# Efficient Dynamic Dispatching with Type Slicing

Joseph (Yossi) Gil[1]
yogi@cs.technion.ac.il
and
Yoav Zibin
yoav@zibin.net
Technion—Israel Institute of Technology

A fundamental problem in the implementation of object-oriented languages is that of a frugal implementation of dynamic dispatching, i.e., a small footprint data structure that supports quick response to runtime dispatching queries of the following format: which method should be executed in response to a certain message sent to a given object. Previous theoretical algorithms for this problem tend to be impractical due to their conceptual complexity and large hidden constants. In contrast, successful practical heuristics lack theoretical support.

The contribution of this paper is in a novel *type slicing* technique, which results in two dispatching schemes: TS and $\mathrm{CT}_d$. We make the case for these schemes both practically and theoretically. The empirical findings on a corpus of 35 hierarchies totaling some 64 thousand types from 8 different languages, demonstrate improvement over previous results in terms of the space required for the representation, and the time required for computing it.

The theoretical analysis is with respect to $\iota$, the best possible compression factor of the dispatching matrix. The results are expressed as a function of a parameter $\kappa$, which can be thought of as a metric of the complexity of the topology of a multiple inheritance hierarchy. In single inheritance hierarchies $\kappa = 1$, but although $\kappa$ can be in the order of the size of the hierarchy, it is typically a small constant in actual use of inheritance; in our corpus, the median value of $\kappa$ is 5, while its average is 6.4.

The TS scheme generalizes the famous interval containment technique to multiple inheritance. TS achieves a compression factor of $\iota/\kappa$, i.e., our generalization comes with an increase to the space requirement by a small factor of $\kappa$. The pay is in the dispatching time, which is no longer constant as in a naive matrix implementation, but logarithmic in the number of different method implementations. In practice, dispatching uses one indirect branch and, on average, only 2.5 binary branches.

The CT schemes are a sequence of algorithms $\mathrm{CT}_1$, $\mathrm{CT}_2$, $\mathrm{CT}_3$, ..., where $\mathrm{CT}_d$ uses $d$ memory dereferencing operations during dispatch, and achieves a compression factor of $\frac{1}{d}\iota^{1-1/d}$ in a *single inheritance* setting. A generalization of these algorithms to a *multiple inheritance* setting, increases the space by a factor of $(2\kappa)^{1-1/d}$. This tradeoff represents the first bounds on the compression ratio of constant-time dispatching algorithms. We also present an *incremental* variant of the $\mathrm{CT}_d$ suited for languages such as Java.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types; structures*; G.4 [**Mathematical Software**]: Algorithm design; analysis

Some of the material presented here was published previously in the proceedings of OOPSLA'02 [Zibin and Gil 2002] and the proceedings of POPL'03 [Zibin and Gil 2003].

General Terms: Algorithms, Design, Measurement, Performance, Theory
Additional Key Words and Phrases: CT, Dispatch, Dynamic-typing, Hierarchy, Incremental, Message, Subtyping, Type slicing

## 1. INTRODUCTION

*Message dispatching* stands at the heart of object-oriented (OO) programs, being the only way objects communicate with each other. Indeed, it was demonstrated [Driesen and Hölzle 1996] that OO programs spend a considerable amount of time in implementing dynamic dispatching. There is a large body of research dedicated to the problem of "efficient" implementation of message dispatching [Conroy and Pelegri-Llopart 1983; Deutsch and Schiffman 1984; Dixon et al. 1989; Kiczales and Rodriguez 1990; Hölzle et al. 1991; Pascal and Royer 1992; Driesen 1993; Vitek and Horspool 1994; Vitek 1995; Driesen and Hölzle 1995; Driesen et al. 1995b; Ferragina and Muthukrishnan 1996; Muthukrishnan and Müller 1996; Vitek and Horspool 1996; Driesen and Hölzle 1996; Zendra et al. 1997; Driesen et al. 1995a; Driesen 1999; Naik and Kumar 2000].

To implement dynamic binding during dispatch, the runtime system of OO languages uses a *dispatching data structure*, in which a *dispatching query* finds the appropriate implementation of the message to be called, according to the dynamic type of the message receiver. A fundamental problem in the implementation of such languages is then a frugal implementation of this data structure, i.e., simultaneously satisfying (i) compact representation of the type hierarchy and the sets of different implementations of each message, (ii) quick response to dispatching queries, and (iii) fast creation of the dispatching data structure.

These are the main variation of the dispatching problem:

(1) **Single inheritance vs. Multiple inheritance** In a single inheritance hierarchy each type has at most one direct supertype, which means that the hierarchy takes a tree or forest topology, as in SMALLTALK [Goldberg 1984], OBJECTIVEC [Cox 1986], and other OO languages. Algorithms in the single inheritance setting tend to be more efficient than the general case of multiple inheritance. Some OO languages fall in between these two variations. For example, JAVA [Arnold and Gosling 1996] has a multiple inheritance *type* hierarchy, but a single inheritance *class* hierarchy.

(2) **Incremental- vs. batch- algorithms** We are also interested in *incremental* algorithms where the hierarchy evolves at runtime. The most important kind of change is the addition of types (together with their accompanying messages and methods) at the bottom of hierarchy, also called *dynamic loading*. (This is the case in JAVA, where types may be added as leaves at run time.) Previous research explored addition of methods to existing nodes, as well as deletion of types and other modifications.

(3) **Statically- vs. dynamically typed languages** Statically typed languages such as EIFFEL [ISE 1997] and C++ [Stroustrup 1997] may provide (partial) type information. The challenge is in utilizing this information at runtime.

Conversely, it is often more difficult to find algorithms for dynamically typed languages (sometimes called *dynamic-typing*).

*Virtual function tables* (VFT) make a simple and well known (see e.g., [Stroustrup 1994]) incremental technique which achieves dispatching in constant time (two dereferencing operations), and very good compaction rates. The VFT of each type is an array of method addresses. A location in this array represents a message, while its content is the address of an implementing method. The VFT of a subtype is an extension of the VFT of its supertype, and messages are allocated locations at compile time in sequential order. The static type of the receiver uniquely determines the location associated with each message. VFTs rely both on *static typing* and *single inheritance*. *Multiple inheritance* implementations exist [Gil and Sweeney 1999], but they are not as elegant or efficient.

The challenge in the dispatching problem is therefore mostly in dealing with *dynamically typed* and/or multiple inheritance languages. Our contribution (described in greater detail in Section 1.3) includes a provable tradeoff between space and dispatching time with extensions to multiple inheritance hierarchies. The pinnacle of the results is an incremental algorithm for maintaining a compact dispatch table in languages such as JAVA.

The empirical evaluation of our algorithms was done on a corpus of 35 hierarchies totaling some 64 thousand types from 8 different languages, with the purpose of making our research *language independent*. We demonstrate improvement over previous results in terms of the space required for the representation, and the time required for computing it. We describe the dispatching sequence for each technique, without any hidden constants, however, we do not present actual dispatching time nor do we measure code-space costs due to the language independent setting. Incorporating our algorithms into a runtime system (e.g., in an industrial JVM), fine tuning various parameters, and measuring runtime and code-space costs are left for future work.

## 1.1 The Problem

We define the dispatching problem in a similar fashion to the *colored-ancestors* abstraction described by Ferragina and Muthukrishnan [Ferragina and Muthukrishnan 1996]: a *hierarchy* is a partially ordered set $\langle \mathcal{T}, \preceq \rangle$ where $\mathcal{T}$ is a set of types[2] and $\preceq$ is a reflexive, transitive and anti-symmetric *subtype relation*. For example, in JAVA, $\mathcal{T}$ is the set of classes and interfaces, and $\preceq$ is the transitive closure of the **`extends`** and **`implements`** relations.

If $a$ and $b$ are types, i.e., $a, b \in \mathcal{T}$, and $a \preceq b$ holds, we say that $a$ is a *subtype* (or a *descendant*) of $b$ and that $b$ is a *supertype* (or an *ancestor*) of $a$. Direct subtypes (supertypes) are called *children* (*parents*).

We use the term *message* for the unique identifier of a set of implementing *methods* (also called member functions, operations, features, implementations, etc.).

A message, which is sometimes called a *selector* (in e.g., SMALLTALK or OBJECTIVEC) or a *signature* (in e.g., JAVA or C++ ), may include, depending on the programming language, components such as name, arity, and even the type of

[2]The distinction between type, class, interface, signature, etc., as it may occur in various languages does not concern us here. We shall refer to all these collectively as *types*.

parameters. (Note that a consequence of feature renaming in EIFFEL, is that the message does not always include the name of a routine.)

The intuition however is the same in all OO languages: when an object receives a message encoded as a selector, *dispatching on the dynamic type of the receiver* must take place at runtime to find and invoke the implementation which is most appropriate for the receiver's dynamic type.

We will use the terms *message* and *selector* interchangeably.

Let the min(·) operator return the set of smallest types in any given set of types, i.e.,

$$\min(X) = \{t \in X \mid \nexists t' \in X \bullet t' \neq t, t' \preceq t\}. \tag{1.1}$$

Given a message $m$, $I(m) \subseteq \mathcal{T}$ are the *implementors of* $m$, i.e., the set of types which have a *method implementation* for $m$. Given a message $m$ and a type $t$, $\mathsf{cand}(m,t)$ is the set of candidates in $I(m)$, i.e., those ancestors of $t$ with an implementation of $m$:

$$\mathsf{cand}(m,t) \equiv I(m) \cap \text{ancestors}(t). \tag{1.2}$$

A *dispatching query* $\mathsf{dispatch}(m,t)$ returns either *the smallest candidate* or null if no such unique candidate exists. (A null result represents either the *message not understood* or *message ambiguous* error conditions.) Specifically,

$$\mathsf{dispatch}(m,t) \equiv \begin{cases} t' & \text{if } \min(\mathsf{cand}(m,t)) = \{t'\}, \\ \text{null} & \text{otherwise.} \end{cases} \tag{1.3}$$

Figure 1.1 depicts a hierarchy which will serve as our running example in this paper. In the figure and henceforth, type names are denoted by uppercase letters, printed in sans-serif font: A, B, C, ..., while lower case letters in the same font denote messages.

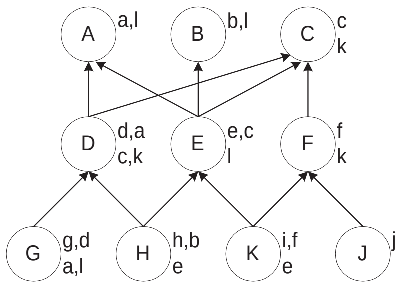


Fig. 1.1. A small example of a hierarchy and the methods implemented in each type

In the figure we see, for example, that type D implements messages d, a, c and k. The implementors of message c are $I(\mathsf{c}) = \{\mathsf{C},\mathsf{D},\mathsf{E}\}$. We also see in Figure 1.1 that for this message

| | |
|---|---|
| $\mathsf{cand}(\mathsf{c},\mathsf{K}) = \{\mathsf{C},\mathsf{E}\}$ | $\mathsf{dispatch}(\mathsf{c},\mathsf{K}) = \mathsf{E}$ |
| $\mathsf{cand}(\mathsf{c},\mathsf{B}) = \emptyset$ | $\mathsf{dispatch}(\mathsf{c},\mathsf{B}) =$ null (*message not understood*) |
| $\mathsf{cand}(\mathsf{c},\mathsf{H}) = \{\mathsf{C},\mathsf{D},\mathsf{E}\}$ | $\mathsf{dispatch}(\mathsf{c},\mathsf{H}) =$ null (*message ambiguous*) |

The type checker of statically typed languages makes sure at compile time that dispatching never results in null. It would therefore be a compilation error in such a langauge to send c to objects whose static type is B or H. Moreover, it is a compilation error even to send this message to any ancestor of H, e.g., type C. The reason is that the type analyzer cannot infer [Gil and Itai 1998] that the dynamic type is not H.

We shall assume a pre-processing stage in which all ambiguities are resolved by an appropriate augmentation of the set implementors. In the example, we add type H to the implementors $I(\mathsf{c})$ since $\mathsf{dispatch}(\mathsf{c}, \mathsf{H})$ resulted in *message ambiguous.* As in previous work [Holst et al. 1998; Pang et al. 1999] in which this assumption was made, our working hypothesis is that the ensuing increase of problem size is insignificant in practice.

Figure 1.1 is an example of a *multiple inheritance* hierarchy, since, e.g., type D has two parents: A and C. *Single inheritance*, in which each type has at most one parent, is mandated by languages such as SMALLTALK and OBJECTIVEC. The fact that single inheritance hierarchies take a simple forest topology, makes single inheritance an important special case, for which very efficient algorithms exist.

DEFINITION 1.1. *Given a hierarchy $\langle \mathcal{T}, \preceq \rangle$, a set of messages $\mathcal{M}$, and the set of implementors $I(m) \subseteq \mathcal{T}$ for every $m \in \mathcal{M}$, the* dispatching problem *is to* encode *the hierarchy in a data structure supporting* $\mathsf{dispatch}(m, t)$ *queries for every message $m \in \mathcal{M}$ and every type $t \in \mathcal{T}$.*

We assume that each object includes an accessible type-id, and tacitly ignore the object space overheads and the time of retrieving such type-id. This assumption is justified by the practice of implementation of OO languages, which invariably attach a run-time type identifier to each object.

Also, we assume that a message is supplied to the dispatching mechanism at runtime as an integer selector. In the batch (non-incremental) variant, we assume that this selector is known at compile time, and accordingly allow any pre-processing which is dependent solely on this selector. Given the object type-id and this selector, the dispatch query means that the runtime system must compute the address of the method defined in the smallest candidate, and jump to it.

A solution to the dispatching problem is measured by the following three metrics: (i) the space that the data structure occupies in memory, (ii) the time required for processing a query, and (iii) the time for generating the data structure from the input hierarchy. The incremental version of the problem, is to maintain this data structure in the face of additions of types (with their accompanying methods) to the bottom of the hierarchy, as done in languages such as JAVA.

We would like to express these metrics as a function of the following parameters of the problem:

—The number of *types* in the hierarchy

$$\mathbf{n} \equiv |\mathcal{T}|. \tag{1.4}$$

—The number of *different messages* that can be sent during runtime

$$\mathbf{m} \equiv |\mathcal{M}|. \tag{1.5}$$

—The total number of *different method implementations*

$$\mathbf{i} \equiv \sum_{m \in \mathcal{M}} |I(m)|. \tag{1.6}$$

—The number of valid *message-type combinations*, i.e., combinations which do not result in null

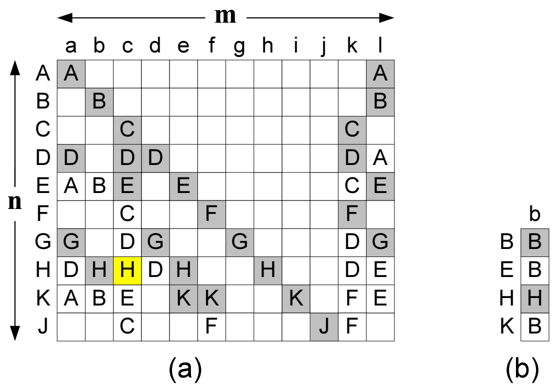
$$\mathbf{w} \equiv |\{\langle m, t\rangle \mid \mathsf{dispatch}(m, t) \neq \text{null}\}|. \tag{1.7}$$

In Figure 1.1 for example, we have $\mathbf{n} = 10$, $\mathbf{m} = 12$, $\mathbf{i} = 27$ and $\mathbf{w} = 46$.

## 1.2 Simple Solutions

The most obvious solution to the dispatching problem is in a $\mathbf{n} \times \mathbf{m}$ *dispatching matrix*, storing the outcomes of all possible dispatching queries. We stress that the order of rows and columns in the dispatching matrix is arbitrary, and the performance of some algorithms for compressing the matrix may depend heavily on the chosen ordering.

The dispatching matrix of our running example is presented in Figure 1.2a, where the $\mathbf{nm}-\mathbf{w}$ type-message pairs which result in null are represented as empty entries.



| | a | b | c | d | e | f | g | h | i | j | k | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | | | | | | | | | | | A |
| B | | B | | | | | | | | | | B |
| C | | | C | | | | | | | | C | |
| D | D | | D | D | | | | | | | D | A |
| E | A | B | E | | E | | | | | | C | E |
| F | | | C | | | F | | | | | F | |
| G | G | | D | G | | | G | | | | D | G |
| H | D | H | H | D | H | | | H | | | D | E |
| K | A | B | E | | K | K | | | K | | F | E |
| J | | | C | | | F | | | | J | F | |

| | b |
|---|---|
| B | B |
| E | B |
| H | H |
| K | B |

Fig. 1.2. (a) The dispatching matrix, and (b) the sorted dictionary for message b

The figure depicts in grey all $\mathbf{i}$ entries which represent a method implemented in a certain type. For example, the top right grey entry is to say that type A has an implementation of message l. (Recall that type H was added to the implementors $I(\mathsf{c})$ to resolve an ambiguity. Therefore, the cell corresponding to $\langle \mathsf{H}, \mathsf{c}\rangle$ is rendered in grey.)

In the matrix representation, queries are answered by a quick indexing operation. However, the space consumption is inhibitively large, e.g., 512MB for the dispatching matrix in the largest hierarchy in our benchmarks (8,793 types and 14,575 messages).

There are two opportunities for compressing the dispatching matrix: *null elimination* and *duplicates elimination.*

1.2.1 *Null elimination.* There is much empirical evidence to show that dispatching matrices are very sparse. *Null elimination* is the attempt to store only the non-null elements in the matrix. Examples of null-elimination schemes are *row displacement* [Driesen 1993; Driesen and Hölzle 1995], *selector coloring* [Dixon et al. 1989; Pascal and Royer 1992], and *virtual function tables* (VFT) [Stroustrup 1994]. In single inheritance and static-typing setting of the problem, the VFT technique uses precisely $\mathbf{w}$ memory cells.

The ratio $(\mathbf{nm})/\mathbf{w}$ is an upper bound on the compression rate which null elimination might achieve. The matrix of Figure 1.2a has $120 = 10 \times 12$ entries, out of which, 46 are non-null. Null elimination in this case gives a compression factor of no more than $120/46 \approx 2.6$. In our benchmarks we found that on average, null elimination might achieve compression by a factor of circa 150.

Null elimination can be achieved by storing each column as a *sorted dictionary*, i.e., a sorted array of ⟨key,value⟩-pairs. In the running example, the sorted dictionary for message b is depicted in Figure 1.2b. In this implementation, the query time is logarithmic in the number of non-null entries in each column. Space is linear in this number.

*Dynamic perfect hashing* (DPH) [Dietzfelbinger et al. 1994] is theoretically better than sorted dictionaries. In this algorithm, each column (or the entire matrix for that matter) is stored as a hash table. Indices (or their concatenation) serve as keys. The space requirement is linear in $\mathbf{w}$. More importantly, query time is constant! Unfortunately, DPH is of mere theoretical interest since it carries large hidden constants, which might offset any saving of space due to null elimination. Even though dispatching time is constant in perfect hashing, it is complicated by the finite-field arithmetic incurred during the computation of the hash function.

The more sophisticated previously published practical algorithms, try, and in most cases achieve, complete, or almost complete null elimination with no hidden constants and constant search time.

1.2.2 *Duplicates elimination.* Even though optimal null elimination may give very good results, it still leaves something to be desired. In one hierarchy of our data set, featuring 3,241 types, an optimal null elimination scheme still requires 2.4MB. *Duplicates elimination* improves on null elimination by attempting to store only the *distinct* $\mathbf{i}$ entries of the dispatching matrix. Therefore, the compression factor of duplicates elimination is at most

$$\iota \equiv (\mathbf{nm})/\mathbf{i}. \tag{1.8}$$

We shall refer to $\iota$ as the *optimal compression factor*, and to schemes attempting to reach $\iota$ as *duplicates-elimination schemes.* In our data-set of 35 large hierarchies (see Section 8), $\iota \approx 725$.

The ratio $\mathbf{w}/\mathbf{i}$ gives the factor by which duplicates elimination can improve on null elimination. This ratio was as high as 122.4 in one of our benchmarks. In the matrix of Figure 1.2a there are 27 distinct entries, i.e., $\mathbf{i} = 27$, so duplicates elimination has the potential of compressing the dispatching matrix by a factor of $120/27 \approx 4.44$.

It is not difficult to come close to full duplicates elimination, with a simple representation of the hierarchy as a graph where types are nodes and immediate inheritance relations are edges. The cost is of course the search time, which becomes $O(\mathbf{n})$, since each dispatch must traverse all the ancestors of a receiver in order to find the smallest candidate. Sophisticated caching algorithms (as employed in the runtime system of SMALLTALK) make the typical case more tolerable than what the worst case indicates.

The two dispatching techniques described in this paper (TS and CT) are both duplicates elimination. Our challenge is to come as close as possible to optimal duplicates elimination, i.e., space linear in the number of implementations $\mathbf{i}$, while still maintaining small, preferably constant, query time.

## 1.3 Contribution

There is a large body of research on the dispatching problem (see e.g., [Deutsch and Schiffman 1984; Dixon et al. 1989; Hölzle et al. 1991; Pascal and Royer 1992; Driesen 1993; Vitek and Horspool 1994; Driesen and Hölzle 1995; Vitek and Horspool 1996; Zendra et al. 1997; Zibin and Gil 2002; 2003]). The focus in these was on "practical" algorithms, which were evaluated empirically, rather than by provable upper bound on memory usage. The main theoretical research on the topic [Ferragina and Muthukrishnan 1996; Muthukrishnan and Müller 1996] produced algorithms (for the single inheritance setting) which using minimal space ($O(\mathbf{i})$ cells) supported dispatching in doubly logarithmic, $O(\lg\lg \mathbf{n})$, time. However, the hidden constants are large, and the implementation is complicated.

Our two new algorithms are simple, efficient in practical terms, and their theoretical analysis does not carry any hidden constants. Both algorithms uses a novel *type slicing* technique, whose efficiency is parameterized by the complexity of the topology of a multiple inheritance hierarchy, denoted $\kappa$. In practice, this factor is small, but in arbitrary hierarchies it might be in the order of the number of types. In all single inheritance hierarchies, $\kappa \equiv 1$.

We are unaware of any non-exponential method for finding $\kappa$. Instead we use the PQ-trees heuristic [Zibin and Gil 2001] which gives an *upper-bound* on $\kappa$. In a benchmark of 19 multiple inheritance hierarchies with 34,810 types, we found the median value of an upper bound for $\kappa$ is 5, the average is 6.4, and the maximum is 18. To achieve fast creation times in the TS algorithm we use an incremental heuristic (see Appendix A) instead of using PQ-trees. When using the incremental heuristic the median value of $\kappa$ increases to 6.5, the average to 7.3, and the maximum to 19.

1.3.1 *TS algorithm.* Informally, we can say that our TS algorithm (first published in [Zibin and Gil 2002]) generalizes the linear space *interval containment* algorithm [Muthukrishnan and Müller 1996; Ferragina and Muthukrishnan 1996] which is restricted to the single inheritance setting. Our main theoretical result is that the generalization to the multiple inheritance case comes with a $\kappa$ factor increase in space, but in practice we find much better results.

In a collection of 35 hierarchies, totaling over 60,000 types, its *space requirement* improves those of the famous *row displacement* (RD) algorithm [Driesen and Hölzle 1995] (arguably the best previously published algorithm in this category), in 32 out of the 35 hierarchies of our data set; the median reduction in space is by a factor of 2.6. The slowest runtime of our TS algorithm was less than a third of a second on a modern processor; this time was on a hierarchy of circa nine thousand types and fourteen thousand messages. In the vast majority of the hierarchies, the creation time was less than a hundredth of a second. This is an improvement by one, two and sometimes three orders of magnitude compared to RD.

The improvement of creation time and of space requirement comes with a penalty of a small increase to dispatching time. Specifically, dispatching requires a binary search in which the number of branches is logarithmic in the number of implementations of the dispatched message, or alternatively, doubly-logarithmic in the number of types. Each dispatch requires about 2.5 branches on average, as well as one dereferencing operation. These numbers may be compared with the two dereferencing steps required by the *Virtual Function Tables* (VFT) [Ellis and Stroustrup 1994] standard implementation strategy of C++ in the single inheritance setting. Note that in contrast with our results and most other dispatching algorithms, the VFT technique is valid only in *statically typed* languages [Vitek and Horspool 1996]. Some dispatching schemes, such as RD and selector coloring (SC), require additional space and one more comparison at runtime in order to work in *dynamically typed* languages.

Interestingly, there is a strong practical evidence that binary searches, which are used in our implementation, may be faster than the simple VFT implementation. The trick is to inline the binary search by generating what was called "static branch code" by the implementors of the SmallEiffel compiler [Zendra et al. 1997], instead of the more general binary search routine. It was shown that with this optimization a binary search between fewer than 50 results was faster than the VFT implementation in most architectures.

One of the explanations of this phenomenon is that indirect branches do not schedule well on modern processors [Driesen et al. 1995b; Driesen and Hölzle 1996; Driesen et al. 1995a; Driesen 1999]. Other, less direct, advantages of inlined binary search is that it can take better advantage of type inference and that it is more susceptible to inlining of method code and any ensuing optimization. The cost of inlining is (of course) in an increase to the code size. Note that several other previous publications suggested using a combination of binary searches, array lookups, and even linear searches [Hölzle et al. 1991; Chambers and Chen 1999; Naik and Kumar 2000; Alpern et al. 2001] for dispatching.

1.3.2 $\mathrm{CT}_d$ *algorithm.* Our $\mathrm{CT}_d$ algorithm (first published in [Zibin and Gil 2003]) presents a different tradeoff than all previous work: constant-time dispatching in $d$ steps, while using at most $d\mathbf{i}\sqrt[d]{\iota}$ cells. Stated differently, our results are that $d$ steps in dispatching (provably) achieve a compression rate of $\frac{\iota}{d\sqrt[d]{\iota}}$. For example, with $d = 2$ the compression is by a factor of at least half of the square root of $\iota$, the optimal compression rate. Also, the compression factor is close to optimal, $\frac{\iota}{2\lg \mathbf{m}}$, when the dispatching time is logarithmic, $\lg \mathbf{m}$.

An important advantage of these results in comparison to previous theoretical algorithms is that they are simple and straightforward to implement, and bear no hidden constants. In fact, our algorithms are based on a successful practical technique, namely *compact dispatch tables* (CT), which was invented by Vitek and Horspool [Vitek and Horspool 1996]. Viewed differently, the results presented here give the first proof of a non-trivial upper bound on practical algorithms.

Even though the algorithms carry on to multiple inheritance with the same time bounds of dispatching, the memory consumption increases by a factor of at most $(2\kappa)^{1-1/d}$.

We give empirical evidence that the algorithms perform well in practice, in many cases even better than the theoretically obtained upper bounds.

We also describe an incremental version of the algorithms for languages such as JAVA, and prove that updates to the dispatching data structures can be made in optimal time. The cost is in a small constant factor increase (e.g., 2) to the memory footprint.

Readers may also take interest in some proof techniques, including the representation of dispatching as search in a collection of partitionings, the elegant Lemma 6.1, and the amortization analysis of the incremental algorithm.

**Outline** The remainder of this article is organized as follows. A survey of prior dispatching techniques is the subject of Section 2.

Our TS algorithm is described in Section 3. Section 4 presents the generalized CT schemes for single inheritance hierarchies. Section 5 shows how these schemes can be made incremental. A (non-incremental) version of these schemes for multiple inheritance hierarchies is described in Section 6.

The data set of the 35 hierarchies used in our benchmarking is presented in Section 7. Section 8 presents the experimental results, comparing the performance of both algorithms with those of previous algorithms. Finally, Section 9 mentions open problems and directions for future research. Appendix A describes our heuristic for performing type slicing.

## 2. PREVIOUS WORK

This section gives an overview of some of the dispatching techniques proposed in the literature. The performance of these techniques might be improved by using various forms of caching at runtime (see e.g., [Conroy and Pelegri-Llopart 1983; Deutsch and Schiffman 1984; Hölzle et al. 1991]).

**VFT:** ***Virtual Function Tables*** [Ellis and Stroustrup 1994] As mentioned above, the VFT technique is valid only in statically typed languages [Vitek and Horspool 1996]. In a single inheritance setting, VFT achieves optimal null elimination and constant dispatch time. A distinguishing property of the technique is that it does not require whole program information. The VFT of any type can be constructed using only information regarding its ancestors.

The multiple inheritance version of the VFT is much more complicated than the single inheritance version, with complicated space and time overheads. Each type stores multiple VFTs, and if a method is inherited along more than one path, then it will be stored in these more than once. Further, in presence of shared (virtual) inheritance, searching for an implementation is carried out by either following a chain of pointers to ancestors, or by additional increase to object size using *inessential virtual base pointers* [Gil and Sweeney 1999]. It was shown [Eckel and Gil 2000] that these space overheads can be very significant. Even with this overhead, dispatching time increases due to what is known in the C++ jargon as **`this`**-*adjustment*.[3]

[3]In general, dispatching in C++ is tightly coupled with its peculiar object-layout, and is therefore not directly applicable to languages with different layout scheme. Simple object-layout have the advantage of fast synchronization, hash-codes, and easier garbage collection.

**RD:** ***Row Displacement*** [Driesen 1993; Driesen and Hölzle 1995] Another null elimination technique is due to Driesen [Driesen 1993] who suggested to displace the rows in the dispatching matrix by different offsets so that they could be merged together in a *master array.* Later [Driesen and Hölzle 1995] it was found that *selector-based RD*, i.e., a displacement of columns rather than rows, gives much better compression values. In fact, this technique comes very close (median value 94.7%) to optimal null elimination.

In dynamically typed languages vanilla RD does not work, since null entries which correspond to *message not understood* will usually become occupied. It is possible to amend RD with an increase to space requirement and adding one more comparison at runtime.[4] We stress that duplicates elimination (which we use) does not suffer from this limitation.

It is obvious that selector-based RD is not incremental, but we note that even type-based RD does not perform well incrementally since it relies on a global heuristic for reordering of the selectors.

Figure 2.1 shows parts of the selector-based RD compression of our running examples. Note that the types are reordered to achieve better compression values.

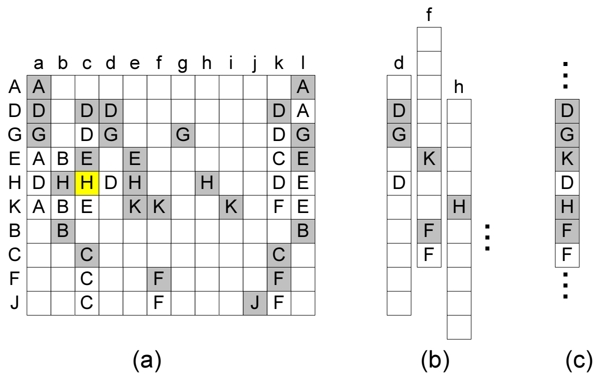


Fig. 2.1. (a) The dispatching matrix of Figure 1.2a with a different type ordering, (b) the columns with different offsets, and (c) the master array of row-displacement

**CT:** ***Compact dispatch Tables*** [Vitek and Horspool 1994; Vitek 1995; Vitek and Horspool 1996] The very good compression results of RD were improved significantly by Vitek and Horspool on some hierarchies. Their CT technique aims at duplicates elimination. The idea is to partition the set of messages $\mathcal{M}$ into disjoint *slices* $\mathcal{M}_1, \ldots, \mathcal{M}_\mathbf{k}$. (Vitek and Horspool recommend that each slice contains 14 messages.) Slicing breaks the dispatching matrix into $\mathbf{k}$ sub-matrices, also called *chunks.* Identical rows within each chunk are then merged. Each slice $\mathcal{M}_i$ has an array $r_i$ of size $\mathbf{n}$. Entry $r_i[t]$ points to the row of $t$ in chunk $i$. Dispatching in CT requires an extra load compared to the dispatching matrix, but the merging of rows may reduce the space requirement.

[4]The trick is to add a prologue to each method which checks that the method indeed corresponds to the sent message.

Figure 2.2 shows a CT representation of the matrix of Figure 1.2a, using 4 slices:

$$\mathcal{M}_1 = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}, \mathcal{M}_2 = \{\mathsf{d}, \mathsf{e}, \mathsf{f}\}, \mathcal{M}_3 = \{\mathsf{g}, \mathsf{h}, \mathsf{i}\}, \mathcal{M}_4 = \{\mathsf{j}, \mathsf{k}, \mathsf{l}\}.$$

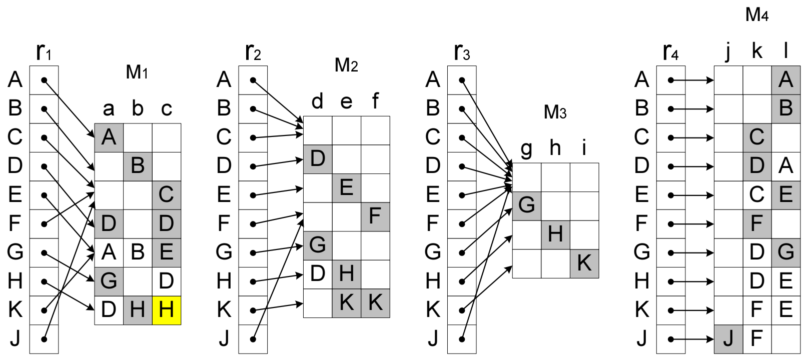


Fig. 2.2. CT representation of Figure 1.2a

Section 4 presents our CT$_d$ algorithms, where CT$_1$ is the dispatching matrix, and CT$_2$ is similar to CT except CT$_2$ performs a precise analysis that dictates an optimal slice size, instead of the arbitrary universal recommendation of 14. Using this optimal slice size, we give the first proof of a non-trivial upper bound on the memory usage of a constant-time dispatching algorithm.

**SC: *Selector Coloring*** [Dixon et al. 1989; Pascal and Royer 1992] SC aims at null elimination by slicing the set of messages. Each slice must satisfy the following property: *no two messages in the slice can be recognized by the same type.* In other words, in each chunk, a row can have at most one non-null entry. This property makes it possible to merge together all the columns in a chunk, resulting in a space requirement of $\mathbf{n} \times \mathbf{k}$.

Note that with the addition of a new type, existing slices may violate the slicing property. Therefore, SC is not suitable for incremental algorithms.

Figure 2.3 shows a SC representation of Figure 1.2a, in which there are a total of eight slices. This is the smallest possible such number, since the row of H (for example) has eight non-null entries.

The performance of SC is improved as the number of slices decreases. Since it is computationally hard to find an optimal slicing, the slices must be found using a heuristic. As in RD, null entries are treated as empty in SC and therefore additional storage and an extra comparison are required in dynamically typed languages. CT also uses SC in each of the chunks.

Our MI algorithms also use a slicing property, and a heuristic for finding a good slicing. As it turns out, slicing the set of types, rather than the set of message, as well as our particular choice of the slicing property, guarantees that the addition of new types does not invalidate existing slices. We rely on the fact that only types

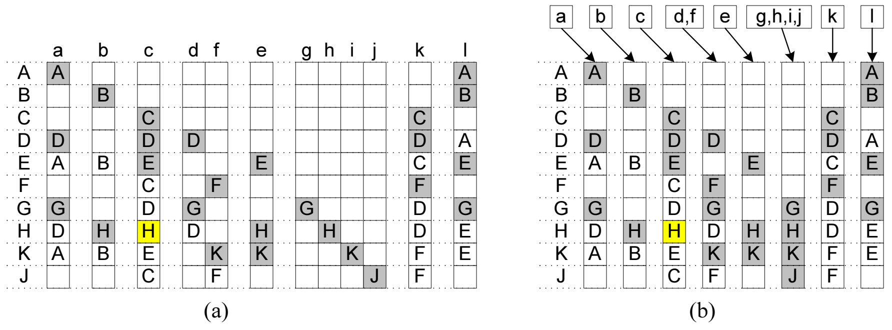


Fig. 2.3. (a) The eight chunks of the dispatching matrix of Figure 1.2a, and (b) their selector coloring representation

with their associated methods can be added and that it is forbidden to add methods to existing types.[5]

***Jalapeño*** [Alpern et al. 2001] Jalapeño, an IBM implementation of Java virtual machine, uses a fast incremental variant of SC in realizing `invokeinterface` instructions (used for dispatching messages sent to an **`interface`**). Messages are hashed into $\mathbf{k}$ slices, where $\mathbf{k}$ is an a-priori fixed number. Each type has an *interface method table* of length $\mathbf{k}$. When the slicing property of SC does not hold, i.e., some type recognizes more than one message in the same slice, then a conflict resolution thunk must be generated by the compiler. Since there is no bound on the number of conflicting messages in each hash table entry, dispatch time is not necessarily constant. It is easy to see that the total memory requirement is $\mathbf{nk}$ for the tables, plus $O(\mathbf{w})$ memory for conflict resolution.

***Interval Containment for single inheritance hierarchies*** [Muthukrishnan and Müller 1996; Ferragina and Muthukrishnan 1996] Interval containment achieves optimal duplicates elimination at the cost of non-constant dispatch time. Our TS technique (Section 3) is a generalization of interval containment for multiple inheritance hierarchies. Let us describe this technique in greater detail.

Interval containment assigns id's to types in a preorder traversal of the tree hierarchy. An important property of the preorder traversal is that descendants of a type $t$ define an *interval.* Therefore, each message $m$, defines a set of intervals, one for each type $t \in I(m)$.

Figure 2.4(a) shows a tree hierarchy with three implementations of a message $\mathsf{a}$ in types: $\mathsf{A}$, $\mathsf{B}$, and $\mathsf{F}$, i.e., $I(\mathsf{a}) = \{\mathsf{A}, \mathsf{B}, \mathsf{F}\}$. Then, as can be seen in Figure 2.4(b), these implementors define three intervals in the preorder traversal: $[1, 7]$, $[5, 7]$, and $[3, 3]$, respectively. The intersections of those three intervals partition the types into four segments: $[1, 2]$, $[3, 3]$, $[4, 4]$, and $[5, 7]$, which correspond to implementors: $\mathsf{A}$, $\mathsf{F}$, $\mathsf{A}$, and $\mathsf{B}$, respectively. The dispatch of message $\mathsf{a}$ on any given type depends only on

[5]Ferragina and Muthukrishnan [1996] study the less natural dispatching problem of a *fixed* SI hierarchy, where *methods* can be dynamically added and removed. Their results include various tradeoffs between space,randomization, query time, and update time. For instance, there is an optimal space algorithm, whose update and query times are $o(\mathbf{n}^{\epsilon})$, $\epsilon \ll 1$ and $O(\sqrt{\log \mathbf{n}})$ (respectively). A randomized version of this algorithm reduces the query time to $O(\log \log \mathbf{n})$.

the segment this type belongs to. If, for example, the receiver is of type G whose id is 6, then we find that it belongs to segment $[5, 7]$, and therefore return B.

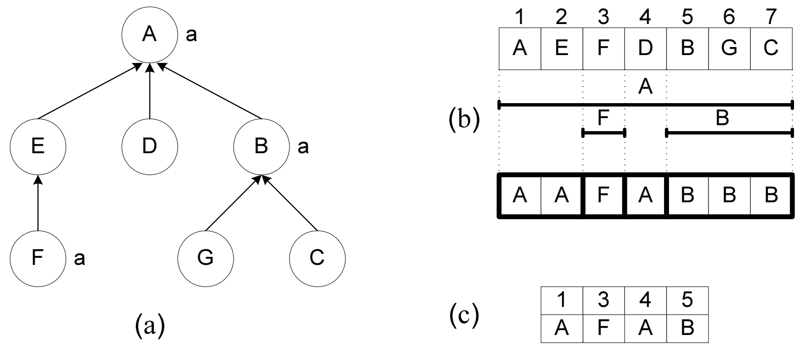


Fig. 2.4. (a) A message a, $I(\mathsf{a}) = \{\mathsf{A}, \mathsf{B}, \mathsf{F}\}$ in a tree hierarchy, (b) the intervals and segments $I(\mathsf{a})$ defines, and (c) the representation of $I(\mathsf{a})$ as a sorted dictionary

Given a message $m$, there are $|I(m)|$ intervals which partition the preorder of $\mathcal{T}$ into at most $2|I(m)| + 1$ *segments*, where all types in a segment have the same dispatching result. Message $m$ is represented as a sorted dictionary, mapping segments' starting point to methods. In our example, Figure 2.4(c) shows a sorted dictionary that represents the segment partitioning. This dictionary serves as the dispatching table for $I(\mathsf{a})$.

Note that the sorted dictionary representation is linear in $|I(m)|$. The total memory for representing all messages is therefore $O(\mathbf{i})$. In fact, the number of memory cells required by this representation is at most

$$\sum_{m \in \mathcal{M}} 2(2|I(m)| + 1) = 2\mathbf{m} + 4 \sum_{m \in \mathcal{M}} |I(m)| = 2\mathbf{m} + 4\mathbf{i}.$$

It remains to describe the representation of the sorted dictionary and the procedure to determine the segment to which a specific type belongs. Algorithmically, the problem is characterized as follows: Given a set of integers $S \subseteq [1, \ldots, \mathbf{n}]$, build a data structure to implement the predecessor operation, $\mathrm{pred}(x)$, defined as

$$\mathrm{pred}(x) = \max\{y \in S \mid y \leq x\}, \tag{2.1}$$

for any integer $x \in [1, \ldots, \mathbf{n}]$. Let $s = |S|$. In our case, $s$, which is smaller than twice the number of different implementations, is typically much smaller than $\mathbf{n}$. We will therefore be more interested by algorithms whose resource demands are dependent on $s$, rather than on $\mathbf{n}$.

In an array implementation it is possible to implement $\mathrm{pred}(x)$ using a *binary search* in $O(\log s)$ time, while the space requirement is $O(s)$. The hidden constants are small.

If the number of integers is not so small, then a theoretically superior algorithm is the *Q-fast trie* [Willard 1984], which achieves $O(\sqrt{\log \mathbf{n}})$ time while still maintaining the space linear in $s$. Stratified trees, also called *van Emde Boas data structure* [van Emde Boas et al. 1977; van Emde Boas 1977], offer a different tradeoff, with space linear in $\mathbf{n}$ and time $O(\log\log\mathbf{n})$. In the randomized version of stratified trees the expected space requirement is reduced to $O(s)$. In practice we expect the simple binary search algorithm to outperform these asymptotically better competitors.

## 3. TS DISPATCHING TECHNIQUE

Our dispatching technique for multiple inheritance hierarchies is a generalization of *interval containment* for single inheritance hierarchies. The idea behind interval containment is that there is an ordering of the tree hierarchy in which the descendants of any given type are consecutive. The difficulty in the multiple inheritance case is that an ordering of $\mathcal{T}$ with the above property might not exist. Figure 3.1 shows the smallest hierarchy for which such an ordering is impossible. The reason is that such an ordering imposes the contradicting constraints that A, B and C must be adjacent to D.

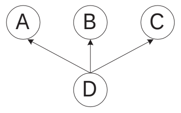


Fig. 3.1. The smallest multiple inheritance hierarchy for which no ordering exists where all descendants of any type are consecutive

Instead of imposing a global ordering, we partition the set of types $\mathcal{T}$ into disjoint *slices* $\mathcal{T}_1, \ldots, \mathcal{T}_\kappa$ and impose a local ordering condition on each of the slices. For a slice $\mathcal{T}_i$ and a type $t$ (not necessarily in $\mathcal{T}_i$), let $D_i(t)$ be the set of descendants of $t$ in $\mathcal{T}_i$, i.e.,

$$D_i(t) = \text{descendants}(t) \cap \mathcal{T}_i\,.$$

Figure 3.2 shows a partitioning of the hierarchy of Figure 1.1 into two slices:

$$\mathcal{T}_1 = \{\mathsf{B}, \mathsf{A}, \mathsf{D}, \mathsf{G}, \mathsf{C}, \mathsf{F}, \mathsf{J}\},$$
$$\mathcal{T}_2 = \{\mathsf{E}, \mathsf{H}, \mathsf{K}\}.$$

The grey entries in any column represent a set of descendants of some type. The sets of descendants of type A, for example, in the two slices are

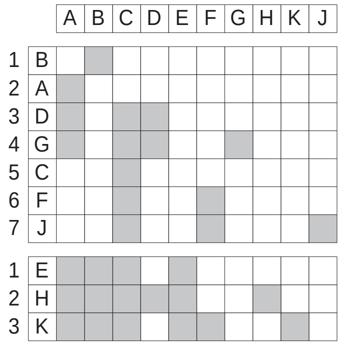
$$\begin{aligned} D_1(\mathsf{A}) &= \{\mathsf{A}, \mathsf{D}, \mathsf{G}\}, \\ D_2(\mathsf{A}) &= \{\mathsf{E}, \mathsf{H}, \mathsf{K}\}. \end{aligned} \tag{3.1}$$

The type slicing technique is based on the demand that the sets $D_i(t)$ are *consecutive* in some ordering of the rows. Visually this means that the grey entries are consecutive within each chunk. For instance, in Figure 3.2 the sets of (3.1) define the *intervals*

$$\begin{aligned} D_1(\mathsf{A}) &= [2, 4], \\ D_2(\mathsf{A}) &= [1, 3]. \end{aligned} \tag{3.2}$$

Formally, each slice $\mathcal{T}_i$ must satisfy the following slicing property:

*There is an ordering of $\mathcal{T}_i$ in which $D_i(t)$ is consecutive for <u>all</u> types $t \in \mathcal{T}$.*



| | | A | B | C | D | E | F | G | H | K | J |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | B | | ▨ | | | | | | | | |
| 2 | A | ▨ | | | | | | | | | |
| 3 | D | ▨ | | ▨ | ▨ | | | | | | |
| 4 | G | ▨ | | ▨ | ▨ | | | ▨ | | | |
| 5 | C | | | ▨ | | | | | | | |
| 6 | F | | | ▨ | | | ▨ | | | | |
| 7 | J | | | ▨ | | | ▨ | | | | ▨ |
| 1 | E | ▨ | ▨ | ▨ | | ▨ | | | | | |
| 2 | H | ▨ | ▨ | ▨ | ▨ | ▨ | | | ▨ | | |
| 3 | K | ▨ | ▨ | ▨ | | ▨ | ▨ | | | ▨ | |

Fig. 3.2. Type slicing for the hierarchy of Figure 1.1

Each type $t$ is identified by a pair $\langle s_t, \mathrm{id}_t\rangle$, where $s_t$ is an id of the slice to which $t$ belongs, and $\mathrm{id}_t$ is the position of $t$ in the ordering of this slice. Thanks to the slicing property, the set $D_i(t)$ defines an *interval* for each $i$, $1 \leq i \leq \kappa$.

DEFINITION 3.1. *The complexity of a hierarchy is the minimal number $\kappa$ such that there exists a partitioning of $\mathcal{T}$ into sets $\mathcal{T}_1, \ldots, \mathcal{T}_\kappa$, such that each $\mathcal{T}_i$ satisfies the slicing property.*

A partitioning of $\mathcal{T}$ into slices which satisfy the slicing property always exists, since this property trivially hold for singletons. We will strive to minimize $\kappa$, the total number of slices.

**Finding the slices** We are unaware of any non-exponential method for finding the minimal number of slices. Instead we use a greedy heuristic: "try to make the current slice as large as possible without violating the slicing property". Specifically, we traverse the types in a topological order, and try to insert each type into each of the slices. If all these insertion attempts fail then a new slice is created.

Given a slice $\mathcal{T}_i$ and a type $t$, PQ-trees [Zibin and Gil 2001; Booth and Leuker 1976] can be used to check whether there is *any* ordering of $\mathcal{T}_i \cup \{t\}$ which satisfies the slicing property, in $O(\mathbf{n} \cdot |\mathcal{T}_i|)$ time. In inserting $\mathbf{n}$ types using this strategy, the total time might be cubic in $\mathbf{n}$, which is highly undesirable.

Instead we use a heuristic which, by not disturbing the existing order of $\mathcal{T}_i$, achieves a run time that depends only on the number of ancestors of $t$. Therefore, the total runtime of the above algorithm for finding the slices is $O(\kappa|\preceq|)$. The exact details of this order preserving heuristic are presented in Appendix A.

**Dispatching using type slicing** Given a type $t$ and a message $m$, a dispatching query returns the smallest type $t' \in I(m)$ such that $t' \succeq t$. Let $\mathcal{T}_i$ be the slice of $t$. Given a type $t'$, we have that $t' \succeq t$ if and only if $t \in D_i(t')$. We therefore must consider all intervals of $D_i(t')$, $D_i(t') \neq \emptyset$, where $t' \in I(m)$. Since there are at most $|I(m)|$ such intervals, we obtain a partition of $\mathcal{T}_i$ into $2|I(m)| + 1$ segments, where the result of the dispatch on $t$ depends only on the segment to which $t$ belongs.

Figure 3.3 shows the dispatching representation for the message $\mathsf{c}$,

$$I(\mathsf{c}) = \{\mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{H}\}$$

in the hierarchy of Figure 1.1. Consider, for example, the first slice. Only types C and D define non empty intervals, which are $[3, 7]$ and $[3, 4]$, respectively. We also consider the implicit interval $[1, 7]$ for the method *message not understood.* Those three intervals partition the types into three segments: $[1, 2]$, $[3, 4]$, and $[5, 7]$. Message c is represented in the first slice using an appropriate data structure storing those three segments, and mapping them to: null (*message not understood*), D, and C, respectively.

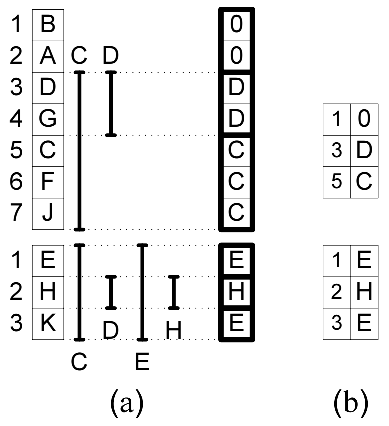


Fig. 3.3. (a) The intervals and segments of message c in the two slices of Figure 3.2, and (b) the message representation in each slice

In general, a message $m$ is encoded in slice $\mathcal{T}_i$ by a data structure of choice which represents a set of segments, mapping each one to the appropriate method implementation. As in vanilla interval containment, this data structure can be a simple array, a Q-fast trie, or a stratified tree. Obviously, each slice has its own unique such data structure.

Dispatching on type $t \in \mathcal{T}$ and message $m \in \mathcal{M}$ is carried out in three stages:

(1) Finding $s_t$, the id of the slice of $t$,

(2) following this slice to find the respective data structure of $m$, and then

(3) carrying on as in single inheritance in a search of $\mathrm{id}_t$ in this data structure to find the dispatching result.

Thus, dispatching in multiple inheritance hierarchies requires only two more steps in comparison to dispatching in single inheritance hierarchies. The space requirement in multiple inheritance hierarchies increases by a factor of at most $\kappa$. Curiously, this factor depends only on the topology of the hierarchy and the quality of the slicing algorithm. It does not depend in any way on the number of messages.

**Reducing the number of slices** We now describe one optimization that given the set of messages reduces the number of slices $\kappa$. In our multiple inheritance benchmarks, $\kappa$ is reduced by an average of 1.35. (In the LOV hierarchy, for example, the number of slices is reduced from 12 to 7.) The key observation is that the dispatching algorithm assumes that each implementor $t \in I(m)$ defined an interval for each slice. Therefore, $D_i(t)$ must be consecutive in $\mathcal{T}_i$, *only* for those types $t$ which are indeed implementors of some message $m$.

Formally, we say that a type $t$ is *significant* if there exists a message $m$ such that $t \in I(m)$, and redefine the slicing property as follows:

> *There is an ordering of $\mathcal{T}_i$ in which $D_i(t)$ is consecutive for all $\underline{\text{significant}}$ types $t \in \mathcal{T}$.*

**Optimizations for statically typed languages** We also note that in *statically typed languages*, the *binary search* algorithm can be optimized. Suppose that we dispatch on an object whose *static type* is $a$. Then, at runtime, the binary search can begin at a smaller interval, restricted only to the interval of descendants of $a$ in each of the slices.

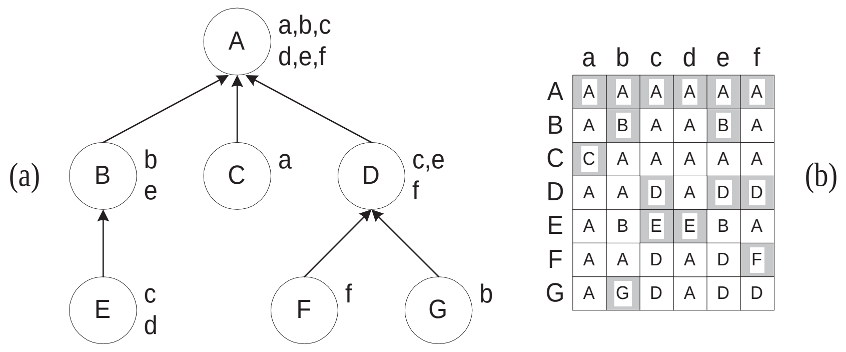
Moreover, we can even discard segments which correspond to *message not understood*, since such a case does not occur in statically typed languages.

## 4. CT DISPATCHING TECHNIQUE FOR SINGLE INHERITANCE HIERARCHIES

For simplicity, assume w.l.o.g. that the hierarchy is a tree (rather than a forest) rooted at a special node $\top \in \mathcal{T}$. There cannot be a *message ambiguous* in a single inheritance setting. To avoid the other error situation, namely *message not understood*, we assume that $\top \in I(m)$ for all $m \in \mathcal{M}$. With this assumption, every dispatching query returns a single implementor. The cost is in (at most) doubling the number of implementations $\mathbf{i}$, since we add at most $\mathbf{m} \leq \mathbf{i}$ methods to the root $\top$. (At the end of this section we will show that the memory toll can be made much smaller.)

Vitek and Horspool's CT algorithm [Vitek and Horspool 1996] partitions the messages $\mathcal{M}$ into $\mathbf{k}$ disjoint *slices* $\mathcal{M} = \mathcal{M}_1 \cup \ldots \cup \mathcal{M}_{\mathbf{k}}$. These slices break the dispatching matrix into $\mathbf{k}$ sub-matrices, also called *chunks*. The authors' experience was that chunks with 14 columns each give best results, and this number 14 was hard-coded into their algorithm.

For example, consider the single inheritance hierarchy in Figure 4.1a.



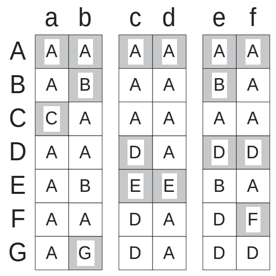
| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| A | A | A | A | A | A | A |
| B | A | B | A | A | B | A |
| C | C | A | A | A | A | A |
| D | A | A | D | A | D | D |
| E | A | B | E | E | B | A |
| F | A | A | D | A | D | F |
| G | A | G | D | A | D | D |

Fig. 4.1. (a) A small example of a single inheritance hierarchy, and (b) its dispatching matrix

Figure 4.2 shows the three chunks of the dispatching matrix of Figure 4.1b for following partitioning:

$$\begin{aligned} \mathcal{M}_1 &= \{\mathsf{a}, \mathsf{b}\}, \\ \mathcal{M}_2 &= \{\mathsf{c}, \mathsf{d}\}, \\ \mathcal{M}_3 &= \{\mathsf{e}, \mathsf{f}\}. \end{aligned} \tag{4.1}$$

As Vitek and Horspool observed, and as can be seen in the figure, there are many identical rows in each chunk. Significant compression can be achieved by merging these rows together, and introducing, in each chunk, an auxiliary array of pointers to map each type to a row specimen.



| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| A | A | A | A | A | A | A |
| B | A | B | A | A | B | A |
| C | C | A | A | A | A | A |
| D | A | A | D | A | D | D |
| E | A | B | E | E | B | A |
| F | A | A | D | A | D | F |
| G | A | G | D | A | D | D |

Fig. 4.2. Three chunks of the dispatching matrix of Figure 4.1b

Why should there be many duplicate rows in each chunk? There are two contributing factors: (i) since the slices are small, there are not too many columns in a chunk, and (ii) that the number of distinct values which can occur in any given column is small, since, as empirical data shows, the number of different implementations of a selector is a small constant. Hence, there could not be too many distinct rows.

However, these considerations apply to any random distribution of values in the dispatching matrix. The crucial observation we make is that a much stronger bound on the number of distinct rows can be set relying on the fact that the values in the dispatching matrix are not arbitrary; they are generated from an underlying structured hierarchy.

Consider for example a chunk with two columns, with $n_1$ and $n_2$ distinct implementations in these columns. Simple counting considerations show that the number of distinct rows is at most $n_1 n_2$. Relying on the fact that the hierarchy is a tree we can show that the number of distinct rows is at most $n_1 + n_2$.

To demonstrate this observation, consider Figure 4.3a which focuses on the first chunk, corresponding to slice $\mathcal{M}_1 = \{\mathsf{a}, \mathsf{b}\}$.

As can be seen in the figure, the rows of types A, D, and F are identical. Figure 4.3b shows the compressed chunk and the auxiliary array. We see that this auxiliary array maps types A, D, and F to the same row.

We call attention to the (perhaps surprising) fact that it is possible to select from the elements of each row in Figure 4.3b a distinguishing representative. These

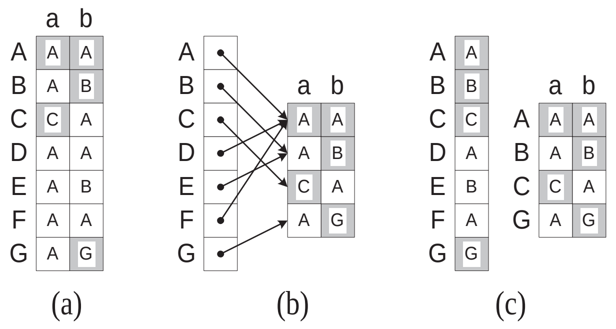


Fig. 4.3. (a) The first chunk of Figure 4.1c, (b) the chunk compressed using an auxiliary array of pointers, and (c) the chunk compressed using an array of labels

representatives are members of what we call the *master-message* $\mathcal{M}_1$,

$$I(\mathcal{M}_1) = I(\mathsf{a}) \cup I(\mathsf{b}) = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{G}\}.$$

(We use the same notation for a slice and its master-message. No confusion will arise since the context will determine if we treat $\mathcal{M}_1$ as a slice or as a master-message.)

The representatives of the four rows in the first chunk are A, B, C and G, in this order. The figure highlights these in grey. Also note that each implementor of the master-message serves as a representative of some row.

Figure 4.3c gives an alternative representation of the chunk, where each row is labeled by its representative. The auxiliary array now contains these labels instead of pointers. For example, the second row is labeled $\mathsf{B} \in I(\mathsf{b})$; the second and the fifth entry of the auxiliary array store B rather than the row specimen address.

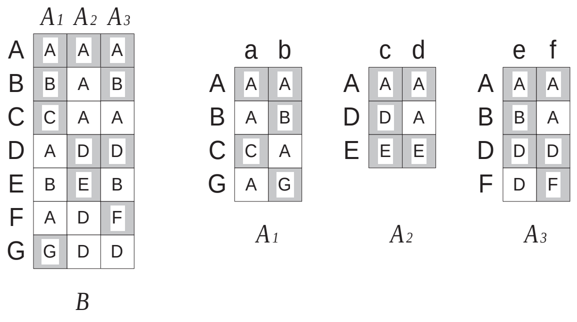
Our improvement is based on the observation that the distinguishing representatives phenomenon is not a coincidence and on the observation that CT applies a *divide-and-conquer* approach to the dispatching problem: The search first determines the relevant master-message, and then continues to select the appropriate result among its implementors.

Let $A_i$ denote the compressed $i^{th}$ chunk of the dispatching matrix, and let $B$ be the master dispatching matrix, whose columns are the auxiliary arrays of the chunks. Figure 4.4 shows matrices $A_1, A_2, A_3$ and $B$, which constitute the complete CT representation for the hierarchy of Figure 4.1. Note that the first column of $B$ is the auxiliary array depicted in Figure 4.3c.

For each slice $\mathcal{M}_i$, its *master-message* is the union of implementors of its messages, i.e., $I(\mathcal{M}_i) \equiv \bigcup_{m \in \mathcal{M}_i} I(m)$. Then, answering the query $\mathsf{dispatch}(m, t)$ at runtime requires three steps:

(1) *Determine the slice of* $m$. That is, the slice $\mathcal{M}_s$, such that $m \in \mathcal{M}_s$. If the partitioning into slices and the message $m$ are known at compile-time, as it is usually the case in dispatching of static-loading languages, then this stage incurs no penalty at runtime.

(2) *Fetch the first dispatching result* $t' = \mathsf{dispatch}(\mathcal{M}_s, t)$. This value is found at the row which corresponds to type $t$ and the column which corresponds to the master-message $\mathcal{M}_s$, i.e., $t' = B[t, s]$.



|   | $A_1$ | $A_2$ | $A_3$ |
|---|---|---|---|
| A | A | A | A |
| B | B | A | B |
| C | C | A | A |
| D | A | D | D |
| E | B | E | B |
| F | A | D | F |
| G | G | D | D |

$B$

|   | a | b |
|---|---|---|
| A | A | A |
| B | A | B |
| C | C | A |
| G | A | G |

$A_1$

|   | c | d |
|---|---|---|
| A | A | A |
| D | D | A |
| E | E | E |

$A_2$

|   | e | f |
|---|---|---|
| A | A | A |
| B | B | A |
| D | D | D |
| F | D | F |

$A_3$

Fig. 4.4. CT representation for the hierarchy of Figure 4.1

(3) *Fetch the final dispatching result* $t'' = \mathsf{dispatch}(m, t)$. This type is found in the row of $t'$ and the column of $m$ in the compressed chunk $A_s$, i.e., $t" = A_s[t', m]$.

The algorithm merges together all the different messages in $\mathcal{M}_s$. At step 2, we find $t' \succeq t$, which is the smallest candidate in the merged master-message. Matrix $B$ (of size $\mathbf{n} \times \mathbf{k}$) is the dispatching matrix of the types $\mathcal{T}$ and the master-messages $\{\mathcal{M}_1, \ldots, \mathcal{M}_\mathbf{k}\}$.

The search then continues with $t'$, to find $t" \succeq t'$, the smallest candidate in $I(m)$. Each matrix $A_i$ (of size $|I(\mathcal{M}_i)| \times |\mathcal{M}_i|$) is the dispatching matrix of the types in $I(\mathcal{M}_i)$ and the messages $\mathcal{M}_i$.

To understand the space saving, consider just two messages $\mathcal{M} = \{m_1, m_2\}$. The naive implementation of dispatch is using *two* arrays, each of size $\mathbf{n} = |\mathcal{T}|$, which map each type $t$ to two types $t"_1 \in I(m_1)$ and $t_2" \in I(m_2)$, such that $t_i" = \mathsf{dispatch}(m_i, t)$, $i = 1, 2$. A more compact representation can be obtained by using a *single* array of size $\mathbf{n}$, to dispatch first on the merged master-message $\mathcal{M}$, $I(\mathcal{M}) = I(m_1) \cup I(m_2)$. Let $t' \in I(\mathcal{M})$ be the result of this dispatch. The crucial point is that the smallest candidate for $t'$, in either $I(m_1)$ or $I(m_2)$, is the same as for $t$. Since there are $|I(\mathcal{M})| \leq |I(m_1)| + |I(m_2)|$ different values of $t'$, a continued search from $t'$ (for either $I(m_1)$ or $I(m_2)$) can be implemented using two arrays, each of size $|I(\mathcal{M})|$. The first such array maps $I(\mathcal{M})$ to $I(m_1)$; the second to $I(m_2)$. Total memory used is $\mathbf{n} + 2|I(\mathcal{M})|$ instead of $2\mathbf{n}$ cells, while the cost is an additional dereferencing operation.

More generally, given a dispatching problem for the messages $\mathcal{M}$, the *CT reduction* partitions $\mathcal{M}$ into $\mathbf{k}$ disjoint slices

$$\mathcal{M} = \mathcal{M}_1 \cup \ldots \cup \mathcal{M}_\mathbf{k}, \tag{4.2}$$

and merges together the messages in each slice by defining a master-message

$$I(\mathcal{M}_i) \equiv \bigcup_{m \in \mathcal{M}_i} I(m), \tag{4.3}$$

for all $i = 1, \ldots, \mathbf{k}$. Let $A_i$ be the matrix whose dimensions are

$$|I(\mathcal{M}_i)| \times |\mathcal{M}_i|, \tag{4.4}$$

corresponding to the $i^{th}$ slice. Then, the query $\mathsf{dispatch}(m, t)$ is realized by the fetch

$$A_s[\mathsf{dispatch}(\mathcal{M}_s, t), m], \tag{4.5}$$

where $m \in \mathcal{M}_s$.

Since both steps 2 and 3 in the dispatching are in essence a dispatching operation, better compaction of the dispatching data structure might be achieved by applying the CT technique recursively to either the matrix $B$, or all the matrices $A_i$. It is not difficult to see that each of the recursive applications will yield the same dispatching data structure, in which the set of selectors is organized in a three-level hierarchy of partitions: messages, master-messages, and master-master-messages (so to speak). We chose to describe this 3-level system by applying the CT technique to the matrix $B$. The (potential) saving in space comes at a cost of another dereferencing step during dispatch. Clearly, we could recursively apply the reduction any number of times.

We need the following notation in order to optimize these recursive applications, i.e., find the optimal number of slices $\mathbf{k}$, and the size of each slice. Let $\text{mem}_d(\mathbf{n}, \mathbf{m}, \mathbf{i})$ denote the memory required for solving the dispatching problem of $\mathbf{n}$ types, $\mathbf{m}$ messages and $\mathbf{i}$ method implementations, using $d$ dereferencing operations during dispatch. A simple dispatching matrix representation gives

$$\text{mem}_1(\mathbf{n}, \mathbf{m}, \mathbf{i}) = \mathbf{nm}. \tag{4.6}$$

Each application of the CT reduction adds another dereferencing, while reducing a dispatching problem with parameters $\langle \mathbf{n}, \mathbf{m}, \mathbf{i} \rangle$ to a new dispatching problem with parameters $\langle \mathbf{n}, \mathbf{k}, \mathbf{i}' \rangle$, where

$$\mathbf{i}' = \sum_{i=1}^{\mathbf{k}} |I(\mathcal{M}_i)| = \sum_{i=1}^{\mathbf{k}} \left| \bigcup_{m \in \mathcal{M}_i} I(m) \right|.$$

Note that $\mathbf{i}' \leq \mathbf{i}$. To see this recall that

$$\mathbf{i} = \sum_{m \in \mathcal{M}} |I(m)| = \sum_{i=1}^{\mathbf{k}} \sum_{m \in \mathcal{M}_i} |I(m)|,$$

and apply the fact that the cardinality of the union of sets is at most the sum of cardinalities of these sets

$$\mathbf{i}' = \sum_{i=1}^{\mathbf{k}} \left| \bigcup_{m \in \mathcal{M}_i} I(m) \right| \leq \sum_{i=1}^{\mathbf{k}} \sum_{m \in \mathcal{M}_i} |I(m)| = \mathbf{i}. \tag{4.7}$$

The reduction generates the matrices $A_1, \ldots, A_{\mathbf{k}}$. To estimate their size suppose that all slices are equal in size, i.e., they all have $x$ messages. (For simplicity we ignore the case that $\mathbf{m}$ is not divisible by $x$, in which slices are *almost* equal.) Then, the total memory generated by the reduction is

$$\sum_{i=1}^{\mathbf{k}} |I(\mathcal{M}_i)| \times |\mathcal{M}_i| = \sum_{i=1}^{\mathbf{k}} |I(\mathcal{M}_i)| \times x = x \sum_{i=1}^{\mathbf{k}} |I(\mathcal{M}_i)| = x\mathbf{i}' \leq x\mathbf{i}.$$

To conclude, the costs of the CT reduction are another dereferencing and an additional space of $x\mathbf{i}$. In return, a dispatching problem with parameters $\langle \mathbf{n}, \mathbf{m}, \mathbf{i} \rangle$ is reduced to a new dispatching problem with parameters $\langle \mathbf{n}, \mathbf{k}, \mathbf{i}' \rangle$, where $\mathbf{k} = \mathbf{m}/x$ and $\mathbf{i}' \leq \mathbf{i}$. Formally,

$$\mathrm{mem}_{d+1}(\mathbf{n}, \mathbf{m}, \mathbf{i}) \leq \mathbf{i}x + \mathrm{mem}_d(\mathbf{n}, \mathbf{m}/x, \mathbf{i}), \tag{4.8}$$

where $x$ is arbitrary.

Let $\mathrm{CT}_d$ be the algorithm obtained by applying the CT reduction $d-1$ times to the original dispatching problem. The recursion is ended by applying simple dispatching matrix at the last step. Thus, $\mathrm{CT}_1$ is simply the dispatching matrix, while $\mathrm{CT}_2$ is similar to Vitek and Horspool's algorithm (with $x = 14$). By making $d-1$ substitutions of (4.8) into itself, and then using (4.6), we obtain

$$\mathrm{mem}_d(\mathbf{n}, \mathbf{m}, \mathbf{i}) \leq \mathbf{i}x_1 + \cdots + \mathbf{i}x_{d-1} + \frac{\mathbf{nm}}{x_1 x_2 \cdots x_{d-1}}, \tag{4.9}$$

where $x_i$ is the slice size used during the $i^{th}$ application of the CT reduction. Symmetry considerations indicate that the bound in (4.9) is minimized when all $x_i$ are equal. We have,

$$\mathrm{mem}_d(\mathbf{n}, \mathbf{m}, \mathbf{i}) \leq (d-1)\mathbf{i}x + \frac{\mathbf{nm}}{x^{d-1}}, \tag{4.10}$$

which is minimized when $x = (\mathbf{nm}/\mathbf{i})^{1/d}$.

Table I summarizes the space and time requirements of algorithms $\mathrm{CT}_d$, where $\iota \equiv (\mathbf{nm})/\mathbf{i}$ is the optimal compression factor.

| Scheme | Slice size | Time | Space | Compression factor |
|---|---|---|---|---|
| $\mathrm{CT}_1$ | N/A | 1 | $\mathbf{i}\iota$ | 1 |
| $\mathrm{CT}_2$ | $\sqrt[2]{\iota}$ | 2 | $2\mathbf{i}\sqrt[2]{\iota}$ | $\frac{\iota}{2\sqrt[2]{\iota}}$ |
| $\mathrm{CT}_3$ | $\sqrt[3]{\iota}$ | 3 | $3\mathbf{i}\sqrt[3]{\iota}$ | $\frac{\iota}{3\sqrt[3]{\iota}}$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $\mathrm{CT}_d$ | $\sqrt[d]{\iota}$ | $d$ | $d\mathbf{i}\sqrt[d]{\iota}$ | $\frac{\iota}{d\sqrt[d]{\iota}}$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $\mathrm{CT}_{\log_x \mathbf{m}}$ | $x$ | $\log_x \mathbf{m}$ | $(\log_x \mathbf{m})\mathbf{i}x$ | $\frac{\iota}{x \log_x \mathbf{m}}$ |

Table I. Generalized CT results for single inheritance hierarchies

The last row in the table is obtained by applying the CT reduction a maximal number of times. In each application the slice size is $x$ (typically, $x = 2$). The messages $\mathcal{M}$ are then organized in a hierarchy of $\log_x \mathbf{m}$ levels, which is also the number of dereferencing steps during dispatch. The memory used in each level is $\mathbf{i}x$ (see (4.8)).

The generalizations (Table I) of $\mathrm{CT}_d$ over Vitek and Horspool's algorithm is in the following directions: (i) a sequence of algorithms which offer a tradeoff between the size of the representation and the dispatching time, and (ii) precise performance analysis, which dictates an optimal slice size, instead of the arbitrary universal recommendation, $x = 14$.

In reflecting on the generalized CT algorithm we see that they are readily adapted to the case where *message not understood* are allowed as is the case in dynamically typed languages. Whenever the search in a master-message $\mathcal{M}$ returns $\top$, we can be certain that the search in every constituent of $\mathcal{M}$ will also return $\top$. Therefore, it is possible to check after each dereferencing operation whether the fetched type is $\top$, and emit the appropriate error message. A more appealing alternative is to continue the search with $\top$, using an array which maps $\top$ into itself for each constituent of $I(\mathcal{M})$. Now, since this array does not depend on the identity of $I(\mathcal{M})$, we can store only one such copy for each application of the CT reduction. The memory toll that $\mathrm{CT}_d$ bears for these arrays is $(d-1)x$ cells.

Note also that Vitek and Horspool's idea of using selector coloring [Dixon et al. 1989; Pascal and Royer 1992] in each chunk is still applicable. If certain columns in a chunk contain many $\top$ elements, it might be possible to collapse these columns together.

## 5. INCREMENTAL CT DISPATCHING TECHNIQUE FOR JAVA

This section describes an incremental variant of the CT scheme for the single inheritance, dynamically typed, and dynamic loading, model. A prime example for the model is the SMALLTALK programming language. Curiously, even though JAVA is a statically typed language, the implementation of the `invokeinterface` bytecode instruction (used for dispatching messages sent to an **`interface`**) matches this model. To see this, recall that all implementations of a method defined in an **`interface`** must reside in **`class`**es, and that these classes take a tree topology. The locations of these implementations in this tree are however totally unrelated, and additional implementations can be introduced as a result of dynamic class loading. By simply ignoring the interfaces, we can view dispatching in JAVA as in a single inheritance and dynamically typed language. Note that the VFT method is unsuitable for `invokeinterface` instructions.

The incremental variant of the CT scheme achieves two important properties: (i) the *space* it uses is at most twice that of the static algorithm, and (ii) its total *runtime* is linear in the final encoding size. (We cannot expect an asymptotically better runtime since the algorithm must at least output the final encoding.) Section 5.1 describes $\mathrm{ICT}_2$, the incremental variant of $\mathrm{CT}_2$. Section 5.2 gives the generalization for $\mathrm{CT}_d$.

The main idea is to *rebuild the entire encoding* whenever the ratio between the current slice size and the optimal one reaches a high- or low-water mark (for example 2 and 1/2). Therefore, some insertions will take longer to process than others. We therefore obtain bounds on the *amortized* time for an insertion.[6] The amortized time of an insertion is asymptotically optimal since the total runtime is linear in the final encoding size. Using techniques of "background copying" [Dietzfelbinger et al. 1994], it is possible to amend the algorithms so that the *worst case* insertion time is optimal as well.

Note that unlike the static version of the problem, we cannot assume that the implementors always include the root $\top$. The reason is that this assumption would

[6]We remind the reader that the *amortized time* of an operation is $c(n)$, if a sequence of $n$ such operations requires at most $nc(n)$ time. The worst case time of any single operation can however be much greater than $c(n)$. For more information on amortized complexity see [Sleator and Tarjan 1985].

require $\top$ to include implementation of *all* messages, and the initial value of the number of messages will jump to $\mathbf{m}$.

## 5.1 Algorithm ICT$_2$ in a Single Inheritance Setting

The CT$_2$ scheme applies a single CT reduction and uses a dispatching matrix for the resulting master-messages. This process divides the dispatching problem into independent sub-problems: one dispatching matrix, and a set of matrices $A_i$, $i = 1, \ldots, \mathbf{k}$, which (in a single inheritance setting) are in fact dispatching matrices as well.

We first describe how to maintain a plain, single-level, dispatching matrix subject to type insertions. The insertion time will be linear in the encoding size, and the cost in dispatching time is in an additional comparison to guard against array overflows.

Each message is assigned a unique identifier in increasing order. The mapping of message-to-identifier is maintained as a hash-table. Consider a newly added type $t$. The newly introduced messages are assigned new identifiers and inserted into the hash-table. Observe that the dispatching result for such a newly introduced message and every other type is always null. However, instead of extending all the other rows with null entries, we perform a range-check before accessing any given row. In the case of array-overflow we return null, otherwise we proceed as usual.

The row of $t$ in the dispatching matrix maps each message to its dispatching result. More precisely, the row of $t$ is an extension of the row of its parent, except for entries corresponding to messages in which $t$ is a member. Note that the insertion time of a type is linear in its row size, and the total runtime is therefore linear in the final encoding size.

The space requirement of CT$_2$ in a single inheritance setting is (see Table I)

$$\mathrm{mem}(x) = \mathbf{i}x + \mathbf{nm}/x, \tag{5.1}$$

which is minimized when the slice size is

$$x_{\mathrm{OPT}} = \sqrt{\mathbf{nm}/\mathbf{i}}. \tag{5.2}$$

Algorithm ICT$_2$ will maintain the following invariant

$$\boxed{\frac{x_{\mathrm{OPT}}}{2} \leq x \leq 2x_{\mathrm{OPT}},} \tag{5.3}$$

and will rebuild the encoding whenever this condition is violated. Algorithm 1 shows the procedure to apply whenever a new type is added to the hierarchy.

Substituting (5.2) in (5.1) we find the optimal encoding size

$$\mathrm{mem}(x_{\mathrm{OPT}}) = 2\sqrt{\mathbf{nmi}}.$$

Let us write this as a function of the problem parameters,

$$f(\mathbf{n}, \mathbf{m}, \mathbf{i}) \equiv \mathrm{mem}(x_{\mathrm{OPT}}) = 2\sqrt{\mathbf{nmi}}.$$

and study the properties of this function.

FACT 5.1. *Function $f$ is monotonic in all three arguments* $\mathbf{n}, \mathbf{m}, \mathbf{i}$.

**Algorithm 1** Insertion of a new type $t$ in ICT$_2$

```
1: Let $x$ be the current slice size.
2: Let $\langle \mathbf{n}, \mathbf{m}, \mathbf{i} \rangle$ be the current problem parameters.
3: $x_{\text{OPT}} \leftarrow \sqrt{\mathbf{nm}/\mathbf{i}}$        // The optimal slice size.
4: If not $\left(\frac{x_{\text{OPT}}}{2} \le x \le 2x_{\text{OPT}}\right)$ then
5:     $x \leftarrow x_{\text{OPT}}$
6:     Rebuild the entire CT$_2$ encoding
7: fi
8: Insert $t$ to the CT$_2$ encoding
```

FACT 5.2. *There are constants $c_1, c_2, c_3$, such that*

$$\begin{aligned}\sum_{i=0}^{\infty} f\left(\frac{\mathbf{n}}{2^i}, \mathbf{m}, \mathbf{i}\right) &\le c_1 f(\mathbf{n}, \mathbf{m}, \mathbf{i}),\\ \sum_{i=0}^{\infty} f\left(\mathbf{n}, \frac{\mathbf{m}}{2^i}, \mathbf{i}\right) &\le c_2 f(\mathbf{n}, \mathbf{m}, \mathbf{i}),\\ \sum_{i=0}^{\infty} f\left(\mathbf{n}, \mathbf{m}, \frac{\mathbf{i}}{2^i}\right) &\le c_3 f(\mathbf{n}, \mathbf{m}, \mathbf{i}).\end{aligned} \tag{5.4}$$

PROOF. Note that

$$\begin{aligned}\sum_{i=0}^{\infty} f\left(\frac{\mathbf{n}}{2^i}, \mathbf{m}, \mathbf{i}\right) &= \sum_{i=0}^{\infty} \sqrt{\frac{\mathbf{n}}{2^i}\mathbf{mi}}\\ &= \sqrt{\mathbf{nmi}} \sum_{i=0}^{\infty} \sqrt{\frac{1}{2^i}}\\ &\le \frac{2}{2-\sqrt{2}}\sqrt{\mathbf{nmi}} \in O(f(\mathbf{n}, \mathbf{m}, \mathbf{i})).\end{aligned}$$

The proof for parameters $\mathbf{m}$ and $\mathbf{i}$ is identical. □

LEMMA 5.3. *The* space requirement *of* ICT$_2$ *is at most*

$2f(\mathbf{n}, \mathbf{m}, \mathbf{i})$.

PROOF. From the algorithm invariant (5.3) it follows that

$$\begin{aligned}\text{mem}(x) &= \mathbf{i}x + \mathbf{nm}/x\\ &\le \mathbf{i}(2x_{\text{OPT}}) + \mathbf{nm}/\left(\frac{x_{\text{OPT}}}{2}\right)\\ &= 2(\mathbf{i}x_{\text{OPT}} + \mathbf{nm}/x_{\text{OPT}})\\ &= 2\,\text{mem}(x_{\text{OPT}}) = 2f(\mathbf{n}, \mathbf{m}, \mathbf{i}).\end{aligned}$$ □

Our next objective is to prove that the total *runtime* of ICT$_2$ is linear in $f(\mathbf{n}, \mathbf{m}, \mathbf{i})$. To do so, we will breakdown the sequence of insertions carried out by the algorithm into *phases*, according to the points in time where rebuilding took place. No rebuilding occurs within a phase, and all that is required is to maintain several plain dispatching matrices. Hence, the total runtime of the insertions in a phase is linear in the encoding size at the end of this phase.

The main observation is that rebuilding happens only when at least one of the problem parameters is doubled. We distinguish between three *kinds* of rebuilds, depending on the parameter which was doubled. We then show that the total runtime of rebuilds of the same kind is linear in $f(\mathbf{n}, \mathbf{m}, \mathbf{i})$.

Formally, phase $i$ begins immediately after phase $i-1$, and ends after the encoding was built for the $i^{th}$ time (the last phase ends when the program terminates). Let $\langle \mathbf{n}_i, \mathbf{m}_i, \mathbf{i}_i \rangle$, $i = 1, \ldots, p$, be the problem parameters at the end of phase $i$. Observe that the problem parameters can only increase, i.e., $\mathbf{n}_{i+1} \geq \mathbf{n}_i$, $\mathbf{m}_{i+1} \geq \mathbf{m}_i$, and $\mathbf{i}_{i+1} \geq \mathbf{i}_i$. Phase $i$ finishes with an encoding size of at most $2f(\mathbf{n}_i, \mathbf{m}_i, \mathbf{i}_i)$, therefore its runtime is linear in $f(\mathbf{n}_i, \mathbf{m}_i, \mathbf{i}_i)$. Thus, the total runtime is linear in

$$\sum_{i=1}^{p} f(\mathbf{n}_i, \mathbf{m}_i, \mathbf{i}_i). \tag{5.5}$$

We need to show that this sum is linear in $f(\mathbf{n}_p, \mathbf{m}_p, \mathbf{i}_p)$.

Lemma 5.4. *Invariant* (5.3) *is violated only when* at least one *of the problem parameters is doubled, i.e., one of the following holds*

$$\begin{aligned} \mathbf{n}_{i+1} &\geq 2\mathbf{n}_i, \\ \mathbf{m}_{i+1} &\geq 2\mathbf{m}_i, \\ \mathbf{i}_{i+1} &\geq 2\mathbf{i}_i. \end{aligned} \tag{5.6}$$

Proof. Let $x_j$ denote the slice size at the beginning of phase $j$, i.e.,

$$x_j \equiv \sqrt{\frac{\mathbf{n}_j \mathbf{m}_j}{\mathbf{i}_j}}. \tag{5.7}$$

At the end of phase $i$ one of the following conditions must hold

$$\begin{aligned} x_{i+1} &\geq 2x_i, \text{ or} \\ x_{i+1} &\leq \frac{1}{2}x_i. \end{aligned} \tag{5.8}$$

From (5.7) and (5.8), we have

$$\begin{aligned} \frac{\mathbf{n}_{i+1}\mathbf{m}_{i+1}}{\mathbf{i}_{i+1}} &\geq \frac{4\mathbf{n}_i\mathbf{m}_i}{\mathbf{i}_i}, \text{ or} \\ \frac{\mathbf{n}_{i+1}\mathbf{m}_{i+1}}{\mathbf{i}_{i+1}} &\leq \frac{\mathbf{n}_i\mathbf{m}_i}{4\mathbf{i}_i}. \end{aligned} \tag{5.9}$$

Since the problem parameters can only increase,

$$\begin{aligned} \mathbf{n}_{i+1}\mathbf{m}_{i+1} &\geq 4\mathbf{n}_i\mathbf{m}_i, \text{ or} \\ \mathbf{i}_{i+1} &\geq 4\mathbf{i}_i, \end{aligned} \tag{5.10}$$

which implies that at least one of the parameters was doubled. □

Lemma 5.5. *The total* runtime *of* ICT$_2$ *is linear in*

$$f(\mathbf{n}_p, \mathbf{m}_p, \mathbf{i}_p).$$

Proof. Let $\{(N_1, M_1, I_1), \ldots, (N_q, M_q, I_q)\}$ be the problem parameters of phases where $\mathbf{n}$ was doubled, i.e., $N_{i+1} \geq 2N_i$. Therefore,

$$N_q \geq 2N_{q-1} \geq \ldots \geq 2^{q-1}N_1. \tag{5.11}$$

Using Fac. 5.2, the total runtime of these phases is linear in

$$\begin{aligned}\sum_{i=1}^{q} f(N_i, M_i, I_i) &\le \sum_{i=1}^{q} f(N_i, M_q, I_q) \\ &\le \sum_{i=1}^{q} f\left(\frac{N_q}{2^{q-j}}, M_q, I_q\right) \\ &\in O(f(N_q, M_q, I_q)).\end{aligned} \tag{5.12}$$

The same consideration applies to phases in which the number of methods or the number of messages was doubled. So, the runtime of the entire algorithm is the total runtime of the three kinds of phases, which is linear in $f(\mathbf{n}_p, \mathbf{m}_p, \mathbf{i}_p)$. □

## 5.2 Algorithm $\mathrm{ICT}_d$ in a Single Inheritance Setting

The generalization to $d > 2$ is mostly technical, as outlined next. Function $\mathrm{mem}(x)$, the space requirement of $\mathrm{CT}_d$ as defined in (4.10) is minimized when the slice size is

$$x_{\mathrm{OPT}} = \sqrt[d]{\mathbf{nm}/\mathbf{i}}.$$

Let function $f_d$ denote the optimal encoding size

$$f_d(\mathbf{n}, \mathbf{m}, \mathbf{i}) \equiv \mathrm{mem}(x_{\mathrm{OPT}}) = d\mathbf{i}\sqrt[d]{\iota}.$$

Algorithm $\mathrm{ICT}_d$ will preserve the following invariant

$$\boxed{\frac{x_{\mathrm{OPT}}}{2^{1/(d-1)}} \le x \le 2x_{\mathrm{OPT}}.} \tag{5.13}$$

LEMMA 5.6. *The space requirement of* $\mathrm{ICT}_d$ *is at most* $2f_d(\mathbf{n}, \mathbf{m}, \mathbf{i})$.

PROOF. Similar to that of Lemma 5.3 □

FACT 5.7. *There are constants* $c_1, c_2, c_3$*, such that*

$$\begin{aligned}\sum_{i=0}^{\infty} f_d\left(\frac{\mathbf{n}}{2^i}, \mathbf{m}, \mathbf{i}\right) &\le c_1 f_d(\mathbf{n}, \mathbf{m}, \mathbf{i}), \\ \sum_{i=0}^{\infty} f_d\left(\mathbf{n}, \frac{\mathbf{m}}{2^i}, \mathbf{i}\right) &\le c_2 f_d(\mathbf{n}, \mathbf{m}, \mathbf{i}), \\ \sum_{i=0}^{\infty} f_d\left(\mathbf{n}, \mathbf{m}, \frac{\mathbf{i}}{2^i}\right) &\le c_3 f_d(\mathbf{n}, \mathbf{m}, \mathbf{i}).\end{aligned} \tag{5.14}$$

LEMMA 5.8. *Rebuilding only takes place when* at least one *of the problem parameters is doubled.*

PROOF. Similar to that of Lemma 5.4 □

LEMMA 5.9. *The total runtime of* $\mathrm{ICT}_d$ *is linear in* $f_d(\mathbf{n}_p, \mathbf{m}_p, \mathbf{i}_p)$.

PROOF. Similar to Lemma 5.5. □

## 6. CT DISPATCHING TECHNIQUE FOR MULTIPLE INHERITANCE HIERARCHIES

This section explains how to generalize the CT reduction as described in Section 4 to the multiple inheritance setting. In a single inheritance hierarchy, there could never be more than one most specific implementor in response to a dispatch query. The fact that this is no longer true in multiple inheritance hierarchies makes it difficult to apply the CT reduction to such hierarchies. Even if the original messages are appropriately augmented to remove all such ambiguities, ambiguities may still occur in the master-messages as they are generated by the reduction.

We will therefore use a novel notion of a *generalized dispatching query*, denoted by $\mathsf{g\text{-}dispatch}(m,t)$, which returns *the entire set* of smallest candidates, rather than null in case that this set is not a singleton. Formally,

$$\mathsf{g\text{-}dispatch}(m,t) \equiv \min(\mathsf{cand}(m,t)).$$

$$\mathsf{dispatch}(m,t) \equiv \begin{cases} t' & \text{if } \mathsf{g\text{-}dispatch}(m,t) = \{t'\}, \\ \text{null} & \text{otherwise.} \end{cases} \tag{6.1}$$

Generalized dispatching is a data-structure transaction rather than an actual run-time operation which must result in a single method to execute. Generalized dispatching is more informative than a regular dispatching query, because instead of returning null in error cases, it returns either an empty set (for *message not understood*) or the entire set of smallest candidates (for *message ambiguous*). When a generalized dispatching query returns a singleton, then the dispatching result is that singleton element.

Consider for example the hierarchy of Figure 6.1.

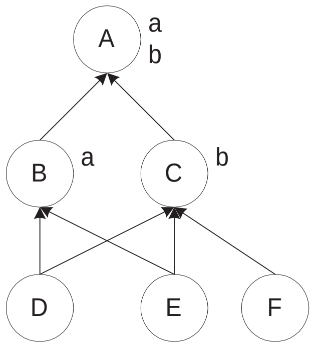


Fig. 6.1. A small example of a multiple inheritance hierarchy with two messages

The figure shows two messages, a and b,

$$\begin{aligned} I(\mathsf{a}) &= \{\mathsf{A},\mathsf{B}\}, \\ I(\mathsf{b}) &= \{\mathsf{A},\mathsf{C}\}. \end{aligned} \tag{6.2}$$

The dispatching matrix of these two messages is depicted in Figure 6.2a. Note that the results of all dispatching queries on types D and E (for example) are the same. The corresponding rows in the table are identical and can be compressed. Figure 6.2b shows a representation of the dispatching matrix obtained by merging together all identical rows and an auxiliary array of pointers to all different rows specimens.

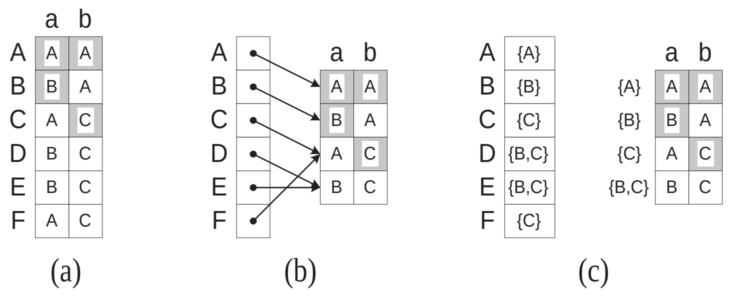


Fig. 6.2. (a) The dispatching matrix of Figure 6.1, (b) the matrix compressed using an auxiliary array of pointers, and (c) the matrix compressed using an array of set-labels

This compressed representation can be understood in terms of a new master-message $\mathcal{M} = \{\mathsf{a}, \mathsf{b}\}$, whose implementors are the union of the implementors of its members:

$$I(\mathcal{M}) \equiv I(\mathsf{a}) \cup I(\mathsf{b}) = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}.$$

The auxiliary array represents all the possible results of a *generalized* dispatch on this master-message. For example,

$$\text{g-dispatch}(\mathcal{M}, \mathsf{D}) = \text{g-dispatch}(\mathcal{M}, \mathsf{E}) = \{\mathsf{B}, \mathsf{C}\}.$$

Therefore, the D and E entries in the auxiliary array point to the same row specimen whose label is the set $\{\mathsf{B}, \mathsf{C}\}$.

In total there are four different results of generalized dispatching with respect to $I(\mathcal{M})$. Implementors $I(\mathcal{M})$ therefore partition the types in the hierarchy into four sets, as shown in Figure 6.2c. The figure shows the same compressed representation of the dispatching matrix, where the results of generalized dispatch are used to label row specimens instead of pointers in the auxiliary array.

In order to derive bounds on the quality of the CT compression in the multiple inheritance setting we need to estimate the number of distinct rows in chunks. The difficulty is that the result of a generalized dispatch is a set rather than a singleton, and hence the number of distinct rows might be exponential in the number implementors. To show that this is not the case, we first define the notion of a partition imposed by the set of implementors, and then show that the size of this partition is at most $2\kappa$ times the number implementors, where $1 \leq \kappa \leq \mathbf{n}$ is the complexity of the hierarchy as defined in Definition 3.1.

## 6.1 Implementors Partitionings

Given a partially ordered set of types $\mathcal{T}$ and a message $m$, the *partitioning of* $\mathcal{T}$ *by* $I(m)$, also called the *implementors partitioning* of $I(m)$, is

$$\nabla I(m) \equiv \{\mathcal{T}_1, \ldots, \mathcal{T}_n\},$$

such that all types in *partition* $\mathcal{T}_i$ have the same generalized dispatch result. In other words, types $a, b \in \mathcal{T}$ are in the same *partition* $\mathcal{T}_i \in \nabla I(m)$ if and only if

$$\textsf{g-dispatch}(m, a) = \textsf{g-dispatch}(m, b). \tag{6.3}$$

Figure 6.3 shows the implementors partitioning of the implementors $I(\textsf{a})$, $I(\textsf{b})$ of (6.2) and their master-message $\mathcal{M} = \{\textsf{a}, \textsf{b}\}$, $I(\mathcal{M}) \equiv I(\textsf{a}) \cup I(\textsf{b})$.

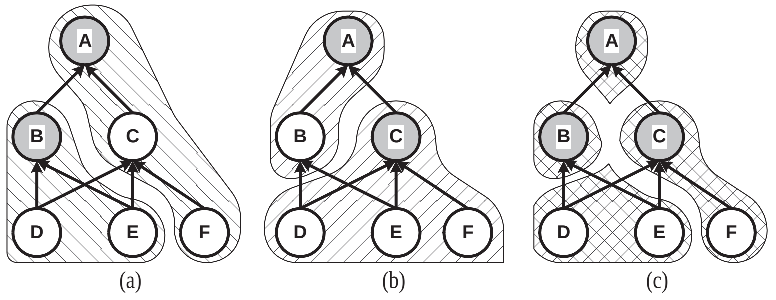


Fig. 6.3. The implementors partitionings of the implementors $I(\textsf{a})$, $I(\textsf{b})$ of (6.2) and their master-message $\mathcal{M}$, $I(\mathcal{M}) \equiv I(\textsf{a}) \cup I(\textsf{b})$

Types D and E, for example, are in the same partition in $\nabla I(\mathcal{M})$ since

$$\textsf{g-dispatch}(\mathcal{M}, \textsf{D}) = \textsf{g-dispatch}(\mathcal{M}, \textsf{E}) = \{\textsf{B}, \textsf{C}\}.$$

The partitionings are

$$\begin{aligned} \nabla I(\textsf{a}) &\equiv \{\{\textsf{A}, \textsf{C}, \textsf{F}\}, \{\textsf{B}, \textsf{D}, \textsf{E}\}\}, \\ \nabla I(\textsf{b}) &\equiv \{\{\textsf{A}, \textsf{B}\}, \{\textsf{C}, \textsf{D}, \textsf{E}, \textsf{F}\}\}, \\ \nabla I(\mathcal{M}) &\equiv \{\{\textsf{A}\}, \{\textsf{B}\}, \{\textsf{C}, \textsf{F}\}, \{\textsf{D}, \textsf{E}\}\}. \end{aligned} \tag{6.4}$$

Figure 6.4 overlays $\nabla I(\textsf{a})$ and $\nabla I(\textsf{b})$. The dotted lines are the partitions of $\nabla I(\textsf{a})$, whereas the full lines are the partitions of $\nabla I(\textsf{b})$.

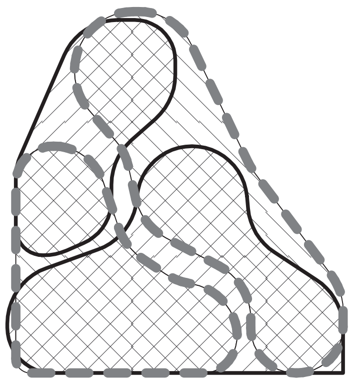


Fig. 6.4. The *overlay* of $\nabla I(\textsf{a})$ and $\nabla I(\textsf{b})$ of Figure 6.3

In comparing Figure 6.3c with Figure 6.4, we see that the partitioning $\nabla I(\mathcal{M})$ can be obtained by a simple overlay of the two partitionings $\nabla I(\mathsf{a})$ and $\nabla I(\mathsf{b})$. We will next prove that this was no coincidence.

Given two partitionings $\pi$, $\pi'$, their *overlay* $\pi \cdot \pi'$ is the coarsest partitioning consistent with both $\pi$ and $\pi'$. Constructively, the overlay is obtained by intersecting all partitions of $\pi$ with all partitions of $\pi'$:

$$\pi \cdot \pi' = \{\mathcal{T}_i \cap \mathcal{T}'_j \mid \mathcal{T}_i \in \pi, \mathcal{T}'_j \in \pi'\}. \tag{6.5}$$

For example, the overlay of $\nabla I(\mathsf{a})$ and $\nabla I(\mathsf{b})$ of (6.4) is

$$\begin{aligned}\nabla I(\mathsf{a}) \cdot \nabla I(\mathsf{b}) &= \{\{\mathsf{A},\mathsf{C},\mathsf{F}\} \cap \{\mathsf{A},\mathsf{B}\}, \{\mathsf{A},\mathsf{C},\mathsf{F}\} \cap \{\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F}\},\\ &\qquad \{\mathsf{B},\mathsf{D},\mathsf{E}\} \cap \{\mathsf{A},\mathsf{B}\}, \{\mathsf{B},\mathsf{D},\mathsf{E}\} \cap \{\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F}\}\}\\ &= \{\{\mathsf{A}\}, \{\mathsf{C},\mathsf{F}\}, \{\mathsf{B}\}, \{\mathsf{D},\mathsf{E}\}\}.\end{aligned} \tag{6.6}$$

LEMMA 6.1. $\nabla I(m_1) \cdot \nabla I(m_2) = \nabla(I(m_1) \cup I(m_2))$ *for all* $m_1$, $m_2$.

PROOF. It is a well known fact that for every partitioning $\pi$ there is a binary *equivalence relation* whose set of equivalence classes are the same as the partitioning $\pi$. Instead of proving that the partitioning $\nabla(I(m_1) \cup I(m_2))$ and $\nabla I(m_1) \cdot \nabla I(m_2)$ are equal, we will prove that their equivalence relations are the same.

On the one hand, types $a, b$ are in the equivalence relation of

$$\nabla(I(m_1) \cup I(m_2))$$

if and only if they have the same generalized dispatching results with respect to $I(m_1) \cup I(m_2)$ (see (6.3)), i.e.,

$$\mathsf{g\text{-}dispatch}(m_1 \cup I(m_2), a) = \mathsf{g\text{-}dispatch}(m_1 \cup I(m_2), b). \tag{6.7}$$

On the other hand, the overlay partitioning, $\nabla I(m_1) \cdot \nabla I(m_2)$, is defined by intersecting all partitions of $\nabla I(m_1)$ with those of $\nabla I(m_2)$ (see (6.5)). Therefore, types $a, b$ are in the equivalence relation of $\nabla I(m_1) \cdot \nabla I(m_2)$ if and only if the following two conditions hold

$$\begin{aligned}\mathsf{g\text{-}dispatch}(m_1, a) &= \mathsf{g\text{-}dispatch}(m_1, b),\\ \mathsf{g\text{-}dispatch}(m_2, a) &= \mathsf{g\text{-}dispatch}(m_2, b).\end{aligned} \tag{6.8}$$

We must show that (6.7) holds if and only if (6.8) holds. Formally, using the definition of generalized dispatch (6.1), we must show that

$$\begin{gathered}\min(\mathsf{cand}(m_1 \cup I(m_2), a)) = \min(\mathsf{cand}(m_1 \cup I(m_2), b))\\ \Leftrightarrow\\ \min(\mathsf{cand}(m_1, a)) = \min(\mathsf{cand}(m_1, b)) \ \wedge\\ \min(\mathsf{cand}(m_2, a)) = \min(\mathsf{cand}(m_2, b)).\end{gathered} \tag{6.9}$$

Since two sets of candidates (for the same message) have the same smallest elements if and only if they are equal, our objective is to prove (see the definition of candidates in (1.2))

$$\begin{aligned}(I(m_1) \cup I(m_2)) \cap \text{ancestors}(a) &= (I(m_1) \cup I(m_2)) \cap \text{ancestors}(b) \\ &\Leftrightarrow \\ I(m_1) \cap \text{ancestors}(a) &= I(m_1) \cap \text{ancestors}(b) \ \wedge \\ I(m_2) \cap \text{ancestors}(a) &= I(m_2) \cap \text{ancestors}(b).\end{aligned} \tag{6.10}$$

Given two sets $X, Y$, their *symmetric difference* is defined as

$$X \bigtriangleup Y \equiv (X \cup Y) \setminus (X \cap Y).$$

Observe that

$$Z \cap X = Z \cap Y \Leftrightarrow Z \cap (X \bigtriangleup Y) = \emptyset. \tag{6.11}$$

By combining (6.10) and (6.11) we find that we need to prove that

$$\begin{aligned}(I(m_1) \cup I(m_2)) \cap (\text{ancestors}(a) \bigtriangleup \text{ancestors}(b)) &= \emptyset \\ &\Leftrightarrow \\ I(m_1) \cap (\text{ancestors}(a) \bigtriangleup \text{ancestors}(b)) &= \emptyset \ \wedge \\ I(m_2) \cap (\text{ancestors}(a) \bigtriangleup \text{ancestors}(b)) &= \emptyset.\end{aligned} \tag{6.12}$$

The above trivially holds since for all sets $X, Y, Z$,

$$\begin{aligned}(X \cup Y) \cap Z &= \emptyset \\ &\Leftrightarrow \\ X \cap Z &= \emptyset \ \wedge \\ Y \cap Z &= \emptyset. \qquad \square\end{aligned}$$

## 6.2 Memory Requirements of the Reduction

As in the single inheritance version, the CT reduction partitions the messages $\mathcal{M}$ into disjoint slices $\mathcal{M}_1, \ldots, \mathcal{M}_{\mathbf{k}}$, and generates for each slice $\mathcal{M}_i$ its master-message, where $I(\mathcal{M}_i)$ are the union of the implementors of messages in $\mathcal{M}_i$. To answer the *generalized dispatching* query $\mathsf{g\text{-}dispatch}(m, t)$, where $m \in \mathcal{M}_i$, we first (recursively) answer the query $\mathsf{g\text{-}dispatch}(\mathcal{M}_i, t)$. This recursive call returns one of the partitions of $\nabla I(\mathcal{M}_i)$. The next step is to find the unique containing partition of $\nabla I(m)$.

To understand this better, recall that $I(m) \subseteq I(\mathcal{M}_i)$. To apply Lemma 6.1 note that there exists a set $X$ such that $I(\mathcal{M}_i) = I(m) \cup X$, and hence

$$\nabla I(\mathcal{M}_i) = \nabla(I(m) \cup X) = \nabla I(m) \cdot \nabla X.$$

Therefore, every partition of $\nabla I(\mathcal{M}_i)$ is contained in a partition of $\nabla I(m)$. A matrix $A_i$ with $|\nabla I(\mathcal{M}_i)|$ rows and $|\mathcal{M}_i|$ columns is used to map each of the partitions of $\nabla I(\mathcal{M}_i)$ to a partition of $\nabla I(m)$, for all $I(\in)\,\mathcal{M}_i$. Matrices $A_1, \ldots, A_{\mathbf{k}}$ are nothing other than the dispatching data structure of the CT reduction. (Clearly, there is an additional data structure which the recursive call uses.)

To bound the size of these matrices, we need to bound $|\nabla I(m)|$. In single inheritance, the root of each partition correspond to a different implementor, and therefore $|\nabla I(m)| \leq |I(m)|$. An easy, but not so useful bound in multiple inheritance, is $|\nabla I(m)| \leq 2^{|I(m)|}$.

We will show below how to use $\kappa$, the complexity of a hierarchy (see Definition 3.1), to give a better bound:

$$|\nabla I(m)| \leq 2\kappa |I(m)|. \tag{6.13}$$

Using slices with $x$ messages in each, the total memory of matrices $A_1, \ldots, A_{\mathbf{k}}$ is

$$\sum_{i=1}^{\mathbf{k}} |\nabla I(\mathcal{M}_i)| \times |\mathcal{M}_i| = \sum_{i=1}^{\mathbf{k}} |\nabla I(\mathcal{M}_i)| \times x \leq x \sum_{i=1}^{\mathbf{k}} 2\kappa |I(\mathcal{M}_i)| \leq 2x\kappa\mathbf{i}.$$

The recursive equations then become

$$\begin{aligned} \mathrm{mem}_1(\mathbf{n}, \mathbf{m}, \mathbf{i}) &= \mathbf{nm}, \\ \mathrm{mem}_{d+1}(\mathbf{n}, \mathbf{m}, \mathbf{i}) &\leq 2\kappa\mathbf{i} \cdot x + \mathrm{mem}_d(\mathbf{n}, \mathbf{m}/x, \mathbf{i}). \end{aligned} \tag{6.14}$$

By using $2\kappa\mathbf{i}$ instead of $\mathbf{i}$, the analysis of the previous section holds.

COROLLARY 6.2. *Let* $\varphi \equiv (\mathbf{nm})/(2\kappa\mathbf{i})$. *In a hierarchy whose complexity is* $\kappa$, $\mathrm{CT}_d$ *performs dispatching in* $d$ *dereferencing operations, and reaches a compression factor of at least* $\frac{1}{d}\varphi^{1-1/d}$ *(when using a slice size of* $\varphi^{1/d}$*).*

In other words, in a hierarchy whose complexity is $\kappa$, the space requirements of $\mathrm{CT}_d$ in the multiple inheritance setting is worse than the single inheritance setting by a factor of at most $(2\kappa)^{1-1/d}$.

## 6.3 Hierarchy Complexity

We now show how to use $\kappa$, the complexity of a hierarchy (see Definition 3.1), to bound the size of a implementors partitioning, i.e., show that $|\nabla I(m)| \leq 2\kappa |I(m)|$. Figure 6.5 is an example of a multiple inheritance hierarchy of complexity 1, i.e., there exists an ordering of $\mathcal{T}$ in which the descendants of every type define an interval. Within each type we write its position in that ordering.

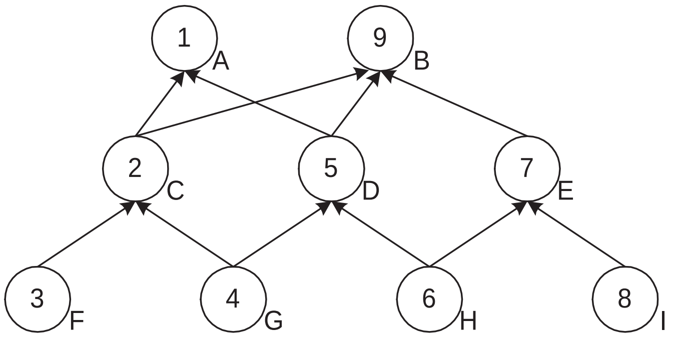


Fig. 6.5. An example of a multiple inheritance hierarchy of complexity 1

Figure 6.6 shows the implementors partitioning of $I(m) = \{\mathsf{A}, \mathsf{B}, \mathsf{E}\}$ in the hierarchy of Figure 6.5. Observe that $|\nabla I(m)| = 5$.

Since the complexity of this hierarchy is 1, the descendants of each type define an *interval.* Therefore the implementors $I(m)$ defines the three intervals depicted in Figure 6.7.

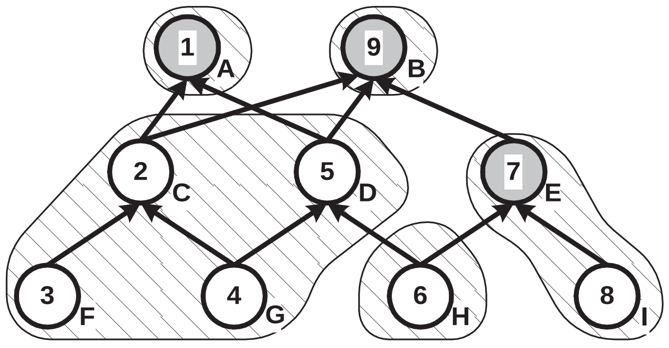


Fig. 6.6. The implementors partitioning of $I(m) = \{\mathsf{A}, \mathsf{B}, \mathsf{E}\}$ in the hierarchy of Figure 6.5

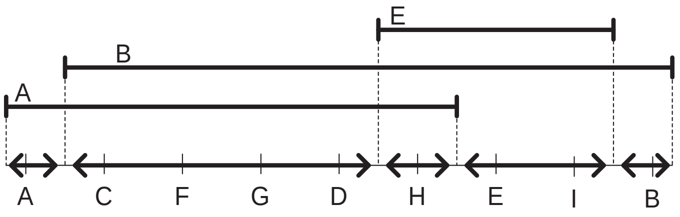


Fig. 6.7. The intervals of the implementors $I(m) = \{\mathsf{A}, \mathsf{B}, \mathsf{E}\}$ in the hierarchy of Figure 6.5

The intervals in Figure 6.7 partition the types into 5 *segments*. (We will show that there are at most $2|I(m)|$ segments.) Types in the same segment have the same set of candidates and therefore belong to the same partition. So we conclude that the number of partitions is at most the number of segments, which in turn is at most $2|I(m)|$. In our example,

$$|\nabla I(m)| = 5 \leq 6 = 2|I(m)|.$$

We need the following fact, whose proof is elementary:

FACT 6.3. *A set of $f$ intervals partition any consecutive set into at most $2f + 1$ segments. Out of these segments at most $2f - 1$ are contained in one interval or more. (See illustration in Figure 6.7.)*

LEMMA 6.4. $|\nabla I(m)| \leq 2\kappa|I(m)|$ *for each message $m$.*

PROOF. Let $f = |I(m)|$. Recall (Definition 3.1) the partitioning of $\mathcal{T}$ into sets $\mathcal{T}_1, \ldots, \mathcal{T}_\kappa$ with their respective ordering. Let $i$ be fixed. We write the list of members of the set $\mathcal{T}_i$, enumerated in its respective order $\pi_i$.

Consider a type $t \in \mathcal{T}_i$. The result of $\mathsf{g\text{-}dispatch}(m, t)$ is uniquely determined by the subset of all types $t' \in I(m)$, such that $t$ is among the descendant of $t'$. From Definition 3.1, we have that the descendants are consecutive in the list of $\mathcal{T}_i$. Implementors $I(m)$ define therefore $f$ intervals (which may be empty) in this list. These intervals partition the list into at most $2f + 1$ segments such that the result of $\mathsf{g\text{-}dispatch}(m, t)$ is uniquely determined by the segment of $t$. These segments give the restriction of $\nabla I(m)$ to $\mathcal{T}_i$.

We have thus obtained $|\nabla I(m)| \leq \kappa(2f+1)$. To obtain a tighter bound we need a more careful counting. Let us remove from $\mathcal{T}_i$ all types which are not descendants of any of the members of $I(m)$. The remaining types are divided by $I(m)$ into $2f-1$ segments. Generalized dispatching on the removed types returns the empty set, irrespective of $i$. The total number of equivalence classes in $\nabla I(m)$ is therefore $\kappa(2f-1)+1 \leq 2\kappa f$. □

Remark 6.5. *The actual partitioning $\mathcal{T}_1, \ldots, \mathcal{T}_\kappa$ is* not *required in order to apply the CT reduction; only the integer $\kappa$ is needed for determining the slice size. We found that in practice the single inheritance analysis closely models even hierarchies which use multiple inheritance heavily. (Therefore there is no need even to find $\kappa$.)*

## 7. DATA SET

Thirty-five hierarchies collected from eight different programming languages and totaling 63,972 types, were assembled from the following sources:

(1) The four hierarchies (Self, Unidraw, LOV, Geode) used in benchmark of RD in multiple inheritance hierarchies [Driesen and Hölzle 1995].
(2) The eight Smalltalk, ObjectiveC and C++ hierarchies used for benchmarking RD and CT [Vitek and Horspool 1996] in single inheritance hierarchies.
(3) The ensemble of seven Java hierarchies used in the definition of the "common programming practice" [Cohen and Gil 2000], augmented by version 1.3.1 of the Java Development Kit. Each of these eight hierarchies, was also used both for benchmarking multiple inheritance dispatching algorithms and, after pruning interfaces, for benchmarking single inheritance dispatching algorithms.
(4) The two Cecil [Chambers 1993] and Dylan [Shalit 1997] hierarchies used in all benchmarking of multiple dispatching algorithms [Holst et al. 1998; Dujardin et al. 1998; Pang et al. 1999; Dujardin 1996] contributed by Eric Dujardin.
We regard each multi-dispatch query as several independent single-dispatch queries on each of the arguments, as done in the first step of the major algorithms for multi-dispatching [Zibin and Gil 2002].
(5) A collection of five other multiple dispatching hierarchies contributed by Wade Holst: Cecil- and Cecil2 are two older versions of the Cecil run time library. Vortex3 is a Cecil compiler written in Cecil, while Vor3 is an old version of this compiler. Harlequin is a commercial implementation of Dylan including its GUI library.

The data set for benchmarking dispatching algorithms has 16 single inheritance hierarchies with 29,162 types, 12 multiple inheritance hierarchies with 27,728 types, and seven multiple dispatch hierarchies with 7,082 types.

This benchmark includes 5 hierarchies out of 13 hierarchies used in previous experimental work on subtyping. (We were unable to obtain information on the definition of messages and methods in the other eight hierarchies.) As observed previously [Eckel and Gil 2000] many of the topological properties of these hierarchies are similar to those of balanced binary trees. The average number of ancestors in these hierarchies is less than 9 for all hierarchies, with the exception of Geode, in which it is 14.0 and Self, in which it is 30.9.

All degenerate messages, i.e., messages with a single implementor (singletons), were eliminated from the data set prior to running the experiments. The reason for doing so is that sending a degenerate message requires no dispatching, and is the same as static procedure call. (In dynamically typed languages there is an earlier step, which is *equivalent* to a subtyping test, in which it is made sure that the message is valid for the receiver type.)

Table II gives a summary of the pruned hierarchies. The three blocks in the table correspond to single inheritance-, multiple inheritance-, and multiple dispatch-hierarchies. We see that the hierarchies span a range of sizes, from about a hundred types up to almost 9,000 types.

The row denoted *Total* in this and some of the subsequent tables corresponds to the total or universal hierarchy obtained by a simple disjoint union of all hierarchies in the ensemble. In most cases, the "Total" row therefore corresponds to an average of the different hierarchies, weighted by size. In Table II, this row indicates that in total the dispatching benchmark spanned some 64 thousand types and 70 thousand messages.

The **i/n** column shows the average number of method implementations per type. Examining the entries along this column we see that in many multiple dispatch hierarchies, there are about one or two methods per type. A typical value of the other hierarchies is four or five implementations per type. The San Francisco (SI: IBM SF) project gives the largest number of methods per type (13.3).

In checking the **i/m** column we find that the number of implementors tend to be small, with average values of around four to six methods per message in most hierarchies. We note that the average number of comparisons in a binary search in the set of implementors is no greater than $\lceil \log_2 \frac{\mathbf{i}}{\mathbf{m}} \rceil$. The reason is that the geometrical mean is no greater than the arithmetical mean, and therefore

$$\begin{aligned}\frac{1}{\mathbf{m}} \sum_{m\in\mathcal{M}} \log_2 |I(m)| &= \log_2 \left( \prod_{m\in\mathcal{M}} |I(m)| \right)^{\frac{1}{\mathbf{m}}} \\ &\le \log_2 \left( \frac{1}{\mathbf{m}} \sum_{m\in\mathcal{M}} |I(m)| \right) = \log_2 \frac{\mathbf{i}}{\mathbf{m}}.\end{aligned} \tag{7.1}$$

Thus, just by inspecting the **i/m** column we learn that the number of comparisons is about 3.

The next **(nm)/w** column gives the best possible factor by which null elimination can improve upon the complete dispatching matrix. As can be seen from the table, this matrix is very sparse. In most cases, 90% or more of its cells are null. In hierarchies such as MI: JDK 1.3.1 and MI: IBM SF we even find that the potential compression is by a factor as high as 300.

How much can duplicates elimination improve on an *optimal* null elimination? The answer is in the **w/i** column. We observe a potential for *additional* compression by factors of about 10. Duplicates elimination performs very well precisely on the multiple dispatch hierarchies, where mere null elimination is not as effective as it is in other hierarchies.

| | Hierarchy | **n** | **m** | **i/n** | **i/m** | **(nm)/w** | **w/i** |
|---|---|---|---|---|---|---|---|
| Single Inheritance | Visualworks1 | 774 | 1,170 | 6.0 | 4.0 | 11.4 | 17.1 |
| | Visualworks2 | 1,956 | 3,196 | 6.9 | 4.2 | 21.6 | 21.3 |
| | Digitalk2 | 535 | 962 | 6.2 | 3.5 | 7.1 | 21.7 |
| | Digitalk3 | 1,357 | 2,402 | 7.0 | 3.9 | 9.0 | 38.3 |
| | IBM Smalltalk 2 | 2,320 | 4,335 | 7.0 | 3.8 | 49.1 | 12.6 |
| | VisualAge 2 | 3,241 | 6,529 | 8.1 | 4.0 | 35.6 | 22.7 |
| | NextStep | 311 | 499 | 6.8 | 4.2 | 9.6 | 7.7 |
| | ET++ | 371 | 296 | 3.8 | 4.8 | 9.0 | 8.6 |
| | SI: JDK 1.3.1 | 6,681 | 4,392 | 3.6 | 5.4 | 228.8 | 5.4 |
| | SI: Corba | 1,329 | 222 | 1.9 | 11.6 | 42.5 | 2.7 |
| | SI: HotJava | 644 | 690 | 4.5 | 4.2 | 18.6 | 8.2 |
| | SI: IBM SF | 6,626 | 11,664 | 13.3 | 7.6 | 268.9 | 3.3 |
| | SI: IBM XML | 107 | 131 | 5.5 | 4.5 | 10.8 | 2.2 |
| | SI: Orbacus | 1,053 | 980 | 3.6 | 3.9 | 55.3 | 4.9 |
| | SI: Orbacus Test | 579 | 368 | 4.1 | 6.5 | 37.6 | 2.4 |
| | SI: Orbix | 1,278 | 535 | 2.3 | 5.4 | 62.7 | 3.8 |
| Multiple Inheritance | Self | 1,802 | 2,459 | 12.1 | 8.8 | 18.9 | 10.8 |
| | Unidraw | 614 | 360 | 3.8 | 6.5 | 27.3 | 3.5 |
| | LOV | 436 | 663 | 6.5 | 4.3 | 20.5 | 5.0 |
| | Geode | 1,318 | 1,413 | 7.2 | 6.7 | 15.2 | 12.9 |
| | MI: JDK 1.3.1 | 7,401 | 5,724 | 3.9 | 5.0 | 300.7 | 4.9 |
| | MI: Corba | 1,699 | 396 | 1.9 | 8.1 | 49.6 | 4.2 |
| | MI: HotJava | 736 | 829 | 4.6 | 4.1 | 24.5 | 7.3 |
| | MI: IBM SF | 8,793 | 14,575 | 13.2 | 8.0 | 328.3 | 3.4 |
| | MI: IBM XML | 145 | 271 | 6.5 | 3.5 | 16.9 | 2.5 |
| | MI: Orbacus | 1,379 | 1,261 | 3.6 | 4.0 | 70.1 | 5.0 |
| | MI: Orbacus Test | 689 | 379 | 4.0 | 7.3 | 34.9 | 2.7 |
| | MI: Orbix | 2,716 | 786 | 1.4 | 4.7 | 95.1 | 6.1 |
| Multiple Dispatching | Cecil | 932 | 1,009 | 4.5 | 4.2 | 12.9 | 17.3 |
| | Dylan | 925 | 428 | 1.9 | 4.2 | 5.6 | 39.5 |
| | Cecil- | 473 | 592 | 5.0 | 4.0 | 17.4 | 6.8 |
| | Cecil2 | 472 | 131 | 1.2 | 4.3 | 3.6 | 30.6 |
| | Harlequin | 666 | 229 | 1.5 | 4.4 | 6.6 | 22.7 |
| | Vor3 | 1,660 | 328 | 1.1 | 5.7 | 35.3 | 8.3 |
| | Vortex3 | 1,954 | 476 | 1.3 | 5.2 | 3.0 | 122.4 |
| | Total | 63,972 | 70,680 | 6.5 | 5.9 | N/A | 8.7 |
| | Median | 1,053.0 | 690.0 | 4.5 | 4.4 | 21.6 | 7.3 |
| | Minimum | 107 | 131 | 1.1 | 3.5 | 3.0 | 2.2 |
| | Maximum | 8,793 | 14,575 | 13.3 | 11.6 | 328.3 | 122.4 |

Table II. Statistical and topological properties of the 35 hierarchies used in benchmarking dispatching algorithms

## 8. EXPERIMENTAL RESULTS

In this section we compare the theoretical prediction of algorithms TS and CT$_d$ with their empirical performance. All tests were run on 900Mhz Pentium III computer, equipped with 256MB internal memory and controlled by a Windows 2000 operating system.

## 8.1 Space requirement

In order to evaluate the quality of the order-preserving heuristic used in our TS technique, we compared it with a much more powerful, but time consuming, heuristic which uses PQ-trees. The superscript PQ shall denote the variant which use the PQ heuristic.

We follow the popular convention of ignoring *code space* requirement, i.e., assuming that there is a single generic dispatching routine which receives a message-selector and a type-id. Although our results indicate that inlining of the binary search might be worthy, further research is required to estimate the incurred code space penalty.

The following definition is pertinent to the comparison of algorithms.

Definition 8.1. *redundancy Let $W$ be the number of 4-bytes words the algorithm uses to encode the dispatching tables of a certain hierarchy, then the algorithm's* redundancy factor *on this hierarchy is $W/\mathbf{i}$.*

In other words, the redundancy factor of a dispatching algorithm in a certain hierarchy is the ratio between the total space requirement of that algorithm and the lower bound ideal implementation which uses 4 bytes for storing the address of each method.

Table III gives the redundancy factor of different algorithms on the 35 hierarchies in our dispatching benchmark. In reading the table, remember that better algorithms have lower redundancy factors.

Algorithms CT, $CT_2$, $CT_3$, TS, and $TS^{PQ}$ attempt to achieve duplicates elimination. The other algorithms rely on null elimination. The results in the table do not include the additional provisions mentioned above for the RD and SC algorithms to support dynamically typed languages. The redundancy factors have to be appropriately adjusted to include selector verification information.

Since we did not have access to the original implementation and heuristics of SC and CT, redundancy factors reported in the respective columns present a lower bounds on these values: In SC, the number of slices is no less than the maximal number of messages that a type understands. In estimating CT, the set of messages was divided into chunks of 14 messages each (as prescribed in [Vitek and Horspool 1996]). We then applied the SC lower bound estimate in each chunk.

Memory usage of $CT_2$ and $CT_3$ were obtained using the empirically found *best* slice size (which may be different than the prescription of column 2 of Table I).

The results of the VFT technique are calculated in the traditional manner [Driesen and Hölzle 1995], under the assumption that there are no virtual bases. The size of a type VFTs equals the sum of its parents VFTs plus the number of newly introduced messages. However, in practice inheritance is usually *shared* (not *repeated*), giving rise to other overheads [Eckel and Gil 2000].

In studying the last column of the table (labeled "Mem") we see that the total space requirement of type slicing ranges between 5KB to almost 1.7MB. When viewed in relative- rather than absolute-terms (in the penultimate column labeled TS showing redundancy factors), we find that the space requirement of type slicing is about three or four times larger than a theoretic optimal duplicates elimination.

In comparing the columns TS and $TS^{PQ}$ we find that using the PQ-heuristic does not always improve the space performance. In fact, in all single inheritance

| | Hierarchy | CT[a] | CT$_2$ | CT$_3$ | VFT | SC[b] | RD | **TS$^{PQ}$** | **TS** | Mem[c] |
|---|---|---|---|---|---|---|---|---|---|---|
| Single Inheritance | Visualworks1 | 18.3 | 17.9 | 9.9 | 17.1 | 24.3 | 17.3 | 2.8 | 2.5 | 45 |
| | Visualworks2 | 37.5 | 28.8 | 13.9 | 21.3 | 39.8 | 21.7 | 2.6 | 2.5 | 134 |
| | Digitalk2 | 15.8 | 17.0 | 9.9 | 21.7 | 59.8 | 22.0 | 3.0 | 2.7 | 35 |
| | Digitalk3 | 29.8 | 26.1 | 13.3 | 38.3 | 92.5 | 38.8 | 3.0 | 2.7 | 98 |
| | IBM Smalltalk 2 | 48.9 | 30.5 | 13.8 | 12.6 | 37.5 | 15.4 | 3.0 | 2.6 | 165 |
| | VisualAge 2 | 63.0 | 35.3 | 15.3 | 22.7 | 62.3 | 29.2 | 3.0 | 2.6 | 267 |
| | NextStep | 10.7 | 12.7 | 8.3 | 7.7 | 21.8 | 7.9 | 2.9 | 2.6 | 22 |
| | ET++ | 9.9 | 12.1 | 8.1 | 8.6 | 26.0 | 8.9 | 2.6 | 2.4 | 13 |
| | SI: JDK 1.3.1 | 91.9 | 45.1 | 18.3 | 5.4 | 67.9 | 6.2 | 2.6 | 2.4 | 219 |
| | SI: Corba | 10.1 | 12.2 | 7.1 | 2.7 | 25.2 | 3.7 | 2.8 | 2.7 | 27 |
| | SI: HotJava | 15.5 | 16.4 | 9.5 | 8.2 | 33.7 | 8.5 | 2.8 | 2.5 | 28 |
| | SI: IBM SF | 66.0 | 26.9 | 10.8 | 3.3 | 26.0 | 3.5 | 2.4 | 2.2 | 744 |
| | SI: IBM XML | 4.2 | 4.8 | 3.8 | 2.2 | 8.4 | 2.5 | 2.5 | 2.1 | 5 |
| | SI: Orbacus | 22.6 | 18.8 | 9.4 | 4.9 | 35.0 | 5.1 | 2.8 | 2.4 | 36 |
| | SI: Orbacus Test | 8.4 | 8.4 | 5.0 | 2.4 | 43.9 | 2.9 | 2.5 | 2.3 | 21 |
| | SI: Orbix | 21.3 | 21.2 | 11.5 | 3.8 | 35.7 | 4.6 | 2.8 | 2.5 | 29 |
| Multiple Inheritance | Self | 17.6 | 14.3 | 7.4 | 10.8 | 27.3 | 11.1 | 3.0 | 2.8 | 240 |
| | Unidraw | 10.7 | 12.8 | 8.2 | 3.5 | 15.3 | 4.0 | 2.7 | 2.5 | 23 |
| | LOV | 12.1 | 14.2 | 9.1 | 12.8 | 11.8 | 5.2 | 4.4 | 4.5 | 50 |
| | Geode | 19.2 | 19.0 | 10.8 | 44.9 | 40.4 | 16.2 | 5.5 | 6.1 | 228 |
| | MI: JDK 1.3.1 | 109.2 | 48.3 | 18.5 | 5.8 | 62.4 | 5.5 | 4.1 | 4.1 | 463 |
| | MI: Corba | 18.5 | 16.4 | 8.5 | 6.5 | 35.6 | 4.9 | 3.4 | 3.3 | 42 |
| | MI: HotJava | 17.3 | 17.4 | 9.7 | 8.5 | 39.0 | 7.6 | 4.2 | 4.6 | 60 |
| | MI: IBM SF | 82.3 | 31.6 | 11.8 | 5.9 | 26.2 | 3.5 | 3.8 | 3.7 | 1,663 |
| | MI: IBM XML | 5.7 | 6.3 | 4.4 | 3.5 | 8.7 | 2.6 | 3.5 | 3.3 | 12 |
| | MI: Orbacus | 28.0 | 20.9 | 10.2 | 6.9 | 37.5 | 5.3 | 4.0 | 3.8 | 75 |
| | MI: Orbacus Test | 8.8 | 8.8 | 5.1 | 3.5 | 45.3 | 3.0 | 3.2 | 3.2 | 35 |
| | MI: Orbix | 45.1 | 31.6 | 14.7 | 7.0 | 64.5 | 6.7 | 3.6 | 3.4 | 49 |
| Multiple Dispatching | Cecil | 19.5 | 27.2 | 15.8 | 34.0 | 34.6 | 17.8 | 4.2 | 4.1 | 68 |
| | Dylan | 20.5 | 23.9 | 13.5 | 46.3 | 71.6 | 40.2 | 3.5 | 3.5 | 24 |
| | Cecil- | 12.7 | 18.3 | 11.8 | 12.7 | 27.7 | 7.2 | 4.5 | 4.8 | 45 |
| | Cecil2 | 11.6 | 11.6 | 6.7 | 100.3 | 69.7 | 31.2 | 3.3 | 3.9 | 9 |
| | Harlequin | 14.2 | 16.7 | 10.0 | 47.9 | 83.3 | 23.5 | 4.3 | 4.4 | 18 |
| | Vor3 | 24.1 | 25.5 | 13.9 | 19.4 | 50.8 | 9.3 | 3.4 | 3.5 | 26 |
| | Vortex3 | 29.2 | 29.2 | 14.8 | 375.7 | 159.7 | 124.0 | 3.5 | 4.1 | 40 |
| | Total | 55.7 | 29.7 | 13.0 | 22.8 | 48.5 | 13.3 | 3.3 | 3.2 | 433 |
| | Median | 18.5 | 18.3 | 10.0 | 8.5 | 37.5 | 7.6 | 3.0 | 2.8 | 42 |
| | Minimum | 4.2 | 4.8 | 3.8 | 2.2 | 8.4 | 2.5 | 2.4 | 2.1 | 5 |
| | Maximum | 109.2 | 48.3 | 18.5 | 375.7 | 159.7 | 124.0 | 5.5 | 6.1 | 1,663 |

[a]The original CT with slice size 14 and SC within each chunk
[b]A lower bound on SC redundancy factor
[c]The space requirements of TS in kilo-bytes

Table III. The redundancy factor of different dispatching algorithms and the total memory requirements of TS in kilo-bytes

hierarchies, and several multiple inheritance hierarchies, it *increases* the memory consumption of the algorithm. The improvement, in the few cases it occurs, is quite small; a maximum of 15% in the Vortex3 hierarchy.

RD is better than our TS algorithm in three out of 35 hierarchies: IBM SF (redundancy factor 3.5 in RD vs. 3.7 in TS), IBM XML (2.6 vs. 3.3), and Orbacus Test (3.0 vs. 3.2) multiple inheritance hierarchies. We see that even in these cases the space requirement of TS is comparable to that of RD.

TS however always wins against CT, VFT, SC, and against RD in all other hierarchies, sometimes by factors as large as 30. For instance, in the Vortex3 hierarchy, RD uses 1.24MB, an optimal null elimination scheme will use 1.22MB, while TS uses 40KB!

The average improvement of TS over RD is by a factor of 4.6, while the median improvement is by a factor of 2.6. In fairness, it should be said that all these algorithms dispatch in constant time, using simple array references, while TS uses a non-constant time binary search. This constant time must be extended to include selector verification in dynamically languages, which is not required in TS. Conversely, as we saw in Section 3, the search time in TS can be reduced in statically typed languages.

In general, the VFT algorithm is the next best algorithm among *single inheritance hierarchies*. The RD algorithm is usually the second best for *multiple inheritance hierarchies*, while the CT techniques perform well on *multiple dispatch hierarchies*. The median improvement of $CT_3$ over $CT_2$ is 44%.

We remind the reader that the comparison presented in Table III is different than that reported in the literature, since even though we used the same hierarchies, we eliminated degenerate messages from the benchmark. Different algorithms compress such messages to different levels.

## 8.2 Creation time

Table IV compares the times for creating the compressed dispatching data structures using RD with those of TS and those of $TS^{PQ}$. Since we could not obtain the original implementations of SC and CT, their runtime is not reported. Vitek and Horspool [Vitek and Horspool 1996] report that CT required 1.5 seconds for NextStep hierarchy, and 4.8 seconds for Visualworks2, on a Sparc station 5. The implementation of VFT is so straightforward and fast that its runtime overhead can be considered as zero for many practical purposes.

TS is consistently better than RD, sometimes by a factor of hundreds. The average improvement of TS over RD is by a factor of 37.4, while the median is 6.3. (Since RD is a heuristic it may sometimes find a good solution quickly.) $TS^{PQ}$ is very slow.

The runtimes for generating the CT encodings (without actually copying the values into matrices) of the first four schemes ($CT_2$ through $CT_5$) were 0.7 Sec, 1.4 Sec, 2.1 Sec and 2.9 Sec. Since our data-set included in total 418,839 methods we find that the time per implementation is measured in microseconds. For example, we found that the creation time *per implementation* ranged between 0.3 and 1.7 $\mu$Sec in $CT_2$ in single inheritance hierarchies (the median being 0.6 $\mu$Sec). These times increase in multiple inheritance hierarchies: the range being 1.1 to 6.7 $\mu$Sec; the median being 2.4 $\mu$Sec.

| | Hierarchy | RD | $CT_2$ | $CT_3$ | $CT_4$ | $CT_5$ | **TS** | $\mathbf{TS^{PQ}}$ |
|---|---|---|---|---|---|---|---|---|
| Single Inheritance | Visualworks1 | 54 | 3 | 5 | 6 | 8 | 5 | 261 |
| | Visualworks2 | 250 | 9 | 17 | 25 | 29 | 13 | 2,430 |
| | Digitalk2 | 54 | 2 | 4 | 4 | 5 | 3 | 130 |
| | Digitalk3 | 281 | 7 | 11 | 16 | 23 | 9 | 1,040 |
| | IBM Smalltalk 2 | 3,430 | 11 | 19 | 26 | 35 | 15 | 3,790 |
| | VisualAge 2 | 18,800 | 20 | 35 | 47 | 63 | 24 | 8,160 |
| | NextStep | 13 | 1 | 1 | 2 | 2 | 1 | 50 |
| | ET++ | 9 | 1 | 1 | 1 | 2 | 1 | 60 |
| | SI: JDK 1.3.1 | 162 | 16 | 28 | 39 | 52 | 26 | 33,600 |
| | SI: Corba | 11 | 1 | 2 | 1 | 2 | 3 | 561 |
| | SI: HotJava | 22 | 1 | 3 | 3 | 4 | 2 | 211 |
| | SI: IBM SF | 1,620 | 45 | 73 | 107 | 134 | 69 | 30,300 |
| | SI: IBM XML | 1 | 1 | 1 | 1 | 1 | 1 | 10 |
| | SI: Orbacus | 27 | 1 | 4 | 5 | 5 | 4 | 401 |
| | SI: Orbacus Test | 12 | 1 | 1 | 1 | 1 | 1 | 110 |
| | SI: Orbix | 18 | 1 | 3 | 3 | 4 | 3 | 571 |
| Multiple Inheritance | Self | 242 | 84 | 159 | 250 | 354 | 30 | 27,600 |
| | Unidraw | 9 | 5 | 10 | 14 | 18 | 3 | 371 |
| | LOV | 18 | 6 | 12 | 19 | 22 | 5 | 3,430 |
| | Geode | 182 | 40 | 83 | 128 | 162 | 38 | 66,800 |
| | MI: JDK 1.3.1 | 240 | 82 | 161 | 237 | 316 | 88 | 324,000 |
| | MI: Corba | 26 | 6 | 10 | 17 | 22 | 9 | 10,400 |
| | MI: HotJava | 30 | 7 | 15 | 26 | 32 | 7 | 3,390 |
| | MI: IBM SF | 903 | 277 | 568 | 874 | 1180 | 307 | 1,740,000 |
| | MI: IBM XML | 2 | 1 | 2 | 4 | 4 | 1 | 140 |
| | MI: Orbacus | 31 | 11 | 22 | 31 | 45 | 11 | 12,700 |
| | MI: Orbacus Test | 11 | 3 | 6 | 8 | 10 | 4 | 1,740 |
| | MI: Orbix | 31 | 9 | 19 | 24 | 28 | 14 | 12,400 |
| Multiple Dispatching | Cecil | 57 | 21 | 42 | 66 | 80 | 9 | 6,410 |
| | Dylan | 48 | 12 | 29 | 36 | 64 | 5 | 1,870 |
| | Cecil- | 18 | 9 | 23 | 32 | 44 | 4 | 2,490 |
| | Cecil2 | 16 | 1 | 3 | 4 | 4 | 1 | 2,650 |
| | Harlequin | 23 | 3 | 6 | 9 | 11 | 2 | 2,710 |
| | Vor3 | 24 | 8 | 17 | 25 | 33 | 9 | 23,400 |
| | Vortex3 | 394 | 16 | 31 | 47 | 59 | 11 | 42,100 |

Table IV. Encoding creation time in milliseconds of different dispatching algorithms

## 8.3 Dispatching time

The dispatching sequence of $CT_d$ has exactly $d$ dereference steps in the batch version. The incremental version has additional constant-time overhead, such as a range check before accessing the dispatching matrix and an additional dereference to retrieve the slice of the dispatching message.

In TS we associate with each message an array of the $\kappa$ addresses of the appropriate sorted dictionaries of each slice. The dispatching sequence therefore starts with a single dereference to retrieve the sorted dictionary, followed by a binary search routine. The main performance metric of such code is the number of conditionals in the search routine.

We computed the average number of such conditionals, taking care to weigh each slice proportionally to the number of types in it. The average number of such conditionals in the 35 hierarchies ranged between 0.6 and 3.4; the median value being 2.5. (Even though the experiments used only non-degenerate messages, i.e., messages with two or more methods, it turned out to be that the number of conditionals was sometime zero, precisely when there was only one method implementation in a slice.)

A potentially better technique eliminates the dereference by coalescing all the sorted dictionaries of each message. Observe that with this technique dispatching time increases from $O(\log|I(m)|)$ to $O(\log \kappa|I(m)|)$. In practice, if this is implemented, then the average number of comparisons ranges between 2.5 to 3.8; the median becomes 2.9. We see that the dereference is substituted by about one or two comparisons on average. We should also say that this coalescing technique reduces the total memory requirement, since it eliminates the array of the $\kappa$ addresses which was associated with each message. We finally note that, for this technique, we can use a weaker definition for the complexity of an hierarchy, which is: *there exists an ordering of $\mathcal{T}$ in which the descendants of any type define at most $\kappa$ intervals.*

## 8.4 Experimental Results for $\mathrm{CT}_d$

Figure 8.1 shows the memory used by the first four CT schemes relative to the $\mathbf{w}$ baseline in the 35 hierarchies in the data-set.

The figure shows that compared to the *optimal* null-elimination, $\mathrm{CT}_2$ is better in 6 hierarchies, $\mathrm{CT}_3$ in 13 hierarchies, $\mathrm{CT}_4$ in 15 hierarchies, and $\mathrm{CT}_5$ in 16 hierarchies. In a few cases, the improvement is by an order of magnitude from the baseline. We also see that $\mathrm{CT}_2$ is at most one order of magnitude worse than this idealized baseline.

We can also learn from Figure 8.1 that the incremental improvement by the series of CT schemes is diminishing. In fact, examining the actual memory requirements, we find that the median incremental improvements are: $\mathrm{CT}_3$ over $\mathrm{CT}_2$: 44%, $\mathrm{CT}_4$ over $\mathrm{CT}_3$: 18%, and $\mathrm{CT}_5$ over $\mathrm{CT}_4$: 8%. This finding is consistent with the theoretical prediction.

The figure also plots another idealized algorithm, i.e., the optimal duplicates-elimination scheme, which uses $\mathbf{i}$ cells. We see that this ideal is about one order of magnitude better than the various CT schemes. Finally, we see a certain correlation between $\mathbf{i}$ and the series of CT schemes, as predicted by the theoretical analysis. When $\mathbf{i} \ll \mathbf{w}$ the CT schemes outperform even an optimal null-elimination scheme.

We now turn to comparing the actual performance of the various CT schemes with the theoretically obtained bounds.

In single inheritance hierarchies, the upper bound on the memory requirement are given by the fourth column of Table I. Figure 8.2a shows the memory requirement relative to these values. We see that in all schemes and in all hierarchies, the memory requirement is significantly smaller than the upper bounds. Also, the extent of improvement of $\mathrm{CT}_d$ over the upper bound increases with $d$.

Corollary 6.2 provides the upper bounds in multiple inheritance hierarchies depending on their complexity $\kappa$. Figure 8.2b shows the memory, relative to these upper bounds, of the actual CT performance. Again, we see that the extent of improvement of $\mathrm{CT}_d$ over the upper bound increases with $d$. Interestingly, in comparing Figure 8.2b with Figure 8.2a, we see that the improvement of the imple-

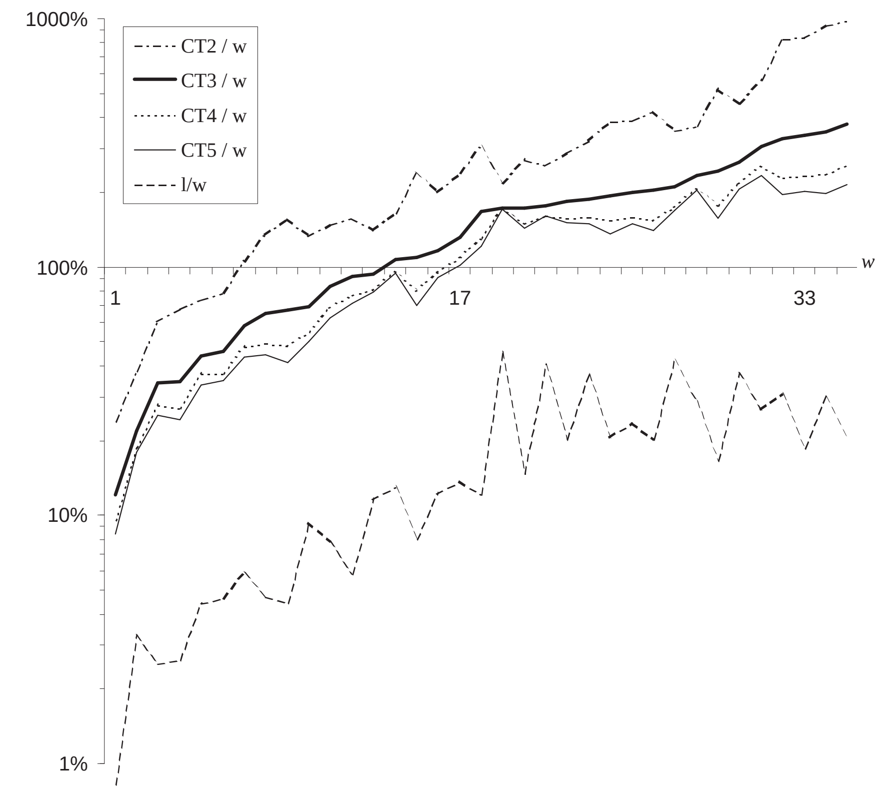


Fig. 8.1. Memory used by $CT_2$, $CT_3$, $CT_4$, $CT_5$ and optimal duplicates-elimination (**i**) relative to optimal null-elimination (**w** – marked as the 100%); hierarchies are sorted in ascending memory used by $CT_3$

mentation upon the upper bound is much greater in multiple inheritance vs. single inheritance hierarchies.

A possible explanation for this seemingly better performance in multiple inheritance hierarchies is exaggerated upper bounds. Examining Corollary 6.2, we see that the upper bounds increase with $\kappa$. Since our heuristics only finds an upper approximation of $\kappa$, it could be that the true upper bounds are actually smaller, and hence the improvement upon the upper bound is not as great.

Figure 8.2c tries to test this hypothesis, by comparing the performance on multiple inheritance hierarchies with the upper bounds obtained by assuming $\kappa = 1$ (as in single inheritance hierarchies).[7] We see that the improvement upon the upper bounds computed thus is almost the same as in single inheritance hierarchies (Figure 8.2a). Such a similarity could not be explained by an overestimation of $\kappa$.

The reason that the CT algorithms perform better than the theoretically obtained bounds is that the analysis of the CT reduction bounded the number of

[7]In fact, we used the bound for single inheritance in Table I, which is smaller by a factor of $(2\kappa)^{1-1/d}$ than the bound for multiple inheritance in Corollary 6.2.

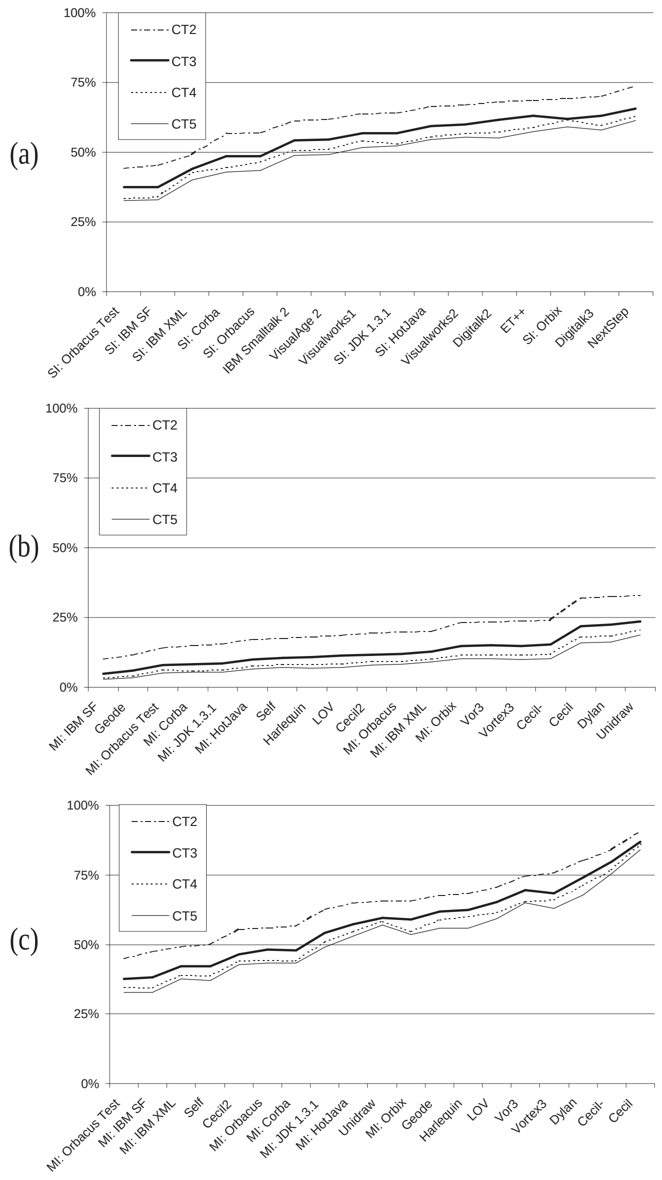


Fig. 8.2. The memory requirement of $CT_2$, $CT_3$, $CT_4$ and $CT_5$ relative to the theoretically obtained upper bounds in single inheritance hierarchies (a), multiple inheritance hierarchies where the upper bound was computed using $\overline{\kappa}$ (b), and multiple inheritance hierarchies, where the upper bound is computed as in single inheritance hierarchies ($\kappa = 1$) (c)

implementors of a master-message by the sum of implementors of its constituents, i.e.,

$$|I(\mathcal{M}_i)| = \left|\bigcup_{m\in\mathcal{M}_i} I(m)\right| \le \sum_{m\in\mathcal{M}_i} |I(m)|.$$

In fact, especially when the messages are large, the probability of finding shared elements may be significant, and the master-message is likely to have a smaller number of implementors. As a result, $\mathbf{i}'$, the number of implementations after the reduction, may be much smaller than the original value $\mathbf{i}$. For example, with $x = 29$ for $\mathrm{CT}_2$ in Digitalk3, the CT reduction transforms the problem $\langle \mathbf{n}, \mathbf{m}, \mathbf{i}\rangle = \langle 1357, 2402, 9444\rangle$ to $\langle 1357, 83, 4616\rangle$, i.e., the number of implementations decreased by a factor of more than 2. Our analysis assumed (see (4.7)) however that $\mathbf{i}' = \mathbf{i}$.

This effect increases also with slice size, which is the reason that choosing a slice size greater than the theoretical prescription may improve the performance of the reduction. In IBM SF, for example, the theoretical analysis suggested that $x_{\mathrm{OPT}} = 30$ as optimal slice size for $\mathrm{CT}_2$. However, by using instead a slice size $x = 70$, we were able to further reduce the number of cells from 3.3M to about 2.4M.

Figure 8.3 compares the actual memory used by the $\mathrm{CT}_2$ scheme with the theoretical prediction (4.10) in the Digitalk3 hierarchy. (The graphs of other hierarchies and higher order schemes are similar.) We see that the extent by which the empirical performance is superior to the theoretically obtained bound increases with the slice size.

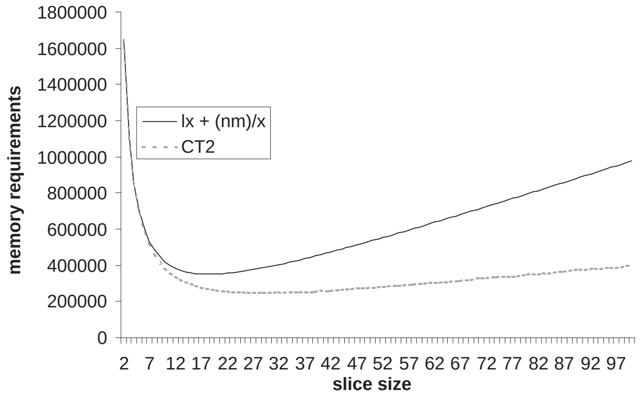


Fig. 8.3. Space requirements vs. slice size in the single inheritance hierarchy of Digitalk3 for $\mathrm{CT}_2$ and its theoretical upper bound (4.10)

## 9. OPEN PROBLEMS

The incremental $\mathrm{CT}_d$ algorithm can be generalized to the multiple inheritance setting, but there are subtle issues in the theoretical analysis of the performance of this generalization. Another natural extension to the incremental dispatching problem is in allowing also *deletion* of leaves from the hierarchy, as supported, at least in part, by JAVA. Other extensions include addition of new methods to existing types, or as it might be the case in knowledge representation, reasoning, database management, and query processing, allowing insertion of types anywhere in the hierarchy (and not just as leaves).

When comparing $\mathrm{CT}_d$ and TS we find a gap in the theoretical bounds for multiple inheritance hierarchies. $\mathrm{CT}_d$ achieves $2\kappa\mathbf{i}\lg\mathbf{m}$ space when $d = \lg\mathbf{m}$. TS however uses only $O(\kappa\mathbf{i})$ cells, while achieving $O(\lg\lg\mathbf{n})$ dispatching time. There is therefore a reason to believe that the tradeoff offered by $\mathrm{CT}_d$ can be improved, especially for higher values of $d$.

In the more pure algorithmic front, it would be both interesting and useful to generalize the PQ-tree data structure to support modifications of existing constraints when a new element is added to the universe.

Our algorithms assumed that ambiguities are resolved by an appropriate augmentation of implementors. Some OO languages resolve ambiguities based on a *linearization* of the partial order $\preceq$. COMMONLOOPS [Bobrow et al. 1986], for example, uses a global type ordering, while CLOS [Bobrow et al. 1988] uses a local type ordering. Extending our algorithms to support linearization based ambiguity resolution appears to be a worthy prospect.

Dispatching also occur in JAVA exception handling, as the following code excerpt shows.

```
try {...}
catch(D d) {...}
catch(E e) {...}
catch(A a) {...}
```

When an object $o$ of a dynamic type $a = a(o)$ is thrown in a `try` block, the program executes the first `catch` block whose argument is a supertype of $a$. Thus, each of the `catch` clauses is a subtyping test. When the number of such clauses is large, it might be worthwhile to choose the exception handler using a dispatching algorithm which will find the clause with the smallest supertype.[8] There is no possibility for ambiguity in JAVA exception handling. The reason is that a type in the `catch` block must be a subtype of the `class Throwable`, and JAVA has a single inheritance `class` hierarchy (and ambiguities cannot occur in a single inheritance hierarchy).

Finally, incorporating our algorithms into a runtime system requires careful attention to details, including selecting a heuristic for determining the optimal slice size, which might perform better than the theoretical value, a wise strategy for background copy to avoid stagnation, tweaking and fine tuning of the partitioning algorithm, etc. We leave this empirical evaluation to continuing research.

## REFERENCES

ALPERN, B., COCCHI, A., FINK, S., GROVE, D., AND LIEBER, D. 2001. Efficient implementation of Java interfaces: `invokeinterface` considered harmless. See OOPSLA'01 [2001].

[8]The above code, if read in C++ [Stroustrup 1997], leads to the same problem. This is due to the separate compilation model of C++ , in spite of the fact that exceptions are caught according to the *static* type of the thrown object.

Arnold, P. and Gosling, J. 1996. *The Java Programming Language.* The Java Series. Addison-Wesley Publishing Company, Reading, Massachusetts.

Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. 1988. Common Lisp object system specification. X3J13 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 *Proc. of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*, N. K. Meyrowitz, Ed. ACM SIGPLAN Notices 21(11), Portland, Oregon, USA, 17–29.

Booth, K. S. and Leuker, G. S. 1976. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. of Comp. and Sys. Sci. 13,* 3 (Dec.), 335–379.

Chambers, C. 1993. The Cecil language, specification and rationale. Tech. Rep. TR-93-03-05, University of Washington, Seattle.

Chambers, C. and Chen, W. 1999. Efficient multiple and predicate dispatching. See OOPSLA'99 [1999], 238–255.

Cohen, T. and Gil, J. 2000. Self-calibration of metrics of Java methods. In *Proc. of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00 Pacific).* Prentice-Hall, Englewood Cliffs, New Jersy 07632, Sydney, Australia, 94–106.

Conroy, T. J. and Pelegri-Llopart, E. 1983. An assessment of method-lookup caches for Smalltalk-80 implementations. In *Smalltalk-80: bits of history, words of advice.* Addison-Wesley Publishing Company, Reading, Massachusetts.

Cox, B. J. 1986. *Object-Oriented Programming - An Evolutionary Approach.* Addison-Wesley Publishing Company, Reading, Massachusetts.

Deutsch, P. and Schiffman, A. 1984. Efficient implementation of the Smalltalk-80 system. In *Proc. of the 11th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'84).* ACM Press, New York, NY, USA, Salt Lake City, Utah, 297–302.

Dietz, P. F. and Sleator, D. D. 1987. Two algorithms for maintaining order in a list. In *Proc. of the 19th Annual ACM Symposium on Theory of Computing (STOC'87).* ACM Press, New York, NY, USA, New York, New York, USA, 365–372.

Dietzfelbinger, M., Karlin, A. R., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., and Tarjan, R. E. 1994. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput. 23,* 4 (Aug.), 738–761.

Dixon, R., McKee, T., Vaughan, M., and Schweizer, P. 1989. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. of the 4th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'89)*, N. K. Meyrowitz, Ed. ACM SIGPLAN Notices 24(10), New Orleans, Louisiana, 211–214.

Driesen, K. 1993. Selector table indexing & sparse arrays. In *Proc. of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, A. Paepcke, Ed. ACM SIGPLAN Notices 28(10), Washington, DC, USA, 259–270.

Driesen, K. 1999. Software and hardware techniques for efficient polymorphic calls. Tech. Rep. TRCS99-24, Computer Sciences Department, University of California, Santa Barbara. July 15,.

Driesen, K. and Hölzle, U. 1995. Minimizing row displacement dispatch tables. In *Proc. of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95).* ACM SIGPLAN Notices 30(10), Austin, Texas, USA, 141–155.

Driesen, K. and Hölzle, U. 1996. The direct cost of virtual functions calls in C++. In *Proc. of the 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96).* ACM SIGPLAN Notices 31(10), San Jose, California, 306–323.

Driesen, K., Hölzle, U., and Vitek, J. 1995a. Message dispatch on modern computer architectures. Technical Report TRCS94-20, University of California, Santa Barbara. Computer Science. Feb. 9,.

Driesen, K., Hölzle, U., and Vitek, J. 1995b. Message dispatch on pipelined processors. In *Proc. of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, W. G. Olthoff, Ed. Lecture Notes in Computer Science, vol. 952. Springer Verlag, Åarhus, Denmark, 253–282.

Dujardin, E. 1996. Efficient dispatch of multimethods in constant time using dispatch trees. Technical Report RR-2892, Inria, Institut National de Recherche en Informatique et en Automatique.

Dujardin, E., Amiel, E., and Simon, E. 1998. Fast algorithms for compressed multimethod dispatch table generation. *ACM Trans. on Prog. Lang. Syst. 20,* 1 (Jan.), 116–165.

Eckel, N. and Gil, J. 2000. Empirical study of object-layout strategies and optimization techniques. In *Proc. of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, E. Bertino, Ed. Lecture Notes in Computer Science, vol. 1850. Springer Verlag, Sophia Antipolis and Cannes, France, 394–421.

Ellis, M. A. and Stroustrup, B. 1994. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts.

Ferragina, P. and Muthukrishnan, S. 1996. Efficient dynamic method-lookup for object oriented languages (extended abstract). In *Proc. of the 4th Annual European Symposium on Algorithms (ESA'96)*, J. Díaz and M. J. Serna, Eds. Lecture Notes in Computer Science, vol. 1136. Springer Verlag, Barcelona, Spain, 107–120.

Gabow, H. N., Bentley, J. L., and Tarjan, R. E. 1984. Scaling and related techniques for geometry problems. In *Proc. of the 16th Annual ACM Symposium on Theory of Computing (STOC'84).* ACM Press, New York, NY, USA, Washington, DC, United States, 135–143.

Gil, J. and Itai, A. 1998. The complexity of type analysis of object oriented programs. In *Proc. of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, E. Jul, Ed. Lecture Notes in Computer Science, vol. 1445. Springer Verlag, Brussels, Belgium, 601–634.

Gil, J. and Sweeney, P. F. 1999. Space- and time-efficient memory layout for multiple inheritance. See OOPSLA'99 [1999], 256–275.

Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley Publishing Company, Reading, Massachusetts.

Holst, W., Szafron, D., Leontiev, Y., and Pang, C. 1998. Multi-method dispatch using single-receiver projections. Tech. Rep. TR-98-03, University of Alberta, Edmonton, Alberta, Canada.

Hölzle, U., Chambers, C., and Ungar, D. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. of the 5th European Conference on Object-Oriented Programming (ECOOP'91)*, P. America, Ed. Lecture Notes in Computer Science, vol. 512. Springer Verlag, Geneva, Switzerland, 21–38.

ISE. 1997. *ISE EIFFEL The Language Reference.* ISE, Santa Barbara, CA.

Kiczales, G. and Rodriguez, L. 1990. Efficient method dispatch in PCL. In *Proc. of the ACM Conference on Lisp and Functional Programming.* ACM Press, New York, NY, USA, Nice, France, 99–105.

Muthukrishnan, S. and Müller, M. 1996. Time and space efficient method-lookup for object-oriented programs. In *Proc. of the 7thSoda, Giovanni (SODA'96).* Society for Industrial and Applied Mathematics, New York / Philadelphia, 42–51.

Naik, M. and Kumar, R. 2000. Efficient message dispatch in object-oriented systems. *ACM SIGPLAN Notices 35,* 3 (Mar.), 49–58.

OOPSLA'01 2001. *Proc. of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01).* ACM SIGPLAN Notices 36(11), Tampa Bay, Florida.

OOPSLA'99 1999. *Proc. of the 14th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99).* ACM SIGPLAN Notices 34 (10), Denver, Colorado.

Pang, C., Holst, W., Leontiev, Y., and Szafron, D. 1999. Multi-method dispatch using multiple row displacement. In *Proc. of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, R. Guerraoui, Ed. Lecture Notes in Computer Science, vol. 1628. Springer Verlag, Lisbon, Portugal, 304–328.

Pascal, A. and Royer, J. 1992. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *Proc. of the 7th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, A. Paepcke, Ed. ACM SIGPLAN Notices 27(10), Vancouver, British Columbia, Canada, 110–126.

Shalit, A. 1997. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley Publishing Company, Reading, Massachusetts.

Sleator, D. and Tarjan, R. 1985. Self-adjusting binary search trees. *Journal of the ACM 32,* 3 (July), 652–686.

Stroustrup, B. 1994. *The Design and Evolution of C++.* Addison-Wesley Publishing Company, Reading, Massachusetts.

Stroustrup, B. 1997. *The C++ Programming Language*, 3rd ed. Addison-Wesley Publishing Company, Reading, Massachusetts.

van Emde Boas, P. E. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Processing Letters 6,* 3, 80–82.

van Emde Boas, P. E., Kaas, R., and Zijlstra, E. 1977. Design and implementation of an efficient priority queue. *Math. Systems Theory 10*, 99–127.

Vitek, J. 1995. Compact dispatch tables for dynamically typed programming languages. M.S. thesis, University of Victoria.

Vitek, J. and Horspool, R. N. 1994. Taming message passing: Efficient method look-up for dynamically typed languages. In *Proc. of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, M. Tokoro and R. Pareschi, Eds. Lecture Notes in Computer Science, vol. 821. Springer Verlag, Bologna, Italy, 432–449.

Vitek, J. and Horspool, R. N. 1996. Compact dispatch tables for dynamically typed object oriented languages. In *Proc. of the 6th International Conference on Compiler Construction (CC'96)*, T. Gyimothy, Ed. Lecture Notes in Computer Science, vol. 1060. Springer Verlag, Linköping, Sweden, 309–325.

Willard, D. E. 1984. New trie data structures which support very fast search operations. *J. of Comp. and Sys. Sci. 28*, 379–394.

Zendra, O., Colnet, D., and Collin, S. 1997. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proc. of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97).* ACM SIGPLAN Notices 32(10), Atlanta, Georgia, USA, 125–141.

Zibin, Y. and Gil, J. 2001. Efficient subtyping tests with PQ-encoding. See OOPSLA'01 [2001], 96–107.

Zibin, Y. and Gil, J. 2002. Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In *Proc. of the 17th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02).* ACM SIGPLAN Notices 37(11), Seattle, Washington, 142–160.

Zibin, Y. and Gil, J. 2003. Incremental algorithms for dispatching in dynamically typed languages. In *Proc. of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03).* ACM Press, New York, NY, USA, New Orleans, Louisiana, USA, 126–138.

## A. AN ORDER-PRESERVING HEURISTIC FOR FINDING THE SLICES

In Section 3 we describe how algorithm TS uses an order-preserving heuristic as an internal procedure in partitioning $\mathcal{T}$ into slices $\mathcal{T}_1, \ldots, \mathcal{T}_\kappa$. Recall that the partitioning is built incrementally as new types are added to the hierarchy. For each new type, we try to find the first slice it can be added to, without violating the slicing property. If no such slice is found, we create a new slice.

The heuristic maintains for each slice an *ordered list* of all types in that slice. Given the order list of slice $\mathcal{T}_i$ and a type $t$, we give an algorithm whose runtime is $|\text{ancestors}(t)|$, which checks whether there is a valid list location for inserting $t$, and if so, finds it. The slicing property is slightly modified so that the sets $D_i(t)$ are consecutive in *the ordered list* of slice $\mathcal{T}_i$.

An *ordered list* is a data structure supporting two kinds of operations: Insert transactions and Order queries of the following sort. Given two positions in the list (usually as pointers to list nodes), determine which one precedes the other. In a paper entitled "Two Algorithms for Maintaining Order in a List", Dietz and Sleator [Dietz and Sleator 1987] give the best algorithm for this problem, achieving $O(1)$ worst-case time per operation. However, the authors comment that their other algorithm "is probably the best algorithm to use in practice", even though it is theoretically inferior, since its amortized[9] insertion time is $O(\log n)$. This other algorithm is based on a technique known as *self-adjustment.* In a nutshell, each list node is assigned an integer position in an increasing order, thus ORDER queries are answered in constant time. "Holes" are left to support future insertions, and if a "hole" is filled, then we redistribute the positions in some "sufficiently large and uneven" list interval. We implemented this simple algorithm and indeed found it to be very fast in practice.

Before describing the order-preserving heuristic we need to make the notions of list locations and list intervals more precise.

DEFINITION A.1. *location A* location *of a linked list is either (i) the beginning of the list, (ii) the end of the list, or (iii) any point between two consecutive nodes of the list. An* interval *in the list is a set of consecutive locations. The* boundary *of an interval comprises its first and last locations. All other locations are called the* interior *of the interval.*

The boundary usually contains two locations, the first and the last. For example, the interval marked as $D_1(\mathsf{A})$ in Figure A.1 has two interior locations and two boundary locations.

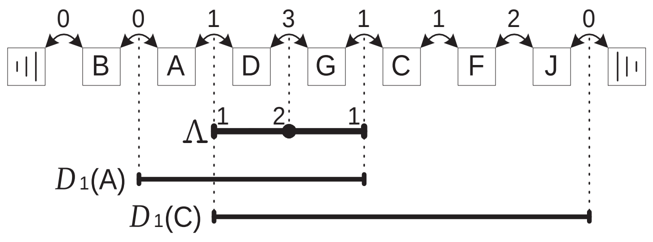


Fig. A.1. Addition of a new type to the first slice of Figure 3.2

The interior of *degenerate intervals* is empty; in such intervals the first and last locations are the same. An *empty interval* has an empty boundary and an empty interior.

DEFINITION A.2. *interval:set The* interval of the set $D_i(t)$ *in the ordered list of* $\mathcal{T}_i$ *includes* all *locations in the sub-list defined by* $D_i(t)$.

In other words, the interval of the set $D_i(t)$ also includes the location prior to the first element of $D_i(t)$, as well as the location following its last element.

[9]The *amortized time* of an operation is $c(n)$, if a sequence of $n$ operations requires at most $nc(n)$ time. The worst case time of any single operation can however be much greater than $c(n)$.

In the example, we see in Figure 3.2 that A has three descendants in the first slice, i.e., $D_1(\mathsf{A}) = \{\mathsf{A}, \mathsf{D}, \mathsf{G}\}$. In Figure A.1 we see that these three types are consecutive in the ordered list of the first slice and that the interval of $D_1(\mathsf{A})$ has four locations.

When inserting a new type $t$ to the ordered list of $\mathcal{T}_i$, we search for a list location where inserting $t$ will not violate the slicing property. Such locations must belong to the interval of $D_i(t')$ for *all* ancestors $t'$ of $t$, i.e., $t' \succeq t$. Let $\mathcal{I}$ denote the set of all such intervals, and let $\Lambda$ denote the intersection of all intervals in $\mathcal{I}$. A list locations in $\Lambda$ is called a *candidate* for inserting $t$.

Algorithmically, $\Lambda$ is computed by finding the largest first location of the intervals in $\mathcal{I}$, and the smallest last location of these intervals. (Comparisons are carried out using simple ORDER queries.) If $\Lambda$ is empty, then we conclude that $t$ cannot be inserted into $\mathcal{T}_i$. The time for computing the intersection and for checking whether it is empty is in the following asymptotic growth class:

$$O(|\text{parents}(t)|) \subseteq O(|\text{ancestors}(t)|).$$

It is also required that $t$ does not "break" any interval of $D_i(t")$, $t" \not\succeq t$. More precisely, a location is an *invalid candidate* if it belongs to the *interior* of these intervals. Although it is possible to check each candidate location $\ell \in \Lambda$ against every interval of a type $t" \in \mathcal{T} \setminus \text{ancestors}(t)$, the running time of this exhaustive search may be linear in the size of the hierarchy!

Figure A.1 shows the ordered list of the first slice of Figure 3.2. We try to insert to that slice a new type whose parents are A and C. We see the intervals of $D_1(\mathsf{A})$ and $D_1(\mathsf{C})$, and their intersection $\Lambda$. The new type can only be inserted in a candidate location $\ell \in \Lambda$. The candidate location between types D and G, for example, is invalid since it belongs to the interior of the interval of $D_1(\mathsf{D})$, and D is not an ancestor of the new type. The other two candidate locations are valid.

The *counts* $\lambda_\ell$ associated with each location in Figure A.1 are a part of a more efficient implementation for determining if a location is an invalid candidate. For each location $\ell$ in the ordered list, let $\lambda_\ell$ be the number of all intervals $D_i(t)$, such that $\ell$ is in the *interior* of $D_i(t)$. For instance, the location between types D and G has a count of 3, since it is in the interior of $D_1(\mathsf{A})$, $D_1(\mathsf{C})$ and $D_1(\mathsf{D})$.

A location $\ell$ in the *interior* of $\Lambda$ is contained in the interior of *all* intervals defined by $D_i(t')$, $t' \succeq t$. Therefore, for all candidate locations $\ell \in \Lambda$ we have that

$$\lambda_\ell \geq |\text{ancestors}(t)|. \tag{A.1}$$

The location is an invalid candidate if it is contained in the interior of any other interval, and therefore

$$\lambda_\ell > |\text{ancestors}(t)|. \tag{A.2}$$

In the example of Figure A.1, the location between types D and G is an invalid candidate, since its count is strictly higher than the number of ancestors.

We must be more careful in checking a location $\ell$ in the *boundary* of $\Lambda$. Let $I \in \mathcal{I}$ be arbitrary. Then, by definition $\ell \in I$. It is not however guaranteed that $\ell$ is in the interior of $I$. We therefore compute the number $n_\ell$ of intervals $I \in \mathcal{I}$ such that $\ell$ is in the interior of $I$. A boundary location $\ell$ is an invalid candidate if and only if

$$\lambda_\ell > n_\ell. \tag{A.3}$$

In our example, both boundary locations are valid candidates.

Although there are several special cases and many nitty-gritty details, it is a straightforward matter to update in $O(1)$ time the counts $\lambda_\ell$ with every insertion. (Note that the count may change only for two locations: before and after the insertion point.) Also, computing $n_\ell$ and checking (A.3) can be done in $O(|\text{ancestors}(t)|)$ time. It is potentially more time consuming to do the check (A.2) since we have no a priori bounds on $|\Lambda|$.

**Non-exhaustive techniques for finding a valid insertion location** We found empirically that if $t$ could not be inserted at the boundary of $\Lambda$, then it was rarely possible to insert it to the interior of $\Lambda$. For example, out of the 4339 types of JDK 1.22, only 22 types (less than 0.5%) were inserted in the interior of $\Lambda$. In all other hierarchies of our data set, the total number of such types was even smaller, and their fraction was always lower than 1%.

Therefore, it does not seem necessary to apply the check (A.2) at all. Nevertheless, we should note that there are ways of implementing (A.2) more efficiently than an exhaustive search. It follows from (A.1) and (A.2) that there exists a valid location $\ell$ in the interior of $\Lambda$ if and only if

$$\min\{\lambda_\ell \mid \ell \text{ is in the interior of } \Lambda\} = |\text{ancestors}(t)|.$$

Therefore, the problem of finding a valid location in the interior of $\Lambda$ is reduced to the famous *range minima* problem [Gabow et al. 1984]. A simple solution to the range minima problem is to maintain a balanced binary search tree (BBST) over the ordered list of $\mathcal{T}_i$, such that each internal node in it stores the minimum of $\lambda_\ell$ of all locations $\ell$ in the subtree rooted at this node. This representation adds $O(\log \mathbf{n})$ time to each insertion operation. It is standard to use this BBST to compute the minimum of any given interval. More sophisticated solutions to the range minima problem require only constant time per operation [Gabow et al. 1984]. It is not clear whether these algorithms have any practical utility.

**Inserting types with a single parent into multiple inheritance hierarchies** Finally, we present an optimization for quickly inserting a type $t$ with a *single* parent $p$. Let $i$ be the slice of $p$, i.e., $p \in \mathcal{T}_i$. Consider the ordered list of $\mathcal{T}_i$, and a list location $\ell$ immediately to the left (or to the right) of $p$. We claim that $\ell$ is valid for $t$. Assume the contrary, i.e., $\ell$ is in the *interior* of some interval defined by $D_i(t")$, and that $t"$ is *not* a supertype of $t$. Combined with the fact that $p$ is adjacent to $\ell$, we conclude that $p \in D_i(t")$, and therefore, $p \preceq t"$. Since $t \preceq p$, it follows that $t \preceq t"$, which contradicts our assumption.

Received October 2005