Fast Algorithm for Creating Space Efficient Dispatching Tables with Application to Multi-Dispatching

YOAV ZIBIN* JOSEPH (YOSSI) GIL

Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel

zyoav | yogi @ cs.technion.ac.il

Abstract

The dispatching problem can be solved very efficiently in the single-inheritance (SI) setting. In this paper we show how to extend one such solution to the multiple-inheritance (MI) setting. This generalization comes with an increase to the space requirement by a small factor of κ. This factor can be thought of as a metric of the complexity of the topology of the inheritance hierarchy.

On a data set of 35 hierarchies totaling some 64 thousand types, our dispatching data structure, based on a novel *type slicing* technique, exhibits very significant improvements over previous dispatching techniques, not only in terms of the time for creating the underlying data structure, but also in terms of total space used.

The cost is in the dispatching time, which is no longer constant, but doubly logarithmic in the number of types. Conversely, by using a simple binary search, dispatching time is logarithmic in the number of different implementations. In practice dispatching uses one indirect branch and, on average, only 2.5 binary branches.

Our results also have applications to the space-efficient implementation of the more general problem of dispatching multi-methods.

A by-product of our type slicing technique is an *incremental* algorithm for constant-time *subtyping tests* with favorable memory requirements. (The incremental version of the subtyping problem is to maintain the subtyping data structure in presence of additions of types to the inheritance hierarchy.)

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features; G.4 [**Mathematical Software**]: Algorithm design and analysis

General Terms

Algorithms, Design, Measurement, Performance, Theory

Keywords

Dispatch,Hierarchy,Incremental,Message,Subtyping,Type slicing

*Contact author

OOPSLA'02, November 4–8, 2002, Seattle, Washington, USA.

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 [24] 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 [14, 16, 20–26, 35, 43, 45, 51, 52, 59, 65–67, 69]. The principal optimization objective adopted by most of this prior research was a compact representation of the dispatching data structure, while maintaining a small, preferably constant, dispatch time. A heavy toll incurred in many cases was the time required for creating the dispatching data structure.

This research revisits the problem, trying to optimize another important complexity metric, *creation time*, i.e., the time required for generating the dispatching data structure. Our motivation is the staggering importance of dynamic compilation and re-compilation systems, as found in JAVA. Previous work tended to be conceptually locked in the static compilation model, with few reports on creation time values; when reported, these times were measured in seconds for modest size hierarchies.

Our novel *type slicing* technique gives rise to a very fast algorithm for creating space efficient dispatching data structure. The creation time is improved by one, two and sometimes three orders of magnitude compared to the famous *row displacement* (RD) algorithm [23]. In a collection of 35 hierarchies, totaling over 60,000 types, the slowest runtime of our 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.

These timing results make it feasible to regenerate the entire dis-

patching data structure each time a new class is loaded. A more demanding requirement from an algorithm for creating a dispatching data structure is that it is *truly incremental*. By this we mean that the asymptotic time complexity of creating the data structure in a piecemeal feed of the type hierarchy, is the same as in the case that the entire hierarchy is supplied up front. In a theoretical adjunct paper [70], we show that there exists a truly incremental version of our algorithm. Currently we work on determining the incurred overheads in a practical implementation of this theoretical result.

The algorithm presented here does not only improve the creation time. Its space requirement improves those of RD (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 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) [33] standard implementation strategy of C++ [61] in the single inheritance (SI) setting. Note that in contrast with our results and most other dispatching algorithms, the VFT technique is valid only in *statically typed* languages [67]. 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 [69], 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 [22, 24–26]. 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 look-ups, and even linear searches [2,10,43,52] for dispatching.

Informally, we can say that our algorithm generalizes the linear space *interval containment* algorithm [35, 51] which is restricted to the SI setting. Our main theoretical result is that the generalization to the multiple inheritance (MI) case comes with a κ (the number of slices) factor increase of space. This factor depends only on the *topology of the MI hierarchy*, and can be thought of as a metric of its complexity. In practice this factor is small, but in arbitrary hierarchies it might be in the order of the number of types.

In all SI hierarchies, $\kappa \equiv 1$. We provide a heuristic for finding an upper bound of κ, and an actual implementation of the generalization. In our data set of 19 MI hierarchies the median value of κ is 6.5, the average is 7.3, and the maximum is 19. We stress that the space increase is by a factor of at most κ; in practice, we find much better results.

Our dispatching technique has also applications to space-efficient implementation of *multi-dispatching*.

Finally, our type slicing technique also provides an *incremental* algorithm for constant-time *subtyping tests* with favorable memory requirements. We provide theoretical analysis of our algorithm, as well as practical evidence that our algorithm is fast even when compared to previous non-incremental algorithms.

1.1 Preliminary Definitions

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*. Formally, a *hierarchy* is a partially ordered set $(\mathcal{T}, \preceq)$ where $\mathcal{T}$ is a set of types and $\preceq$ is a reflexive, transitive and anti-symmetric *subtype relation*. If a and b are types, 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*).

Similarly, we abstract away from the nomenclature of different languages, and use the term *message* for the unique identifier of a family of *methods* (also called member functions, operations, features, implementations, etc.). A message, which is sometimes called a *selector* (in e.g., SMALLTALK [40] or OBJECTIVE-C [15]) or a *signature* (in e.g., JAVA [5] or C++ [61]), may include, depending on the programming language, components such as name, arity, and even the type of parameters. We will use the terms *message* and *selector* interchangeably. Note that a consequence of feature renaming in EIFFEL [49], 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 type of the receiver* must take place at runtime to find and invoke the implementation which is most appropriate for the receiver's type.

We use the following notation. A method implementation in type a for a message μ is denoted $\mu(a)$. A *method family* F_μ,

$$F_\mu = \{\mu(a_1), \ldots, \mu(a_{f_\mu})\},$$

is the set of all methods of a certain message. Given two methods $\mu(a), \mu(b) \in F_\mu$, we say that $\mu(a)$ is *more specific* than $\mu(b)$ if $a \preceq b$. Given a method $\mu(b)$, we say that $\mu(b)$ is *applicable* to type a if $a \preceq b$. Finally, given a message μ and a type a, a *dispatching query* $\mathsf{dispatch}(\mu, a)$, results in *the most specific applicable method* $\mu(b) \in F_\mu$.

In a slight abuse of notation, we can safely omit the μ symbol in writing the set F_μ as

$$F_\mu = \{a_1, \ldots, a_{f_\mu}\} \subseteq \mathcal{T}.$$

No confusion will arise since a dispatch query $\mathsf{dispatch}(\mu, a)$ will return the lowest type $b \in F_\mu$ such that $b \succeq a$.

In dynamically typed languages there are two more possible outcomes: an error message to say that the *message is not under-*

stood (MNU), and an error message to say that the *message is ambiguous*, i.e., there are two or more most specific methods for the type of the receiving object. On the other hand, the type checker of statically typed languages makes sure at compile time that dispatching never results in an *error message*.

Figure 1.1 depicts a hierarchy which serves as the running example of this paper. Type names are written with uppercase letters; messages with lower case letters.

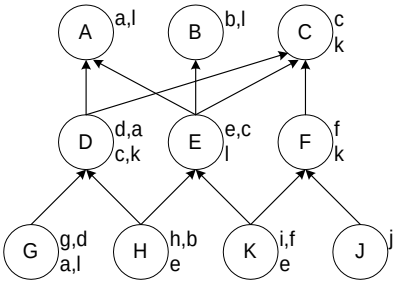


Figure 1.1: A small example of a hierarchy and the methods implemented in each type

We see in the figure that for message c:

$$
\begin{aligned}
F_{\mathsf{c}} &= \{\mathsf{c(C)}, \mathsf{c(D)}, \mathsf{c(E)}\} \\
\mathsf{dispatch(c, K)} &= \mathsf{c(E)} \\
\mathsf{dispatch(c, B)} &= \textit{message not understood} \text{ (MNU)} \\
\mathsf{dispatch(c, H)} &= \textit{message ambiguous}
\end{aligned}
\tag{1.1}
$$

It would therefore be a compilation error in statically typed language 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 [38] 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 method families. In the example, we add c(H) to F_{c} since dispatch(c, H) resulted in *message ambiguous*. As in previous work [42, 57] 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* (MI) hierarchy, since, e.g., type D has two parents: A and C. *Single-inheritance* (SI), in which each type has at most one parent, is mandated by languages such as SMALLTALK and OBJECTIVE-C. The fact that SI hierarchies take a simple forest topology, makes SI an important special case, for which very efficient algorithms exist. The general case of multiple-inheritance is more difficult, and will be our main concern here.

DEFINITION 1.1. *Given a hierarchy $(\mathcal{T}, \preceq)$, a set of messages $\mathcal{M}$, and a method family $F_\mu \subseteq \mathcal{T}$ for each message $\mu \in \mathcal{M}$, the* dispatching problem *is to* encode *the hierarchy in a data structure supporting dispatching queries of the form* dispatch(μ, t), *where* $\mu \in \mathcal{M}, t \in \mathcal{T}$.

From a practical point of view 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. Also, the message is given at runtime as an integer selector. We usually 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 most specific applicable method 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. We would like to express these metrics as a function of the following parameters of the problem:

- The number of types in the hierarchy
$$n \equiv |\mathcal{T}|. \tag{1.2}$$
- The number of different messages that can be sent during runtime
$$m \equiv |\mathcal{M}|. \tag{1.3}$$
- The total number of different method implementations
$$\ell \equiv \sum_{\mu \in \mathcal{M}} |F_\mu|. \tag{1.4}$$
- The number of valid message-type combinations, i.e., combinations which do not result in *message not understood*
$$w \equiv |\{\langle \mu, t \rangle \mid \mathsf{dispatch}(\mu, t) \neq \text{MNU}\}|. \tag{1.5}$$

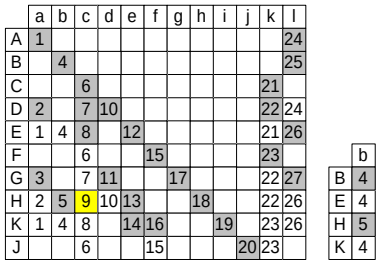
1.2 Simple algorithms

The most obvious solution to the dispatching problem is in a $n \times m$ *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 1.2(a), where the $nm - w$ type-message pairs which result in *message not understood* are represented as empty entries. The figure depicts in grey all ℓ 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. The cell corresponding to $\langle \mathsf{H}, \mathsf{c} \rangle$ is rendered in a lighter shade of gray since c(H) was added to F_{c} to resolve an ambiguity.

In the matrix representation queries are answered by a quick indexing operation. However, the space consumption is inhibitivly 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:



	a	b	c	d	e	f	g	h	i	j	k	l
A	1											24
B		4										25
C			6								21	
D	2		7	10							22	24
E	1	4	8		12						21	26
F			6			15					23	
G	3		7	11			17				22	27
H	2	5	9	10	13			18			22	26
K	1	4	8		14	16			19		23	26
J			6			15				20	23	

(a)

	b
B	4
E	4
H	5
K	4

(b)

Figure 1.2: (a) The dispatching matrix, and (b) the sorted dictionary for message b

Null elimination There is much empirical evidence to show that dispatching matrices are very sparse. *Null elimination*, which was the objective of almost all previous work, is the attempt to store only the non-empty elements in the matrix.

The ratio $(nm)/w$ is an upper bound on the compression rate which null elimination might achieve. The matrix of Figure 1.2(a) 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 1.2(b). 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) [19] 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 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.

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.

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* ℓ entries of the dispatching matrix. Therefore, the compression factor of duplicates elimination is at most $(nm)/\ell$, which was around 725 in our benchmarks.

The ratio w/ℓ 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 1.2(a) there are 27 distinct entries, i.e., $\ell = 27$, 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 $O(n)$, since each dispatch must traverse all the ancestors of a receiver in order to find the most specific applicable method implementation. Sophisticated caching algorithms make the typical case more tolerable than what the worst case indicates. This is the implementation in languages such as SMALLTALK.

Our challenge here is to come as close as possible to optimal duplicates elimination, i.e., space linear in the number of implementations ℓ, while still maintaining small, preferably constant, query time.

Outline The remainder of this article is organized as follows. A survey of prior dispatching techniques including a detailed description of interval containment is the subject of Section 2. Our new slicing technique is described in Section 3. The data set of the 35 hierarchies used in our benchmarking, collected from both single and multiple dispatching languages, is presented in Section 4. Section 5 presents the experimental results, comparing the performance of our algorithm with those of previous algorithms. The application of our results to the problem of multiple dispatching is presented in Section 6. An incremental algorithm for constant-time subtyping tests is the concern of Section 7. Finally, Section 8 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., [14, 16, 43]).

VFT: *Virtual Function Tables* [33] As mentioned above, the VFT technique is valid only in statically typed languages [67]. In an SI 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 MI version of the VFT is much more complicated than the SI 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* [39]. It was shown [29] 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*.[1]

RD: *Row Displacement* [21, 23] Another null elimination tech-

[1] 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.

nique is due to Driesen [21] 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 [23] 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.[2] We stress that duplicates elimination (which we use) does not suffer from this limitation.

CT: *Compact dispatch Tables* [65–67] 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* $\mathcal{M}_1, \ldots, \mathcal{M}_k$. Slicing breaks the dispatching matrix into k sub-matrices, also called *chunks*. Identical rows within each chunk are then merged. Each type t has an array r_t of size k. Entry $r_t[i]$ 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.

Our MI algorithms adopt the slicing idea. However, we slice the set of types rather than the set of messages.

SC: *Selector Coloring* [20, 59] 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 $n \times k$.

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.

***Jalapeño* [2]** JAVA's `invokeinterface` bytecode instruction, i.e., messages sent to receivers whose static type is an **`interface`**, cannot be implemented using the VFT technique. Jalapeño, an IBM implementation of JAVA virtual machine, uses a fast incremental variant of SC in realizing these instructions. Messages are hashed into k slices, where k is an a-priori fixed number. Each type has an *interface method table* of length 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 nk for the tables, plus $O(w)$ memory for conflict resolution.

***Interval Containment for SI hierarchies* [35, 51]** Interval containment achieves optimal duplicates elimination at the cost of non-constant dispatch time. Our dispatching technique is a generalization of interval containment for MI 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 method family F_μ, defines a set of intervals, one for each method $\mu(t) \in F_\mu$.

Figure 2.1(a) shows a tree hierarchy with three implementations of a message μ: μ_1 in A, μ_2 in B, and μ_3 in F. Then, as can be seen in Figure 2.1(b), the methods μ_1, μ_2, and μ_3 define three intervals in the preorder traversal: $[1, 7]$, $[5, 7]$, and $[3, 3]$, respectively. The intersections of those intervals partition the types into four segments: $[1, 2]$, $[3, 3]$, $[4, 4]$, and $[5, 7]$, which correspond to the methods: μ_1, μ_3, μ_1, and μ_2, respectively. The dispatch of message μ on any given type depends only on 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 execute method μ_2.

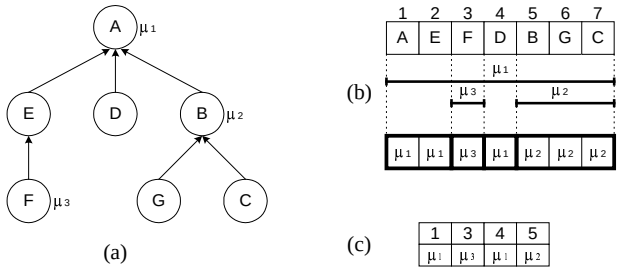


Figure 2.1: (a) A method family $F_\mu = \{\mu_1, \mu_2, \mu_3\}$ in a tree hierarchy, (b) the intervals and segments F_μ defines, and (c) the representation of F_μ as a sorted dictionary

If $|F_\mu| = f_\mu$ then there are f_μ intervals which partition $\mathcal{T}$ into at most $2f_\mu + 1$ *segments*, where all types in a segment execute the same method in response to message μ. Family F_μ is represented as a sorted dictionary, mapping segments' starting point to methods. In our example, Figure 2.1(c) shows a sorted dictionary that represents the segment partitioning. This dictionary serves as the dispatching table for μ.

Note that the sorted dictionary representation is linear in f_μ. The total memory for representing all method families is therefore $O(\ell)$. In fact, the number of memory cells required by this representation is at most

$$\sum_{\mu \in \mathcal{M}} 2(2f_\mu + 1) = 2m + 4 \sum_{\mu \in \mathcal{M}} f_\mu = 2m + 4\ell.$$

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 $S \subseteq [1, \ldots, n]$, 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 $x \in [1, \ldots, n]$. Let $s = |S|$. In our case, s, which is smaller than twice the number of different implementations, is typically much smaller than n. We will therefore be more interested

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

by algorithms whose resource demands are dependent on s, rather than on 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* [68], which achieves $O(\sqrt{\log n})$ time while still maintaining the space linear in s. Stratified trees, also called *van Emde Boas data structure* [63, 64], offer a different tradeoff, with space linear in n and time $O(\log \log n)$. 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. Dispatching using Type Slicing (TS)

Our dispatching technique for MI hierarchies is a generalization of *interval containment* for SI 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 MI 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.

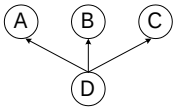


Figure 3.1: The smallest MI hierarchy for which no ordering exists where all descendants of any type are consecutive

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.,

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

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

$$\begin{aligned} \mathcal{T}_1 &= \{\mathsf{B}, \mathsf{A}, \mathsf{D}, \mathsf{G}, \mathsf{C}, \mathsf{F}, \mathsf{J}\}, \\ \mathcal{T}_2 &= \{\mathsf{E}, \mathsf{H}, \mathsf{K}\}. \end{aligned} \tag{3.1}$$

The grayed squares 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

$$\begin{aligned} D_1(\mathsf{A}) &= \{\mathsf{A}, \mathsf{D}, \mathsf{G}\}, \\ D_2(\mathsf{A}) &= \{\mathsf{E}, \mathsf{H}, \mathsf{K}\}. \end{aligned} \tag{3.2}$$

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

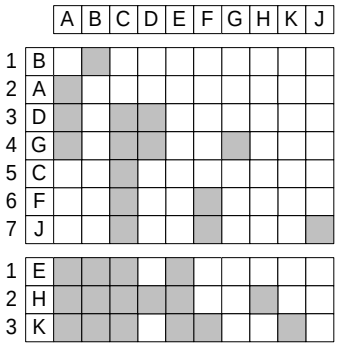


Figure 3.2: Type slicing for the MI hierarchy of Figure 1.1

that the grayed entries are consecutive within each chunk. For instance, in Figure 3.2 the sets of (3.2) define the *intervals*

$$\begin{aligned} D_1(\mathsf{A}) &= [2, 4], \\ D_2(\mathsf{A}) &= [1, 3]. \end{aligned} \tag{3.3}$$

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 $t \in \mathcal{T}$.*

In this data structure representation 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 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*.

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 κ, 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 [8, 71] can be used to check whether there is *any* ordering of $\mathcal{T}_i \cup \{t\}$ which satisfies the slicing property, in $O(n \cdot |\mathcal{T}_i|)$ time. In inserting n types using this strategy, the total time might be cubic in 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 the heuristic are presented in Appendix A.

Dispatching using type slicing Given a type t and a method family F_μ, $|F_\mu| = f_\mu$, a dispatching query returns the most specific method $\mu(t') \in F_\mu$ such that $t \preceq t'$. Let $\mathcal{T}_i$ be the slice of t. A method $\mu(t')$ is applicable for t iff $t \in D_i(t')$. We therefore must consider all intervals of $D_i(t')$, $D_i(t') \neq \emptyset$, where t' has an implementation of μ. Since there are at most f_μ such intervals, we

obtain a partition of $\mathcal{T}_i$ into $2f_\mu + 1$ 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 method family

$$F_{\mathsf{c}} = \{\mathsf{c(C)}, \mathsf{c(D)}, \mathsf{c(E)}, \mathsf{c(H)}\} \triangleq \{\mathsf{c}_6, \mathsf{c}_7, \mathsf{c}_8, \mathsf{c}_9\}$$

in the hierarchy of Figure 1.1. Consider, for example, the first slice. Only methods c_6 and c_7 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 the methods: *message not understood*, c_7, and c_6, respectively.

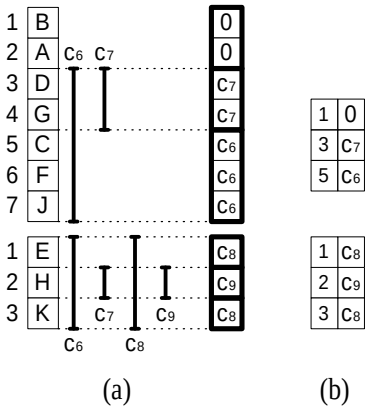


Figure 3.3: (a) The intervals and segments of message c in the two slices of Figure 3.2, and (b) the message representation in each slice

In general, a message μ 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 $\mu \in \mathcal{M}$ 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 μ, and then
3. carrying on as in SI in a search of id_t in this data structure to find the appropriate implementation.

Thus, dispatching in MI hierarchies requires only two more steps in comparison to dispatching in SI hierarchies. The space requirement in MI hierarchies increases by a factor of at most κ. 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 κ. In our MI benchmarks, κ 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 method $\mu(t)$ defined an interval for each slice. Therefore, $D_i(t)$ must be consecutive in $\mathcal{T}_i$, *only* for those types t which defined some method $\mu(t)$.

Formally, we say that a type t is *significant* if it defined some method implementation, and redefine the slicing property as follows:

> *There is an ordering of $\mathcal{T}_i$ in which $D_i(t)$ is consecutive for all 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. 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 MI-hierarchies [23].
2. The eight SMALLTALK, OBJECTIVE-C and C++ hierarchies used for benchmarking RD and CT [67] in SI-hierarchies.
3. The ensemble of seven JAVA hierarchies used in the definition of the "common programming practice" [13], augmented by version 1.3.1 of the Java Development Kit. Each of these eight hierarchies, was also used both for benchmarking MI dispatching algorithms and, after pruning interfaces, for benchmarking SI dispatching algorithms.
4. The two CECIL [9] and DYLAN [60] hierarchies used in all benchmarking of multiple dispatching algorithms [27, 28, 42, 57] contributed by Eric Dujardin.

   We used these hierarchies for benchmarking single-dispatch algorithms, by projecting each multi-method on each of its arguments. (The details are in Section 6.)
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.

In total, the data set for benchmarking dispatching algorithms has 16 SI-hierarchies with 29,162 types, 12 MI-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 [29] 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 method families, i.e., families of exactly one method, were eliminated from the data set prior to running the experiments, since no runtime dispatching is required for such families.

We stress that by eliminating degenerate families we only made the input *more difficult* for our new dispatching algorithm and any other duplicates elimination scheme, including CT. The reason is that degenerate families, in which there are only two distinct values in their corresponding columns, have the greatest potential for duplicates elimination.

Table 1 gives a summary of the pruned hierarchies. The three blocks in the table correspond to SI-, MI- 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 1, this row indicates that in total the dispatching benchmark spanned some 64 thousand types and 70 thousand messages.

The ℓ/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 ℓ/m column we find that method families tend to be small, with average values of around four to six methods in a family in most hierarchies. We note that the average number of comparisons in a binary search in method families is no greater than $\lceil \log_2 \frac{\ell}{m} \rceil$. The reason is that the geometrical mean is no greater than the arithmetical mean, and therefore

$$\begin{aligned} \frac{1}{m} \sum_{\mu \in \mathcal{M}} \log_2 |F_\mu| &= \log_2 \left(\prod_{\mu \in \mathcal{M}} |F_\mu| \right)^{\frac{1}{m}} \\ &\leq \log_2 \left(\frac{1}{m} \sum_{\mu \in \mathcal{M}} |F_\mu| \right) \\ &= \log_2 \frac{\ell}{m}. \end{aligned} \tag{4.1}$$

Thus, just by inspecting the ℓ/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 empty. 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. (The 1,249.0 bound for the universal hierarchy is meaningless.)

	Hierarchy	n	m	ℓ/n	ℓ/m	$(nm)/w$	w/ℓ
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	1,242.0	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 1: Statistical and topological properties of the 35 hierarchies used in benchmarking dispatching algorithms

How much can duplicates elimination improve on an *optimal* null elimination? The answer is in the w/ℓ 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.

5. Experimental Results

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.

Space requirement 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 5.1. *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/ℓ.*

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 2 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.

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

[a] A lower bound on SC redundancy factor
[b] The space requirements of TS in kilo-bytes

Table 2: The redundancy factor of different dispatching algorithms and the total memory requirements of TS in kilo-bytes

Algorithms CT, 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, CT, 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 [67]). We then applied the SC lower bound estimate in each chunk.

The results of the VFT technique are calculated in the traditional manner [23], 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 [29].

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 SI-hierarchies and several MI-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 main 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) MI 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 *SI hierarchies*. The RD algorithm is usually the second best for *MI hierarchies*, while CT performs well on *multiple dispatch hierarchies*.

We remind the reader that the comparison presented in Table 2 is

different than that reported in the literature, since even though we used the same hierarchies, we eliminated degenerate families from the benchmark. Different algorithms compress such families to different levels.

Creation time Table 3 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 [67] 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.

	Hierarchy	RD	**TS**	**TS^{PQ}**
Single Inheritance	Visualworks1	54	5	261
	Visualworks2	250	13	2,430
	Digitalk2	54	3	130
	Digitalk3	281	9	1,040
	IBM Smalltalk 2	3,430	15	3,790
	VisualAge 2	18,800	24	8,160
	NextStep	13	1	50
	ET++	9	1	60
	SI: JDK 1.3.1	162	26	33,600
	SI: Corba	11	3	561
	SI: HotJava	22	2	211
	SI: IBM SF	1,620	69	30,300
	SI: IBM XML	1	1	10
	SI: Orbacus	27	4	401
	SI: Orbacus Test	12	1	110
	SI: Orbix	18	3	571
Multiple Inheritance	Self	242	30	27,600
	Unidraw	9	3	371
	LOV	18	5	3,430
	Geode	182	38	66,800
	MI: JDK 1.3.1	240	88	324,000
	MI: Corba	26	9	10,400
	MI: HotJava	30	7	3,390
	MI: IBM SF	903	307	1,740,000
	MI: IBM XML	2	1	140
	MI: Orbacus	31	11	12,700
	MI: Orbacus Test	11	4	1,740
	MI: Orbix	31	14	12,400
Multiple Dispatching	Cecil	57	9	6,410
	Dylan	48	5	1,870
	Cecil-	18	4	2,490
	Cecil2	16	1	2,650
	Harlequin	23	2	2,710
	Vor3	24	9	23,400
	Vortex3	394	11	42,100

Table 3: Encoding creation time in milliseconds, on a 900 Mhz Pentium III, of different dispatching algorithms

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.

Dispatch time Recall that in TS we associate with each message an array of the κ addresses of the appropriate binary search code in each slice. The main performance metric of such code is the number of conditionals.

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 families, i.e., families 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 jump by coalescing the jump and the binary branch code of each message. If this is implemented, the average number of comparisons now ranges between 2.5 to 3.8; the median becomes 2.9. We see that the indirect jump 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 κ addresses which was associated with each message.

6. Multiple Dispatching

Interestingly, our results have applications also to the more general multiple dispatching problem.

6.1 Introduction to Multiple Dispatching

Remember that in ordinary dispatching, the method to be invoked depends only on the type of a single receiver. In contrast to this single dispatching, *multiple dispatching* is the dispatch over several arguments. Consider, for example, a geometric modeling application, in which shapes such as rectangles, triangles, circles, are to be depicted on various drawing canvases, such as screens, printers and files. Then, the appropriate drawing method is to be selected according to both the shape and canvas kind. Languages such as POLYGLOT [1], KEA [50], COMMONLOOPS [7], CLOS [6], CECIL [9], DYLAN [60] make only a partial list of the new generation OO languages which support multiple-dispatching in the form of *multi-methods*.

Even though multi-methods are believed to be more expressive, natural and readable than *mono-methods*, they did not find their way into more mainstream languages. One of the reasons is probably the perceived cost of implementation. The prospect of efficient multiple dispatching drew much research effort [4, 10, 11, 27, 28, 36, 42, 46, 57]. The contribution that this paper makes is in improving the memory requirements of two existing practical techniques of multiple-dispatching.

We note that multiple dispatching can be viewed as dispatching over tuples. Given a hierarchy

$$\mathcal{H} = (\mathcal{T}, \preceq),$$

we define the *c-tuple* hierarchy, denoted $\mathcal{H}^c = (\mathcal{T}^c, \preceq')$, where

$$(a_1, \ldots, a_c) \preceq' (b_1, \ldots, b_c) \text{ iff } \forall i = 1, \ldots, c : a_i \preceq b_i.$$

A multi-method $\mu(t_1, \ldots, t_c)$ in $\mathcal{H}$, can be thought of as a mono-method of the *multi-type* $(t_1, \ldots, t_c) \in \mathcal{T}^c$. However, this perspective does not lead to any efficient algorithms because of the size of $\mathcal{H}^c$.

6.2 Review of Algorithms for Multiple Dispatching

The best practical techniques for multiple dispatching known today are *Compressed N-dimensional Tables* (CNT) [4,28,46] and *Single-Receiver Projections* (SRP) [42]. Both techniques begin with the

same *mono-dispatch stage*, in which c independent single-dispatch queries are executed for a multi-method of arity c. The results of these queries are then used in the *resolution stage* which is technique specific.

The mono-dispatch stage quickly reduces the number of candidate methods using the following observation. For a given multi-method family F_μ, $|F_\mu| = f_\mu$, let $T_i(\mu)$ be the set of all types which occur in the i^{th} position in an implementation of μ. Then, the dispatching of μ on a multi-type $(t_1, \ldots, t_c)$, can be made easier by first using a single-dispatch algorithm for finding for each $i = 1, \ldots, c$, the most specific $t'_i \in T_i(\mu)$, such that $t'_i \succeq t_i$.

Consider, for example, the multi-method family

$$F_\mu = \{\mu(\mathsf{A},\mathsf{A}), \mu(\mathsf{A},\mathsf{D}), \mu(\mathsf{B},\mathsf{D}), \mu(\mathsf{E},\mathsf{D})\}, \tag{6.1}$$

defined over the type hierarchy of our running example (Figure 1.1). Then,

$$\begin{aligned} T_1(\mu) &= \{\mathsf{A},\mathsf{B},\mathsf{E}\},\\ T_2(\mu) &= \{\mathsf{A},\mathsf{D}\}. \end{aligned} \tag{6.2}$$

In dispatching the multi-type (G,H), the mono-dispatch stage first determines that $t'_1 = \mathsf{A}$ and that $t'_2 = \mathsf{D}$. The resolution stage then continues with the multi-type (A,D).

FACT 6.1. (DUJARDIN ET AL. [28, P. 129]). *If dispatching never result in an error message then there is always a unique such t'_i. Further, dispatching on $(t_1, \ldots, t_c)$ is the same as dispatching on $(t'_1, \ldots, t'_c)$, i.e.,*

$$\mathsf{dispatch}(\mu, t_1, \ldots, t_c) = \mathsf{dispatch}(\mu, t'_1, \ldots, t'_c).$$

The CNT technique creates a c-dimensional dispatch table with entries for each multi-type in the cartesian product

$$T_1(\mu) \times \cdots \times T_c(\mu).$$

The dispatching table for the multi-method family of (6.1) is shown in Table 4.

	A	D
A	$\mu(\mathsf{A},\mathsf{A})$	$\mu(\mathsf{A},\mathsf{D})$
B	null	$\mu(\mathsf{B},\mathsf{D})$
E	$\mu(\mathsf{A},\mathsf{A})$	$\mu(\mathsf{E},\mathsf{D})$

Table 4: CNT representation for the multi-method family of (6.1)

The resolution stage in CNT requires only $O(c)$ time. The number of memory cells for representing the multi-dimensional dispatch table is reduced from $O(n^c)$ to

$$|T_1(\mu)| \times \cdots \times |T_c(\mu)| = O((f_\mu)^c), \tag{6.3}$$

which might still be very large.

SRP gives a different tradeoff in which the time of resolution increases to $O(cf_\mu)$, while the space (in bits) is

$$f_\mu\,(|T_1(\mu)| + \cdots + |T_c(\mu)|) = O(c(f_\mu)^2). \tag{6.4}$$

An asymptotic comparison of the bound (6.4) with $O((f_\mu)^c \log f_\mu)$, the bound on number of bits in the CNT representation obtained from (6.3), as well as practical experience, shows that SRP is usually more space efficient than CNT.

SRP uses an encoding of subsets of F_μ as bit vectors of length f_μ. The positions in this bit vector are given in a topological order, so that more specific methods are positioned first. For all $i = 1, \ldots, c$, and for all $t \in T_i(\mu)$, the technique encodes the set of all multi-methods which might be applicable if the i^{th} argument is of type t, i.e., the set

$$\{\mu(t_1, \ldots, t_i, \ldots, t_c) \in F_\mu \mid t \preceq t_i\}. \tag{6.5}$$

At the resolution phase, the intersection of all c sets defined by (6.5) is computed by *AND*ing the bit-vector representation of these sets. The most specific method in the intersection is then found using a *find-first-set* operation, which can often be implemented as a single machine instruction.

Assuming that the multi-methods in (6.1) are positioned in the following order:

$$\{\mu(\mathsf{E},\mathsf{D}), \mu(\mathsf{B},\mathsf{D}), \mu(\mathsf{A},\mathsf{D}), \mu(\mathsf{A},\mathsf{A})\},$$

then the bit-vectors assigned with the sets $T_1(\mu)$ and $T_2(\mu)$ of (6.2) are shown in Table 6.

$T_1(\mu)$	vector
A	0011
B	0100
E	1111

$T_2(\mu)$	vector
A	0001
D	1111

Table 6: SRP representation for the multi-method family of (6.1)

6.3 Reducing the Space Requirement of the Mono-dispatch Stage with Type Slicing

We applied the mono-dispatch reduction on multiple-dispatching benchmarks, drawn from various languages. The resulting hierarchies were used as benchmarks to single-dispatching algorithms. Degenerate multi-method families, and degenerate arguments were removed.[3]

The mono-dispatch stage in SRP or CNT [4, 28, 42, 46] is currently carried out using either the technique of SC or RD for single dispatching, which are both null elimination schemes.

Table 5 compares the average number of *bits* per multi-method family for the mono-dispatch stage and the resolution stage. The *mono-dispatch stage* is carried out using either our type slicing (TS) technique, or using an ideal null elimination scheme which requires w entries. The *resolution stage* is carried out using either SRP or CNT. The results were broken down by arity of the multi-method, which ranged between 2 to 4.

The space requirements presented in the table are in a way a lower bound, since we used a bit granularity rather than byte. For instance, in the ideal null elimination scheme, an entry for message μ occupies $\lceil \log_2 |F_\mu| \rceil$ bits. Also, in the c-dimensional matrix of CNT, the number of bits in one matrix entry is not necessarily divisible by 8. The same is true for the bit-vector size in SRP, or the

[3] A family F_μ is degenerate if $|F_\mu| = 1$. The i^{th} argument is degenerate if $|T_i(\mu)| = 1$.

Hierarchy	Arity											
	2				3				4			
	TS	w^a	SRP	CNT	TS	w^a	SRP	CNT	TS	w^a	SRP	CNT
Cecil	296	718	234	110	380	1,718	168	501	263	2,798	16	16
Dylan	228	1,100	115	142	496	3,903	609	8,906	697	4,031	801	38,475
Cecil-	269	177	180	473	327	137	30	73	408	272	16	16
Cecil2	241	373	270	644	286	740	30	73	352	272	16	16
Harlequin	283	466	148	185	284	471	123	238	0	0	0	0
Vor3	330	303	347	925	485	278	666	1,449	328	320	16	16
Vortex3	351	2,100	294	720	469	5,996	302	828	472	320	16	16

[a] An ideal null elimination scheme

Table 5: Average number of bits per family for the mono-dispatch stage and the resolution stage

size of entries in our array implementation of TS. Therefore, shift and mask operations are needed in order for the assumption to hold.

We observe the following in the table:

1. The *relative advantage* of SRP over CNT (in the *resolution stage*) increases with the arity. For example, in the DYLAN hierarchy SRP improves on CNT by 19% for an arity of 2, by 93% for an arity of 3, and by 98% for an arity of 4. This fact is in agreement with the theoretical analysis of SRP in (6.4) and of CNT in (6.3).

2. The space requirement of the *mono-dispatch stage* using an ideal null elimination scheme dominates those of the *resolution stage* using SRP. In other words, the benefits of a space efficient resolution stage are wasted if we simply use RD or SC in the mono-dispatch stage.

   The reason that null elimination performs so poorly in the multiple dispatching benchmark is that many multi-methods have *root-type arguments* to handle unexpected combination of arguments. Null elimination schemes cannot compress such multi-methods. Therefore, it was even suggested [42] to compare different algorithms on data sets without such multi-methods. Duplicates elimination schemes, such as CT and our TS, performs especially well on such cases.

3. Using TS instead of a null elimination scheme reduces, in most cases, the space requirement of the *mono-dispatch stage*. In the Cecil- and Vor3 hierarchies (and in the Cecil2 hierarchy for the case of an arity 4) an ideal null elimination scheme is better than TS. However, in the other five hierarchies TS is better by as much as 92%.

7. Incremental Algorithm for Constant-time Subtyping Tests

7.1 Introduction

In the *incremental version* of the subtyping problem the type hierarchy may grow during program execution when new types are added as leaves. Such additions are allowed, e.g., in JAVA [5]. This dynamic hierarchy model gains increasing popularity since it shortens the initialization time of applications loaded from a local storage device, such as a disk, and even more so from a remote device such as the network. Also, in mission critical systems, in which an application cannot be restarted, it is convenient to make updates to the running software by simply loading more types.

Almost all previous work on the subtyping problem [34, 44, 47, 48, 58, 62] mention an incremental extension of the proposed algorithm. However, these after thought additions invariably suffer from the limitation that the total time for building the associated data structures is much greater in a piecemeal feed of the type hierarchy, than if the entire hierarchy is supplied up front.

An algorithm for the dispatching problem is also a solution to the subtyping problem, since if we associate with each type t a unique message μ_t, then $a \preceq b$ holds precisely when a recognizes the message μ_b. We know of no opposite reduction. Indeed, solutions of the subtyping problem tend to be more efficient then their dispatching counterparts.

After this reduction, applying TS gives constant-time subtyping tests. The reason is that the dispatch time is $O(\log f_\mu)$, where $f_\mu = |F_\mu| = 1$. (Recall that for each type we created a unique method family containing only that type.) For completeness we describe the subtyping algorithm in detail.

7.2 Previous Work on Subtyping Tests

(B)PE: *(Bit) Packed Encoding* [47] SC was specialized into a subtyping test scheme called Packed Encoding (PE), by Vitek, Horspool and Krall. They also suggested packing several identifiers into the same byte, resulting in an encoding called Bit Packed Encoding (BPE).

NHE: *Near Optimal Hierarchical Encoding* [48] *Bit-vector encoding* embeds the hierarchy in the lattice of subsets of $\{1, \ldots, \beta\}$. In this scheme, each type a is encoded as a vector vec_a of β bits, such that relation $a \preceq b$ holds iff

$$\text{vec}_b \wedge \text{vec}_a = \text{vec}_b \,. \tag{7.1}$$

The challenge in building a bit-vector encoding is in finding the minimal β for which such an embedding is possible. The problem is NP-hard [41], but several good heuristics were proposed. Currently, NHE due to Krall, Vitek and Hoorspool, is the best (in terms of smallest β) algorithm for bit vector encoding.

Bommel and Beck [62] describe an incremental technique for updating a bit-vector encoding. Although no asymptotic results are given, and testing was limited to "randomly generated hierarchies", it appears from the authors description that the technique is useful for small hierarchies, with at most 300 types.

PQE: *PQ-Encoding* [71] PQE encoding, which uses PQ-trees [8] gives one of the best compression results of the subtyping matrix,

Hierarchy	PQE [a]		NHE [b]		(B)PE [c]		**(B)TS** [a]		**(B)TS^{PQ}** [a]	
	Total (mSec)	Per type (μSec)	Total (mSec)	Per type (μSec)	Total (mSec)	Per type (μSec)	Total (mSec)	Per type (μSec)	Total (mSec)	Per type (μSec)
IDL	1	15	-	-	5	75	0.1	1	1	15
Laure	3	10	21	71	9	31	0.5	2	90	305
Unidraw	1	2	93	151	10	16	1.6	3	90	147
JDK 1.1	1	4	19	84	10	44	0.3	1	30	133
Self	48	27	1,367	759	22	12	20.2	11	12,100	6,715
Ed	29	67	136	313	12	28	1.7	4	711	1,638
LOV	42	96	168	385	10	23	2.0	5	941	2,158
Eiffel4	146	73	-	-	29	15	19.5	10	11,400	5,703
Geode	311	236	1,902	1,443	28	21	20.6	16	22,700	17,223
JDK 1.18	15	9	-	-	26	15	10.0	6	2,520	1,479
JDK 1.22	81	19	-	-	77	18	38.1	9	32,500	7,490
JDK 1.30	113	21	-	-	90	17	53.8	10	49,800	9,158
Cecil	24	26	-	-	13	14	4.4	5	2,000	2,146
Total	815	42	-	-	341	17	172.8	9	134,883	6,880
Median	29	21	136	313	13	18	4.4	5	2,000	2,146

[a] 900 Mhz Pentium III
[b] 500 Mhz 21164 Alpha
[c] 750 Mhz Pentium III, user time in Linux

Table 7: Total time (in mSec) and average time per type (μSec) for generating a subtyping encoding

while maintaining constant time for queries. PQE is not incremental since it requires feeding whole program information into a very sophisticated data structure.

Dynamic subtyping in SI Dietz [17, 18] suggested an asymptotically optimal solution to the dynamic subtyping problem, i.e., linear space requirement and constant time for queries and additions. The idea is to maintain the pre- and post-orders of the tree in an *ordered list* (see Appendix A). Subtyping tests are answered by using two ORDER queries relying on the fact that $a \preceq b$ iff a occurs before b in the post-order and b occurs before a in the pre-order.

A different incremental algorithm for SI is Cohen's algorithm [12]. Let $l_t = |\text{ancestors}(t)|$ denote the level of t. The algorithm associates with each type t an array of length l_t, storing the type-id of each $t' \succeq t$ in position $l_{t'}$. Cohen's algorithm gives *simple* and constant-time subtyping tests. The cost is that the space requirement might be $O(n^2)$ if the hierarchy is, for instance, a long chain. In practice, since the maximal number of ancestors is relatively small, the space requirement of Cohen's encoding is tolerable. Jalapeño [3], IBM implementation of the JAVA virtual machine (JVM), uses Cohen's algorithm for subtyping tests where the supertype is a class.

7.3 Subtyping using Type Slicing Scheme

Our incremental subtyping algorithm is based on the order-preserving heuristic for maintaining the slices (described in Appendix A). The non-incremental variant is described next.

Figure 3.2 showed the slicing of the running example into two slices. We associate with type A, for example, the following data,

$$\begin{aligned} s_{\mathsf{A}} &= 1, \\ \mathrm{id}_{\mathsf{A}} &= 2, \\ D_1(\mathsf{A}) &= [2,4], \\ D_2(\mathsf{A}) &= [1,3]. \end{aligned} \tag{7.2}$$

Encoding a hierarchy in this fashion requires at most $2\kappa n + 2n$ memory cells.

Since $\text{descendants}(t) = \bigcup_{1 \le i \le \kappa} D_i(t)$, we have that $a \preceq b$ holds iff the position of a is within the appropriate interval of b, i.e.,

$$\mathrm{id}_a \in D_{s_a}(b). \tag{7.3}$$

For instance, we test whether $\mathsf{G} \preceq \mathsf{A}$, by retrieving the slice of G, $s_{\mathsf{G}} = 1$, and its identifier, $\mathrm{id}_{\mathsf{G}} = 4$. We then determine whether this identifier falls inside the appropriate interval of A. In this example, we conclude that $\mathsf{G} \preceq \mathsf{A}$ since $4 \in [2,4]$.

7.4 Experimental Results for TS

To make the comparison of incremental and non-incremental algorithms meaningful, we do not include in the space requirement pointers or other auxiliary data used in computing the encoding or in maintenance of the dynamic data structure. In the case of our *type slicing* (TS) algorithm this auxiliary data is a small number of (about four) words per type.

The *BTS variant* of the basic TS algorithm applies *bit packing* to compress the identifiers of types in small slices, in a manner similar to BPE. Note that the BTS and the BPE variants are slower than their non-packing counterparts, since they are obliged to use shifts and masks to unpack the type identifiers. As mentioned in Section 5, the superscript PQ shall denote the variant which use the PQ heuristic.

Creation time Table 7 compares both the total and the per-type run time of different subtyping algorithms on modern computing platforms. In the worst case hierarchy (Geode), the average time a modern computing platform requires to insert a type using (B)TS algorithms is as little as 16 micro-seconds. We also see that the PQ variants of TS are very slow, requiring 17.223 mSec *per type* in this hierarchy, whereas the basic TS algorithms require just a little more (21 mSec) to process the *entire* hierarchy.

To estimate the cost of using the PQE in an incremental fashion, we can compare the *total time* of PQE with the *per-type time* of the incremental (B)TS. In doing so we find that (B)TS is three to four orders of magnitude faster than PQE. Even the *total runtime* of

the (B)TS algorithms is, on average, three times faster than that of PQE.

Despite the fact that the data on the NHE runs was generated on a different architecture, we argued [71] that PQE is in general faster than NHE.

Space requirement The main metric of subtyping algorithms is the encoding length, i.e., the number of bits per type. Table 8 compares the encoding length obtained by TS and its three variants with those of some other algorithms over the standard ensemble of 13 MI-hierarchies.

Hierarchy	PQE	NHE	BPE	PE	**BTS^{PQ}**	**BTS**	**TS^{PQ}**	**TS**
IDL	0	17	32	96	56	64	40	56
Laure	6	23	63	128	72	80	88	152
Unidraw	2	30	63	96	72	72	88	88
JDK 1.1	1	19	32	64	64	64	56	56
Self	39	53	126	344	88	96	120	152
Ed	36	54	94	216	144	152	376	408
LOV	42	57	94	216	144	152	376	408
Eiffel4	65	72	157	312	160	176	312	344
Geode	80	95	157	408	248	264	600	632
JDK 1.18	25	39	94	128	104	112	152	184
JDK 1.22	36	62	157	184	152	168	280	312
JDK 1.30	41	65	188	216	160	192	280	376
Cecil	22	58	94	192	104	112	184	216
Total	40	61	145	227	144	161	266	315
Median	36	54	94	192	104	112	184	216
Minimum	0	17	32	64	56	64	40	56
Maximum	80	95	188	408	248	264	600	632

Table 8: The encoding lengths of different subtyping algorithms

In comparing the last two columns of the table we learn that our quick order-preserving heuristic can be improved, sometimes by as much as 40% by applying the PQ heuristic. However, in going through the BTS column we discover that bit-packing is a more effective compression technique, outperformed by TS^{PQ} in only two out of the 13 hierarchies. Therefore, it seems worthwhile to spend the little extra time in the subtyping tests of BTS.

Note also that applying the PQ heuristic on top of bit packing does not yield much: the maximal compression of the encoding length in doing so is 16.7%. Therefore, our basis of comparison of the incremental algorithms with their static counterparts will be the BTS column.

The BTS encoding is better than PE in all hierarchies, but is only better than BPE in the Self hierarchy. It is slightly worse than BPE in all but the Geode hierarchy. BTS does not yield as good encoding length as NHE and PQE. However, since BTS is incremental, it can answer subtyping queries during any stage of the creation process—a task in which PE, BPE, NHE and PQE fail.

8. Open Problems

The most important problem this paper leaves open is an incremental dispatching algorithm, i.e., allowing additions of types, along with their methods, at the bottom of the hierarchy. Another natural extension worth investigating 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.

Preliminary results include an incremental variant of TS with the same theoretical space and dispatch-time bounds [70]. An insertion of a method $\mu(t)$ to the dispatching data structure takes $O(\kappa \log f_\mu + |\text{descendants}(t)|)$ amortized time.

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 method families. Some OO languages resolve ambiguities based on a *linearization* of the partial order $\preceq$. COMMONLOOPS [7], for example, uses a global type ordering, while CLOS [6] uses a local type ordering. Extending our algorithms to support linearization based ambiguity resolution appears to be a worthy prospect.

Dispatching and linearization 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 most specific supertype.[4] Ambiguities are resolved using the order of the **catch** blocks chosen by the programmer.[5]

Acknowledgements Tal Cohen contributed the database of JAVA hierarchies. Eric Dujardin made the multi-method hierarchies of CECIL and DYLAN available for our experiments. Wade Holst contributed the other five multi-method hierarchies. All the other dispatching data was supplied by Karel Driesen. Driesen's help in supplying the RD implementation and answering our questions in porting it was invaluable. We pay tribute to Jan Vitek for his inspiring remarks.

9. References

[1] R. Agrawal, L. Demichiel, and B. Lindsay. Static type checking of multi-methods. In *Proceedings of the 6th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Phoenix, Arizona, USA, Oct.

[4] The above code, if read in C++ [61], 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.

[5] In fact, 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 SI **class** hierarchy (and ambiguities cannot occur in an SI hierarchy).

6-11 1991. OOPSLA'91, ACM SIGPLAN Notices 26(11) Nov. 1991.

[2] B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber. Efficient implementation of JAVA interfaces: `invokeinterface` considered harmless. In OOPSLA'01 [53].

[3] B. Alpern, A. Cocchi, and D. Grove. Dynamic type checking in Jalapeño. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *Java Virtual Machine Research and Technology Symposium*, Monterey, California, Apr. 2001. USENIX.

[4] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 244–258, Portland, Oregon, USA, Oct. 23-27 1994. OOPSLA'94, ACM SIGPLAN Notices 29(10) Oct. 1994.

[5] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[6] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp object system specification. X3J13 Document 88-002R, June 1988.

[7] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 17–29, Portland, Oregon, USA, Sept. 29 - Oct. 2 1986. OOPSLA'86, ACM SIGPLAN Notices 21(11) Nov. 1986.

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

[9] C. Chambers. The Cecil language, specification and rationale. Technical Report TR-93-03-05, University of Washington, Seattle, 1993.

[10] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In OOPSLA'99 [56], pages 238–255.

[11] W. Chen, V. Turau, and W. Klas. Efficient dynamic look up strategy for multi-methods. In *Proceedings of the 8th European Conference on Object-Oriented Programming* [31], pages 408–431.

[12] N. H. Cohen. Type-extension tests can be performed in constant time. *ACM Trans. Prog. Lang. Syst.*, 13:626–629, 1991.

[13] T. Cohen and J. Y. Gil. Self-calibration of metrics of Java methods. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems*, pages 94–106, Sydney, Australia, Nov. 20-23 2000. TOOLS Pacific 2000, Prentice-Hall.

[14] T. J. Conroy and E. Pelegri-Llopart. *An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations*. Addison-Wesley, Menlo Park,CA 94025, 1983.

[15] B. J. Cox. *Object-Oriented Programming - An Evolutionary Approach*. Addison-Wesley, Reading, Massachusetts, 1986.

[16] P. Deutsch and A. Schiffman. Efficient implementation of the Smalltalk-80 system. In *11th Symposium on Principles of Programming Languages, POPL'84*, pages 297–302, Salt Lake City, Utah, Jan. 1984. ACM SIGPLAN — SIGACT, ACM Press.

[17] P. F. Dietz. Maintaining order in a linked list. In *Proc. of the 14th Ann. ACM Symp. on Theory of Computing*, pages 122–127, San Francisco, California, United States, 1982. ACM Press.

[18] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing*, pages 365–372, New York, New York, United States, 1987. ACM Press.

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

[20] R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings of the 4th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–214, New Orleans, Louisiana, Oct. 1-6 1989. OOPSLA'89, ACM SIGPLAN Notices 24(10) Oct. 1989.

[21] K. Driesen. Selector table indexing & sparse arrays. In OOPSLA'93 [54], pages 259–270.

[22] K. Driesen. Software and hardware techniques for efficient polymorphic calls. Technical Report TRCS99-24, University of California, Santa Barbara. Computer Science., July 15, 1999.

[23] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 141–155, Austin, Texas, USA, Oct. 15-19 1995. OOPSLA'95, ACM SIGPLAN Notices 30(10) Oct. 1995.

[24] K. Driesen and U. Hölzle. The direct cost of virtual functions calls in C++. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 306–323, San Jose, California, Oct. 6-10 1996. OOPSLA'96, ACM SIGPLAN Notices 31(10) Oct. 1996.

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

[26] K. Driesen, U. Hölzle, and J. Vitek. Message dispatch on pipelined processors. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, number 952 in Lecture Notes in Computer Science, pages 253–282, Aarhus, Denmark, Aug. 7–11 1995. ECOOP'95, Springer Verlag.

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

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

[29] N. Eckel and J. Y. Gil. Empirical study of object-layout strategies and optimization techniques. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 394–421, Sophia Antipolis and Cannes, France, June 12–16 2000. ECOOP 2000, Springer Verlag.

[30] ECOOP'91. *Proceedings of the 5th European Conference on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, Geneva, Switzerland, July15–19 1991. Springer Verlag.

[31] ECOOP'94. *Proceedings of the 8th European Conference on Object-Oriented Programming*, number 821 in Lecture Notes in Computer Science, Bologna, Italy, July 4-8 1994. Springer Verlag.

[32] ECOOP'99. *Proceedings of the 13th European Conference on Object-Oriented Programming*, number 1628 in Lecture Notes in Computer Science, Lisbon, Portugal, June 14–18 1999. Springer Verlag.

[33] Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, Jan. 1994.

[34] A. Fall. Sparse term encoding for dynamic taxonomies. In P. W. Eklund, G. Ellis, and G. Mann, editors, *Proceedings of the Fourth International Conference on Conceptual Structures (ICCS-96): Knowlegde Representation as Interlingua*, volume 1115 of *LNAI*, pages 277–292, Berlin, Aug. 19–22 1996. Springer.

[35] P. Ferragina and S. Muthukrishnan. Efficient dynamic method-lookup for object oriented languages. In J. Díaz and M. Serna, editors, *Algorithms—ESA '96, Fourth Annual European Symposium*, volume 1136 of *Lecture Notes in Computer Science*, pages 107–120, Barcelona, Spain, 25–27 Sept. 1996. Springer.

[36] P. Ferragina, S. Muthukrishnan, and M. de Berg. Multi-method dispatching: A geometric approach with applications to string matching problems. In *Proc. of the 31st Ann. ACM Symp. on Theory of Computing*, pages 483–491, Atlanta, Georgia, United States, 1999. ACM Press.

[37] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 135–143, Washington, DC, United States, 1984. ACM Press.

[38] J. Gil and A. Itai. The complexity of type analysis of Object Oriented programs. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 601–634, Brussels, Belgium, July 20–24 1998. ECOOP'98, Springer Verlag.

[39] J. Y. Gil and P. Sweeney. Space- and time-efficient memory layout for multiple inheritance. In OOPSLA'99 [56], pages 256–275.

[40] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Massachusetts, 1984.

[41] M. Habib and L. Nourine. Bit-vector encoding for partially ordered sets. In V. Bouchitte and M. Morvan, editors, *International Workshop on Orders, Algorithms, and Applications (ORDAL'94)*, number 831 in Lecture Notes in Computer Science, pages 1–12, Lyon, France, July 1994. Springer Verlag.

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

[43] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the 5th European Conference on Object-Oriented Programming* [30].

[44] H. Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operation. *ACM Trans. Prog. Lang. Syst.*, 11:115–146, 1989.

[45] G. Kiczales and L. Rodriguez. Efficient method dispatch in PCL. In *1990 ACM Conference on Lisp and Functional Programming*, pages 99–105, Nice, France, June 1990. ACM, ACM Press.

[46] E. Kidd. Efficient Compression of Generic Function Dispatch Tables. Technical Report TR2001-404, Dartmouth College, Computer Science, Hanover, NH, June 2001.

[47] A. Krall, J. Vitek, and R. N. Horspool. Efficient type inclusion tests. In OOPSLA'97 [55], pages 142–157.

[48] A. Krall, J. Vitek, and R. N. Horspool. Near optimal hierarchical encoding of types. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 128–145, Jyväskylä, Finland, June 9-13 1997. ECOOP'97, Springer Verlag.

[49] B. Meyer. *EIFFEL the Language*. Object-Oriented Series. Prentice-Hall, Hemel Hempstead, Hertfordshire, UK, 1992.

[50] W. Mugridge, J. Hamer, and J. Hosking. Multi-methods in a statically-typed programming languages. In *Proceedings of the 5th European Conference on Object-Oriented Programming* [30].

[51] S. Muthukrishnan and M. Müller. Time and space efficient method-lookup for object-oriented programs. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 42–51, New York/Philadelphia, Jan. 28–30 1996. ACM/SIAM.

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

[53] OOPSLA'01. *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Tampa Bay, Florida, Oct. 14–18 2001. ACM SIGPLAN Notices 36(10) Oct. 2001.

[54] OOPSLA'93. *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, USA, Sept. 26 - Oct. 1 1993. ACM SIGPLAN Notices 28(10) Oct. 1993.

[55] OOPSLA'97. *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Atlanta, Georgia, Oct. 5-9 1997. ACM SIGPLAN Notices 32(10) Oct. 1997.

[56] OOPSLA'99. *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, Nov.1–5 1999. ACM SIGPLAN Notices 34(10) Nov. 1999.

[57] C. Pang, W. Holst, Y. Leontiev, and D. Szaforon. Multi-method dispatch using multiple row displacement. In Pang, *et al.* [32], pages 304–328.

[58] O. Raynaud and E. Thierry. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. In J. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 165–181, Budapest, Hungary, June 12–16 2001. ECOOP 2001, Springer Verlag.

[59] A. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *Proceedings of the 7th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 110–126, Vancouver, British Columbia, Canada, Oct.18-22 1992. OOPSLA'92, ACM SIGPLAN Notices 27(10) Oct. 1992.

[60] A. Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[61] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997.

[62] M. F. van Bommel and T. J. Beck. Incremental encoding of multiple inheritance hierarchies. In *Proceedings of the 8th International Conference on Information Knowledgement (CIKM-99)*, pages 507–513, N.Y., Nov. 2–6 2000. ACM Press.

[63] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

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

[65] J. Vitek. Compact dispatch tables for dynamically typed programming languages. Master's thesis, University of Victoria, 1995.

[66] J. Vitek and R. N. Horspool. Taming message passing: Efficient method lookup for dynamically typed object-oriented languages. In *Proceedings of the 8th European Conference on Object-Oriented Programming* [31].

[67] J. Vitek and R. N. Horspool. Compact dispatch tables for dynamically typed object oriented languages. In T. Gyimothy, editor, *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 309–325, Linköping, Sweden, 24–26 Apr. 1996. Springer.

[68] D. E. Willard. New trie data structures which support very fast search operations. *J. Comput. Sys. Sci.*, 28:379–394, 1984.

[69] O. Zendra, C. Colnet, and S. Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In OOPSLA'97 [55], pages 125–141.

[70] Y. Zibin and J. Y. Gil. Theory and practice of incremental subtyping tests and message dispatching. http://www.cs.technion.ac.il/~zyoav. Manuscript.

[71] Y. Zibin and J. Y. Gil. Efficient subtyping tests with PQ-encoding. In OOPSLA'01 [53], pages 96–107.

APPENDIX

A. An Order-Preserving Heuristic for Finding The Slices

The algorithm for creating the slices uses the order-preserving heuristic as an internal procedure in the following fashion. We traverse the types in a topological order, i.e., as if the hierarchy is given to us incrementally where new types can be added only as leaves. For each such 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.

Given a slice $\mathcal{T}_i$ and a type t, we give an algorithm whose runtime is $|\text{ancestors}(t)|$, which checks whether there is an eligible list location for inserting t, and if so, finds it. The idea is to maintain an *ordered list* for all types in a slice. 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 [18] 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[6] 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

[6]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 $c(n)$.

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. *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.

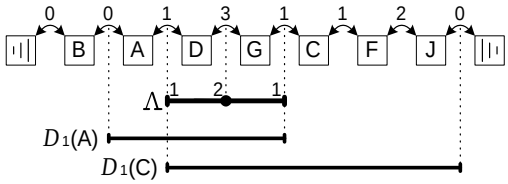


Figure A.1: Addition of a new type to the first slice of Figure 3.2

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. *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.

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 Λ denote the intersection of all intervals in $\mathcal{I}$. A list locations in Λ is called a *candidate* for inserting t.

Algorithmically, Λ 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 Λ 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:

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

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 $t" \in \mathcal{T} \setminus \mathrm{ancestors}(t)$, 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 Λ. 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* λ_ℓ 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 ℓ in the ordered list, let λ_ℓ be the number of all intervals $D_i(t)$, such that ℓ 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 ℓ in the *interior* of Λ 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

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

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

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

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 ℓ in the *boundary* of Λ. Let $I \in \mathcal{I}$ be arbitrary. Then, by definition $\ell \in I$. It is not however guaranteed that ℓ is in the interior of I. We therefore compute the number n_ℓ of intervals $I \in \mathcal{I}$ such that ℓ is in the interior of I. A boundary location ℓ is an invalid candidate iff

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

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 λ_ℓ with every insertion. (Note that the count may change only for two locations: before and after the insertion point.) Also, computing n_ℓ and checking (A.3) can be done in $O(|\mathrm{ancestors}(t)|)$ 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 an eligible insertion location We found empirically that if t could not be inserted at the boundary of Λ, then it was rarely possible to insert it to the interior of Λ. For example, out of the 4339 types of JDK 1.22, only 22 types (less than 0.5%) were inserted in the interior of Λ. 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 an interior candidate location ℓ is valid iff

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

Therefore, the problem of finding an eligible location in the interior of Λ is reduced to the famous *range minima* problem [37]. 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 λ_ℓ of all locations ℓ in the subtree rooted at this node. This representation adds $O(\log n)$ 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 [37]. It is not clear whether these algorithms have any practical utility.

Inserting SI types into MI 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 ℓ immediately to the left (or to the right) of p. We claim that ℓ is eligible for t. Assume the contrary, i.e., ℓ is in the *interior* of some interval defined by $D_i(t")$, $t" \not\succeq t$. Combined with the fact that p is adjacent to ℓ, 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.

Incremental subtyping algorithm Recall that each slice is kept in an *ordered list*. Instead of associating integer values with id_t and $D_i(t)$ as in (7.2), we now use pointers to cells in the ordered list. The test (7.3) can be carried out in constant time using two ORDER queries. We show next how to update this representation as new types are added.

When a type t is added to the ordered list of slice $\mathcal{T}_i$, only the list intervals of its ancestors can change. Therefore, for each $t' \succeq t$ we check if t was added at the boundary of $D_i(t')$, and if so update it. Updating all list intervals $D_i(t')$ takes $O(|\text{ancestors}(t)|)$ time. Since the insertion time of the heuristic is $O(\kappa|\text{ancestors}(t)|)$, the asymptotic time bound remains the same.

When a new slice is created, the arrays which store $D_i(t)$, $i = 1, \ldots, \kappa$, must be extended. Note that with the cost of a constant factor increase of the space requirement, the amortized time for extending an array is constant. Using techniques of "background copying" [19], the *worst case* time for an array extension operation becomes constant as well.