# Fast Algorithms for Compressed Multimethod Dispatch Table Generation

ERIC DUJARDIN
Sun Microsystems
and
ERIC AMIEL
Cambridge Technology Partners
and
ERIC SIMON
INRIA

The efficiency of dynamic dispatch is a major impediment to the adoption of multimethods in object-oriented languages. In this article, we propose a simple multimethod dispatch scheme based on compressed dispatch tables. This scheme is applicable to any object-oriented language using a method precedence order that satisfies a specific monotonous property (e.g., as Cecil and Dylan) and guarantees that dynamic dispatch is performed in constant time, the latter being a major requirement for some languages and applications. We provide efficient algorithms to build the dispatch tables, provide their worst-case complexity, and demonstrate the effectiveness of our scheme by real measurements performed on two large object-oriented applications. Finally, we provide a detailed comparison of our technique with other existing techniques.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures*; *procedures, functions and subroutines*; D.3.4 [**Programming Languages**]: Processors—*optimization*; E.1.3 [**Data**]: Data Structures—*tables*; E.2.1 [**Data**]: Data Storage Representations—*contiguous representations*

General Terms: Algorithms, Languages, Measurement, Performance

Additional Key Words and Phrases: Dispatch tables, late binding, multimethods, optimization, pole types, run-time dispatch

## 1. INTRODUCTION

In traditional object-oriented systems such as Smalltalk [Goldberg and Robson 1983] and C++ [Ellis and Stroustrup 1992], functions have a specially designated *target* argument, sometimes called receiver, whose type, determined either at run-

This work was done while the authors were at INRIA.
Author's addresses: E. Dujardin, SunSoft, Embedded Systems Software Group, 6 avenue Gustave Eiffel, 78182 St Quentin-en-Yvelines, France; email: eric.dujardin@france.sun.com; E. Amiel, CTP, 16 chemin des Coquelicots, CH-1214 Genève, Switzerland; email: eric.amiel@ctp.com; E. Simon, INRIA, 78153 Le Chesnay, France; email: eric.simon@inria.fr.

time or at compile-time, serves to select the method to execute in response to a function invocation. Multiple dispatching, first introduced in CommonLoops [Bobrow et al. 1986] and CLOS [Bobrow et al. 1988], generalizes this form of dynamic binding by selecting a method, depending on the run-time type of a subset of the arguments of a function invocation. Methods in this generalized scheme are called multitargetted methods, or multimethods for short. As explained in Chambers and Leavens [1995], multimethods bring an increased expressive power over single-targetted methods. For this reason, multimethods are becoming a key feature of recently developed object-oriented languages as Polyglot [Agrawal et al. 1991], Kea [Mugridge et al. 1991], Cecil [Chambers 1992], and Dylan [Apple Computer 1995]. They also have been integrated as part of the SQL3 standard [Melton 1994; 1996] currently under development.

However, several problems still hamper the wide acceptance of multimethods in object-oriented systems, among which the inefficiency of multiple dispatching caused by the multiple-selection criteria of methods. By contrast, there exist algorithmical solutions that offer a fast constant-time dispatching for mono-methods using a two-dimensional table, called dispatch table, that holds the precalculated result of dispatching for all possible function invocations. Optimization techniques have been developed to minimize the size of the dispatch table by eliminating entries corresponding to function invocations for which there is no applicable method [Dixon et al. 1989; Driesen and Hölzle 1995; Ellis and Stroustrup 1992; Huang and Chen 1992]. The reduction factor is experimentally measured to be up to 66 in Driesen and Hölzle [1995].

The dispatch table technique does not naturally scale up for multimethods because additional target arguments create new dimensions in the table. Since each dimension is indexed by the set of types, which is usually large (e.g., above 100), the size of the table dramatically increases and cannot be handled anymore. Furthermore, since the number of target arguments varies between functions, there must be one dispatch table per function.

The major contribution of this article is to propose a simple, rigorously defined, multimethod dispatch scheme based on compressed dispatch tables that guarantees a dynamic dispatching in constant time. Our compression technique is widely applicable: representing a method dispatch scheme as a function that applies to an invocation $m(s)$ and returns a set of applicable methods ordered with a precedence ordering, our technique only assumes that this ordering obeys a single monotonicity property. We formally state that languages such as Cecil and Dylan use a precedence ordering that does satisfy this property, whereas CLOS does not.

The basic idea of the table's compression scheme is first to eliminate entries corresponding to invocations for which there is no applicable method, and second to group identical $(n-1)$-dimensional rows in a table of dimension $n$. Although this scheme can achieve a high compression factor, naive algorithms to group identical rows in a table of dimension $n$ have a worst-case complexity of $O(n \times |\Theta|^{n+1})$ where $|\Theta|$ represents the total number of types in this system. We show that in a compressed dispatch table, each dimension is indexed by a subset of the types, called the *pole types* of this dimension. We then provide fast algorithms to compute pole types, and hence a compressed table's structure, with a worst-case complexity of $O(|m|+|\Theta| \times \mathcal{E})$, where $|m|$ is the number of methods associated with a generic func-

tion $m$, and $\mathcal{E}$ is the number of edges in the type graph. We prove the correctness of our algorithms and demonstrate the effectiveness of our compression scheme using two real object-oriented applications with multimethods (in Cecil and Dylan). Finally, we provide a detailed comparison of our multimethod dispatch scheme with other existing schemes, including the recent proposal of Chen and Turau [1995], which uses a kind of "dispatch tree."

The article is organized as follows. Section 2 introduces preliminary notions and notations on multimethods and method dispatch. Section 3 presents the problem and an overview of our solution. Section 4 introduces our dispatch table compression scheme, based on a formal definition of pole types, and proves its correctness. Section 5 presents the pole type computation algorithm, while Section 6 presents the table fill-up algorithm. Section 7 describes our implementation and provides experimental results. Section 8 is devoted to a presentation of related work. Finally, we conclude in Section 9.

## 2. BACKGROUND

We briefly review some terminology mainly introduced in Agrawal et al. [1991]. We denote *subtyping* by $\preceq$. Given two types $T_1$ and $T_2$, if $T_1 \preceq T_2$, we say that $T_1$ is a subtype of $T_2$ and that $T_2$ is a supertype of $T_1$. The set of types is denoted $\Theta$. We assume the existence of a relation *isa* between types, such that the subtyping relation $\preceq$ is the transitive closure of *isa*. If $T$ *isa* $T'$, $T$ is a *direct subtype* of $T'$, and $T'$ is a *direct supertype* of $T$. Intuitively, *isa* corresponds to the user's declarations of supertypes.

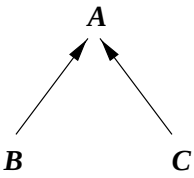
A generic function is defined by its name and its arity. To each generic function $m$ of arity $n$ corresponds a set of methods $m_k(T_k^1, \ldots, T_k^n) \rightarrow R_k$, where $T_k^i$ is the type of the $i$th formal argument, and where $R_k$ is the type of the result. We call the list of arguments $(T_k^1, \ldots, T_k^n)$ of method $m_k$ the *signature* of $m_k$. We also define a relation $\preceq$ on signatures, called *precedence*, such that $(T^1, \ldots, T^n) \preceq (T'^1, \ldots, T'^n)$ if and only if for all $i$, $T^i \preceq T'^i$. An invocation of a generic function $m$ is denoted $m(T^1, ..., T^n)$, where $(T^1, \ldots, T^n)$ is the signature of the invocation, and the $T^i$'s represent the types of the expressions passed as arguments.

Object-oriented languages support *late binding* of method code to invocations: the method that gets executed is selected based on the run-time type(s) of the target argument(s). This selection process is called *method dispatch*. In traditional object-oriented systems, generic functions have a single specially designated *target* argument, also known as *receiver*. In systems that support multimethods, a subset of the arguments of a generic function are target arguments. From now on, we shall only consider multimethods, and without a loss of generality we assume that all arguments are target arguments.

Operationally, given an invocation $m(s)$, method dispatch follows a two-step process: first, based on the types of the target arguments, the set of applicable methods, henceforth noted $applicable(m(s))$, is found:

*Definition* 2.1. A method $m_i(T_i^1, \ldots, T_i^n)$ is applicable to a signature $(T^1, \ldots, T^n)$, denoted by $m_i \succeq (T^1, \ldots, T^n)$, if and only if $(T_i^1, \ldots, T_i^n) \succeq (T^1, \ldots, T^n)$.

If the set of applicable methods is empty then the method dispatch is undefined. Otherwise, a precedence ordering is used to select the most specific method to




$m_1(A, B)$

$m_2(B, A)$

Fig. 1. Cecil types and methods.

execute. If there are several most specific methods, they are said to be *ambiguous*. Given a signature $s = (T_1, \ldots, T_n)$ and a generic function invocation $m(s)$, if $m_i$ and $m_j$ are applicable to $m(s)$, and according to a particular method precedence ordering, $m_i$ is more specific than $m_j$ for $s$, denoted as $m_i \preceq_s m_j$, then $m_i$ is a closer match for the invocation. Indeed, looking at existing languages, the result of method dispatch can be more accurately described as a set of applicable methods ordered with $\preceq_s$. We illustrate this with two existing languages: Cecil and Dylan.

Cecil offers either a *static* or a *dynamic* checking mode to the programmer. In both modes, Cecil uses a precedence ordering over applicable methods, called *argument subtype precedence* in Agrawal et al. [1991]:

*Definition* 2.2. Let $m_i(T_i^1, \ldots, T_i^n)$ and $m_j(T_j^1, \ldots, T_j^n)$ be two methods associated with a generic function $m$ of arity $n$; if $(T_i^1, \ldots, T_i^n) \preceq (T_j^1, \ldots, T_j^n)$, then for any method invocation $m(s)$ such that $m_i$ and $m_j$ are applicable to $m(s)$, we have $m_i \preceq_s m_j$.

With a static checking mode, Cecil guarantees that there always exists a unique most specific applicable method (henceforth, called UMSA) for all possible run-time arguments of each generic function invocation. This means that an invocation signature may remain ambiguous as long as the type-checker can prove that it will never occur at run-time in the actual codes of methods. With a dynamic checking mode, the detection of possible ambiguities is performed at run-time. If an ambiguity is detected for a given invocation then an error is signaled, and the set of ambiguous methods is returned.

*Example* 2.3. Consider the type hierarchy and methods of Figure 1. With a static checking mode, Cecil's type-checker attempts to prove that no invocation $m(B, B)$ may occur at run-time. If it succeeds then no ambiguity will arise: method dispatch will returns $m_1$ for an invocation $m(C, B)$ and $m_2$ for invocations $m(A, C)$ or $m(B, C)$. With a dynamic checking mode, if an invocation $m(B, B)$ occurs then an error is signaled, and the set $\{m_1, m_2\}$ is returned.

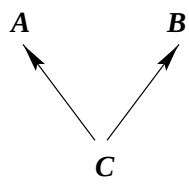
Dylan distinguishes classes as a variety of types.[1] When a class is created, a *linearization* of its supertype classes, including itself, is defined and represented as a Class Precedence List (CPL) that strictly orders all its supertype classes. For convenience, we define a specific order between types as follows:

*Definition* 2.4. Given a type $T$,

(1) if $T$ is not a class, then $\preceq_T$ is the subtyping order;[2]

[1]Here, we use the same definitions for "types" and "classes" as the ones of Dylan [Apple Computer 1995].

[2]More precisely, the subtyping order between types we use here denotes what is called "proper subtyping" in Dylan.




$m_1(A, B)$ CPL $(C) = (A, B)$
$m_2(B, A)$
$m_3(C, A)$
$m_4(C, B)$

Fig. 2. Dylan types and methods.

(2) if $T$ is a class, and $T_1, T_2$ are types, then $T_1 \preceq_T T_2$ if and only if $T_1 \preceq T_2$ or $T_1$ precedes $T_2$ in T's CPL.

Note that $\preceq_T$ is an order, since subtyping is an order, and by definition in Dylan, $T_1$ cannot precede $T_2$ in any CPL if $T_2$ is a subtype of $T_1$. Using this definition, Dylan's precedence ordering between methods can be defined as follows:

*Definition* 2.5. Let $m_1(T_1^1, \ldots, T_1^n)$ and $m_2(T_2^1, \ldots, T_2^n)$ be two methods associated with a generic function $m$ of arity $n$, and let $s = (T_1, \ldots, T_n)$ be the signature of an invocation of $m$. Then $m_1 \preceq_s m_2 \Leftrightarrow \exists i$, $1 \leq i \leq n$, such that $T_i^1 \preceq_{T_i} T_i^2$ and $\forall\, j$ such that $j \neq i$, $T_j^2 \not\preceq_{T_i} T_j^1$.

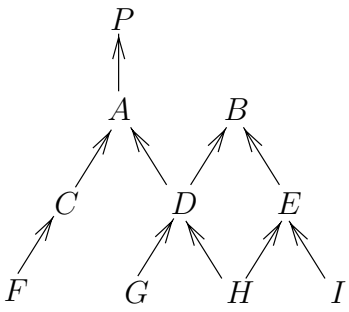
Initially, given an invocation $m(s)$, method dispatch looks for the most specific applicable method according to the above precedence (partial) ordering. If there is more than one such method, an error is signaled. Otherwise, the method, say $m_i$, is returned. However, $m_i$ may invoke a function called *next-method* that returns the "next" method that is immediately less specific than $m_i$ in the set of applicable methods to $m(s)$. If there is no such method, or it is not unique, an error is signaled. Thus, conceptually, the method dispatcher of Dylan can be interpreted as returning the subset of most specific applicable methods that is totally ordered according to the precedence ordering.

*Example* 2.6. Consider the type hierarchy and methods of Figure 2. Let $s = (C, C)$ be the signature of an invocation of $m$. All four methods are applicable. Since $A \preceq_C B$, $m_1(A, B)$ and $m_2(B, A)$ are ambiguous, and $m_3 \preceq_s m_4$. Thus, conceptually, method dispatch returns $\{m_3, m_4\}_{\preceq_s}$, and function *next-method* operates over this set.

Formally, we represent a method dispatch as a function denoted $MS$ (Most Specific) that applies to an invocation $m(s)$ and returns a set of applicable methods ordered with a precedence ordering $\preceq_s$. The dispatch table compression scheme presented in this article assumes that the precedence ordering used by $MS$ satisfies the *monotonicity* property defined as follows:

*Definition* 2.7. Let $s$ and $s'$ be two signatures of arity $n$ such that $s \preceq s'$. If for any methods $m_1$ and $m_2$ applicable to $m(s)$ and $m(s')$, $m_1 \preceq_s m_2 \Leftrightarrow m_1 \preceq_{s'} m_2$, then $\preceq_s$ is said to be a *monotonous* order.

The notion of monotonous order on classes was first introduced in Ducournau et at. [1992]. We generalized this notion for method orders in the above definition. Languages such as Cecil [Chambers 1993] and Dylan [Barrett et al. 1996] use a precedence ordering that satisfies the monotonicity condition. This will be formally stated in Section 4. On the contrary, CLOS [Bobrow et al. 1988] does not




$m_1(A, B)$
$m_2(A, D)$
$m_3(B, B)$
$m_4(B, C)$
$m_5(B, D)$

Fig. 3. Example schema.

enforce monotonicity. However, Ducournau et al. [1994] proposes a monotonous class precedence order that would yield a monotonous order on methods.

## 3. PROBLEM STATEMENT AND OVERVIEW OF THE SOLUTION

We address the problem of multimethods dispatch in languages requiring constant-time method selection. Therefore, our goal is to explore the dispatch table approach both in statically and dynamically typed languages.

### 3.1 Dispatch Tables for Multimethods

As generic functions can have different arities, the single global table organization is not applicable anymore. We need to use a dispatch table for each generic function. For a set of types $\Theta$, the dispatch table of a generic function $m$ of arity $n$ is an $n$-dimensional array with the types of $\Theta$ as indices of each dimension. We denote the dispatch table by $\mathcal{D}_m$. Every entry in $\mathcal{D}_m$ represents a signature in $\Theta^n$. The entry at position $(T_1, \ldots, T_n)$ holds the result of $MS(m(T_1, \ldots, T_n))$ and a null value if $MS$ returns an empty set. In the next examples, $MS$ always return a single method called the Most Specific Applicable (MSA) method. For clarity, we assume that (1) a table entry contains the address of the method indicated by its index and (2) "$-$" denotes the null value.

*Example* 3.1.1. We compute the dispatch table for the type hierarchy and the methods of Figure 3. To this end, we determine $MS$ for every possible invocation using argument subtype precedence and the additional order saying that $m_1$ is more specific than $m_3$ and saying that $m_2$ is more specific than $m_5$. $\mathcal{D}_m$ is illustrated in Figure 4.

Dispatching the methods of a generic function $m$ using a dispatch table is achieved as follows. Assume a unique index is attributed to every type and that every object contains the index of its type as an attribute named *type_index*. Then, in a statically typed language, which ensures that $MS$ always returns an MSA method for every possible invocation,[3] an invocation $m(o_1, \ldots, o_n)$, where the $o_i$ are objects, is translated into the following pseudocode:

```
ms = D_m[o_1.type_index, ..., o_n.type_index]
call ms(o_1, ..., o_n)
```

[3]Take, for instance, Cecil with static checking mode.

| | $P$ | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ | $H$ | $I$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $P$ | – | – | – | – | – | – | – | – | – | – |
| $A$ | – | – | 1 | – | 2 | 1 | – | 2 | 2 | 1 |
| $B$ | – | – | 3 | 4 | 5 | 3 | 4 | 5 | 5 | 3 |
| $C$ | – | – | 1 | – | 2 | 1 | – | 2 | 2 | 1 |
| $D$ | – | – | 1 | 4 | 2 | 1 | 4 | 2 | 2 | 1 |
| $E$ | – | – | 3 | 4 | 5 | 3 | 4 | 5 | 5 | 3 |
| $F$ | – | – | 1 | – | 2 | 1 | – | 2 | 2 | 1 |
| $G$ | – | – | 1 | 4 | 2 | 1 | 4 | 2 | 2 | 1 |
| $H$ | – | – | 1 | 4 | 2 | 1 | 4 | 2 | 2 | 1 |
| $I$ | – | – | 3 | 4 | 5 | 3 | 4 | 5 | 5 | 3 |

Fig. 4. Dispatch table of generic function $m$.

In a dynamically typed language, which does not ensure the existence of an MSA method for every possible invocation,[4] an additional test is required to check the content of the table entry: it may be empty, or there may be several ambiguous methods. In any case, the computation of $MS$ for an invocation is performed very fast, using a single $n$-dimensional array access. However, the size of $\mathcal{D}_m$ is $|\Theta|^n$, which is prohibitive ($|\Theta|$ being usually above 100 and the arity of multimethods between 2 and 4).

### 3.2 Compressing Dispatch Tables

The core of the problem of multimethod dispatch tables is their size, which is in the power of the number of arguments. We propose to reduce the number of entries of each dispatch table and get a compressed table for $m$ that we denote by $\mathcal{D}^c_m$. Assuming a generic function of arity $n$, our compression scheme is based on two techniques:

—eliminating entries holding only null values and

—grouping entries which have all their values identical.

The following example sketches the idea.

*Example* 3.2.1. Consider the table of Figure 4. First, columns $P$ and $A$ and line $P$ can be eliminated, as they only contain null values. This means that invocations where $P$ appears as the first or second argument, or where $A$ appears as the second argument, are actually type errors. In these cases, examining the type of the argument is sufficient to determine that there is no MSA. Second, lines $A$, $C$, and $F$ are identical, as are lines $B$, $E$, and $I$ and lines $D$, $G$, and $H$. The same is true for columns $B$, $E$, and $I$, columns $C$ and $F$, and columns $D$, $G$, and $H$. Such rows can be grouped yielding the following table, where indices of the table may be groups of types. Entry at position $(\{T_1, \ldots, T_p\}, \{U_1, \ldots, U_q\})$ holds the MSA method for all invocations $m(T_i, U_j), i \in \{1, \ldots, p\}, j \in \{1, \ldots, q\}$:

| | $\{B, E, I\}$ | $\{C, F\}$ | $\{D, G, H\}$ |
|---|---|---|---|
| $\{A, C, F\}$ | 1 | – | 2 |
| $\{B, E, I\}$ | 3 | 4 | 5 |
| $\{D, G, H\}$ | 1 | 4 | 2 |

[4] As in Cecil with dynamic checking mode, or Dylan.

We call the groups of types that index the dimensions of a compressed table *index-groups*. In each dimension, index-groups are attributed an index. In our example, index-group $\{A, C, F\}$ of the first dimension has index 1. The index of a type in a dimension of a compressed table is the index of its group.

### 3.3 Dispatch Using Compressed Tables

Once the table is compressed, dispatch must be performed differently. Indeed, the index of a type is not anymore the same in every dimension of a dispatch table and in every dispatch table. Thus, the unique index of each type cannot be directly used to access entries in compressed dispatch tables. In the compressed table of Example 3.2.1, the index of type $A$ is 1 (index of its group) when $A$ appears as the first argument, while $A$ does not appear in the second dimension.

We map, for every generic function $m$ and every argument position $i$, a type to its index in the $i$th dimension of the compressed dispatch table of $m$. This can be achieved using $n$ *argument arrays*, where $n$ is the arity of $m$. Each argument array is a one-dimensional array indexed by the types. The $i$th argument 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. A "0" indicates that this type cannot appear as the $i$th argument.

*Example* 3.3.1. Here are the two argument arrays $m\_arg_1$ and $m\_arg_2$ of $m$ for the compressed dispatch table of Example 3.2.1:

| | $P$ | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ | $H$ | $I$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $m\_arg_1$ | 0 | 1 | 2 | 1 | 3 | 2 | 1 | 3 | 3 | 2 |
| $m\_arg_2$ | 0 | 0 | 1 | 2 | 3 | 1 | 2 | 3 | 3 | 1 |

In a statically typed language that ensures an MSA method for every invocation, an invocation $m(o_1, \ldots, o_n)$ is translated into the following code:

```
ms = D^c_m[m_arg_1[o_1.type_index], ..., m_arg_n[o_n.type_index]]
call ms(o_1, ..., o_n)
```

In a dynamically typed language, run-time type checking is needed to verify the content of $m\_arg_i[o_i.type\_index]$ and $\mathcal{D}^c_m$ before calling the $MS$ method.

### 3.4 Argument-Array Coloring

Coloring has been proposed in Dixon et al. [1989] to compress dispatch tables of mono-methods. This technique is applicable to argument arrays for grouping into one array several arrays which hold nonnull cells at different positions.

We shall call *selector*, noted $(m, i)$, an argument position $i$ in a generic function $m$. A selector is associated with a set of types, which are the types allowed at run-time for this method and this argument position. Two selectors are said to *conflict* if their sets of types intersect. When two or more selectors do not conflict with each other, their colors can be the same, i.e., their argument arrays can be grouped, as shown in Figure 5. These arrays do not necessarily belong to the same generic function.

When argument arrays are grouped, the information represented by a null value in the original argument arrays is lost in the group argument array. This information

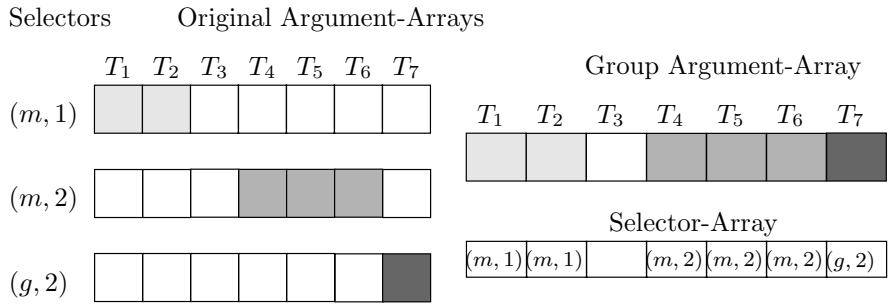



Fig. 5. Grouping of argument arrays.

is used by run-time type checking, in dynamically typed languages like Smalltalk or Self, or in Amiel et al. [1996]. If a type $T$ is not allowed for a selector $(m, i)$, type checking makes it necessary to mark that the entry of $T$ in the group argument array is related to another selector. For this, we associate with each group argument array a selector array. For each type, this array holds the selector with which it is associated in the group argument array. At run-time, the type of the object that appears as the $i$th argument in an invocation of $m$ yields both an entry number (coming from the group argument array) and a selector (coming from the selector array). Type checking consists in comparing the selector to $(m, i)$, and raises an error if they differ. If not, the entry number is used against the dispatch table to obtain $MS$.

### 3.5 Naive Approach to Compress a Dispatch Table

The naive approach starts from an uncompressed dispatch table and then compresses it. In this section, we consider the case of a language that ensures an MSA method for every invocation. Building the uncompressed dispatch table simply involves computing the MSA method for every possible invocation signature. With a generic function of arity $n$, compressing the table comes down first to scanning all entries in all dimensions to detect the empty entries and eliminate them and then compare the remaining entries, to group the ones that are identical. However, this process is unrealistic, both for memory space and processing time considerations:

3.5.1 *MSA Computation Time.* Assuming an application with 100 types, and the use of a library like the programming environment of Smalltalk which roughly counts 800 more types, there are 810,000 possible invocation signatures for two-targetted generic functions, and 729,000,000 for three-targetted generic functions, for which the MSA method has to be computed.

3.5.2 *Uncompressed Table Size.* With $|\Theta| = 900$, the uncompressed table of a three-targetted method takes 695MB, and the one of a four-targetted method takes 610GB.

3.5.3 *Empty Entries Elimination.* Searching the empty entries needs at worst to scan $n$ times the entire table, once for each dimension. Moreover, eliminating the empty entries requires moving a lot of data in memory.

3.5.4 *Identical Entries Grouping.* Grouping the entries needs at worst, for each dimension and each entry, to compare it to $|\Theta| - 1$ other entries. Each of these entries is associated with $|\Theta|^{n-1}$ values. Hence grouping takes at worst $n \times |\Theta| \times (|\Theta|-1) \times |\Theta|^{n-1} = n \times (|\Theta|^{n+1} - |\Theta|^n)$ equality comparisons. For $n = 3$, $|\Theta| = 900$, assuming each comparison is done in one cycle by a 200MHz processor, this nearly takes three hours. Grouping also involves a lot of memory moves.

In the following, we present an efficient algorithm to directly build compressed dispatch tables. For this, we first analyse the set of formal argument types in each dimension, to determine the empty entries and the index groups. Then, we build the compressed dispatch table by computing $MS$ only once for each index-group signature. In this way, both processing time and memory space are spared, and obtaining the compressed dispatch table becomes realistic.

## 4. COMPRESSION APPROACH

The central issue in our approach to dispatch table compression is to determine which entries can be eliminated and which ones can be grouped without having to scan the whole table. Let us first formally define the condition under which an entry can be eliminated and two entries can be grouped. Consider the $i$th argument position of a generic function $m$ of arity $n$.

—*Elimination Condition:* The entry for type $T$ in the $i$th dimension of the dispatch table of $m$ can be eliminated if and only if

$$\forall (T_1, \ldots, T_{n-1}) \in \Theta^{n-1}, MS(m(T_1, \ldots, T_{i-1}, T, T_i, \ldots, T_{n-1})) = \emptyset. \quad (1)$$

—*Grouping Condition:* The two entries for types $T$ and $T'$ in the $i$th dimension of the dispatch table of $m$ can be grouped if and only if

$$\begin{aligned} &\forall (T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_n) \in \Theta^{n-1}, \\ &MS(m(T_1, \ldots, T_{i-1}, T, T_{i+1}, \ldots, T_n)) \\ &\qquad = MS(m(T_1, \ldots, T_{i-1}, T', T_{i+1}, \ldots, T_n)). \end{aligned} \quad (2)$$

The first condition says that $T$ is not allowed as the type of the $i$th argument of an invocation of $m$. The second condition says that $T$ can replace $T'$ as the $i$th argument of any invocation without any change to $MS$ and vice-versa. If this condition holds, then $T$ and $T'$ belong to the same index-group in the $i$th dimension and thus have the same index in the $i$th argument array of $m$.

Coming back to Example 3.2.1, it is useful to observe that in every index-group there is a type supertype of all the other types of the group. For instance, in the groups of the first dimension, types $A$, $B$, and $D$ are respectively the greatest types for groups $\{A, C, F\}$, $\{B, E, I\}$, and $\{D, G, H\}$. We call these types *$i$-poles*, where $i$ is the argument position. Thus, $A$, $B$, and $D$ are 1-poles and $B$, $C$ and $D$ are 2-poles. We shall show that there always exists a unique $i$-pole for every group in the $i$th dimension. In fact, we show how to determine for every dimension the $i$-poles and their associated index-groups, called *influence*. Then, we establish that

—the types that do not belong to an influence can be eliminated from the $i$th dimension and

—the types belonging to the same influence can be grouped.

### 4.1 Poles and Influences

We first introduce some auxiliary definitions and then formally define the notions of $i$-pole and influence.

*Definition* 4.1.1. $T_1$ and $T_2$ are incomparable, denoted $T_1 \prec\!\!\succ T_2$, if and only if

$$T_1 \not\preceq T_2 \text{ and } T_1 \not\succeq T_2.$$

*Definition* 4.1.2. The cover of a type $T$, denoted $cover(T)$, is the set of subtypes of $T$:

$$cover(T) = \{T' \mid T' \preceq T\}$$

The cover of a set of types $\{T_1, \ldots, T_n\}$, denoted $cover(\{T_1, \ldots, T_n\})$, is the union of the covers of each type in the set:

$$cover(\{T_1, \ldots, T_n\}) = \bigcup_{i=1}^{n} cover(T_i)$$

*Definition* 4.1.3. The $i$th static arguments of a generic function $m$, denoted $Static_m^i$, are the types of the $i$th formal arguments of the methods of $m$:

$$Static_m^i = \{T_k^i \mid \exists m_k(T_k^1, \ldots, T_k^n)\}$$

*Definition* 4.1.4. The $i$th dynamic arguments of a generic function $m$, denoted $Dynamic_m^i$, are the cover of the $i$th static arguments of $m$:

$$Dynamic_m^i = cover(Static_m^i)$$

They represent the types that can appear at the $i$th position in invocations of $m$ at run-time. $Dynamic_m$ is the cross product of the $i$th dynamic arguments sets:

$$Dynamic_m = \prod_{i=1}^{n} Dynamic_m^i$$

*Fact* 4.1.5. $s \in Dynamic_m \Leftrightarrow applicable(m(s)) \neq \emptyset$.

*Definition* 4.1.6. A type $T \in \Theta$ is an $i$-pole of a generic function $m$ of arity $n$, $1 \leq i \leq n$, if and only if[5]

$$T \in Static_m^i \tag{3}$$

or

$$|min_{\preceq} \{T' \in \Theta \mid T' \text{ is an } i\text{-pole of } m \text{ and } T' \succ T\}| > 1. \tag{4}$$

The set of $i$-poles of $m$ is denoted $Pole_m^i$. The poles that are included in $Static_m^i$ are called *primary poles*, and the others are called *secondary poles*.

*Fact* 4.1.7. $Pole_m^i \subset Dynamic_m^i$.

[5]In this definition and the sequel of this article, $(E, \leq)$ being an ordered set, we define $min_{\leq} E$ as $\{x \in E \mid \forall y \in E - \{x\}, y \not\leq x\}$.

*Definition* 4.1.8. Let $m$ be a generic function of arity $n$, $T \in \Theta$, and $1 \leq i \leq n$; the set of closest poles of $T$ is

$$closest\text{-}poles^i_m(T) = min_{\preceq}\{T' \in Pole^i_m \mid T \prec T'\}.$$

Using this notation, condition (4) can also be expressed as

$$|closest\text{-}poles^i_m(T)| > 1. \tag{5}$$

LEMMA 4.1.9. *Given a generic function $m$ of arity $n$, for each $i$, $1 \leq i \leq n$ and each $T \in \Theta$, we have*

$$closest\text{-}poles^i_m(T) \neq \emptyset \Rightarrow T \in Dynamic^i_m.$$

PROOF. If $closest\text{-}poles^i_m(T) \neq \emptyset$ then there exists some $T' \in Pole^i_m$ such that $T \preceq T'$. To establish the lemma, we show that there exists $T'' \in Static^i_m$ such that $T \preceq T' \preceq T''$, i.e., $T \in cover(T'')$. We proceed by contradiction. Suppose

$$\forall T' \in Pole^i_m(T) \text{ s.t. } T \preceq T',\ \nexists T'' \in Static^i_m \text{ s.t. } T' \preceq T''. \tag{6}$$

In particular $T' \notin Static^i_m$. Since $T'$ is an $i$-pole, by condition (4) $\exists T_0 \in Pole^i_m$ such that $T' \preceq T_0$. Hence $T \preceq T_0$, and by (6), $T_0 \notin Static^i_m$. Iterating the same reasoning, we can prove the existence of an infinite sequence of distinct poles $T_0, T_1, \ldots$, which contradicts the fact that $\Theta$ is finite. □

PROPOSITION 4.1.10. *Given a generic function $m$ of arity $n$, and $i \in \{1, \ldots, n\}$, we have*

$$(\forall T' \in Static^i_m,\ \forall T \in \Theta \text{ s.t. } T \prec T',\ T \text{ has exactly one supertype}) \Rightarrow (Pole^i_m = Static^i_m).$$

PROOF. We have to show that $Pole^i_m \subset Static^i_m$, i.e., $\forall T \in \Theta$, $T$ is not a secondary pole. Let $T \in \Theta$. If $closest\text{-}poles^i_m(T) = \emptyset$ then $\{T' \in Static^i_m \mid T \prec T'\} = \emptyset$, and the proposition trivially holds. If $closest\text{-}poles^i_m(T) \neq \emptyset$ then by Lemma 4.1.9, $T \in Dynamic^i_m$. Thus, by definition, there exists $T' \in Static^i_m$ such that $T \in cover(T')$. We show that $|closest\text{-}poles^i_m(T)| = 1$, which by condition (5) means that $T$ is not a secondary pole. We proceed by contradiction.

Let $T_a$ and $T_b$ be two distinct poles of $closest\text{-}poles^i_m(T)$. Let $T_1, \ldots, T_k$ such that $T$ *isa* $T_1 \ldots$ *isa* $T_k$ *isa* $T'$. By the left-hand side of the proposition, each type $T, T_1, \ldots, T_k$ has only one direct supertype. Since $\{T, T_1, \ldots, T_k, T'\}$ is totally ordered with respect to $\preceq$ and $T_a$ and $T_b$ are incomparable by definition, this set does not include both $T_a$ and $T_b$. Suppose that $T_b$ is not in this set. By the left-hand side of the proposition, the path from $T$ to $T_b$ in the type graph necessarily goes through $T'$; hence $T' \preceq T_b$. As $T_b \notin \{T, T_1, \ldots, T_k, T'\}$, $T' \prec T^b$.

Since $T' \in Static^i_m$, $T'$ is an $i$-pole, and by Definition 4.1.8, $T_b \preceq T'$, a final contradiction. □

COROLLARY 4.1.11. *In systems supporting or actually using only single inheritance, for each generic function $m$, and each $i$ in $\{1, \ldots, n\}$, $Pole^i_m = Static^i_m$.*

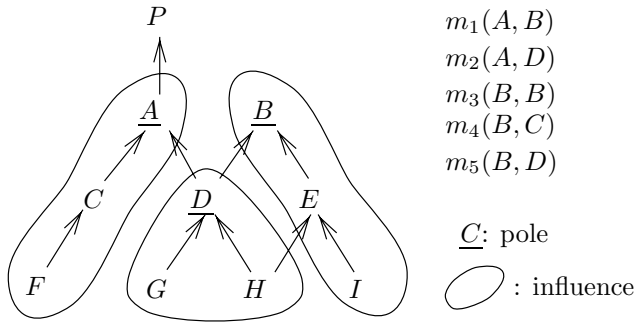



Fig. 6. Example schema with poles and influences.

*Example* 4.1.12. We determine the 1- and 2-poles in the example of Figure 3. We have $Static^1_m = \{A, B\}$, $Static^2_m = \{B, C, D\}$, $Dynamic^1_m = \{A, B, C, D, E, F, G, H, I\}$, and $Dynamic^2_m = \{B, C, D, E, F, G, H, I\}$. As $Static^1_m = \{A, B\}$, $A$ and $B$ are 1-poles. Moreover, they are the closest poles of $D$, so $D$ is also a 1-pole. So $Pole^1_m = \{A, B, D\}$. As $Static^2_m = \{B, C, D\}$, $B$, $C$, and $D$ are 2-poles. They do not satisfy condition (4) with respect to any type, so $Pole^2_m = \{B, C, D\}$. The 1-poles are underlined on Figure 6.

*Definition* 4.1.13. With every $i$-pole $T$ of a generic function $m$ is associated the set of subtypes of $T$, noted $Influence^i_m(T)$, which is defined as

$$Influence^i_m(T) = \{T' \preceq T \mid \forall T'' \in Pole^i_m, T' \not\preceq T'' \text{ or } T \preceq T''\}$$

*Example* 4.1.14. We determine $Influence^i_m$ for the 1-poles of Example 4.1.12. The influence of $A$ contains $A$, $C$, and $F$, as $A$ is their only 1-pole supertype. For the same reason, the influence of $B$ contains $B$, $E$, and $I$. $D$ does not belong to the influence of either $A$ or $B$, as $D$ is a 1-pole, and it is supertype of itself and a subtype of $A$ and $B$. The influence of $D$ contains $D$, $G$, and $H$, as $D$ is the single closest 1-pole. The influences are surrounded by a blob on Figure 6.

PROPOSITION 4.1.15. *Given a generic function $m$ of arity $n$, and $i \in \{1, \ldots, n\}$, let $Pole^i_m = \{T_1, \ldots, T_l\}$, then*

*(1)* $\{Influence^i_m(T_1), \ldots, Influence^i_m(T_l)\}$ *is a partition of* $Dynamic^i_m$ *and*
*(2)* $\forall T \in Dynamic^i_m$

$$(T \in Influence^i_m(T_k) \Leftrightarrow min_{\preceq}\{T' \in Pole^i_m \mid T \preceq T'\} = \{T_k\}).$$

PROOF. Let us prove the first assertion. We first show that influences are pairwise disjoint. We proceed by contradiction. Suppose that $T_1$ and $T_2$ are in $Pole^i_m$, and $T \in Influence^i_m(T_1) \cap Influence^i_m(T_2)$. Hence, $T \preceq T_1$ and $T \preceq T_2$. By definition of $Influence^i_m(T_1)$ and $T \preceq T_2$, we infer that $T_1 \preceq T_2$, and symmetrically that $T_2 \preceq T_1$. Therefore $T_1 = T_2$, a contradiction.

We now show that $\bigcup_{k \preceq l} Influence^i_m(T_k) = Dynamic^i_m$.

$\subseteq$:If $T \in Incluence^i_m(T_k)$ then $closest\text{-}poles^i_m(T) = \{T_k\}$, and by Lemma 4.1.9, $T \in Dynamic^i_m$.

$\supseteq$:Let $T \in Dynamic^i_m$. If $T$ is a pole then $T \in Influence^i_m(T)$. Otherwise, $|closest\text{-}poles^i_m(T)| \leq 1$, by condition (4). Since $T \in Dynamic^i_m$, there exists an $i$-pole $T'$ such that $T \in cover(T')$. Thus, $|closest\text{-}poles^i_m(T)| > 0$. Then, there exists $T'$ such that $\{T'\} = closest\text{-}poles^i_m(T)$, and by definition 4.1.13, $T \in Influence^i_m(T')$. We prove now the second assertion.

$\Leftarrow$: This is shown by what precedes.

$\Rightarrow$: If $T \in Influence^i_m(T_k)$ then let $\{T'_k\} = min_{\preceq}\{T_p \in Pole^i_m \mid T \prec T_p\}$. By assertion (1), $T_k = T'_k$. □

As a consequence, each type of $Dynamic^i_m$ is in the influence of one and only one pole of $Pole^i_m$. We use this to define $pole^i_m$.

*Definition* 4.1.16. For each type $T$ in $\Theta$, we define $pole^i_m$:

—$\forall T \in Dynamic^i_m,\ pole^i_m(T) = T' \ \Leftrightarrow\ T \in Influence^i_m(T')$

—$\forall T \in \Theta - Dynamic^i_m,\ pole^i_m(T) = 0$

From this definition, and Proposition 4.1.15, we have the following

Corollary 4.1.17. *Given a generic function $m$ of arity $n$, $i \in \{1, \ldots, n\}$, for each type $T \in Dynamic^i_m$,*

$$min_{\preceq}\{T' \in Pole^i_m \mid T \preceq T'\} = \{pole^i_m(T)\}.$$

## 4.2 Main Results

The following two theorems respectively establish that in the $i$th dimension, the entries for types which do not belong to any influence can be eliminated and that the entries for types belonging to the same influence can be grouped.

Theorem 4.2.1. *For every generic function $m$ of arity $n$, and $i \in \{1, \ldots, n\}$, the entry for type $T$ can be eliminated from the dimension $i$ of the dispatch table of $m$ if $T$ does not belong to the influence of any $i$-pole.*

Proof. By condition (1), we note $T_i = T$, and we prove that if $T_i$ does not belong to the influence of any $i$-pole, then it verifies

$$\forall (T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_n) \in \Theta^{n-1}, MS(m(T_1, \ldots, T_i, \ldots, T_n)) = \emptyset.$$

By assertion (1) of Proposition 4.1.15, $T_i \notin Dynamic^i_m$. By Definition 4.1.4, $(T_1, \ldots, T_n) \notin Dynamic_m$. By Fact 4.1.5, $applicable(m(T_1, \ldots, T_n)) = \emptyset$. Hence by Definition 2.2, $MS(m(T_1, \ldots, T_n)) = \emptyset$.

As the cell associated with $(T_1, \ldots, T_n)$ is composed of members of $MS(m(T_1, \ldots, T_n))$, it is also empty. Consequently the entry for $T$ can be eliminated. □

Lemma 4.2.2. *Given a generic function $m$ of arity $n$, for all $s = (T_1, \ldots, T_n)$ in $Dynamic_m$, with $s_p = (pole^1_m(T_1), \ldots, pole^n_m(T_n))$, we have*

$$MS(m(s)) = MS(m(s_p))$$

PROOF. We have to prove that $applicable(m(s)) = applicable(m(s_p))$ and that $\preceq_s$ is the same order as $\preceq_{s_p}$. We first prove $applicable(m(s)) = applicable(m(s_p))$.

$\subseteq$ : As $s \in Dynamic_m$, $applicable(m(s))$ is not empty. Let $m_k(T_k^1, \ldots, T_k^n) \in applicable(m(s))$. From Definition 2.1, for all $i$, $1 \leq i \leq n$, $T_i \preceq T_k^i$. By Definition (4.1.6), $T_k^i$ is an $i$-pole. By Corollary 4.1.17, $pole_m^i(T^i) \preceq T_k^i$. Hence by Definition 2.1, $m_k$ is applicable to $s_p$.

$\supseteq$ : As $applicable(m(s)) \neq \emptyset$ and $applicable(m(s)) \subseteq applicable(m(s_p))$, we have $applicable(m(s_p)) \neq \emptyset$. Let $m_k \in applicable(m(s_p))$. From Definition 2.1, $\forall i$, $1 \leq i \leq n$, we have $pole_m^i(T_i) \preceq T_k^i$, and by Corollary 4.1.17, $T_i \preceq pole_m^i(T_i)$. By transitivity, $T^i \preceq T_k^i$, and by Definition 2.1, $m_k \in applicable(m(s))$.

We now prove that $\preceq_s = \preceq_{s_p}$. By Corollary 4.1.17, $\forall i$, $1 \leq i \leq n$, $T^i \preceq pole_m^i(T^i)$. Hence $s \preceq s_p$. By Definition 2.7, $\forall m_1, m_2 \in applicable(s), m_1 \preceq_{s_p} m_2$ if and only if $m_1 \preceq_s m_2$. □

THEOREM 4.2.3. *Let $m$ be a generic function of arity $n$, for each $i$, $1 \leq i \leq n$, and for each type $T \in Dynamic_m^i$, the entries for type $T$ and $pole_m^i(T)$ can be grouped in the dispatch table.*

PROOF. We prove that for each $i$, $1 \leq i \leq n$, $\forall T \in Dynamic_m^i$, $\forall (T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_n) \in \Theta^{n-1}$, and $T_i = T$, we have

$$MS(m(T_1, \ldots, T_i, \ldots, T_n)) = MS(m(T_1, \ldots, pole_m^i(T), \ldots, T_n)). \tag{7}$$

If there exists $j$, $j \neq i$, such that $T_j \notin Dynamic_m^i$ then $(T_1, \ldots, T_n) \notin Dynamic_m$, and by Fact 4.1.5, both sides of (7) are $\emptyset$. Assuming now that $(T_1, \ldots, T_n) \in Dynamic_m$, by Lemma 4.2.2,

$$MS(m(T_1, \ldots, T_n)) = MS(m(pole_m^1(T_1), pole_m^n(T_n))).$$

By Fact 4.1.7, $pole_m^i(T_i) \in Dynamic_m^i$ and

$$MS(m(T_1, \ldots, pole_m^i(T_i), \ldots, T_n)) = MS(m(pole_m^1(T_1), \ldots, pole_m^i(pole_m^i(T)), \ldots, pole_m^n(T_n))).$$

By Definitions 4.1.16 and 4.1.13, $pole_m^i(T_i) \in Pole_m^i$ and $pole_m^i(T_i) \in Influence_m^i(pole_m^i(T_i))$. Hence by Definition 4.1.16, $pole_m^i(pole_m^i(T)) = pole_m^i(T)$, which proves (7).

As the cells associated with $(T_1, \ldots, T_n)$ and $(T_1, \ldots, pole_m^i(T), \ldots, T_n)$ are both computed from the same ordered set, their values are identical, and by condition (2), the entries associated with $T$ and $pole_m^i(T)$ can be grouped. □

PROPOSITION 4.2.4. *Method precedence is monotonous in Cecil and Dylan.*

PROOF. In Cecil, method precedence only depends on subtyping:

$$\forall s \in Dynamic_m, m_1(T_1^1, \ldots, T_n^1) \preceq_s m_2(T_1^2, \ldots, T_n^2) \Leftrightarrow \forall i, 1 \leq i \leq n, T_i^1 \preceq T_i^2$$

As a consequence, for all $s, s'$ in $Dynamic_m$, $m_1 \preceq_s m_2 \Leftrightarrow m_1 \preceq_{s'} m_2$.

Dylan enforces a monotonicity constraint on CPLs [Barrett et al. 1996]:

*Fact* 4.2.5. If a class $T$ has a class $T'$ as supertype, and classes $T_a$ and $T_b$ appear in the CPLs of both $T$ and $T'$, they appear in the same order.

Consequently, $\preceq_T$ is monotonous, i.e., $\forall T, T', T_a, T_b \in \Theta$ such that $T \preceq T', T' \preceq T_a$, and $T' \preceq T_b$; thus we have

$$T_a \preceq_T T_b \quad \Leftrightarrow \quad T_a \preceq_{T'} T_b. \tag{8}$$

Let us show that method precedence is monotonous. Let $s = (T_1, \ldots, T_n)$ and $s' = (T'_1, \ldots, T'_n)$ be two signatures in $Dynamic_m$ such that $s \preceq s'$. Let $m_1(T^1_1, \ldots, T^1_n)$ and $m_2(T^2_1, \ldots, T^2_n)$ be two methods applicable to $s$ and $s'$ such that $m_1 \preceq_{s'} m_2$. Let $i$, $1 \leq i \leq n$, be such that $T^1_i \preceq_{T'_i} T^2_i$. From (8), if follows that $T^1_i \preceq_{T_i} T^2_i$. As, by Definition 2.5, $\forall j, j \neq i, T^1_j \not\preceq_{T'_i} T^2_j$, it follows from (8) that $\forall j, j \neq i, T^1_j \not\preceq_{T_i} T^2_j$. □

Method dispatch both in Cecil and Dylan uses the applicability criteria and obeys the monotonicity rule, which are the basic conditions for Theorems 4.2.1 and 4.2.3. As a consequence, both theorems apply to Cecil and Dylan.

### 4.3 General Algorithm

From the above definitions and theorems, the general algorithm to compute the compressed dispatch table of a generic function $m$ is the following:

(1) Compute the $i$-poles and their influence, for each selector $(m, i)$.

(2) Attribute an index to each $i$-pole.

(3) Compute the selector conflicts, define the color of each selector, and give an identifier to each selector of the same color.

(4) For every type $T$ and each argument position $i$, store the index of its $i$-pole in the entry of $T$ in the group argument array associated with selector $(m, i)$, and store the selector number at the same position in the corresponding selector array.

(5) For every signature $(T_1, \ldots, T_n) \in \prod_{i=1}^{n} Pole^i_m$, determine $MS(m(T_1, \ldots, T_n))$, and store the result (address of a set or index of a method) in the corresponding entry of the table, i.e., in $\mathcal{D}^c_m[m\_arg_1[T_1.type\_index], \ldots, m\_arg_n[T_n.type\_index]]$.

## 5. COMPUTING POLES

Computing the $i$-poles and influences of a generic function $m$ amounts to successively computing the pole of each type in $\Theta$, i.e., $pole^i_m$. This is needed because $T$ is a pole if and only if $pole^i_m(T) = T$, and if $T'$ is a pole then $Influence(T') = \{T \in \Theta \mid pole^i_m(T) = T'\}$. Moreover, the values of $pole^i_m$ are also needed to build the argument arrays, whereas the values of $Influence$ are not needed in the general algorithm given above.

We show that the computation of $pole^i_m$ can be done in a single pass over the set of types using $closest\text{-}poles^i_m$. We then present two techniques to optimize the computation of $closest\text{-}poles^i_m$, first by minimizing the set in which they need to be searched, then by using bit vectors to efficiently represent the subtyping relationship.

### 5.1 Single-Pass Traversal of the Type Graph

We consider a generic function $m$ of arity $n$, and an argument position $i$. We show that $pole^i_m(T)$ only depends on $closest\text{-}poles^i_m(T)$.

THEOREM 5.1.1. *Given a generic function $m$ of arity $n$, for each $i$, $1 \leq i \leq n$, for each $T$ in $\Theta$,*

*(1)* $(pole^i_m(T) = 0) \Leftrightarrow (T \notin Static^i_m$ *and* $closest\text{-}poles^i_m(T) = \emptyset)$,
*(2)* $(pole^i_m(T) = T',\ T' \neq T) \Leftrightarrow (T \notin Static^i_m$ *and* $closest\text{-}poles^i_m(T) = \{T'\})$, *and*
*(3)* $(pole^i_m(T) = T) \Leftrightarrow (T \in Static^i_m$ *or* $|closest\text{-}poles^i_m(T)| > 1)$.

PROOF. We first show that

$$T = pole^i_m(T) \Leftrightarrow T \in Pole^i_m. \tag{9}$$

If $T = pole^i_m(T)$, from Definition 4.1.16, $T \in Influence^i_m(T)$, and by Definition 4.1.13, $T \in Pole^i_m$. Conversely if $T \in Pole^i_m$, as $T \in \{T' \in Pole^i_m \mid T \preceq T'\}$, by Corollary 4.1.17 $T = pole^i_m(T)$.

From (3), $T \in Static^i_m \Rightarrow T \in Pole^i_m$; hence by (9), $pole^i_m(T) = T$. Thus, we have

$$T \in Static^i_m \Rightarrow pole^i_m(T) = T. \tag{10}$$

We finally have the following fact:

$$T \notin Pole^i_m \Rightarrow \{T' \in Pole^i_m \mid T \prec T'\} = \{T' \in Pole^i_m \mid T \preceq T'\} \tag{11}$$

(1) $\Rightarrow$: As $pole^i_m(T) \neq T$, from (10) we have $T \notin Static^i_m$. Suppose that $closest\text{-}poles^i_m(T) \neq \emptyset$; then by Lemma 4.1.9, $T \in Dynamic^i_m$ and by Definition 4.1.16, $pole^i_m(T) \neq 0$.

$\Leftarrow$: From Definition 4.1.6, $Static^i_m \subseteq Pole^i_m$; hence $closest\text{-}poles^i_m(T) = \emptyset$ implies $\{T' \in Static^i_m \mid T \prec T'\} = \emptyset$. As $T \notin Static^i_m$, from Definition 4.1.4, $T \notin Dynamic^i_m$. Using Definition 4.1.16, this implies $pole^i_m(T) = 0$.

(2) $\Rightarrow$: As $pole^i_m(T) \neq T$, from (10), $T \notin Static^i_m$. From Corollary 4.1.17 we have $\{T'\} = min_{\preceq}\{T'' \in Pole^i_m \mid T \preceq T''\}$. From (9), $T \notin Pole^i_m$; hence (11) applies, and from the definition of $closest\text{-}poles^i_m$, $\{T'\} = closest\text{-}poles^i_m(T)$.

$\Leftarrow$: From Lemma 4.1.9, $T \in Dynamic^i_m$; hence from Corollary 4.1.17, $\{pole^i_m(T)\} = min_{\preceq}\{T' \in Pole^i_m \mid T \preceq T'\}$. From (3) and (5), $T \notin Pole^i_m$. Hence (11) applies, and $\{pole^i_m(T)\} = min_{\preceq}\{T' \in Pole^i_m \mid T \prec T'\}$. Then by definition of $T'$ and of $closest\text{-}poles^i_m$, $pole^i_m(T) = T'$.

(3) $\Rightarrow$: From (9) $T \in Pole^i_m$; hence by Definition 4.1.6 and (5), $T \in Static^i_m$ or $|closest\text{-}poles^i_m(T)| > 1$.

$\Leftarrow$: If $T \in Static^i_m$, from (10), $pole^i_m(T) = T$. If $|closest\text{-}poles^i_m(T)| > 1$, from (5), $T \in Pole^i_m$ and from (9), $pole^i_m(T) = T$. □

We define now the linear order $\leq$ over $\Theta$, which is used by our algorithm to compute the poles. As $\preceq$ is a partial order over $\Theta$, it is possible to find a total order $\alpha$ over $\Theta$ that is an extension of $\preceq$, i.e.,

```
input      : an ordered set of types Θ≤, a generic function m, an argument position i
output     : a set of poles Poles
side-effect: updated pole attribute of the types

Poles ←∅
for T in (Θ, ≤) do
    if T ∈ Static^i_m then                          // case 3a of Theorem 5.1.1
        T.pole ← T                                  // T is a primary pole
        insert T into Poles
    else
        if closest-poles^i_m(T) = ∅ then            // case 1 of Theorem 5.1.1
            T.pole ← 0
        else
            if closest-poles^i_m(T) = {T_p} then    // case 2 of Theorem 5.1.1
                T.pole ← T_p
            else                                    // case 3b of Theorem 5.1.1
                T.pole ← T                          // T is a secondary pole
                insert T into Poles
```

Fig. 7. Pole computation algorithm.

$$(\forall T, T' \in \Theta,\ (T\ \alpha\ T' \text{ or } T'\ \alpha\ T)) \text{ and } (\forall T, T' \in \Theta,\ (T \preceq T' \Rightarrow T\ \alpha\ T')).$$

The topological sort of Knuth [1973] is a classical algorithm to obtain a linear extension from a partial order. We define $\leq$ to be the opposite of a linear extension of $\preceq$; hence the highest types according to $\preceq$ are the first according to $\leq$:

$$(\forall T, T' \in \Theta,\ (T \leq T' \text{ or } T' \leq T)) \text{ and } (\forall T, T' \in \Theta,\ (T \preceq T' \Rightarrow T' \leq T)).$$

We note $(\Theta, \leq)$ as the set of all types totally ordered by $\leq$. Our algorithm starts from the highest types and traverses the type graph downward to the most specific types. Thus, each type is treated before its subtypes and after its supertypes. Since the closest poles of $T$ only involve poles that are supertypes of $T$, and all the supertypes of $T$ have already been treated before $T$, all poles supertypes of $T$ are known when $T$ is treated. Thus, the body of the algorithm merely implements the case analysis given in Theorem 5.1.1.

In the algorithm of Figure 7, we use a record-like representation of types, with an attribute *pole*. A variable *Poles* records the set of poles.

*Example* 5.1.2. Figure 8 illustrates the computation of the 1-poles of method $m$. Poles appear in boxes, and primary poles are written in bold. The dashed arrow shows the order in which types are considered. The closest poles for each type are given between curly brackets. The closest poles of $D$ are $A$ and $B$; thus $D$ is a pole. The poles supertypes of $H$ are $D$ and $B$, and as $D$ is a subtype of $A$, the single closest pole of $H$ is $D$. Hence $H$ is not a pole, and its pole is $D$.

## 5.2 Computing the Closest Poles

The computation of the closest poles can be optimized by reducing the number of poles that need to be compared at each step. The following theorem reduces this

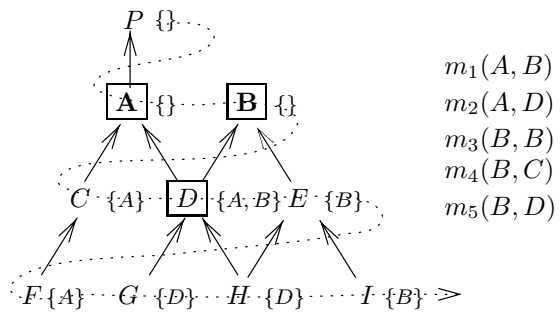



Fig. 8. Pole computation example.

number of comparisons to the number of direct supertypes of the current type.

THEOREM 5.2.1. *For all $T$ in $\Theta$, we have*

$$closest\text{-}poles^i_m(T) = min_{\preceq}\{pole^i_m(T') \mid T \ isa \ T'\}. \tag{12}$$

PROOF. Let $T_1, \ldots, T_N$ be the totally ordered list of types in $(\Theta, \leq)$. The proof is by induction on $k$, $1 \leq k \leq N$.

*Basis of the Induction* $T_1$ has no supertype; thus $\{pole^i_m(T') \mid T_1 \ isa \ T'\} = \emptyset$, and since $closest\text{-}poles^i_m(T_1) = \emptyset$, (12) holds.

LEMMA 5.2.2. *Let $(S, \preceq)$ be a partially ordered set; if $A$ and $B$ are two subsets of $S$, then*

$$((A \subset B) \ and \ (min_{\preceq}B \subset A)) \ \Rightarrow (min_{\preceq}A = min_{\preceq}B). \tag{13}$$

PROOF. First, let $A$ and $B$ be two subsets of $S$:

$$A \subset B \Rightarrow \forall x \in min_{\preceq}B, \ \nexists y \in min_{\preceq}A \text{ s.t. } y \prec x \tag{14}$$

To see that, suppose that $x \in min_{\preceq}B$ and $y \in min_{\preceq}A$. As $A \subset B$, $y \in B$. If $y \prec x$ then $x \notin min_{\preceq}B$, a contradiction.

Suppose that $A \subset B$ and $min_{\preceq}B \subset A$.

$\subseteq$: Let $x \in min_{\preceq}A$. As $A \subset B$, $x \in B$. Since $min_{\preceq}B \subset A$, by (13), $\nexists y \in min_{\preceq}(min_{\preceq}B), y \prec x$, i.e., $\nexists y \in min_{\preceq}B, \ y \prec x$. Hence, $x \in min_{\preceq}B$.

$\supseteq$: Let $x \in min_{\preceq}B$. As $min_{\preceq}B \subset A$, $x \in A$. Since $A \subset B$, by (13), $\nexists y \in min_{\preceq}A, \ y \prec x$. Hence, $x \in min_{\preceq}A$. □

We now come to the induction part.

*Induction:* Suppose that (12) holds for some $k$. Let $A = \{pole^i_m(T_{k'}) \mid T_{k+1} \ isa \ T_{k'}, 1 \leq k' \leq k\}$ and $B = \{T' \in Pole^i_m \mid T_{k+1} \prec T'\}$.

$A \subset B$: Let $T' \in A$. $T' \in Pole^i_m$ and $\exists k'$ such that $T_{k+1} \prec T_{k'} \preceq T'$. Thus, $T' \in B$.

$min_{\preceq}B \subset A$: Let $T' \in min_{\preceq}B$. As $T_{k+1} \prec T'$ either (i) $T_{k+1} \ isa \ T'$ or (ii) there exists an ascending chain C of the partially ordered set $(\Theta, isa)$ starting from $T_{k+1}$ and finishing at $T'$.

*Case (i)*: By (9), $pole^i_m(T') = T'$. Hence, $T' \in A$.

*Case* (*ii*): Let $C = T_{k+1}\ isa\ T^0 \ldots isa\ T'$. We first prove that $closest\text{-}poles^i_m(T^0) = \{T'\}$.

$T' \in closest\text{-}poles^i_m(T^0)$: Suppose there exists $T'' \in Pole^i_m$ such that $T^0 \preceq T'' \prec T'$; then $T^{k+1} \preceq T'' \prec T'$, which contradicts that $T' \in min_{\preceq}B$.

$closest\text{-}poles^i_m(T^0) \subseteq \{T'\}$: Suppose there exists $T'' \in closest\text{-}poles^i_m(T^0)$ such that $T' \neq T''$. Then, by Theorem 5.1.1, $pole^i_m(T^0) = T^0$. As $T_{k+1} \prec T^0 \prec T'$, $T' \notin closest\text{-}poles^i_m(T_{k+1})$, which contradicts that $T' \in min_{\preceq}B$.

Thus $pole^i_m(T^0) = T'$, and since $T_{k+1} \preceq T^0$, $T' \in A$.

We can now apply the above lemma, which gives that $min_{\preceq}\{pole^i_m(T_k) \mid T_{k+1}\ isa\ T_{k'}, 1 \leq k' \leq k\} = min_{\preceq}\{T' \in Pole^i_m \mid T_{k+1} \prec T'\} = closest\text{-}poles^i_m(T_{k+1})$. Thus (12) holds for $k+1$. □

Note that the poles of the direct supertypes of a type $T$ are also supertypes of $T$; thus this verification is not needed in the computation of the closest poles of $T$.

Nonetheless, (12) still requires to compute the minimum over a set, and $min_{\preceq}E$ has a cost of $|E|^2$. By Theorem 5.1.1, the Pole Computation Algorithm only needs to know if $|closest\text{-}poles^i_m(T)|$ is 0, 1, or more, and if there is a single closest pole, we need to compute it. In this latter case, the closest pole of $T$ has the greatest rank in $(\Theta, \leq)$ among all the poles of $T$.

Based on these considerations, we propose to compute the *pseudo-closest* poles of $T$. We assume that pole types are ranked in the ascending order of $\leq$. The algorithm consists of three iterations over the set $\{pole^i_m(T') \mid T\ isa\ T'\}$. In the first iteration, this set is assigned to the variable *candidates*. In the next iteration, the type $T^c$ with the greatest rank is returned, and finally it is checked if it is a subtype of all the other poles. If there is one pole $T'$ for which $T^c$ is not a subtype then $T'$ is also returned,and the computation stops.

The algorithm assumes that each type has a *pole-rank* attribute used for pole types only. Because poles are found in the ascending order of $\leq$, pole ranks are also generated in this ascending order. This algorithm is used instead of *closest-poles*.

### 5.3 Testing the Subtyping Relationship

In this section, we show how to optimize the many subtyping tests that are needed in the *Peudo-Closest-Poles* Algorithm.

We associate with each pole the set of poles of its direct supertypes, implemented as a vector of bits. Because this number of poles is usually small, a few bit vectors are needed. The vector is built during the computation of poles and is released when all poles in a given dimension have been obtained. The subtyping test can then be performed in constant-time by testing the value of a bit.

The poles in a given dimension form an ascending chain of $(\Theta, \leq)$, denoted $(T^k)_{1 \leq k \leq l}$, where $k$ indicates the rank of the pole in the chain.

*Definition* 5.3.1. For each $k$ in $\{1, \ldots, l\}$, the bit vector $\mathcal{H}^k$ of length $l$ associated with $T^k$ is defined as

$$\forall k',\ 1 \leq k' \leq l,\ \mathcal{H}^k(k') = 1 \Leftrightarrow T^{k'} \succ T^k$$
$$\mathcal{H}^k(k') = 0 \Leftrightarrow T^{k'} \not\succ T^k.$$

The following theorem allows us to build these arrays at a very low cost:

```
input    : a type T, an argument position i, a generic function m
output   : a set of poles pseudo-closest

Iteration 1: compute candidates
candidates ← {pole^i_m(T') | T isa T'}

if candidates = ∅ then
   pseudo-closests = ∅
else

   Iteration 2: search for type T^c with greatest rank
   T^c ← any-element(candidates)
   for T' in candidates
      if T'.pole-rank > T^c.pole-rank then
         T^c ← T'

   Iteration 3: check T^c is a subtype of all members of candidates
   pseudo-closest ← {T^c}
   for T' in candidates
      if T^c ⋠ T' then
         insert T' into pseudo-closest
         break
```

Fig. 9. Pseudo-Closest Poles algorithm.

THEOREM 5.3.2. *For each $k$ in $\{1, \ldots, l\}$, we have*

$$\mathcal{H}^k = \bigvee_{T^{k'} \in \{pole^i_m(T') \mid T^k \ isa \ T'\}} \mathcal{H}^{k'} \vee (2^{k'}). \tag{15}$$

The symbol $\vee$ denotes the logical-OR operation, and $2^{k'}$ is the bit vector with a "1" at position $k'$ and "0" everywhere else.

PROOF. As $\mathcal{H}^k$ is a bit vector, (15) is equivalent to

$$\mathcal{H}^k[k'] = 1 \Leftrightarrow (\ \exists k'' \text{ s.t. } (\ \ T^{k''} \in \{pole^i_m(T') \mid T^k \ isa \ T'\} \text{ and } (\ \mathcal{H}^{k''}[k'] = 1 \text{ or } k' = k'')))$$

Using Definition 5.3.1, this is equivalent to

$$\begin{aligned} T^k \prec T^{k'} &\Leftrightarrow (\ \exists T^{k''} \in \{pole^i_m(T') \mid T^k \ isa \ T'\} \text{ s.t. } (\ T^{k''} \prec T^{k'} \text{ or } T^{k''} = T^{k'}\ )) \\ &\Leftrightarrow (\ \exists T^{k''} \in \{pole^i_m(T') \mid T^k \ isa \ T'\} \text{ s.t. } T^{k''} \preceq T^{k'}\ ) \end{aligned}$$

We prove the last equivalence.

$\Leftarrow$: As $T^k \prec T' \preceq T^{k''}$, and $T^{k''} \preceq T^{k'}$, by transitivity of $\preceq$, $T^k \prec T^{k'}$.

$\Rightarrow$: As $T^{k'} \in Pole^i_m$ and $T^k \prec T^{k'}$, by Definition 4.1.8, $\exists T^{k''} \in closest\text{-}poles^i_m(T^k)$ such that $T^{k''} \preceq T^{k'}$. Let $A = \{pole^i_m(T') \mid T^k \ isa \ T'\}$. From Theorem 5.2.1, $closest\text{-}poles^i_m(T^k) = min_{\preceq} A$. Hence $T^{k''} \in min_{\preceq} A$, and $T^{k''} \in A$. □

This result enables us to easily build a vector $\mathcal{H}^k$ as the logical-OR between the bit vectors associated with the poles of the direct supertypes of $T^k$ and a bit vector in which one-valued bits refer to the direct supertypes of $T^k$.

```
input    : two poles T^k and T^k'
output   : true if and only if T^k ≺ T^k', else false

if T^k.pole-rank > T^k'.pole-rank then
    if H^k[k'] = 1 then
        return(true)
return(false)
```

Fig. 10. Subtyping test.

```
input    : A new pole T, its closest poles closest-poles
output   : modified T, modified H, modified last-pole-rank

increment last-pole-rank
T.pole-rank ← last-pole-rank
for T_c in {pole^i_m(T') | T isa T'} do
    H^{T.pole-rank} ← H^{T.pole-rank} ∨ H^{T_c.pole-rank} ∨ 2^{T_c.pole-rank}
```

Fig. 11. Pole initialization.

The simple test of Figure 10 tests if a pole $T^k$ is a subtype of a pole $T^{k'}$. It assumes that $\mathcal{H}$ is implemented as an array of arrays of bits, such that $\mathcal{H}^k$ is the $k$th array of bits, and $\mathcal{H}^k[k']$ is the $k'$th bit in this array.

The ranking of poles in $(\Theta, \leq)$ and the construction of bit vectors are progressively done as long as a new pole is found by the routine in Figure 11, which must be invoked by the Pole Computation Algorithm (Figure 7).

As a pole $T^k$ cannot be a subtype of $T^{k'}$ if $k < k'$, we have $\forall k, k', (k < k' \Rightarrow \mathcal{H}^k(k') = 0)$. Hence $\mathcal{H}^k$ can have an effective length of $k - 1$, which allows us to allocate this vector as soon as pole $T^k$ is found.

### 5.4 Worst-Case Complexity

In this section, we denote $|m|$ as the number of methods of a generic function $m$, $\mathcal{E}$ the number of edges in the type graph, $\mu$ the number of types with two supertypes or more, and we use the following definition:

*Definition* 5.4.1. Let $T \in \Theta$. The set of direct supertypes of $T$, denoted as $dst(T)$, is defined as

$$dst(T) = \{T' \mid T \; isa \; T'\}.$$

*Cost Units.* Table I summarizes the cost units of the basic operations used in our algorithms. All these operations are supposed to execute in constant time. Bit vectors have a fixed length; hence they can be represented as an array of integers, and hence *BitTest* and *BitSet* execute in constant time. For *BitOr*, we denote $\omega$ as the size of a memory word. We do not assume any duplicate removal in sets; hence *InsSet* is constant. We also assume that the number of elements is memorized into the set representation; hence *SetCard* executes in constant time.

*Subtyping Test.* the cost of the subtyping test in Figure 10 is

$$SbTest = IntComp + BitTest.$$

Table I. Cost Units

| Unit | Description of Basic Operation |
|---|---|
| *IntComp* | comparison between two integers |
| *VarAssg* | variable assignment |
| *BitTest* | test the value of an entry in an array of bits |
| *BitSet* | sets the value of an entry in an array of bits |
| *BitOr* | logical-OR between two memory words |
| *InsSet* | insertion of an element into a set |
| *SetCard* | yields the cardinality of a set |
| *SetElem* | take an element out of a set |

*Pseudo-Closest-Poles.* The *Pseudo-Closest-Poles* algorithm (Figure 9) essentially consists of three loops over the set of direct supertypes of a type $T$.

—*Loop 1:* Collects in *candidates* the poles of the direct supertypes. This costs

$$L_1(T) = |dst(T)| \times InsSet.$$

—*Loop 2:* Searches for the pole $T^c$ with the greatest rank. Each iteration consists at worst of a comparison between integers and an assignment. This costs

$$L_2(T) = |dst(T)| \times (IntComp + VarAssg).$$

—*Loop 3:* Checks that $T^c$ is a subtype of all the other poles. It includes the comparison $T' \not\leq T^c$,which consists of an equality comparison between the ranks of $T'$ and $T^c$, and a subtyping test. Hence each iteration consists at worst of an equality comparison, a subtyping test, and an insertion into *pseudo-closest*. This costs

$$\begin{aligned} L_3(T) &= |dst(T)| \times (IntComp + SbTest + InsSet) \\ &= |dst(T)| \times (2 \times IntComp + BitTest + InsSet). \end{aligned}$$

To sum up, this algorithm costs

$$\begin{aligned} PCP(T) &= L_1(T) + L_2(T) + L_3(T) \\ &= |dst(T)| \times (3 \times IntComp + BitTest + 2 \times InsSet). \end{aligned}$$

*Pole Initialization.* The pole initialization routine in Figure 11 essentially consists of a loop over the direct supertypes. $|\mathcal{H}|$ being the number of bits in a vectors, a vector is stored in $(|\mathcal{H}|/\omega)+1$ memory words (we assume integer division). Hence the cost of pole initialization is

$$PI(T) = |dst(T)| \times ((\frac{|\mathcal{H}|}{\omega} + 1) \times BitOr + BitSet + VarAssg).$$

$|\mathcal{H}|$ is at least the total number of poles to be computed, which cannot be predicted. As the number of primary poles is bounded by $|m|$, and the number of secondary poles is bounded by $\mu$, we use $|\mathcal{H}| = |m| + \mu$. In the following, we use

$$PI'(T) = |dst(T)| \times ( (\frac{|m| + \mu}{\omega} + 1) \times BitOr + BitSet + VarAssg ).$$

```
input   : an ordered set of types (Θ, ≤), a generic function m, an argument position i
output  : a set of poles Poles, modified types
1.  Poles ← ∅
    Step 1: mark primary poles
2.  increment current-mark
3.  for m_k in m.methods do
4.      m_k.formals[i].mark ← current-mark
    Step 2:
5.  for T in (Θ, ≤) do
6.      if T.mark = current-mark then   // tests if T ∈ Static^i_m (case 3a of Theorem 5.1.1)
7.          pole-init(T)                // T is a primary pole
8.          T.pole ← T
9.          insert T into Poles
10.     else
11.         closest ← pseudo-closest-poles(T, m, i)
12.         size-closest ← |closest|
13.         if size-closest = 0 then    // case 1 of Theorem 5.1.1
14.             T.pole ← 0
15.         else
16.             if size-closest = 1 then   // case 2 of Theorem 5.1.1
17.                 T.pole ←first(closest)
18.             else                    // case 3b of Theorem 5.1.1
19.                 pole-init(T)        // T is a secondary pole
20.                 T.pole ← T
21.                 insert T into Poles
```

Fig. 12. Detailed algorithm for pole computation.

*Pole Computation.* We compute the worst-case complexity of pole computation on the detailed algorithm of Figure 12. In particular, the membership test of a type $T$ in $Static^i_m$ can be done immediately by marking each type of $Static^i_m$. An integer mark is generated before each pole computation and assigned to the variable *current-mark*. This new mark is used to identify the elements of $Static^i_m$, by assigning it to the attribute *mark* of each type in $Static^i_m$. Then, to test if a type $T$ is a member of $Static^i_m$, it suffices to test if $T.mark$ is equal to *current-mark*. We use a record-like representation of generic functions and methods. Generic functions have an attribute *methods* that represents the set of methods, and methods have an attribute $formals$ that represents the array of their formal arguments, which are primary poles. The algorithm essentially consists of two loops:

—*Loop 1 (lines 3–4):* This loop marks the $i$th formal argument of each method of the generic function $m$, which costs

$$L_1 = |m| \times VarAssg.$$

—*Loop 2 (lines 5–21):* This is the main loop over $\Theta$, which consists of several tests.
—If $T$ is a primary pole, only lines 6–9 are executed, which costs

$$L^1_2(T) = IntComp + PI'(T) + VarAssg + InsSet.$$

—If $T$ has no pole, lines 6 and 11–14 are executed, which costs

$$L_2^2(T) = 2 \times IntComp + PCP(T) + 3 \times VarAssg + SetCard.$$

—If $pole_m^i(T) = T', T \neq T'$, lines 6, 11–13, and 16–17 are executed, which costs

$$L_2^3(T) = 3 \times IntComp + PCP(T) + 3 \times VarAssg + SetCard + SetElem.$$

—If $T$ is a secondary pole, lines 6, 11–13, 16, and 19–21 are executed, which costs

$$L_2^4(T) = 3 \times IntComp + PCP(T) + 3 \times VarAssg + SetCard + PI'(T) + InsSet.$$

In the worst case, the cost is $max\{L_2^1(T), L_2^2(T), L_2^3(T), L_2^4(T)\}$, which is bounded by

$$L_2(T) = 3 \times IntComp + PCP(T) + PI'(T) + 3 \times VarAssg + SetCard + InsSet + SetElem.$$

The total cost of pole computation is

$$\begin{aligned}
PC &= L_1 + \sum_{T \in \Theta} L_2(T) \\
&= |m| \times VarAssg + \sum_{T \in \Theta} (3 \times IntComp + PCP(T) + PI'(T) + 3 \times VarAssg \\
&\qquad + SetCard + InsSet + SetElem) \\
&= |m| \times VarAssg + \sum_{T \in \Theta} (3 \times IntComp + 3 \times VarAssg + SetCard + InsSet \\
&\qquad + SetElem + |dst(T)| \times (3 \times IntComp + BitTest \\
&\qquad\qquad + 2 \times InsSet + VarAssg \\
&\qquad\qquad + BitSet + (\tfrac{|m|+\mu}{\omega} + 1) \times BitOr)) \\
&= |m| \times VarAssg + |\Theta| \times (3 \times IntComp + 3 \times VarAssg + SetCard + InsSet \\
&\quad + SetElem) + (\sum_{T \in \Theta} dst(T)) \times (3 \times IntComp + BitTest + 2 \times InsSet \\
&\qquad\qquad + (\tfrac{|m|+\mu}{\omega} + 1) \times BitOr + BitSet + VarAssg).
\end{aligned}$$

As $\sum_{T \in \Theta} dst(T) = \mathcal{E}$, we finally have

$$PC = \mathcal{O}(VarAssg \times |m| + (3 \times IntComp + \ldots + SetElem) \times |\Theta| + BitOr \times \frac{|m| + \mu}{\omega} \times \mathcal{E}).$$

As $\omega > 1$, the cost of pole computation is at worst in $|m| + |\Theta| + (|m| + \mu) \times \mathcal{E}$. $|m| + \mu$ is our bound of the number of poles, which is also bounded by $|\Theta|$; hence another bound to the complexity of pole computation is $|m| + |\Theta| \times \mathcal{E}$.

## 6. FILLING UP DISPATCH TABLES

The computation of poles determines the structure of the dispatch table and the associated argument arrays. Filling up the dispatch table requires that we compute $MS$ for each signature of poles. Thus, the more the table is compressed, the less are computations needed. In this section, we optimize the computation of $MS$ when it returns a set consisting of the most specific applicable methods. In particular, this covers the case of Cecil with static and dynamic checking. We first show that

the most specific methods can be computed on a subset of the applicable methods obtained by examining the signatures of poles in an order compatible with signature precedence. We explain how such an order is obtained. We finally give the algorithm to fill up the dispatch table and its worst-case complexity.

### 6.1 MSA Method Computation

We first define the notions of pole signature, candidate signature, and conflicting methods.

*Definition* 6.1.1. Given a generic function $m$ of arity $n$, the set of pole signatures is

$$Poles_m = \prod_{i=1}^{n} Pole_m^i.$$

*Definition* 6.1.2. For each $(T_1, \ldots, T_n)$ in $Dynamic_m$, the set of candidate signatures of $(T_1, \ldots, T_n)$ is

$$\begin{aligned} &candidate\text{-}signatures_m((T_1, \ldots, T_n)) \\ &\quad = \{(T'_1, \ldots, T'_n) \mid \exists i \in \{1, \ldots, n\}, \exists T \in \Theta \text{ s.t. } T'_i = pole_m^i(T) \text{ and } T_i \textit{ isa } T \\ &\qquad\qquad \textit{and } \forall j \neq i,\ T'_j = T_i\}. \end{aligned}$$

*Example* 6.1.3. Consider the types and methods in Figure 3. The 1-poles are $A$, $B$, and $D$, and the 2-poles are $B$, $C$, and $D$. The candidate signatures of $(D, D)$ are obtained by substitution with the poles of the direct supertypes. The 1-poles of $D$'s direct supertypes are $A$ and $B$, and their 2-poles is simply $B$. Hence the candidate signatures of $(D, D)$ are $(A, D), (B, D)$, and $(D, B)$.

*Definition* 6.1.4. The set of conflicting methods of an invocation $m(s)$, denoted as $conflicting(m(s))$, is

$$conflicting(m(s)) = min_{\preceq_s} MS(m(s)) = min_{\preceq_s} applicable(m(s)).$$

When $m(s)$ is not ambiguous, we have

$$conflicting(m(s)) = \{MSA(m(s))\}.$$

The above definition exactly matches the dynamic checking mode of Cecil, which needs the MSA method of unambiguous invocations, and the conflicting methods of ambiguous invocations. It also applies to its static checking mode, in which the type-checker searches possible ambiguous invocations in the source code, and where the MSA methods are needed at run-time. Finally, the definition captures the case of Dylan when "next-method" is not used.

THEOREM 6.1.5. *For each $s \in Poles_m$, which is not the signature of a method, we have*

$$conflicting(m(s)) = min_{\preceq_s} \bigcup_{s' \in candidate\text{-}signatures_m(s)} conflicting(m(s')).$$

PROOF. Let $s = (T^1, \ldots, T^n)$. If $\mathit{conflicting}(m(s)) = \emptyset$, then $\mathit{applicable}(m(s))$ is empty. From Definition 6.1.2, the candidate signatures of $s$ are more generic than $s$. By transitivity of $\preceq$, no method is also applicable to these signatures; hence both sides of the equality are $\emptyset$. We assume now that $\mathit{conflicting}(m(s)) \neq \emptyset$, and we denote $A = \bigcup_{s' \in \mathit{candidate\text{-}signatures}_m(s)} \mathit{conflicting}(m(s'))$ and $B = \{m_k \mid m_k \succeq s\}$.

—$A \subset B$: Let $m_k \in A$, and $s' \in \mathit{candidate\text{-}signatures}_m(s)$ such that $m_k \in \mathit{conflicting}(m(s'))$. From Definition 6.1.2, $s \preceq s'$. From Definition 2.1, $m_k$ is transitively applicable to $s$. By definition of $B$, $m_k \in B$.

—$min_{\preceq_s} B \subset A$: We proceed by contradiction. Let $m_s \in min_{\preceq_s} B$, i.e., $m_s \in \mathit{conflicting}(m(s))$. We assume that $m_s \notin A$. Let $T_s^1, \ldots, T_s^n$ be the formals of $m_s$; we have $\mathit{conflicting}(m(T_s^1, \ldots, T_s^n)) = \{m_s\}$. As $m_s \notin A$, we have $(T_s^1, \ldots, T_s^n) \notin \mathit{candidate\text{-}signatures}_m(s)$. By Definition 2.1, and as $s$ is not the signature of a method, $s \prec (T_s^1, \ldots, T_s^n)$. We build a candidate signature $s_c$ such that $s_c \prec (T_s^1, \ldots, T_s^n)$. Two cases may occur:

*Case 1:* $\exists i \in \{1, \ldots, n\}$, with $i$ being unique, such that $T_s^i \succ T^i$. As $T_s^i \in \mathit{Pole}_m^i$, by Corollary 4.1.17 we have $\mathit{pole}_m^i(T_s^i) = T_s^i$. We show by contradiction that $T_s^i$ is not a direct supertype of $T^i$: assuming $T^i \ \mathit{isa}\ T_s^i$, we have $(T_s^1, \ldots, T_s^n) \in \mathit{candidate\text{-}signature}_m(s)$, a contradiction. Hence $\exists T_0 \in \Theta$ such that $T^i \ \mathit{isa}\ T_0 \prec T_s^i$. As $T_s^i \in \mathit{Static}_m^i$, by Definition 4.1.4, $T_0 \in \mathit{Dynamic}_m^i$. Let $T_0' = \mathit{pole}_m^i(T_0)$. As $T_s^i \in \mathit{Pole}_m^i$ and $T_0 \prec T_s^i$, by Corollary 4.1.17, $T_0' \preceq T_s^i$. As $T^i \ \mathit{isa}\ T_0$, $T_0' = \mathit{pole}_m^i(T_0)$ and $(T_s^1, \ldots, T_s^n) \notin \mathit{candidate\text{-}signatures}_m(s)$, by Definition 6.1.2, $T_0' \neq T_s^i$. Hence $s_c = (T^1, \ldots, T^{i-1}, T_0', \ldots, T_n)$ is a candidate signature, and $s_c \prec (T_s^1, \ldots, T_s^n)$.

*Case 2:* $\exists i, j \in \{1, \ldots, n\}$, with $i \neq j$, such that $T_s^i \succ T^i$ and $T_s^j \succ T^j$. Let $T_0$ such that $T_i \ \mathit{isa}\ T_0 \preceq T_s^i$. As $T_s^i \in \mathit{Static}_m^i$, by Definition 4.1.4, $T_0 \in \mathit{Dynamic}_m^i$. Let $T_0' = \mathit{pole}_m^i(T_0)$. As $T_s^i \in \mathit{Pole}_m^i$ and $T_0 \prec T_s^i$, by Corollary 4.1.17, $T_0' \preceq T_s^i$. Let $s_c$ be the candidate signature $(T^1, \ldots, T^{i-1}, T_0, \ldots, T^n)$. As $T_j \prec T_s^j$, $s_c \prec (T_s^1, \ldots, T_s^n)$.

As $s_c \prec (T_s^1, \ldots, T_s^n)$, from Definition 2.1, $m_s$ is applicable to $s_c$. As $m_s \notin A$ and $s_c$ is a candidate signature of $s$, by construction of A, $m_s \notin \mathit{conflicting}(m(s_c))$. Hence by Definition 6.1.4, $\exists m_s' \in \mathit{applicable}(m(s_c))$ such that $m_s' \prec_{s_c} m_s$. As $s \preceq s_c$, by Definition 2.7, we have $m_s' \prec_s m_s$ also. As $m_s' \in \mathit{applicable}(m(s_c))$ and $s \preceq s_c$, from Definition 2.1, $m_s' \in \mathit{applicable}(m(s))$. From Definition 6.1.4, because of $m_s'$, $m_s \notin \mathit{conflicting}(s)$, a contradiction. Hence $m_s \in A$.

As the set of methods applicable to $m(s)$, ordered by $\preceq$, is a partially ordered set, by (12), we have $min_{\preceq_s} A = min_{\preceq_s} B$. □

This theorem is useful if the conflicting methods of the candidate signatures are known before computing $\mathit{conflicting}(m(s))$. We propose to ensure this property by considering signatures in the ascending order of $\leq$, $\leq$ being the opposite of a linear extension of the precedence order $\preceq$:

$$(\forall s, s' \in \Theta^n, (s \leq s' \text{ or } s' \leq s))$$

and

$$(\forall s, s' \in \Theta^n, (s \preceq s' \Rightarrow s' \leq s))$$

As for pole computation, this order of pole signatures ensures that for each signature $s$, the signatures $s'$, such that $s \prec s'$, are considered before $s$, because $s' < s$. As all the candidate signatures of $s$ are in this case, their conflicting methods are computed before considering $s$, and Theorem 6.1.5 can be applied. Note that the same order is used in the method disambiguation algorithm presented in Amiel and Dujardin [1996], which allows us to perform table fillup and method disambiguation at the same time.

### 6.2 Ordering the Pole Signatures

Ordering the pole signatures in an order that is compatible with precedence comes down to transforming a partially ordered set $(\mathcal{S}, \preceq)$ into a totally ordered set $(\mathcal{S}', \leq)$, where $\mathcal{S}$ is the set of pole signatures. A classical algorithm is given in Knuth [1973]. The basic idea is to pick a first element that has no predecessor, remove this element from $\mathcal{S}$, append it to the originally empty set $\mathcal{S}'$, and start over until $\mathcal{S}$ is empty. In the case of pole signatures, it is necessary to scan the set of pole signatures to find that a given signature has no predecessor. Hence, ordering the pole signatures has an a priori complexity of $O(|Poles_m|^2)$.

However, it is possible to obtain a complexity of $O(|Poles_m|)$ if the poles of each dimension, $Pole_m^i$, are themselves sorted in an order compatible with signature precedence. Indeed, it suffices to produce the signatures in the lexicographic ordering generated by the total orders on the poles. This ordering is inexpensive, as poles are already produced in this order by the algorithm of Figure 7. The total order $\leq$ on $\Theta^n$ is defined as follows:

*Definition* 6.2.1. Given a generic function $m$ of arity $n$, $(T_1, \ldots, T_n)$ and $(T'_1, \ldots, T'_n)$ in $Poles_m$, we have

$((T_1, \ldots, T_n) \leq (T'_1, \ldots, T'_n)) \Leftrightarrow$ ( $(T_1, \ldots, T_n) = (T'_1, \ldots, T'_n)$ or $(\exists i_0,\ i_0 = min\{i \mid T_i \neq T'_i\}$ and $T_{i_0} < T'_{i_0})$.

THEOREM 6.2.2. *Given a generic function $m$, the relation $\leq$ defines a total order on $Poles_m$ which is an extension of the precedence order $\preceq$.*

PROOF. Let $n$ be the arity of $m$. Let $s = (T_1, \ldots, T_n), s' = (T'_1, \ldots, T'_n) \in Poles_m$.

*Antisymmetry.* We assume that $s \leq s'$ and $s' \leq s$. We proceed by contradiction: assuming $s \neq s'$, let $i_0 = min\{i \mid T_i \neq T'_i\}$. By Definition 6.2.1, $s \leq s'$ implies $T_{i_0} < T'_{i_0}$, and $s' \leq s$ implies $T'_{i_0} < T_{i_0}$, a contradiction.

*Transitivity.* Let $s'' = (T''_1, \ldots, T''_n) \in Poles_m$. We assume that $s \leq s'$ and $s' \leq s''$. We consider two cases:

—$s = s'$ or $s' = s''$: By substitution of $s'$ with $s$ or with $s''$, $s \leq s''$.

—$s \neq s'$ and $s' \neq s''$: Let $i_0 = min\{i \mid T_i \neq T'_i\}$, $i'_0 = min\{i \mid T'_i \neq T''_i\}$, and $i''_0 = min\{i_0, i'_0\}$. Then $\forall i, i < i''_0, T_i = T''_i$. If $i''_0 = i_0$ and $i''_0 < i'_0$, then $T_{i''_0} < T'_{i''_0}$ and $T'_{i''_0} = T''_{i''_0}$; hence $T_{i''_0} < T''_{i''_0}$. In the same way, if $i''_0 = i_0$ and $i''_0 < i'_0$, then $T_{i''_0} < T''_{i''_0}$. Finally if $i''_0 = i_0$ and $i''_0 = i'_0$, then $T_{i''_0} < T'_{i''_0} < T''_{i''_0}$. By Definition 6.2.1, this implies $s \leq s''$.

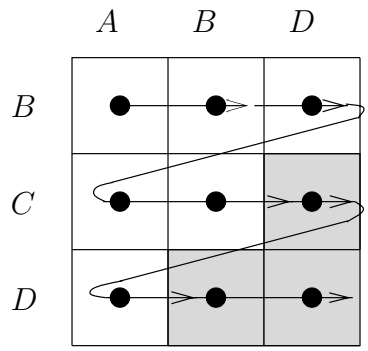



Fig. 13. Order of pole signatures.

```
input      : a generic function m, an array of pole lists P, an empty table dispatch
side-effect: filled dispatch table

Step 1: initialization
1. (Poles_m, ≤) ← OrderedPoleSignatures(P)

Step 2: dispatch of method signatures
2. for m_k in m.methods do
3.     dispatch[m_k.formals] ←m_k

Step 3: computation of the MSA methods
4. for s in (Poles_m, ≤) do
5.     if dispatch[s] = 0 then            // s is not a method signature
6.         dispatch[s] ← most-specific(s, dispatch)
```

Fig. 14. Table Fill Up algorithm.

*Total Order.* If $s = s'$ then $s \leq s'$. If not, then $\{i \mid T_1 \neq T'_i\} \neq \emptyset$, hence $i_0 = min\{i \mid T_i \neq T'_i\}$ is defined. As $\leq$ is a total order over $\Theta$, either $T_i \leq T'_i$ or $T'_i \leq T_i$, and respectively $s \leq s'$ or $s' \leq s$.

*Extension of* $\preceq$. We assume that $s \preceq s'$. If $s = s'$ then $s \leq s'$. If not, let $i_0 = min\{i \mid T_i \neq T'_i\}$. As $s \preceq s'$, by definition of $\preceq$, $T_{i_0} \preceq T'_{i_0}$. As $T_{i_0} \neq T'_{i_0}$, $T_{i_0} \prec T'_{i_0}$. As $<$ is an extension of $\leq$ on $\Theta$, $T_{i_0} < T'_{i_0}$. Hence by Definition 6.2.1, $s \preceq s'$. □

*Example* 6.2.3. The table in Figure 13 represents the pole signatures of the methods and types of Figure 3. The order on 1-poles (resp., 2-poles) in lines (resp., columns) is compatible with argument subtype precedence. A total order of $Poles_m$ is a path through this table. Such a path is compatible with signature precedence if it traverses each signature $s$ before the signatures on the right and below $s$. The path given by a lexicographic ordering, as shown on Figure 13, satisfies the condition. For example, the signatures that are more specific than $(B, C)$ are all included in the gray area.

### 6.3 Table Fill Up Algorithm

Figure 14 gives the fillup algorithm of table *dispatch* for a generic function $m$. This algorithm proceeds in three steps. The first step invokes the function *OrderedPoleSignature* that builds the ordered poles signatures $(Poles_m, \preceq)$, using the array of pole lists $P$, where $P[i]$ is $(Pole^i_m, \leq)$. The second step fills the cells of *dispatch* whose indices are signatures of methods. The value of each of these cells is a single method when method dispatch disambiguation is performed (as in Amiel and Dujardin [1996]), or it is a pointer to the set of conflicting methods. The third step considers the other pole signatures ordered by $\leq$ using Theorem 6.1.5. It assigns the cell associated with the current signature in *dispatch*, using the *Most-Specific* algorithm. This algorithm needs *dispatch* as argument in order to collect the conflicting methods of candidate signatures.

### 6.4 Most Specific Method Computation

To compute the complexity of the Table Fill Up Algorithm, we need to specify the *Most-Specific* function used to compare methods. This function depends on the method specificity criterion of the language. In this section, for simplicity, we use argument subtype precedence, without any additional criterion.

In this case, the specificity relationship on methods is the same as the relation $\preceq$ on their signatures. Hence the order $\leq$ on pole signatures also defines a linear extension of the specificity order for methods. Given a generic function $m$, we rank all methods of $m$ in the ascending order of their signatures in $(Poles_m, \leq)$, the rank of method $m_k(T^1_k, \ldots, T^n_k)$ being $k$. As in Section 5.3, with each method $m_k$ we associate a bit vector called higher-methods and denoted as $higher\text{-}methods(m_k)$. The $k'$th bit of $higher\text{-}methods(m_k)$ is 1 if and only if $m_k$ is more specific than $m_{k'}$. As shown by Theorem 5.3.2 in the context of higher-poles's arrays, $higher\text{-}methods(m_k)$ is computed as a combination of the methods higher than the MSA for the candidate signatures of $(T^1_k, \ldots, T^n_k)$.

Operationally, the ranking of methods and the construction of the higher-methods vectors are done during the traversal of $(Poles_m, \leq)$ required by the Table Fill Up Algorithm. In this traversal, the signatures of methods are identified by existing table's cells. The corresponding methods are then treated by the Higher Methods Initialization Algorithm in Figure 15, which assumes that both the rank and the higher methods are represented as attributes *rank* and *higher-methods*. This solution allows to treat the methods in the ascending order of $\leq$. Thus, the higher methods of the MSA methods of the candidate signatures are already defined. Finally, the routine collects each method in the order of their rank into the list *ordered-methods*, which is needed by the *Most-Specific* algorithm.

The computation of the most specific method using the higher-methods bits, in Figure 16, proceeds in two steps. The first step marks the conflicting methods of candidate signatures, as described for marking the members of $Static^i_m$ in Section 5.4. The second step collects conflicting methods in the reverse order of *ordered-methods*, i.e., from the method with the highest rank downward. Each marked method $m$ is compared with the current set of the conflicting methods $m^c$, using the higher-methods bits. As each $m^c$ has been collected in a previous iteration, it has a greater mark and cannot be more generic than $m$. Hence it is only relevant to test if $m \succ m^c$. If no conflicting method $m^c$ is more specific than $m$,

```
input      : a method meth, a table dispatch
side-effect: modified meth, modified last-method-rank, modified ordered-methods
1. meth.rank ← last-method-rank
2. increment last-method-rank
3. for other in {dispatch[s'] | s' ∈ candidate-signatures(meth.formals)} do
4.     meth.higher-methods ←meth.higher-methods ∨ other.higher-methods ∨ 2^other.rank
5. append meth to ordered-methods
```

Fig. 15. Higher Methods's Initialization algorithm.

```
input : a signature s, a table dispatch, a list ordered-methods
output : list of conflicting methods conflicting

Step 1: mark candidate methods
1. for s' in candidate-signatures(s)
2.     for m in dispatch[s']
3.         mark(m)

Step 2: compute conflict set
4. conflicting ← ∅
5. for m in reverse(ordered-methods)
6.     if m is marked then                    // m is a candidate
7.         if (∀m^c ∈ conflicting, m^c.higher-methods[m.rank] ≠ 1) then
8.             insert m into conflicting
```

Fig. 16. Most-Specific algorithm.

then $m$ is added to *conflicting*.

In systems that do not authorize method selection ambiguities at compile-time, *conflicting* may only contain 0 or 1 method; hence the loop at line 7 reduces to one test. As $\leq$ is an extension of $\preceq$, the most specific method is the marked method $m$ with the highest rank, and other marked methods have to be more generic. Once $m$ is found, the algorithm may return or finish the loop over *ordered-methods* to detect ambiguities.

In the case of Cecil's static checking mode, selection ambiguities may remain at compile-time (as long as those invocations cannot occur at run-time). However the conflicting methods of those invocations are not needed at run-time. We are then able to show that in this mode, line 7 can also be safely reduced to one test.

In the case of Dylan, table cells have to contain a list of most specific methods ordered by precedence. Besides, the method precedence test is more complex and depends on the invocation signature. Consequently, the Most Specific Algorithm would have to be changed, and the higher-methods arrays could not be used. It is likely that using the candidate signatures would again allow to optimize the Most Specific Algorithm. Further work is needed to formally establish this conjecture.

### 6.5 Worst-Case Complexity

We use the cost units introduced in Table I.

*Table Access.* The cost of an access in a dispatch table of dimension $n$ is propor-

tional to $n$:

$$TabAcc(n) = n \times TabAcc(1)$$

*Higher Methods Initialization Algorithm.* This algorithm essentially consists of an iteration over the MSA methods of candidate signatures. For a given method $m_k$, we denote $csg(m_k)$ as the number of candidate signatures for the formals of $m_k$, and we denote $|\mathcal{H}|$ as the size of the *higher-methods* array:

$$HMI(m_k) = csg(m_k) \times ( (\frac{|\mathcal{H}|}{\omega} + 1) \times BitOr + BitSet + VarAssg)$$

$|\mathcal{H}|$ is simply equal to the number of methods $|m|$. Let $K$ be the maximum number of direct supertypes of all types. The number of candidate signatures is less than $n \times K$. Hence the worst-case value of $HMI(m_k)$ is

$$HMI' = n \times K \times ( (\frac{|m|}{\omega} + 1) \times BitOr + BitSet + VarAssg).$$

*Most Specific Methods.* This routine essentially consists of two loops:

*Step 1.* There are at most $n \times K$ candidate signatures and at most $|m|$ conflicting methods for each. Hence this loop costs at worse $n \times K \times (|m| \times VarAssg + TabAcc(n))$.

*Step 2.* There are at most $|m|$ methods in the current list of ordered methods as well as in the current set of conflicting methods. Hence this loop costs at worse $|m| \times (IntComp + InsSet + |m| \times (BitTest + VarAssg))$.

The cost of this algorithm is

$$\begin{aligned} MSM \;=\; & n \times K \times (TabAcc(n) + |m| \times VarAssg) \\ & + |m| \times (IntComp + InsSet + |m| \times (BitTest + VarAssg)). \end{aligned}$$

When no selection ambiguity occurs, line 7 reduces to one test and $dispatch[s']$ on line 2 holds at most one method. Hence the worst-case complexity is

$$\begin{aligned} MSM' \;=\; & n \times K \times (VarAssg + TabAcc(n)) \\ & + |m| \times (IntComp + InsSet + BitTest + VarAssg). \end{aligned}$$

*Table Fill Up.* The algorithm in Figure 14 essentially consists of three iterations.

—*Iteration 1 (line 1):* builds the list of pole signatures, which costs

$$I_1 = |Poles_m| \times InsSet.$$

—*Iteration 2 (lines 2-3):* assigns the cells of the signatures of methods, which costs

$$I_2 = |m| \times (TabAcc(n) + VarAssg).$$

—*Iteration 3 (lines 4-6):* assigns the other cells.

—The detection of the signatures of methods costs

$$I_3^1 = TabAcc(n) + IntComp.$$

—The assignment in line 6 costs

$$I_3^2 = TabAcc(n) + VarAssg + MSM.$$

As $I_3^2$ does not apply to method signatures, and taking into account the initialization of the higher-methods bits of method signatures, in the worst case this iteration costs

$$\begin{aligned}
I_3 &= I_3^1 + (|Poles_m| - |m|) \times I_3^2 + |m| \times HMI' \\
&= |Poles_m| \times (TabAcc(n) + IntComp) \\
&\quad + (|Poles_m| - |m|) \times (TabAcc(n) + VarAssg + MSM) \\
&\quad + |m| \times n \times K \times (\ (\tfrac{|m|}{\omega} + 1) \times BitOr + BitSet + VarAssg) \\
&= |Poles_m| \times (TabAcc(n) + IntComp) + (|Poles_m| - |m|) \\
&\quad \times (TabAcc(n) + VarAssg + n \times K \times (TabAcc(n) + |m| \times VarAssg) \\
&\quad + |m| \times (IntComp + InsSet + |m| \times (BitTest + VarAssg))) \\
&\quad + |m| \times n \times K \times (\ (\tfrac{|m|}{\omega} + 1) \times BitOr + BitSet + VarAssg).
\end{aligned}$$

To sum up, table fill up costs

$$\begin{aligned}
TFU &= I_1 + I_2 + I_3 \\
&= |Poles_m| \times (InsSet + n \times TabAcc(1) + IntComp) \\
&\quad + |m| \times (n \times TabAcc(1) + VarAssg \\
&\quad + n \times K \times (\ (\tfrac{|m|}{\omega} + 1) \times BitOr + BitSet + VarAssg\ )) \\
&\quad + (|Poles_m| - |m|) \times (TabAcc(n) + VarAssg \\
&\quad + n \times K \times (TabAcc(n) + |m| \times VarAssg) \\
&\quad + |m| \times (IntComp + InsSet + |m| \times (BitTest + VarAssg))) \\
&= \mathcal{O}(TabAcc(1) \times n \times |Poles_m| + \tfrac{BitOr}{\omega} \times |m| \times n \times K \times |m| \\
&\qquad + (|Poles_m| - |m|) \times (n \times K \times (TabAcc(1) \times n + VarAssg \times |m|) \\
&\qquad + (BitTest + VarAssg) \times |m|^2) \\
&= \mathcal{O}(\tfrac{BitOr}{\omega} \times |m|^2 \times n \times K \\
&\qquad + (|Poles_m| - |m|) \times (TabAcc(1) \times n^2 \times K + VarAssg \times |m| \times n \times K \\
&\qquad + (BitTest + VarAssg) \times |m|^2).
\end{aligned}$$

As $|Poles_m| \leq |\Theta|^n$ and $|m| \leq |Poles_m|$, we obtain

$$\begin{aligned}
TFU < \mathcal{O}(&\tfrac{BitOr}{\omega} \times |m|^2 \times n \times K \\
&+ (TabAcc(1) \times n^2 \times K + VarAssg \times |m| \times n \times K \\
&+ (BitTest + VarAssg) \times |m|^2) \times |\Theta|^n.
\end{aligned}$$

In real cases, $\Theta$ and $|m|$ are the only factors likely to vary a lot; hence table fillup is performed in $|\Theta|^n \times |m|^2$, typical values for $n$ being 2 and 3. Taking into account the cost reduction brought by table compression, it is performed in $|Poles_m| \times |m|^2$. When no method ambiguity occurs (e.g. with Cecil's static mode) this complexity reduces to $|Poles_m| \times |m|$.

## 7. IMPLEMENTATION AND MEASUREMENTS

This section focuses on the run-time dispatching using compressed dispatch tables. We only describe the implementation in the case of a language that ensures the existence of a UMSA for any invocation. We first describe the representation of the run-time data structures. In particular, we optimize the size of argument arrays using bytes or 16-bit words. We also describe the dispatch code, since its size

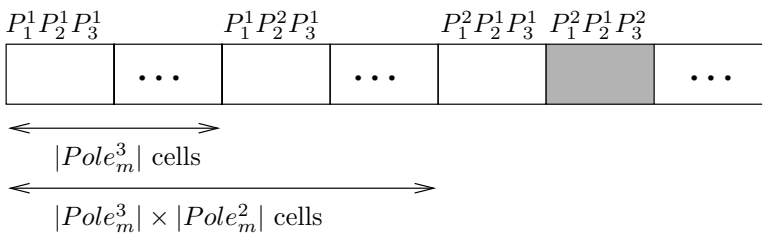



Fig. 17. Implementation of a dispatch table.

must also be counted. We then describe the global algorithm to build the dispatch structures with group argument arrays using bytes or 16-bit words. Finally, we present two applications with multimethods. In the first application consisting of the Cecil compiler, we measure the size of the dispatch table and the total size of dispatch structures using our proposed implementation. With the second application, consisting of the Dylan library, we only compute the poles and give the compression factors of the dispatch table. We then give upper and lower bounds for the size of the dispatch tables.

### 7.1 Run-Time Structures and Dispatch Code

The data structures used at run-time are slightly different from those used at compile time. In the following, we consider an $n$-ary generic function $m$. We denote as $\omega$ the size in bytes of a memory word (in practice, $\omega = 4$ or 8). Furthermore, since run-time dispatch only needs the address of the code to execute, we assume that at run-time each dispatch table's cell holds the address of a method's code.

We assume that these addresses are stored contiguously, starting from a base address $\beta$. We also assume that a $n$-dimensional table is stored as a nesting of one-dimensional arrays. The top-level array is associated with the first argument position, and the low-level arrays are associated with the last argument position. For each dimension, the associated argument array stores the offset of the cells with respect to the base of a one-dimensional array. The function $m\_arg_i$ (introduced in Section 3.3) associates each type with the corresponding offset in the $i$th dimension. This offset is the multiplication of (pole number − 1) by a constant $\gamma_m^i$, which depends on the number of poles for the next argument positions. Hence given an invocation signature $m(o_1, \ldots, o_n)$, the address of the code to execute appears at address $\beta + \sum_{i=1}^{n} m\_arg_i[o_i.type\_index]$. Depending on the value of the largest offset in an argument array, its content can be stored in 8-bit, 16-bit, or 32-bit memory words.[6] In practice, 32-bit words were never needed in our measures; hence we only assume 8- and 16-bit words in the following.

To illustrate this, consider a 3-dimensional dispatch table for a generic function $m$. Let $P_i^j$ be the $j$th pole in dimension $i$, and let $|Pole_m^i|$ be the number of poles in dimension $i$. Thus, we have $\gamma_m^1 = |Pole_m^2| \times |Pole_m^3| \times \omega$, $\gamma_m^2 = |Pole_m^3| \times \omega$, and $\gamma_m^3 = \omega$. Figure 17 portrays the implementation of the dispatch table. Consider three types $T_1$, $T_2$, and $T_3$, such that $pole_m^1(T_1) = P_1^2$, $pole_m^2(T_2) = P_2^1$,

[6]This does not matter on some processors that cannot deal easily with bytes or 16-bit words, such as the Dec Alpha (see Sites [1992])

and $pole^3_m(T_3) = P^2_3$. Hence, we will have $m\_arg_1(T_1) = |Pole^2_m| \times |Pole^3_m| \times \omega$, $m\_arg_2(T_2) = 0$, and $m\_arg_3(T_3) = \omega$. Thus, the code to execute for an invocation $m(T_1, T_2, T_3)$ appears at $\beta + |Pole^2_m| \times |Pole^3_m| \times \omega + \omega$, which corresponds to the gray cell of Figure 17.

We assume that the operations necessary to perform the dispatch are replicated for each generic function. This allows to have the base addresses of the argument arrays and table as constants in the code.

For each invocation, the application's object code includes the parameter-passing code and a call to the dispatch routine. This is equivalent to a static invocation and cannot be counted as method selection overhead. Then the dispatch routine finds the address of the code to execute, based on the relevant arguments, and jumps to this code. Finally this code performs the necessary stack adjustments. Again, these adjustments cannot be counted as method selection overhead.

In the case of a SPARC processor, the dynamic dispatch code for $n$ target arguments needs $5n + 3$ operations. We give in Figure 18 the dispatch code[7] for a generic function with two target arguments, without type checking. It assumes that both argument arrays are made of bytes and that the arguments of the invocation are referred to by registers `i0` and `i1`. Apart from giving the space overhead incurred by multiple dispatching, this code also shows that run-time dispatch is done efficiently and in constant time, the effective time depending on the version of the processor. Furthermore, the blocks of instructions that fetch the table offsets (lines 1–4 and 5–8) are clearly independant, and the offset sum computation (line 9) can be performed in pipe-line mode. For $n$ target arguments, this amounts to $3 \times n - 1$ additions and $2 \times n$ loads from memory that can be performed in parallel, in a sufficiently superscalar processor ($n$ being typically 2 or 3). The common section of code (lines 10–13) involves one addition, one load,and one indirect jump. $A$ being the cost of an addition, $L$ the load latency, and $B$ the branch misprediction penalty of the target processor (we assume it does not predict indirect branches), the total cost of dispatch, assuming use of a superscalar processor, is $4 \times A + 2 \times L + B$. Compared to the time overhead for monomethods, the time overhead of $n$ targets is $2 \times A + L$.

### 7.2 Implementation of Group Argument-Arrays

We use the same coloring algorithm as Dixon et al. [1989] to construct group argument arrays. Each selector is associated with the set of types to which it is applicable, i.e., to which one of its formal arguments is applicable. The conflict set of a selector is the set of the other selectors which set of applicable types intersect with its own set. We order selectors according to the size of their conflict sets in decreasing order, and we allocate the colors by considering the selectors in this order. For a given selector, we first search in the list of colors ordered by creation time from the earliest, a color for which selectors do not conflict with the selector being considered. If no such color exists, a new color is created and associated with the selector. Following André and Royer [1992], we only consider, in the sets of applicable types, the types which do not have any subtypes.

[7]This code was obtained from the compilation of C code, the order of the assembler operations being changed for clarity purposes.

```
                ! find in 1st argument array the offset associated with 1st target argument
1.      ld      [%i0+4],%o0         ! o0=type number of 1st argument
2.      sethi   %hi(_aa1),%o2
3.      or      %o2,%lo(_aa1),%o2   ! o2=base address of arg-array 1
4.      lduh    [%o0+%o2],%o0       ! o0=offset in dispatch table
                ! find in 2nd argument array the offset associated with 2nd target argument
5.      ld      [%i1+4],%g1         ! g1=type number of 2nd argument
6.      sethi   %hi(_aa2),%l1
7.      or      %l1,%lo(_aa2),%l1   ! l1=base address of arg-array 2
8.      ldub    [%g1+%l1],%g1       ! g1=ωγ²π₂ =offset in dispatch table
                ! add offsets
9.      add     %o0,%g1,%g1         ! g1=global offset in table (n − 1 additions)
                ! find code address
10.     sethi   %hi(_meth),%o4
11.     or      %o4,%lo(_meth),%o4  ! o4=table base address
12.     ld      [%o4+%g1],%g1       ! g1=address of code to execute
13.     jmpl    %g1,%g0             ! jump to the code
```

Fig. 18. Sparc implementation of the dispatch routine using dispatch tables.

As some argument arrays use short (8-bit) words, and others long (16-bit) words, the coloring algorithm distinguishes the corresponding selectors. Thus, we do not group two argument arrays whose elements do not have the same size.

It often happens that no overriding occurs on some arguments of a generic function $m$: for some position $i$, the set of $i$th formal arguments types is a singleton. In this case, the MSA method selection does not depend on the run-time type at this position; hence there is no corresponding dimension in the dispatch table. These selectors are called *inactive selectors*, the others being called *active selectors*. The associated formal argument is the single element of $Pole_m^i$, called a *single pole*.

As a consequence, the argument arrays of inactive selectors may only be useful for type-checking, to efficiently differentiate the subtypes of the single pole from the other types. The coloring of these arrays must only yield the selector arrays. Also, note that if two selectors have the same type as single pole, they can share the same selector array. We do not detail the construction of these selector arrays of inactive selectors, as they are not needed for run-time dispatch. Hence in the following, "selector" means "active selector", unless specified otherwise.

Consequently, our dispatch table computation algorithm with coloring is as follows:

(1) *Compute and Store All Poles:* For each generic function $m$, for each argument position $i$ of $m$, compute $Pole_m^i$ and store it, as well as the value of function $pole_m^i$, in a temporary array.

(2) *Create Active Selectors:* This comes down to find the selectors with at least two poles.

(3) *Prepare Argument-Array Fillup:* For each selector $(m, i)$, compute the corresponding $\omega \times \gamma_m^i$ and the corresponding breadth (8 bits or 16 bits). It is 16 bits if $\omega \times \gamma_m^i \times (|Pole_m^i| - 1) > 255$, because $(|Pole_m^i| - 1)$ is the highest pole number.

(4) *Build Conflict Graph for Coloring:* For each selector, find the applicable types which have no subtypes, by propagating the selector down the subtyping graph using a recursive function. The conflict set of each type without subtypes $T$ is composed of the selectors applicable to $T$. For each such conflict set, add each selector to the conflict set of the other selectors.

(5) Order the selectors by size of the conflict set, from the biggest one.

(6) *Compute the Colors:* A color is a set of selectors, and the color list is empty at the beginning. For each selector $(m, i)$ of the list, look for a compatible color in the list of colors. A compatible color is composed of selectors which do not conflict with $(m, i)$ and which have the same breadth as $(m, i)$. If no such color exists, create a new color and add it to the list; then add $(m, i)$ to this color. As an optimization, two color lists can be built, respectively for 8-bit selectors and 16-bit selectors.

(7) Create the colored argument arrays, selector array, and associate each color with its colored argument array and selector array.

(8) *Identify Selectors:* For each color, sequentially number the selectors of the color.

(9) *Fill Up the Argument Arrays:* For each selector $(m, i)$, using the temporary array of the function $pole^i_m$, and for each type $T$ such that $pole^i_m(T)$ is not null, $\pi_i$ being the pole number of $pole^i_m(T)$, store $\omega \times \gamma^i_m \times \pi_i$ at $T$'s place in the colored argument array of $(m, i)$ and store $(m, i)$'s number in the associated selector array.

(10) Fill up the dispatch table, using the algorithm of Figure 14.

As coloring differentiates active selectors with 8-bit argument arrays from those with 16-bit argument arrays, the width of argument arrays must be computed before coloring. This involves computing the number of poles in all dimensions. As coloring takes into account all selectors of all generic functions, the pole computation has to be done for all of these selectors. The argument arrays are filled up before the dispatch table because, for each cell of the latter, we need to use the values of the preceding cells, based on Theorem 6.1.5. The argument arrays are needed to quickly access these cells.

### 7.3 The Cecil Hierarchy

Our results have been conducted on the Cecil compiler[8] [Chambers 1993], which is a real object-oriented application with multimethods, as the compiler is written in Cecil itself. This application includes 932 types and 3,990 generic functions, composed of 7,873 methods. Appendix A is a gzip'd tar archive of files that give the declarations of the types, subtyping links and method signatures of this application, in the format that we have used to conduct our tests. Table II gives the number of generic functions and methods respectively with 0, 1, 2, 3, and 4 selectors.

Most of these functions have 0 or 1 active selector. Indeed, despite many generic functions have many arguments, it often occurs that their type is the same for all methods of the generic function. We say that an argument of a generic function is a target only if its selector is active. Note that in the Cecil terminology, the

[8]The lists of classes and methods of the Cecil compiler were graciously put at our disposal by Craig Chambers and Jeff Dean from University of Washington.

Table II. Distribution of Generic Functions and Methods in the Cecil Compiler

| Number of selectors | Number of generic functions | Number of methods |
|---|---|---|
| 0 | 2,761 | 2,761 |
| 1 | 1,147 | 4,465 |
| 2 | 75 | 586 |
| 3 | 6 | 59 |
| 4 | 1 | 2 |

Table III. Sizes of Dispatch Tables, Cecil Compiler

| Number of Selectors | Size of Uncompressed Tables | Size of Compressed Tables |
|---|---|---|
| 2 | 248.5MB | 7.5KB |
| 3 | 18.1GB | 3KB |
| 4 | 11TB | 64B |

Table IV. Number of Argument Arrays

| | Uncolored Argument Arrays | | Colored Argument-Arrays | |
|---|---|---|---|---|
| | on 8 bits | on 16 bits | on 8 bits | on 16 bits |
| Multitargetted Gen. Fun. | 159 | 18 | 66 | 7 |
| Monotargetted Gen. Fun. | 1147 | | 99 | |

$i$th argument of a generic function $m$ is a *dispatching* argument if and only if $Static_m^i$ includes at least one type which is not *any*, the common superclass of all classes. Hence, the set of target arguments is a subset of the set of dispatching arguments. In our view, dispatching arguments which are not target arguments are only considered for type-checking. No type checking is necessary for nondispatching arguments, because all types are subtypes of *any*.

### 7.4 Measurements with Cecil

We first give the results that show (1) the effectiveness of compression in terms of memory space and (2) the computation time for the algorithms that construct dispatch tables.

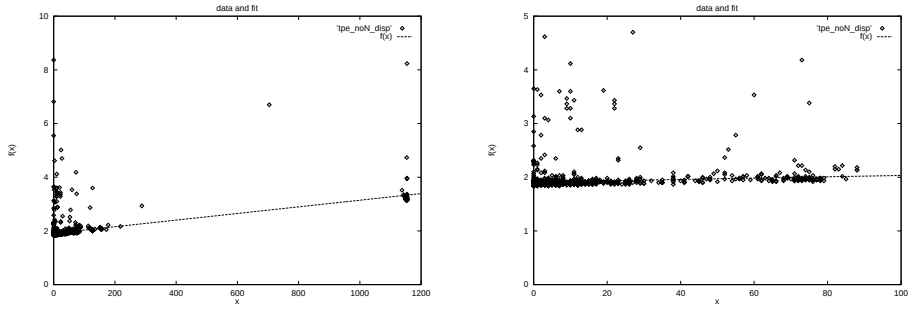
7.4.1 *Size of Dispatch Tables.* Table III gives the total memory size in bytes of uncompressed and compressed dispatch tables, for the generic functions with 2, 3, and 4 target arguments. We assume a system with memory addresses using 32-bit words. These totals take into account all generic functions, and as there are fewer generic functions with three target arguments instead of two, the total size of the corresponding compressed tables is also smaller.

7.4.2 *Effectiveness of Coloring.* We first consider generic functions with at least two target arguments, then generic functions with one target argument. Regarding the former, we distinguish the argument arrays, depending on their width, i.e., 8-bit or 16-bit word. We also consider monotargetted generic functions in order to give the total size of dispatch structures for the whole application.

7.4.3 *Total Size of Dispatch Structures.* The total size of the structures needed for run-time dispatch tables includes the compressed tables (Table III), the colored argument arrays (Table IV), the selector arrays (in case of dynamic type-checking) and the dynamic dispatch code (shown in Figure 18). Assuming a 32-bit system,

Table V. Size of Dispatch Structures (in bytes)

| | Static Type-Checking | Dynamic Type-Checking |
|---|---|---|
| Multitargetted Generic Functions | 89,788 | 161,264 |
| Monotargetted Generic Functions | 371,052 | 507,220 |
| Total | 460,840 | 668,484 |




Full data set Ignoring "$Object \in Dynamic$" cases

Fig. 19. Pole computation time.

we summarize the results in Table V in the case of multitargetted generic functions, monotargetted ones, and all generic functions. For monotargetted generic functions, we assume monodimensional dispatch tables. The dispatch code is replicated for each generic function if type-checking is used, or for each colored dispatch table otherwise.

7.4.4 *Pole Computation Time.* We measured the computation time for poles. In practice, the most costly operation is the Pseudo-Closest-Poles algorithm (Figure 9). It iterates twice over the poles of the direct supertypes of each type to build the *candidates* list and to find the one with the greatest rank. For a type $T$, the size of this candidate list is the number of direct supertypes of $T$ which have a pole, i.e., which are in $Dynamic^i_m$. Therefore, we show in Figure 19 the graph of the pole computation time in function of $\sigma = \sum_{T \in \Theta} |\{T' \in Dynamic^i_m | T \; isa \; T'\}|$. As above, pole computation is performed on active selectors. We compute the line $t = f(x)$, which is the closest to the graph, using the Levenberg-Marquardt method.

The following observations can be made. As *Object* is the supertype of all types in the Cecil hierarchy, when $Object \in Static^i_m$, we have $\sigma = \mathcal{E} = 1155$, i.e., the rightmost points on the graph. On the opposite, when the primary poles are on the bottom of the type hierarchy, most types do not have any supertype pole; hence $\sigma$ is close to 0. This case occurs quite often, which explains why there are many more points on the left of the graph. This also explains why there are more abnormal values on the left, which represent abnormally long pole computations. This result shows that our pole computation algorithm performs better when primary poles have less subtypes, because types then have less supertype poles.

Table VI. Distribution of Generic Functions and Methods in the Dylan Library

| Number of Selectors | Number of Generic Functions | Number of Methods |
|---|---|---|
| 0 | 1559 | 1559 |
| 1 | 271 | 1012 |
| 2 | 61 | 461 |
| 3 | 9 | 129 |
| 4 | 2 | 40 |

Table VII. Sizes of Dispatch Tables, Dylan library

| Number of Selectors | Size of Uncompressed Tables | Minimum Size of Compressed Tables | Maximum Size of Compressed Tables |
|---|---|---|---|
| 2 | 199.1MB | 7.8KB | 28.4KB |
| 3 | 26.5GB | 60.4KB | 246.2KB |
| 4 | 5.3TB | 60.3KB | 184.7KB |

### 7.5 The Dylan Library

Our second results have been conducted on the Dylan library, which is included in the Apple "Technical Release" distribution.[9] A particular aspect of Dylan is to allow us the use of "singleton types" as targets. These are simply instances, for example character strings, which allows the use of the run-time value of character string arguments for method dispatch. We simply considered each of these strings as a direct subtype of "string," and similarly for other singleton types. The resulting library includes 925 types, and 1,902 generic functions composed of 3,201 methods.

Table VI gives the number of generic functions and methods respectively with 0, 1, 2, 3, and 4 target arguments.

### 7.6 Measurements with Dylan

In the case of Dylan, we only compute the size of the dispatch tables in terms of the number of entries in each dimension, i.e., the number of poles in each dimension. The reason is that, as said earlier, our current implementation of the construction of dispatch tables assumes that there exists an MSA for any invocation. However, this is not guaranteed by Dylan for which $MS$ returns a list of most specific methods ordered by precedence. Furthermore, the test of method precedence used in our implementation is different from the one of Dylan. Nevertheless, we give useful boundaries on the size of the dispatch table. A lower bound is obtained by considering the case where only the most specific applicable method is stored in each cell of the dispatch table. This would correspond to a case where no method uses the function *next-method*. The upper bound is obtained by considering that each cell of the dispatch table stores all the corresponding applicable methods. This would correspond to a case where the applicable methods for each pole signature are totally ordered. For the latter, we assume an implementation where each nonempty cell holds a reference to a null-terminated array of method references.[10]

[9]Many thanks go to Glenn S. Burke, Bill St. Clair, and Dave Moon, who gave very helpful informations on how to list the Dylan types and methods.

[10]This structure could probably be optimized by detecting the cells $c_1$ whose array's end is identical to the array of another cell $c_2$, so that $c_2$ could refer to the end of $c_1$'s array.

Table VII gives the total memory size in bytes of uncompressed and compressed dispatch tables, for the generic functions with two, three, and four target arguments. The two columns on the right indicate the upper and lower bounds. As above, we assume a system with memory adresses using 32-bit words.

### 7.7 Summary

We have described an optimized implementation of our multimethod dispatch scheme. Ignoring arguments that do not take part in the dispatch process saves both space and time. Storing the offsets of the cells in the argument arrays, instead of pole numbers, ensures true constant-time dispatch. Replicating the dispatch code for each generic function saves some instructions at run-time. Using the minimal memory word width for argument arrays, and grouping the argument arrays that have the same color, optimizes the memory space needed by argument arrays with no extra cost at run-time. This is of particular importance, as argument arrays are significantly larger than any of the other structures. In the case of multitargetted generic functions without type-checking and the Cecil classes and methods, they represent 83% of the total 89,788 bytes. The structures used for type-checking are another kind of argument arrays. Argument arrays could be further optimized in the following ways:

—ordering optimally the argument positions of each generic function to increase the number of 8-bit-wide argument arrays.

—replacing coloring with a better compression scheme for bidimensional tables. An example is the "minimized row displacement" scheme of Driesen and Hölzle [1995].

—grouping identical argument arrays. If two selectors have the same formal arguments, the associated *pole* functions are then identical; hence their argument arrays can be made identical, by defining the corresponding argument positions as the last of their respective generic functions. Indeed, the associated dimensions in the tables are then represented by low-level arrays, and the offsets (stored in the arguments-arrays) coincide with pole numbers.

On the opposite, the $n$-dimensional compressed tables only take 10,804 bytes with Cecil, a compression ratio of more than 99.99% with respect to Table III, considering only generic functions with two targets. Altogether, the compressed structures take a very small fraction of the memory space required by the uncompressed structures.

## 8. RELATED WORK

### 8.1 Monomethod Dispatch and Generalizations

Several techniques have been proposed to optimize the dynamic dispatch of monomethods. Their presentation is relevant here because first they can be related to some of the techniques we used in our algorithms, and second they have been generalized to multimethod dispatch. We classify these techniques, depending on whether they guarantee that method selection is done in constant or nonconstant time. The latter techniques are mainly used in dynamically typed pure object-oriented languages. They try to optimize the average method dispatch time. Constant-time techniques are mostly used in statically typed nonpure object-oriented languages.

8.1.1 *Variable Time Techniques.* The technique known as *caching* consists in memorizing the MSA methods found by previous searches into a *cache*. The cache is then searched first, and if a method is not found then the MSA is retrieved using the schema search approach. Several caching schemes exist, and systems typically use a combination of them: a single global cache in Smalltalk and Sather [Deutsch 1983; Schmidt and Omohundro 1991; Ungar and Patterson 1983], local caches (local to a generic function or an invocation) in CLOS, PCL, and Sather [Kiczales and Rodriguez 1990; Schmidt and Omohundro 1991], inline caches in Smalltalk and Self [Chambers et al. 1989; Deutsch and Schifman 1984; Hölzle et al. 1991; Ungar 1986; Ungar and Patterson 1987]. Kiczales and Rodriguez [1990] propose to extend its cache per generic function scheme to multimethod dispatch. Hashing is applied to the types of all the arguments, instead of a single argument, to access a cache entry.

The second technique, known as *inlining*, performs type analysis and run-time type tests to avoid method dispatch and *inline* the code of the MSA method. Several sophisticated techniques exist to achieve inlining: type prediction [Deutsch and Schifman 1984; Goldberg and Robson 1983; Ungar and Patterson 1983] and type casing [Johnson et al. 1988] in Smalltalk, customization, splitting [Chambers et al. 1989; Chambers and Ungar 1991; Ungar et al. 1992] , type feedback, and adaptive optimization [Hölzle and Ungar 1996] in Self.

Recently, Ferragina and Muthukrishnan [1996] reduce method dispatch to the inclusion test of integer intervals. In this way, the worst-case complexity of method dispatch is $O(log(log|\Theta|))$. The data structures needed for run-time dispatch are small and can be updated quickly when a new type or a new method is added. This technique currently only applies to single dispatch and single inheritance.

8.1.2 *Constant-Time Techniques: Dispatch Tables.* In the case of monotargetted generic functions, dispatch tables are bidimensional and can be organized in three ways. The first one is a unique global two-dimensional array with the types and the generic functions as indices. The second organization associates to every generic function a one-dimensional array indexed by the types with an entry for every type. The third organization associates to every type a one-dimensional array indexed by the generic functions. The last two organizations are just two different ways of slicing the global table. Independently of the organization, the number of entries in the dispatch table is $|\Theta| \times |F|$; hence in practice the tables need to be compressed. For example in a Smalltalk-80 system, $|\Theta| \times |F|$ amounts to about 3.5 millions entries.

In the case of single-inheritance C++, each type owns a single table (the *_vtbl*). Finding the MSA method requires that we get the base address of the dispatch table stored in the target argument and perform one array access using the index of the generic function. This amounts to two indirections and one addition. This scheme eliminates all empty cells and was adapted when multiple inheritance was added to C++.[11] If a class $C$ inherits classes $A$ and $B$, $C$'s instances are composed of two parts, which correspond to the properties respectively inherited from $A$ and $B$. Because of polymorphism, each of these parts begins with references to distinct dispatch tables, which are used when the instances of $C$ are considered as $A$'s or

[11] The cost of dynamic dispatch may then rise to three indirections and two additions.

as $B$'s instances. This scheme does not seem to generalize well to multimethods. Indeed, the method tables would then be associated with $n$-tuples of classes, and in case of multiple inheritance each $n$-tuple of objects should be split in several parts, which is impossible since each objet should have several cut-outs.

Coloring was originally used to compress the bidimensional dispatch tables of monomethods. Lines are labeled with generic functions, and columns with types. As for the merging of argument arrays described in Figure 5, coloring consists of merging lines in which nonempty cells are not associated with the same type. As an extension, Huang and Chen [1992] propose to merge both lines and columns of the dispatch tables. *Row displacement* [Driesen 1993; Driesen and Hölzle 1995] also proceeds by merging lines and columns. Before being merged, these lines and columns are shifted by a variable number of cells. Driesen and Hölzle [1995] show that row displacement is more efficient on lines than on columns. Experimentally, the best compression factor obtained is a factor of 66. These techniques might be used to compress multimethod dispatch tables, but they only aim at eliminating empty cells. Note however that row displacement could be used to compress our argument arrays, possibly yielding a better compression rate.

Besides, Vitek and Horspool [1994] also proposes to group similar lines of monomethod dispatch tables. When few corresponding cells differ between two lines, the resulting cell in the merged line contains the address of an intermediary routine that chooses the adequate method at run-time. This technique however does not offer constant-time selection. More recently, Vitek and Horspool [1996] proposed instead to slice tables in partitions, each type belonging to one single partition. Then grouping occurs inside a partition, between strictly identical lines, which offers constant-time dispatch. Measurements (only done in the context of single inheritance) show similar compression rates for these two techniques, which significantly outperform coloring and row dispacement. Here again, the technique of Vitek and Horspool [1996] could be profitably used to enhance the compression of our argument arrays.

Finally, Queinnec [1995] proposes to use a different structure than tables to store the precalculated MSA method of each class. It consists of decision trees, composed of three kinds of nodes, traversed by dynamic dispatch using the class of the target at run-time. "`cst`" nodes hold one method reference; they can be nested in "`if`" nodes to group the classes having the same MSA method as a common superclass. Finally, "`xif`" nodes hold arrays that relate each subclass of a class to its MSA method. These trees hence have a limited number of empty cells and enable us to group some cells with the same contents. Each node of the decision tree is used in constant time at run-time, but if the decision tree is not balanced, dynamic selection is not performed in constant time. Queinnec [1995] recognizes that many decision trees can be associated with a given generic function. However, it seems that finding the smallest tree requires us to build all possible trees and compare them. Finally, this proposal does not take into account multiple inheritance and multiple targetting.

The general problem of compressing sparse tables was already studied in the context of compiler construction for parser tables. Dencker et al. [1984] review six compression techniques, notably coloring and row displacement, and their optimizations. The "line elimination scheme," originally proposed by Bell [1974], proceeds

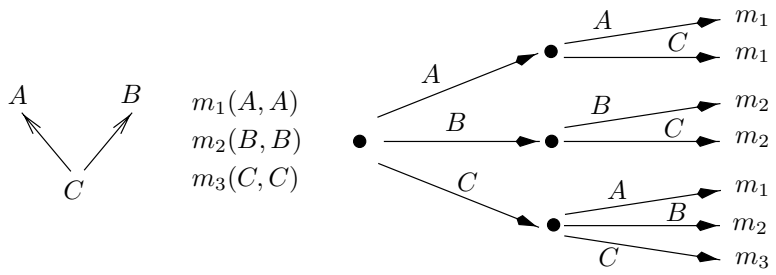



Fig. 20. Type graph and generic function with dispatch tree.

by extracting rows, then columns, in which the values of nonempty cells are identical, and by storing these values in two arrays. The system iterates again over rows, then columns, until a fixpoint is reached. As in the naive approach of Section 3.5, the uncompressed table has to be computed first, and its compression uses many comparisons. Moreover, a table of bits of the dimension of the original table has to be built to memorize which of the original cells are empty. Other techniques reviewed by Dencker et al. [1984] are the Significant Distance Scheme, which suppresses heading and trailing empty cells (as proposed by Driesen [1993]), and two others that require some list scanning to retrieve a cell, which precludes constant-time lookup.

## 8.2 Dispatching Using Lookup Automata

Chen and Turau [1995] present algorithms to dispatch multimethods using some sort of "dispatch trees,"[12] which as our dispatch tables, hold the precalculated MSA method for each possible invocation of a generic function. Their multiple-dispatching scheme offers a quasiconstant time. Furthermore, dispatch trees are compressed using a notion very similar to our poles.

The formalism presented by Chen and Turau [1995] is very different from ours, and its exposure is quite long. Thus, we first present our own description of the principles of their algorithms and then draw a comparison between their algorithms and ours.

8.2.1 *General Principle.* The *dispatch tree* of a $n$-ary generic function $m$ is a directed, balanced tree of depth $n$. Each invocation signature is associated with a unique path in this tree. Each path starts from the root, and each type $T_i$ of the signature determines the choice of one of the possible branches of the tree to expand the path of length $i-1$ into a path of length $i$. The leaves of the tree are labeled with the MSA method of the invocation.

*Example* 8.2.1.1. Consider in Figure 20 the graph of types $A$, $B$, and $C$, the generic function $m$, and the dispatch tree associated with $m$. The arcs are labeled with types. Each signature for which there is an MSA method is associated with a path in the tree. This path leads to a leaf labeled with the corresponding MSA method. An invocation with signature $(A, C)$ corresponds to a path starting from the root that follows the arcs labeled by $A$ and $C$, yielding an MSA method $m_1$.

[12]Dispatch trees is our own terminology to describe what the authors call a lookup automaton.

An invocation with signature $(A, B)$ has no associated path, since $m(A, B)$ has no MSA method.

There is a single path $T_1, T_2, \ldots, T_{i-1}$ that starts from the root and reaches a node $N_i$ of depth $i$. Furthermore, from node $N_i$, there is exactly one output arc for each type $T_i \in \Theta$ such that there exists an invocation $m(T_1, \ldots, T_i, \ldots)$ for which there is an MSA method.

In fact, since an invocation $m(T_1, \ldots, T_n)$ has the same MSA method as $(T'_1, \ldots, T'_n)$, where $\forall i,\ T'_i \in Pole^i_m$, arcs of the tree can be associated with poles only. Moreover, suppose that a node $N$ of depth $i$ is reached by a path $T'_1, \ldots, T'_{i-1}$ where each $T'_i$ is an $i$-pole; the set of methods that are applicable to an invocation where the $i-1$ first types are $(T'_1, \ldots, T'_{i-1})$ forms a subset $\mathcal{M}'$ of the set of methods associated with $m$. The subtree with root $N$ only needs to explore this set $\mathcal{M}'$ of applicable methods. Thus, the number of output arcs for $N$ is determined by the number of $i$-poles that occur at the $i$th position in the methods of $\mathcal{M}'$. We call these $i$-poles the *local poles* of $N$, and their set is noted $Pole^N$. Dujardin [1996] gives a formal definition of these notions.

*Example* 8.2.1.2. In the tree of Figure 20, all types on the labels of arcs are poles. Consider the two nodes $N_1$ and $N_2$ reached by signatures respectively starting with $A$ and $B$ at the first position. The set of possible applicable methods to these signatures is $\{m_1, m_2\}$. The set of 2-poles is $\{A, B, C\}$. However, $Pole^{N_1} = \{A\}$ and $Pole^{N_2} = \{B\}$. Thus, the output arcs of $N_1$ and $N_2$ labeled by $C$ can be discarded from the tree.

A further compression can be obtained by *node unification*. Two nodes $N_1$ and $N_2$ of a given depth can be unified into one node $N$, if it is possible to superpose the two subtrees with respective roots $N_1$ and $N_2$. This superposition must take into account the types labeling the arcs and the MSA methods that label the leave nodes. Then the arcs that lead to $N_1$ and $N_2$ lead to $N$, and the dispatch tree becomes a direct, acyclic graph. This unification spares the space taken by one node and all its subtree.

*Example* 8.2.1.3. Considering the dispatch tree of Figure 21, two unifications can be done. The first one unifies the nodes $N_1$ and $N_2$, and the second unifies $N'_1$, $N'_2$, and $N'_3$. This spares six nodes.

8.2.2 *Comparison.* The approach of Chen and Turau [1995] supports languages whose precedence ordering is Inheritance Order Precedence, as defined by Argrawal et al. [1991]. This order is a particular case of Argument Subtype Precedence with monotonicity, considered throughout this article. No implemented language currently supports the former order,[13] while both Cecil and Dylan support the latter.

The proposal of Chen and Turau [1995] includes a pole computation algorithm (called there a closure algorithm). This algorithm does not compute the influences. It operates in two steps. The first step duplicates the type graph. The second step traverses the type graph from the most generic to the most specific types. In this

[13]Inheritance Order Precedence was presented by Agrawal et al. [1991] to model the precedence algorithm of CLOS; however, the former is monotonic by definition, while the latter is not, as shown by Ducournau et al. [1992].

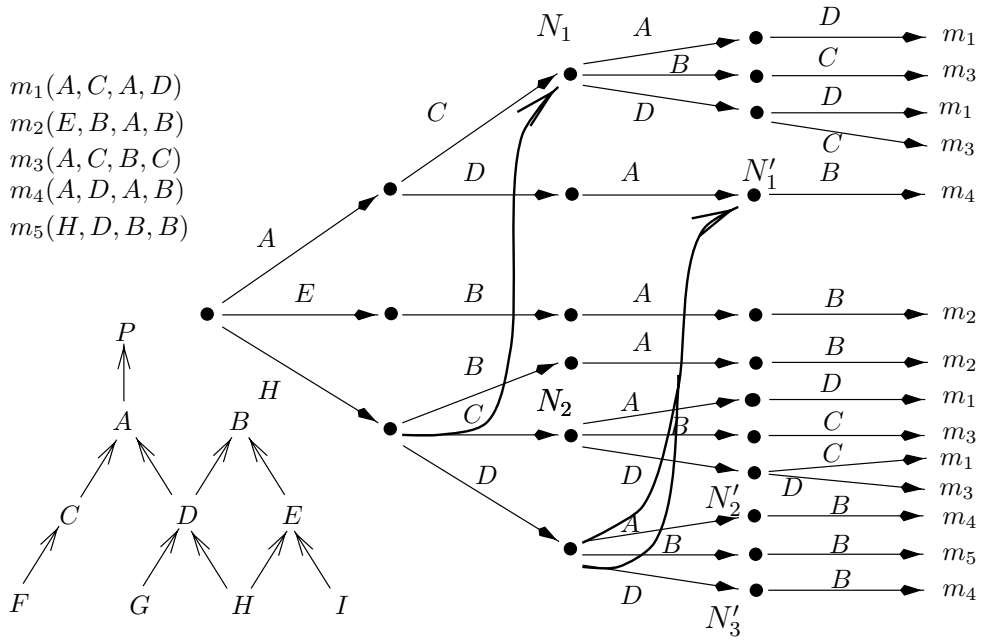



Fig. 21. Dispatch tree with four arguments.

traversal, each type which is found not to be a pole is removed from the graph, and its direct subtypes become direct subtypes of its pole. This way, the poles of the supertypes appear as direct supertypes. In contrast, our proposal associates a pole with each type, which avoids to update the type graph.

For each type, their traversal computes the most specific supertype poles. The technique used for subtyping comparisons is that of Agrawal et al. [1989]. This technique associates with each pole a sequential number and the list of its supertypes's numbers. Testing if $T_1$ is a subtype of $T_2$ then reduces to looking for $T_2$'s number in $T_1$'s list. The experimental complexity of this technique is described by Agrawal et al. [1989] as "essentially constant". Chen and Turau [1995] base the complexity of their pole computation algorithm on this experimental complexity. However, the worst-case complexity of this technique is $O(log|\Theta|)$. Indeed, each test implies a loop through a list, which can be long if $T_1$ has many supertypes. Our proposal is different, due to the representation of the collection of supertypes, which allows us to perform the test with a direct access in a small and constant number of basic operations.

To summarize, the complexity of their algorithm is $\mathcal{O}(|\Theta| + \mathcal{E})$. However, this does take into account neither the worst-case complexity of the subtype comparison nor the construction of the list of supertypes's numbers.

The construction of the dispatch tree requires more pole computations than the dispatch table. Indeed, one pole computation must be done for each node. The number of nodes is bounded by $\sum_{i=1}^{n} |\Theta|^i$. In contrast, the number of pole computations in a dispatch table is $n$. Moreover, node unification also takes supplementary time in dispatch trees. On the other hand, filling up the dispatch tables requires more MSA computations than with dispatch trees. Finally, as we do not assume the

same precedence ordering to compute the MSA methods, it is difficult to compare the effective construction time.

The nodes of the dispatch trees are implemented as two arrays, which respectively hold the local poles and the succeeding nodes of each local pole. Run-time dispatch traverses the dispatch tree using the types of the successive arguments of the invocation. Each type $T$ is used against one node $N$ to obtain the succeeding node in two steps. The first step looks for the pole of $T$ in the array of local poles of $N$. If this pole is found at position $k$, the second step retrieves the succeeding node at position $k$ in the array of succeeding nodes. The first step is done in time linear to $|Pole^v|$, by finding the first pole supertype of $T$. To keep this time bounded by a constant, nodes have a different structure when the number of local poles is greater than an arbitrary constant $c$. This structure is then a single array that associates with each type of the schema its successor node, which is found in a constant time.

The time needed for dynamic dispatch hence obviously depends on $c$. If $|Pole^v| \leq c$, the time complexity of finding the successor vertex is $O(|Pole^v|)$ (assuming that the subtyping tests are done in constant time). If $|Pole^v| > c$, this time is a constant. $|Pole^v|$ is bounded by $|\Theta|$; hence the worst-case time complexity of dynamic dispatch is $O(n \times |\Theta|)$. In contrast, the time for dispatch tables is $n \times \gamma$, $\gamma$ being a constant.

The memory space taken by dispatch trees also depends on $c$. If $c$ is high, dispatch trees benefit from the progressive decrease of the number of local poles and the absence of argument arrays. However, dynamic dispatch is guaranteed to be done in constant time only if $c$ is 1. Then many nodes have a size of $\Theta$ words. This is equivalent to having many argument arrays for each generic function, and in this case the dispatch trees take a lot more space than dispatch tables. Indeed, the measurements in Section 7 show the importance of argument arrays.

To sum up, the dispatch time with this proposal is linear but not constant, as long as its memory cost is competitive. Furthermore, the run-time dispatch with dispatch trees cannot benefit from superscalar architectures, because each step is based on the result of the previous step.

In Dujardin [1996], we present dispatch trees that offer constant-time dispatch. In that study, trees offer better compression rates than tables only for $n \geq 3$. Tridimensional compressed tables however do not seem to take a significant space, as shown in Section 7. Thus, dispatch trees should be considered only for $n \geq 4$, and with enough methods. We also observed that tree construction was significantly slower than table construction, due to the number of pole computations.

## 9. CONCLUSION

We proposed a simple multimethod dispatch scheme based on compressed dispatch tables. The salient features of our proposal are the following. First, it guarantees a dynamic dispatching in constant time, an important property for some object-oriented applications. Second, unlike the proposals such as Chen and Turau [1995], it is applicable to most existing object-oriented languages, both statically and dynamically typed, since it only assumes that method selection does not contradict argument subtype precedence, a most common property of method orderings. Last, our scheme is simple to implement and quite effective: in most cases, it yields a minimal dispatch table [Amiel et al. 1994].

We provided efficient algorithms to obtain the structure of a dispatch table in a linear time in the number of types and to fill it up with MSA methods's addresses. Our measurements show that the compression of dispatch tables is very effective, resulting in a very small fraction of the space required by the uncompressed structures. Further compression can be obtained at the expense of implementation complexity and increased compile time, using dispatch trees [Dujardin 1996] (for four targets or more) or better schemes of argument arrays compression, such as row displacement [Driesen and Hölzle 1995] or partitionning [Vitek and Horspool 1996].

Several issues are left for future work. First, it would be useful to quantify the effectiveness of our compression scheme on other real applications that use multimethods. Second, in Amiel et al. [1994], we presented another possible optimization, which consists of sharing compressed dispatch tables between generic functions. A study of multimethods's definition patterns in real applications would allow the development of heuristics to guide the use of this optimization. Third, applying our scheme to interactive programming environments requires that we develop incremental versions of our algorithms. If new methods are added, new primary poles may appear, and the poles must be recomputed. To do that efficiently, we would need to keep and maintain extra information such as the bit vectors associated with the poles, so that poles can only be recomputed on a subpart of the graph of types. Similarly, the maintenance of additional information would be needed to compute the new dispatch table from the existing one. These incremental algorithms should also take care of method disambiguation, as presented in Amiel and Dujardin [1996].

## ONLINE-ONLY APPENDIX

## A. SOURCE DATA FOR CECIL HIERARCHY EXPERIMENTS (SECTION 7.3)

Appendix A is available only online. You should be able to get the online-only appendix from the citation page for this article:

http://www.acm.org/pubs/citations/journals/toplas/1998-20-1/p116-dujardin/

Alternative instructions on how to obtain online-only appendices are given on the back inside cover of current issues of ACM TOPLAS or on the ACM TOPLAS web page:

http://www.acm.org/toplas

### ACKNOWLEDGMENTS

The initial ideas about compressed multimethod dispatch tables emerged during fruitful conversations with Olivier Gruber when he was at INRIA. Our warmest thanks thus naturally go to him. We would also like to thank Pascal André, Marie-Jo Bellosta, Scott Danforth, Tim Griffin, and Michel Habib for their insightful comments on earlier versions of this article, and Roland Ducournau for pointing out that monotonicity needed to be formally extended to method precedence. Finally, we thank all the referees for their extremely careful reviews and helpful comments.

### REFERENCES

Agrawal, R., Borgida, A., and Jagadish, H. V. 1989. Efficient management of transitive relationships in large data and knowledge bases. In *ACM SIGMOD Conference Proceedings*. ACM, New York.

Agrawal, R., DeMichiel, L. G., and Lindsay, B. G. 1991. Static type checking of multi-methods. In *OOPSLA Conference Proceedings*. ACM, New York.

Amiel, E., Bellosta, M.-J., Dujardin, E., and Simon, E. 1996. Type-safe relaxing of schema consistency rules for flexible modelling in OODBMS. *VLDB J. 5,* 2 (Apr.).

Amiel, E. and Dujardin, E. 1996. Supporting explicit disambiguation of multi-methods. In *ECOOP Conference Proceedings*. Springer-Verlag, Berlin, Germany.

Amiel, E., Gruber, O., and Simon, E. 1994. Optimizing multi-methods dispatch using compressed dispatch tables. In *OOPSLA Conference Proceedings*. ACM, New York.

André, P. and Royer, J.-C. 1992. Optimizing method search with lookup caches and incremental coloring. In *OOPSLA Conference Proceedings*. ACM, New York, 110–126.

Apple Computer. 1995. *Dylan reference manual, draft*. Apple Computer, Cupertino, Calif. Available via http://www.harlequin.com/books/DRM/.

Barrett, K., Cassels, B., Haahr, P., Moon, D. A., Playford, K., and Withington, P. T. 1996. A monotonic superclass linearization for Dylan. In *OOPSLA Conference Proceedings*. ACM, New York.

Bell, J. 1974. A compression method for compiler precedence tables. In *Proceedings of the IFIP Congress, booklet 2*. North-Holland, Amsterdam, The Netherlands.

Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S., Kiczales, G., and Moon, D. A. 1988. Common Lisp Object System specification. *ACM SIGPLAN Not. 23,* Special Issue (Sept.). ANSI X3J13 committee Document 88-002R.

Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. 1986. CommonLoops: Merging Lisp and object-oriented programming. In *OOPSLA Conference Proceedings*. ACM, New York.

Chambers, C. 1992. Object-oriented multi-methods in Cecil. In *ECOOP Conference Proceedings*. Springer-Verlag, Berlin, Germany.

Chambers, C. 1993. The Cecil language, specification and rationale. Tech. Rep. 93-03-05, Dept. of Computer Science and Engineering, Univ. of Washington, Seattle, Wash.

Chambers, C. and Leavens, G. T. 1995. Typechecking and modules for multimethods. *ACM Trans. Program. Lang. Syst. 17,* 6 (Nov.).

Chambers, C. and Ungar, D. 1991. Making pure object-oriented languages practical. In *OOPSLA Conference Proceedings*. ACM, New York.

Chambers, C., Ungar, D., and Lee, E. 1989. An efficient implementation of SELF, a dynamically typed object-oriented language based on prototypes. In *OOPSLA Conference Proceedings*. ACM, New York, 49–70.

Chen, W. and Turau, V. 1995. Multiple dispatching based on automata. *Theory Pract. Obj. Syst. 1,* 1.

Dencker, P., Drre, K., and Heuft, J. 1984. Optimization or parser tables for portable compilers. *ACM Trans. Program. Lang. Syst. 26,* 4 (Oct.), 546–572.

Deutsch, L. P. 1983. The Dorado Smalltalk-80 implementation: Hardware architecture's impact on software architecture. In *Smalltalk-80: Bits of History and Words of Advice*. Addison-Wesley, Reading, Mass.

Deutsch, L. P. and Schifman, A. 1984. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the 11th Annual ACM POPL Conference Proceedings*. ACM, New York.

Dixon, R., McKee, T., Schweizer, P., and Vaughan, M. 1989. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA Conference Proceedings*. ACM, New York, 211–214.

Driesen, K. 1993. Selector table indexing and sparse arrays. In *OOPSLA Conference Proceedings*. ACM, New York.

Driesen, K. and Hölzle, U. 1995. Minimizing row displacement dispatch tables. In *OOPSLA Conference Proceedings*. ACM, New York.

Ducournau, R., Habib, M., Huchard, M., and Mugnier, M. 1992. Monotonic conflict resolution mechanisms for inheritance. In *OOPSLA Conference Proceedings*. ACM, New York.

Ducournau, R., Habib, M., Huchard, M., and Mugnier, M. 1994. Proposal for a monotonic multiple inheritance linearization. In *OOPSLA Conference Proceedings*. ACM, New York.

DUJARDIN, E. 1996. Efficient dispatch of multimethods in constant time with dispatch trees. Tech. Rep. 2892, INRIA, Rocquencourt, France.

ELLIS, M. A. AND STROUSTRUP, B. 1992. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Mass.

FERRAGINA, P. AND MUTHUKRISHNAN, S. 1996. Efficient dynamic method-lookup in object oriented programs. In *Proceedings of the European Symposium on Algorithms*. Springer-Verlag, Berlin, Germany.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Mass.

HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. 1991. Optimizing dynamically typed object-oriented languages using polymorphic inline caches. In *ECOOP Conference Proceedings*. Springer-Verlag, Berlin, Germany, 21–36.

HÖLZLE, U. AND UNGAR, D. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst. 18,* 4 (July).

HUANG, S.-K. AND CHEN, D.-J. 1992. Two-way coloring approaches for method dispatching in object-oriented programming system. In *Proceedings of the Annual International Computer Software and Applications Conference*.

JOHNSON, R. E., GRAVER, J. O., AND ZURAWSKI, L. W. 1988. TS: An optimizing compiler for Smalltalk. In *OOPSLA Conference Proceedings*. ACM, New York, 18–26.

KICZALES, G. AND RODRIGUEZ, L. 1990. Efficient method dispatch in PCL. In *Proceedings of ACM Conference on Lisp and Functional Programming*. ACM, New York.

KNUTH, D. 1973. *The Art of Computer Programming. Vol. 1, Fundamental Algorithms*, 2nd ed. Addison-Wesley, Reading, Mass.

MELTON, J., Ed. 1994. *SQL Persistent Stored Modules (SQL/PSM)*. Number ANSI X3H2-94-331. ANSI, New York.

MELTON, J. 1996. An SQL3 snapshot. In *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society, Washington, D.C.

MUGRIDGE, W. B., HAMER, J., AND HOSKING, J. G. 1991. Multi-methods in a statically-typed programming language. In *ECOOP Conference Proceedings*. Springer-Verlag, Berlin, Germany.

QUEINNEC, C. 1995. Fast and compact dispatching for dynamic object-oriented languages. Current state of the work available via ftp://ftp.inria.fr/INRIA/Projects/icsla/Papers/dispatch.ps.

SCHMIDT, H. W. AND OMOHUNDRO, S. 1991. CLOS, Eiffel, and Sather: A comparison. Tech. Rep. TR-91-047, International Computer Science Inst., Berkeley, Calif.

SITES, R. L. 1992. *Alpha Architecture Reference Manual*. Digital Press, Maynard, Mass.

UNGAR, D. 1986. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, Cambridge, Mass.

UNGAR, D. AND PATTERSON, D. 1983. Berkeley Smalltalk: Who knows where the time goes? In *Smalltalk-80: Bits of History and Words of Advice*. Addison-Wesley, Reading, Mass.

UNGAR, D. AND PATTERSON, D. 1987. What price Smalltalk? *IEEE Comput. 20,* 1.

UNGAR, D., SMITH, R. B., CHAMBERS, C., AND HÖLZLE, U. 1992. Object, message and performance: How they coexist in SELF. *IEEE Comput. 25,* 10 (Oct.).

VITEK, J. AND HORSPOOL, R. N. 1994. Taming message passing: Efficient method look-up for dynamically typed languages. In *ECOOP Conference Proceedings*. Springer-Verlag, Berlin, Germany.

VITEK, J. AND HORSPOOL, R. N. 1996. Compact dispatch tables for dynamically typed languages. In *Compiler Construction Conference Proceedings*. Springer-Verlag, Berlin, Germany.

Received September 1996; revised August 1997; accepted November 1997