus α one-dimensional argument-arrays accesses for the argument-arrays. At first glance, it seems that less indirections are needed by the dispatch table approach. However, a multi-dimensional array is internally represented by a linear array (either implicitly by a compiler or explicitly by a program), so that an index transformation from the multi-dimensional one into the linear one is needed. For example, suppose a three-dimensional array $\mathfrak{D}_3[1 .. n_1, 1 .. n_2, 1 .. n_3]$ is internally represented by a linear array $\mathfrak{D}_1[1 .. n_1 n_2 n_3]$. The index $[i_1, i_2, i_3]$ in $\mathfrak{D}_3$ is transformed into the index in $\mathfrak{D}_1$ as follows:

$$n_3 \cdot [(i_1 - 1)n_2 + (i_2 - 1)] + i_3.$$

Counting the cost of the index transformation, we can not find an obvious difference between the cost of an α-dimensional array access and the cost of α nested one-dimensional array accesses. We test that on a Sun-Sparc station 2: When $\alpha = 3$ and $n_1 = n_2 = n_3 = 100$, repeating 10^6 times the formal strategy (a three-dimensional array access) needs total system-time 0.150s, the later (three nested one-dimensional array accesses) needs 0.140s.[2] This difference is very small such that the time efficiency for both strategies can be regarded as the same.

On the other hand, for dynamically typed languages, the grouped transition-array approach needs to perform the null-checking for the α indices; the dispatch table approach also needs to perform the same null-checking on the α argument arrays. Both checks take the same time cost. Thus, the time efficiencies for both structures are the same.

8 Space Efficiency

With respect to the space efficiency there is a considerable difference between the transition-array structure and dispatch table structure. In this section we analyze and compare the space costs of common benchmarks for multiple dispatching. Furthermore, several experimental results are given. Overall, for binary multi-methods the difference is not obvious; however, when the method arity is greater than 2 the transition-array structure will be more efficient than dispatch table structure, especially when multi-methods are just *sparsely* defined.

Operator Pattern. The first pattern is with respect to operators. We consider a type hierarchy $\mathcal{T}$, on which the multi-methods are defined with α-ary signatures $(T_i, T_i, ..., T_i)$, $T_i \in \mathcal{T}$. The lookup automaton for this pattern has the following properties:

- At each level k, $1 \leq k \leq a$, there are total n states, so that the automaton has total $(\alpha - 1)n + 1$ non-final states;
- For each non-final state there are n state transitions;
- Grouping the transition-arrays is not effective.

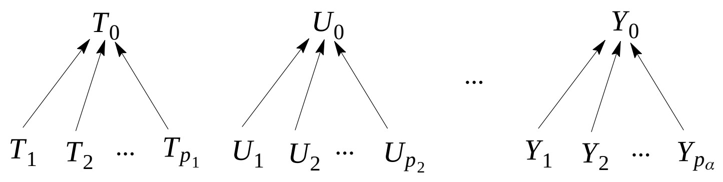
The space cost of the transition-arrays corresponding to this automaton is $n + (\alpha - 1)n^2$. For comparison, the space cost for the compressed dispatch table for this pattern is n^α [5], where the grouping technique is not effective. The following table lists the space costs between these two structures when $n = 100$:

	$\alpha = 2$	$\alpha = 3$	$\alpha = 4$
Transition-arrays	10.1K	20.1K	30.1K
Dispatch Table	10K	1M	100M

The space cost for transition-arrays is essentially reduced in comparison to the space cost for the dispatch table when $\alpha > 2$.

Cross Product Pattern. This is a typical usage of multi-methods. In general, consider a type hierarchy consisting of α separated parts (see Fig. 5) and multi-methods are defined with α-ary signatures $(T_i, U_j, ..., Y_l)$, where all subscripts $i, j, ..., l$ are greater than 0. Furthermore, an additional method is defined on signature $(T_0, U_0, ..., Y_0)$ which plays the role of error-handling function.

2. The testing program is written in C. In fact each time cost listed above includes also the time costs used for a loop-control (from 1 to 10^6), which is same for both strategies.



T_0
U_0
Y_0
...
T_1 T_2 ··· T_{p_1}
U_1 U_2 ··· U_{p_2}
Y_1 Y_2 ··· Y_{p_α}

Fig. 5 A type hierarchy $\mathcal{T}$ consists of the α separated parts

The lookup automaton for this pattern has the following properties:

- At each level k, $1 \leq k \leq a$, there are $p_1p_2...p_k + 1$ states;
- The state at level k can reach state –1 on every input symbols (types) not belonging to the kth separated part of the type hierarchy;
- Grouping the transition-arrays at each level from 2 to $\alpha - 1$ is effective.

The space cost for the transition-array structure is

$$N{\cdot}\alpha + \ [(p_1 + 1)(p_2 + 1)] \ + ... + \ [(p_1p_2...p_{\alpha-1} + 1)(p_\alpha + 1)] \tag{3}$$

where N is the number of types in these α separated parts.

On the other hand, the space cost for the compressed dispatch table for this pattern is

$$N{\cdot}\alpha + (p_1 + 1)(p_2 + 1)...(p_\alpha + 1) \tag{4}$$

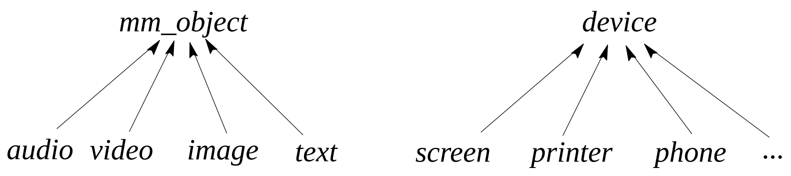
where the first term $N{\cdot}\alpha$ is the cost for argument-arrays.

Expressions (3) and (4) differ only slightly. When $p_1 = p_2 = p_3 = p_4 = 10$, the following table shows the space costs for both structures:

	$\alpha = 2$	$\alpha = 3$	$\alpha = 4$
Transition-arrays	165	1.33K	12.4K
Dispatch Table	165	1.43K	14.8K

In the cross-product pattern, the space cost for transition-array structure and dispatch table structure are nearly the same.

Semi-Cross Product Pattern. In fact, the full cross product pattern is just an extreme case in real applications. In practice, the semi-cross product pattern is more likely. For example, consider a typical application that displays multimedia objects, such as audios, videos, images, and texts on certain devices, such as screens, printers, phones, and so on. Thus, binary multi-methods *displayOn* can be defined for the pairs of types (*obj_type*, *dev_type*) where *obj_type* and *dev_type* are the types belonging to two separate parts of the following type hierarchy



However, it makes no sense to define a full cross product, since functions such as *displayOn*(*audio*, *printer*) and *displayOn*(*image*, *phone*) are not practical. Instead, a semi-cross product is defined while the method *displayOn*(*mm_object*, *device*) plays the role of an error-handling function. For instance, the above non-sense method invocation *displayOn*(*image*, *phone*) would be dispatched to the method *displayOn*(*mm_object*, *device*), by which the error message will be signalled.

Let $\mathcal{M}$ be the set of multi-methods defined on the type hierarchy shown in Fig. 5. In a semi-cross product pattern, $\mathcal{M}$ is a subset of multi-methods in the (full) cross product pattern. However, the error-handling function $m(T_0, U_0, ..., Y_0)$ always belongs to $\mathcal{M}$.

For this set $\mathcal{M}$, the space cost of the dispatch table structure is

$$N \cdot \alpha + q_1 q_2 ... q_\alpha \tag{5}$$

where N is number of types in the type hierarchy, and $q_k = |\{T^k \mid m(T^1, ..., T^k, ..., T^\alpha) \in \mathcal{M}\}|$ is the length in the kth dimension of the dispatch table, $1 \le k \le \alpha$.

On the other hand, the space cost of the transition-array structure is

$$N \cdot \alpha + r_1 q_2 + r_2 q_3 + ... + r_{\alpha-1} q_\alpha \tag{6}$$

where $r_k = |\{(T^1, ..., T^k) \mid m(T^1, ..., T^k, ..., T^\alpha) \in \mathcal{M}\}|$ is the number of states at level k, $1 \le k \le \alpha$. According to the definition of semi-cross product, the following relation holds:

$$r_k \le (q_1 - 1)(q_2 - 1)...(q_k - 1) + 1.$$

However, in general we have $r_k << (q_1 - 1)(q_2 - 1)...(q_k - 1) + 1$. In this case the expression (6) is far less than expression (5) when $\alpha > 2$, i.e. the space cost of the transition-array structure is far less than the cost for dispatch table.

Consider an example of the semi-cross product where multi-methods are sparsely defined: In the type hierarchy shown in Fig. 5, let $p_1 = p_2 = ... = p_\alpha$ and multi-methods are defined with signatures

$$(T_i, U_i, ..., Y_i), \ i = 0, 1, ..., p_1.$$

By formula (5) the space cost for dispatch table structure is

$$N \cdot \alpha + (p_1 + 1)^\alpha \tag{7}$$

By formula (6) the space cost for transition-array structure is

$$N \cdot \alpha + (\alpha - 1)(p_1 + 1)^2 \tag{8}$$

The difference between (7) and (8) is obvious. For example, when $p_1 = p_2 = p_3 = p_4 = 40$, the space costs for both structures are as follows:

	$\alpha = 2$	$\alpha = 3$	$\alpha = 4$
Transition-arrays	1.8K	3.7K	5.7K
Dispatch Table	1.8K	69.2K	2.8M

Experimental Results. After analyzing and comparing the space efficiencies for the common benchmarks, we now present several experimental results to further examine the space efficiencies. We define a type hierarchy with 60 types, based on which the signatures for multi-methods with a same arity (between 2 and 4) are randomly generated by a program. The following table shows the results of six experiments.

α	$\lvert\mathcal{M}\rvert$	Size of transition-arrays	Size of dispatch table
2	86	982	960
2	157	1024	1029
3	101	4.4K	81.5K
3	200	8.2K	117.8K
4	180	9.8K	30.6M
4	301	14.8K	34.6M

The table shows that for binary methods there is no substantial difference between the approaches, while for methods with greater arities, the transition-array structure is superior to the dispatch table structure.

9 Conclusions

In this paper we presented a time and space efficient dispatch strategy for both partially and totally ordered multi-methods, applicable to both statically and dynamically typed languages. The techniques presented are concluded as follows:

- Given a type hierarchy $\mathcal{T}$ and a set of multi-methods $\mathcal{M}$, the lookup automaton is constructed;
- In the automaton, the states and state transitions are represented by a series of transition-arrays;
- The transition-arrays can be compressed using a grouping technique;
- The code of dispatching for a method invocation is translated into at most 2α and at least α nested one-dimensional array accesses, where α is the arity of the method invocation.

The trick of our approach is to combine the advantages of both the lookup automaton structure [12] and the compressed dispatch table structure [5]. The former one provides a good space compression property, while the later provides a fast run-time dispatch mechanism. Thus, the features of our techniques are two-fold. First, the time efficiency is same as for dispatch table structure. Second, the lookup structure effectively prevents the space usage to be combinatorially explosive, especially when multi-methods are just sparsely defined. Analysis and experiments support this claim.

References

1. Agrawal, R., Borgida, A., and Jagsdish, H. V. Efficient Management of Transitive Relationships in Large Data And Knowledge Bases. In *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, 1989.
2. Agrawal, R., DeMichiel, L. G., and Lindsay, B. G. Static Type Checking of Multi-Methods. In *Proc. Conf. OOPS-LA*, 1991.
3. Aho, A. V., and Ullman. J. D(. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, INC. 1972.
4. Aho, A. V., Sethi, R., and Ullman. J. D. *Compilers Principle, Techniques, and Tools.* Addison-Wesley, 1986.
5. Amiel E., Gruber, O., and Simon E. Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables. In *Proc. Conf. OOPSLA*, 1994.
6. André, P., and Royer, J.-C. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *Proc. Conf. OOPSLA*, 1992.
7. Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. CommonLoops: Merging Lisp and Object-Oriented Programming. In *Proc. Conf. OOPSLA*, 1986.
8. Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. Common Lisp Object System Specification X3J13. In *SIGPLAN Notice* 23, *special issue*, Sept. 1988.
9. Cardelli, L., and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4), Dec. 1985.
10. Chambers, C. Object-oriented Multi-Methods in Cecil. In *Proc. Conf. ECOOP*, 1991.
11. Chambers, C, and Leavens, G. T. . Typechecking and Modules for Multi-Methods In *Proc. Conf. OOPSLA*, 1994
12. Chen, W. Turau, V., Klas, W. Efficient Dynamic Lookup Strategy for Multi-methods. In *Proc. Conf. ECOOP*, 1994. An extended version is also accepted in *J. TAPOS*, 1(1), 1995.
13. Dixon, R. Vaughan, P., and Schweizer, P. A Fast Method Dispatcher for Compiled Language with Multiple Inheritance. In *Proc. Conf. OOPSLA*, 1989.
14. Ducournau, R., Habib, M., Huchard, M., and Mugnier, M. L. Monotonic Conflict Resolution Mechanisms for Inheritance. In *Proc. Conf. OOPSLA*, 1992.
15. Dussud, P. H. TICLOS: An Implementation of CLOS for the Explorer Family. In *Proc. Conf. OOPSLA*, 1990.
16. Ingalls, D. H. H. A Simple Technique for Handing Multiple Polymorphism. In *Proc. Conf. OOPSLA*, 1986.
17. Kiczales, G., and Rodriguez, L. Efficient Method Dispatch in PCL. In *Proc. Conf. on Lisp and Functional Programming*, 1990.
18. Lécluse, C., and Richard, P. Manipulation of Structured Values on Object-Oriented Databases. In *Proc. Second Int'l Workshop on Database Prog. Lang.*, 1989.
19. Moon, D, and Weinreb, D. *Lisp Machine Manual*, MIT AI Lab, 1981, Chapter 20.
20. Moon, D. Object Oriented Programming with Flavors. In *Proc. Conf. OOPSLA*, 1986.
21. Mugridge, W. G., Hamer, and Hosking, J. G. Multi-Methods in a Statically-Typed Programming Language. In *Proc. Conf. ECOOP*, 1991.

Appendices

A. Proof of Theorem 1

Given a method invocation $m(T^1, ..., T^\alpha)$, assume

$$\Omega(k, T^k) = \left(\omega_{ij}^k\right)_{n\times n}, 1 \le k \le \alpha.$$

Now we have

$$\omega_{ij} = \omega_{ij}^1 \,\&\, \omega_{ij}^2 \,\&\, ... \,\&\, \omega_{ij}^\alpha.$$

According to the definition of operator '&' (see Fig. 2), ω_{ij} = 'x' iff $\exists k$, $1 \le k \le \alpha$, such that ω_{ij}^k = 'x'. Thus, according to the definition of matrix $\Omega(k, T^k)$, it is easy to check that $\mathcal{A} = \{m_i \in \mathcal{M} \mid \omega_{ii} \ne \text{'x'}\}$ is the set of applicable methods for the invocation $m(T^1, ..., T^\alpha)$.

On the other hand, For POMM, ω_{ij} = '<' iff $\forall k$, $1 \le k \le \alpha$, either ω_{ij}^k = '<' or ω_{ij}^k = '='. For TOMM, ω_{ij} = '<' iff $\exists k$, $1 \le k \le \alpha$, such that (1) $\forall l$, $1 \le l < k$, ω_{ij}^l = '='; (2) ω_{ij}^k = '<'; (3) $\forall l$, $k < l \le \alpha$, $\omega_{ij}^l \ne$ 'x'. Thus, in any case, ω_{ij} = '<' iff (1) m_i and m_j are applicable to the invocation $m(T^1, ..., T^\alpha)$; (2) m_i is more specific than m_j with respect to the invocation. Based on this fact, it is easy to check that the set

$$\mathfrak{D} = \mathcal{A} - \{m_j \in \mathcal{A} \mid \exists i \text{ such that } \omega_{ij} = \text{'<'}\}$$

consists of the (locally) most specific methods with respect to the invocation for both POMM and TOMM.

B. Algorithm of Constructing a Lookup Automaton

A lookup automaton is a deterministic finite automaton [3] which is defined as a 5-tuple $(Q, \Sigma, succ, q_0, F)$, where Q is a finite set of *states*; Σ a finite set of *input symbols*; *succ* a *state transition function*, which is a mapping $Q \times \Sigma \rightarrow Q$; $q_0 \in Q$ the *initial state* of the finite state control; and $F \subseteq Q$ the set of *final states*.

The following routine *construct* constructs the automaton. For each state $q \in Q$, let $q.\Omega$ denote the corresponding matrix. Furthermore, let Q_k denote the set of states created at level k, $0 \le k \le \alpha$. Thus, $F = Q_\alpha$.

```
routine construct()
{       create an initial state $q_0$ such that $q_0.\Omega = (=)_{n\times n}$
        $Q_0 \leftarrow \{q_0\}$
        for $k = 1$ to $\alpha$ do
            foreach $q \in Q_{k-1}$ do
                foreach $T \in \mathcal{T}$ do
                    if $\exists q_{old} \in Q_k$ such that $q_{old}.\Omega = q.\Omega \wedge \Omega(k, T)$ then $succ(q, T) \leftarrow q_{old}$
                    else
                    {   create a new state $q_{new}$ such that $q_{new}.\Omega = q.\Omega \wedge \Omega(k, T)$
                        add $q_{new}$ into $Q_k$
                        $succ(q, T) \leftarrow q_{new}$
                    }
        merge all states $q$ such that $q.\Omega = (x)_{n\times n}$ into a new final state –1
        compute the dispatch result at each final state $q \in Q_\alpha$ by means of the matrix $q.\Omega$
}
```