Efficient Subtyping Tests with PQ-Encoding*

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

Subtyping tests, i.e., determining whether one type is a subtype of another, are a frequent operation during the execution of object-oriented programs. The challenge is in encoding the hierarchy in a small space, while simultaneously making sure that subtyping tests have efficient implementation. We present a new scheme for encoding multiple and single inheritance hierarchies, which, in the standardized hierarchies, reduces the footprint of all previously published schemes. The scheme is called *PQ-encoding* after *PQ-trees*, a data structure previously used in graph theory for finding the orderings that satisfy a collection of constraints. In particular, we show that in the traditional object layout model, the extra memory requirements for single inheritance hierarchies is zero. In the PQ-encoding subtyping tests are constant time, and use only two comparisons. Other than PQ-trees, PQ-encoding uses several novel optimization techniques. These techniques are applicable also in improving the performance of other, previously published, encoding schemes.

1. Introduction

One of the basic operations in the run time environment of object-oriented (OO) programs is a *subtype test*. Given an object o and a type b, a subtype test is to determine whether $a = a(o)$, the type of o, is a subtype of b, i.e., a is a descendant of b in the inheritance hierarchy. Such tests (also known as *type inclusion* tests), occur either implicitly in type cast operations, e.g., **dynamic_cast** in C++ [32], **?=** in EIFFEL [29], or explicitly in the execution of dedicated lingual constructs such as JAVA's [2] **instanceof**, and SMALLTALK's [18] `isKindOf:` method.

Subtyping tests are more frequent than one might think. Covariant overriding of arguments in EIFFEL makes it necessary to make subtyping test in conjunction with calls to procedures which use this feature[1]. The covariance nature of arrays in JAVA renders subtyping tests necessary in assignments to elements of arrays whose dynamic type is unknown. Consider the following code fragment

```
void f(Object x[]) {
    x[1] = new A();
}
f(new B[3]);
```

It may be a bit surprising that the assignment to `x[1]` in `f` requires a subtyping test. To understand why, consider the call to function `f`, in which the argument `x` receives a value of type `B[3]` (an array of three elements of type `B`). The function call is legal since type `B[3]` is a subtype of `Object[]` (an array of objects). However, the assignment

```
x[1] = new A();
```

is legal only if type `A` is a subtype of type `B`. Otherwise, the runtime environment raises an `ArrayStoreException` exception.

Formally, a *hierarchy* is a partially ordered set $(T, \prec)$ where T is a set of types[2] and $\prec$ is a reflexive, transitive and anti-symmetric *subtype relation*. If a,b are types, and $a \prec b$ holds, we say that they are *comparable*, that a is a *subtype* of b and that b is a *supertype* of a. Given a hierarchy $(T, \prec)$, $|T| = n$, the subtyping problem is to build a data structure supporting queries of the sort $a \overset{?}{\prec} b$. The generation of this data structure is called *encoding* of the hierarchy. The subtyping problem has enjoyed considerable attention recently (see e.g., [21, 1, 9, 20, 7, 15, 16, 26, 25, 28, 33, 30]), the challenge being in simultaneously optimizing its four complexity measures:

Space Encoding methods associate certain data with each type. We measure the average number of bits per type, also called the *encoding length*.

Instruction count This is the number of machine instructions in the *test code*, on a certain hardware architecture. There are indications [25] that the space consumed by the test code, which can appear many times in a program, can dominate

†Contact author

*Contact the authors for patent information

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA 01 Tampa Florida USA
Copyright ACM 2001 1-58113-335-9/01/10…$5.00

[1] Various mechanisms of static analysis have been proposed to eliminate this requirement, but none of these have been implemented.

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

the encoding length. An encoding is said to be *uniform*[3] if there exists an implementation of the test code in which the instruction count does not depend on the size of the hierarchy. Only uniform encodings will interest us.

Test time The time complexity of the test code is of major interest. Since the test code might contain loops, the time complexity may not be constant even in uniform encodings. Our main concern here are constant time encodings (which are always uniform). To improve timing performance, loops of non-constant time encodings may be unrolled, giving rise to non-constant instruction count, without violating the uniformity condition. (Bit-vector encoding, presented in Section 4, is an example of a uniform encoding which is non-constant time.)

In most uses of the subtyping lingual constructs, the supertype b, is known at compile time. The test code can then be *specialized*, by precomputing values depending on b only (henceforth denoted by a "#" prefix), and emitting them as part of the test code. Specialization thus benefits both instruction count and test time, and may even reduce the encoding length.

Encoding creation time Another important complexity measure is the time for generating the actual encoding. This task is usually computationally difficult, so different creation algorithms have been proposed to the same encoding scheme. These algorithms differ in their running time and encoding length.

The most obvious (uniform) representation as a *binary matrix* (BM) gives constant subtyping tests, but the encoding length is n. This method is useful for small hierarchies and is used e.g., for encoding the JAVA *interfaces* hierarchy[4] in CACAO 64-bit JIT compiler [19, 24]. However, for a hierarchy containing 5500 types the total size of the binary matrix is 3.8MB. The BM can be (non-uniformly) implemented using a zero encoding length and $O(n)$ instruction count: relying on specialization, the test code for $a \overset{?}{\prec} b$ then checks whether a is among the possibly $O(n)$ descendants of b. More generally, a non-uniform encoding is tantamount to representing the encoding data structure as part of the test code, and therefore will not interest us.

The observation that stands behind the work on subtyping tests is that the BM representation is in practice very sparse, and therefore susceptible to massive optimization. Nevertheless, the number of partially ordered sets (*posets*) with n elements is $2^{\Theta(n^2)}$, so the representation of some posets requires $\Omega(n^2)$ bits[5]. Thus, the encoding length is $\Omega(n)$. In other words, for arbitrary hierarchies the performance of binary matrix is asymptotically optimal.

Let the relation $\prec_d$ be the *transitive reduction* of $\prec$, i.e., a minimal relation whose *transitive closure* is $\prec$. More precisely, relation $\prec_d$ is defined by the condition that $a \prec_d b$ if and only if $a \prec b, a \neq b$, and there is no $c \in T$ such that $a \prec c \prec b, a \neq c \neq b$.

[3] The term is borrowed from circuit complexity. A family of circuits for the size dependent incarnations of a certain problem is called uniform, if this family can be generated by a single Turing machine.

[4] Krall, personal communication, Feb. 2001

[5] The number of bipartite graphs with n elements is clearly $2^{\Theta(n^2)}$, and every bipartite graph is also a poset.

Then, another obvious solution to the subtyping problem is *DAG-encoding* that is based on the directed acyclic graph (DAG) defined by types as nodes and edges from $\prec_d$. Figure 1 depicts a DAG topology representation of a hierarchy which will serve as the running example of this paper. We employ the usual convention that edges are directed from the subtype to the supertype, and that types drawn higher in the diagram are considered greater in the subtype relationship. Thus, G $\prec_d$ C and H $\prec$ A.

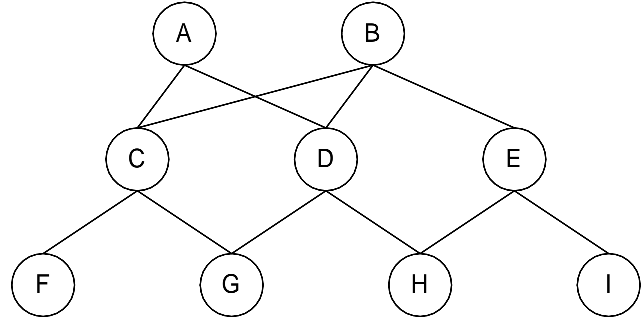

Figure 1: A small example of a multiple subtyping hierarchy

In DAG-encoding, a list of parents is stored with each type, resulting in total space of $(n + |\prec_d|)\lceil \log n \rceil$ bits[6]. The DAG encoding length is therefore $(1 + |\prec_d|/n)\ \lceil \log n \rceil$. Take note that the average number of parents, $|\prec_d|/n$, is small. We will see that in the standard benchmark hierarchies, it is always less than 2. Unfortunately, a subtyping test in DAG-encoding is $O(n)$ time.

The *Closure-encoding* presents another obvious point of tradeoff between space and test time. In this encoding, with each type we store a sorted array of *all* of its ancestors. Thus, a subtyping test is then implemented using a binary search in time $O(\log n)$. The encoding length is $(|\prec|/n)\ \lceil \log n \rceil$.

BM-, DAG- and Closure-encoding are not very appealing techniques. Previous contributions in this field included many sophisticated encoding schemes which come close to DAG-encoding in space, while keeping the test time constant or "almost" constant.

An important special case of the problem is single inheritance (SI), which occurs when the hierarchy DAG takes a tree or forest topology as mandated by the rules of languages such as SMALLTALK [18] and OBJECTIVE-C [12]. The general case of multiple-inheritance (MI), is more difficult, and will be our main concern here. Based on *PQ-trees* [6], a technique for searching an ordering satisfying prescribed constraints, our *PQ-encoding* (PQE) algorithm improves the encoding length of all previous results, in the de facto standard benchmark hierarchies. Thanks to specialization and other optimization techniques, PQE achieves, in a standard object layout model, an encoding length of zero for all SI hierarchies and even in some MI hierarchies.

Yet another demanding constraint is the time for computing the encoding. It is essential that a compiler with whole program information will be able to finish its computation in a reasonable time. PQE compares favorably to previous results in this respect as well.

Outline The remainder of this article is organized as follows. Sec-

[6] Here and henceforth, all logarithms are based two.

tion 2 makes some pertinent definitions. The data set of the 13 hierarchies used in our benchmarking is presented in Section 3. A survey of prior research is the subject of Section 4. This section describes the *slicing* technique of partitioning a hierarchy for the purpose of subtyping tests. The technique is common to many previous algorithms for the problem; it also stands as the basis of the PQE algorithm which is described in Section 5. Section 6 presents our new optimization techniques, improving instruction count, test time and encoding length. The penultimate Section 7 presents the results of running these algorithms on our benchmark. Finally, some open problems and directions for future research are mentioned in Section 8.

2. Definitions

Given a type $a \in T$, we define the following sets: $\text{descendants}(a)$ and $\text{ancestors}(a)$ (which are the subtypes and supertypes of a, respectively), as well as $\text{children}(a)$ and $\text{parents}(a)$ (the sets of all immediate subtypes and supertypes of a). More precisely,

$$\begin{aligned} \text{descendants}(a) &\equiv \{b \in T \mid b \prec a\} \\ \text{ancestors}(a) &\equiv \{b \in T \mid a \prec b\} \\ \text{children}(a) &\equiv \{b \in T \mid b \prec_d a\} \\ \text{parents}(a) &\equiv \{b \in T \mid a \prec_d b\} \end{aligned} \tag{1}$$

Also for $a \in T$, the value $\text{level}(a)$ is the length in nodes of the longest directed (upgoing) path starting from a. The *height* of the hierarchy is the maximal level among all types in T. The kth*-level* of the hierarchy is the set of all types a for which $\text{level}(a) = k$.

$$\begin{aligned} \text{level}(a) &\equiv 1 + \max\{\text{level}(b) \mid b \in \text{parents}(a)\} \\ \text{height}(T) &\equiv \max\{\text{level}(a) \mid a \in T\} \end{aligned} \tag{2}$$

(In the above definition of $\text{level}(a)$, the maximum over an empty set, is defined as zero. In other words, nodes without any parents are defined as being in level 1.)

In Figure 1 we have that

$$\begin{aligned} \text{descendants}(\mathsf{A}) &= \{\mathsf{A}, \mathsf{C}, \mathsf{D}, \mathsf{F}, \mathsf{G}, \mathsf{H}\} \\ \text{ancestors}(\mathsf{F}) &= \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{F}\} \\ \text{children}(\mathsf{A}) &= \{\mathsf{C}, \mathsf{D}\} \\ \text{parents}(\mathsf{F}) &= \{\mathsf{C}\} \\ \text{level}(\mathsf{F}) &= 3 \end{aligned}$$

This hierarchy has three levels: with two, three, and four types.

The following definitions will also become pertinent:

$$\begin{aligned} \text{roots}(T) &\equiv \{a \in T \mid \text{parents}(a) = \emptyset\} \\ \text{leaves}(T) &\equiv \{a \in T \mid \text{children}(a) = \emptyset\} \end{aligned} \tag{3}$$

In our running example we have

$$\begin{aligned} \text{roots}(T) &= \{\mathsf{A}, \mathsf{B}\} \\ \text{leaves}(T) &= \{\mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}\} \end{aligned}$$

3. Data Set

To benchmark the algorithms, we use the 9 multiple inheritance hierarchies used by Eckel and Gil [14] in their benchmark of object layout techniques. Three new JAVA hierarchies, and the Cecil compiler hierarchy [10], were added to this benchmark. This data set represents an array of large hierarchies drawn from various OO languages. In particular, the set includes all multiple-inheritance hierarchies used in previous studies of encoding schemes [28, 30, 26, 25]. The reader is referred to [14] for the detailed description of these hierarchies. One of the findings in [14] is that many topological properties of typical hierarchies are similar to those of balanced trees. This makes it possible to find more efficient encoding for hierarchies used in practice. Comparison of different encoding schemes is done over these 13 hierarchies which have now become a de facto standard benchmark.

Some statistical information on our data sets can be found in Table 1. The number of types ranges between 66 and 5,438. In total the 13 hierarchies represent over 19,500 types.

Hierarchy	n	$\lvert\prec_d\rvert/n$	$\lvert\prec\rvert/n$	α[a]	β[b]	γ[c]	$\lvert T'\rvert/n$
IDL	66	0.98	3.83	8	6	7	15%
Laure	295	1.07	8.13	16	11	9	18%
Unidraw	613	0.78	3.02	9	8	10	4%
JDK 1.1	225	1.04	3.17	7	6	8	15%
Self	1801	1.02	29.89	40	16	11	9%
Ed	434	1.66	7.99	23	10	9	61%
LOV	436	1.71	8.50	24	9	9	62%
Eiffel4	1999	1.28	8.78	39	17	11	46%
Geode	1318	1.89	13.99	50	13	11	75%
JDK 1.18	1704	1.10	4.35	16	9	11	18%
JDK 1.22	4339	1.19	4.37	17	9	13	22%
JDK 1.30	5438	1.17	4.37	19	9	13	21%
Cecil	932	1.21	6.47	23	12	10	33%

[a] $\max\{|\text{ancestors}(a)| \mid a \in T\}$
[b] height
[c] $\lceil \log n \rceil$

Table 1: Topological properties of hierarchies in our data set

We see in that table that the average number of parents, $|\prec_d|/n$, is always less than 2. On the other hand, the average number of ancestors, $|\prec|/n$, is large. In the Self hierarchy a type has in average almost 30 ancestors! The maximal number of ancestors plays an important factor in the complexity of some of the algorithms. We see that there exists a type in the Geode hierarchy which has 50 ancestors in total. In comparing the height of the hierarchy with $\log n$ we see that the hierarchies are shallow; their height is similar to that of a balanced binary tree.

Finally, we can learn a bit more on the topology of inheritance hierarchy by considering the set T', which can be thought as the *MI core* of the hierarchy. Formally, a type is in the core if it has a descendant with more than one parent. Conversely, the set $T \setminus T'$ is a collection of maximal subtrees discovered in a bottom-up search of

the hierarchy. It was previously noticed [25] that encoding is easier if the core is considered first, and the *bottom trees* of $T \setminus T'$ are added to the encoding later. In Table 1 we see that in most hierarchies the core is rather small, typically less than half the number of types. Treating the core and the bottom trees separately will reduce the run time of PQE.

4. Previous Work

This section gives an overview of various encoding methods proposed in the literature. We describe the data structure used in each such encoding, and how it is deciphered to implement subtyping tests. Little if any attention is devoted to describing the actual generation of the data structure and the theory behind it.

Perhaps the most elegant algorithm for encoding is *relative numbering* [31] (also called *Schubert's numbering*) which guarantees both optimal encoding length of $\lceil \log n \rceil$ bits and constant time subtyping tests. However, these achievements are only possible in a SI hierarchy. For a type $b \in T$, let r_b denote its ordinal (i.e., an integer in the range $1, \ldots, n$) in a postorder traversal of T. A basic property of postorder traversal is that

$$r_b = \max\{r_a \mid a \prec b\}. \quad (4)$$

Let l_b be defined by

$$l_b = \min\{r_a \mid a \prec b\}. \quad (5)$$

Combining (4), (5) with the fact that in postorder traversal the descendants of any type are assigned consecutively, we find that $a \prec b$ iff

$$\#l_b \leq r_a \leq \#r_b. \quad (6)$$

Thus, in relative numbering, each type a is encoded by an interval $[l_a, r_a]$ as exemplified by Figure 2. We argue that since b is known at compile time, the subtyping test code (6) can be specialized by eliminating the memory fetch of constants l_b and r_b. In doing so, we find that the values l_b need not be stored. The total encoding length in our improved version of relative numbering is $\lceil \log n \rceil$.

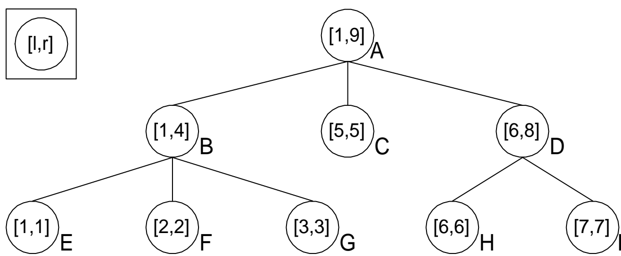

Figure 2: Relative numbering in a tree hierarchy

Relative numbering is used in CACAO [19, 24] for representing the JAVA class inheritance hierarchy[7]. (Recall that BM is used in CACAO for the interfaces hierarchy.) Range-compression [1], described below, is a generalization of relative numbering for MI.

Cohen's algorithm [11], which can be thought of as a variant of Dijkstra's displays [13], is yet another algorithm initially designed for SI hierarchies, and later generalized by Packed Encoding to MI. The algorithm, which is implemented as part of the DEC SRC MODULA-3 system [8], relies on hierarchies being relatively shallow, and more so, on types having a small number of ancestors. As Table 1 shows, this is indeed the case even in our MI hierarchies. A type a is allocated with an array r_a of size

$$\text{level}(a) \leq |\text{ancestors}(a)|$$

(in SI, $\text{level}(a) = |\text{ancestors}(a)|$), with entries for each

$$b \in \text{ancestors}(a).$$

Specifically, $b \succ a$ is stored in location level(b) in array r_a. Thus, checking whether $b \succ a$ can be carried out by checking whether b indeed occurs in location level(b) of array r_a. The encoding is optimized by not storing b itself in this location, but rather an *id*, which is unique among all types in its level.

Since different levels come in different sizes, some id's may require fewer bits than others. Typically, an id is stored in either a single byte or in a 16 bits word. It is even possible to pack several id's into a single byte. As a result of this compression the entries of r_a, which are not of equal size, cannot be referenced using ordinary array access operations. We say that r is a *pseudo array*, and use the notation $r@i$ instead of $r[i]$ for denoting array access. Note that if the index i is known at compile time, then a pseudo-array access is the same as record member selection, and is no slower than a non-pseudo array access. Pseudo-arrays are only used, if in all of their indexing operations, the index is known at compile time.

Cohen's encoding stores with each type a its level, $l_a = \text{level}(a)$, its unique id within this level id_a, as well as the pseudo array r_a, such that for each $b \in \text{ancestors}(a)$,

$$r_a @ l_b = \# \, \text{id}_b \,. \quad (7)$$

The test $a \preceq b$ is carried out by checking that $l_a \geq \#l_b$ and then that (7) holds. Note that l_b is known at compile time.

An example of the actual encoding is given in Figure 3.

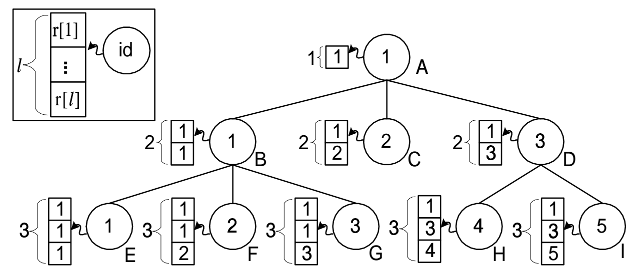

Figure 3: Cohen's encoding of the tree hierarchy of Figure 2

The array boundary check $l_a \geq \#l_b$ is not elegant. We found that it can be eliminated in the price of allocating globally unique id's. Then, it is possible to concatenate the arrays, making sure that the largest array is at the end. Even if there is an overflow in the array access $r_a[l_b]$, the location found will not contain id_b.

Cohen's algorithm was generalized to MI by Vitek, Horspool and Krall [25] into what is called *Packed Encoding* (PE) and *Bit-Packed Encoding* (BPE), which are both constant time methods. A common theme to Cohen's algorithm, PE, BPE and our algorithm is

[7] Krall, personal communication, Feb. 2001

slicing, in which the set T is partitioned into disjoint *slices* (sometimes called buckets) $S_1, \ldots, S_k$. For each slice S_i we store the entire information required to answer queries of the sort $a \preceq b$, $a \in T$ and $b \in S_i$, i.e., queries in which the supertype is drawn from S_i. Type a has a pseudo array r_a of length k, where $r_a@i$ holds information for slice S_i. In essence, we store, in a very compressed format, the set of descendants of each element in S_i. The compression is possible since there is a great deal of sharing in the descendants set of different members of S_i.

PE associates with each type $a \in T$ a unique integer id_a within its slice s_a, so that a is identified by the pair $\langle s_a, \mathrm{id}_a \rangle$. Also associated with a is a byte array r_a, such that for all $b \in \mathrm{ancestors}(a)$, index s_b stores id_b, i.e.,

$$r_a[s_b] = \# \, \mathrm{id}_b \, . \tag{8}$$

A necessary and sufficient condition for $a \prec b$ to hold is then (8). It should be clear that no two ancestors of a can be on the same slice. Thus, the number of slices is at least the size of the largest ancestors set, which is described in Table 1.

Comparing (8) with (7), we see that slices play a role similar to that of levels in Cohen's algorithm. In fact, Cohen's algorithm partitions the hierarchy into $\mathrm{height}(T)$ anti-chains[8], while PE partitions the hierarchy into anti-chains where no two elements in an anti-chain have a common descendant. Fall [15], who observed that this technique might be used for subtyping tests, noted that it is NP-hard to find a minimal such partition, and stopped short of finding a constant time subtyping test. The heuristic suggested in [25] along with the constant time subtype test made PE viable.

PE constrains each slice to a maximum of 255 types, so that id_a can always be represented by a single byte. The encoding length is then $8k$, where k is the number of slices. The inventors of PE observed that k is usually the maximal number of ancestors unless MI is heavily used.

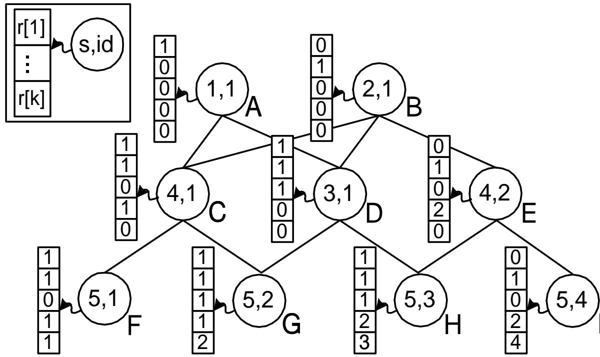

Figure 4: PE representation of the hierarchy of Figure 1

Consider Figure 4 for an example of PE representation of the hierarchy of Figure 1. The types of the hierarchy are partitioned into five different slices: $S_1 = \{\mathsf{A}\}$, $S_2 = \{\mathsf{B}\}$, $S_3 = \{\mathsf{D}\}$, $S_4 = \{\mathsf{C}, \mathsf{E}\}$, and $S_5 = \{\mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}\}$. This is the smallest possible number of slices, since type F (for example) has five ancestors.

[8] An *anti-chain* is a set of types, where no two types are comparable. Clearly, each level is an anti-chain.

The only difference between BPE and PE is that BPE permits two slices or more to be represented within a single byte. Thus, in BPE r_a is a pseudo array, and the array access in (8) becomes a pseudo array access:

$$r_a@s_b = \# \, \mathrm{id}_b \, . \tag{9}$$

Starting from Figure 4 we can represent slices S_1, S_2 and S_3 using a single bit, S_4 using two bits, and S_5 in three bits, for a total of seven bits, which can fit into a single byte.

Bit-vector encoding is one of the most explored directions in prior art. In this scheme, each type a is encoded as a vector vec_a of k bits. If $\mathrm{vec}_a[i] = 1$ then we say that a has *gene* i. Let $\phi(a)$ be the set of genes of a. Relation $a \prec b$ holds iff $\phi(a) \supseteq \phi(b)$, which can be easily checked by masking vec_b against vec_a, specifically, applying the test:

$$\mathrm{vec}_b \wedge \mathrm{vec}_a = \mathrm{vec}_b \, . \tag{10}$$

Figure 5 gives an example of bit-vector encoding of the hierarchy of Figure 1.

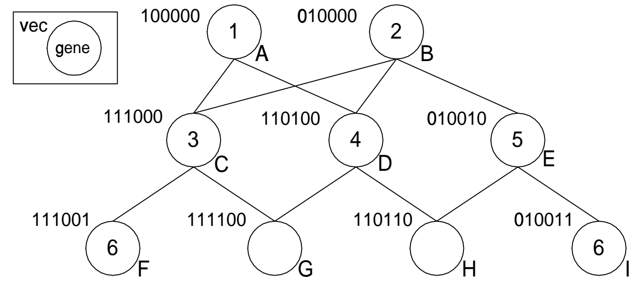

Figure 5: Bit-vector encoding of the hierarchy of Figure 1. (In each type we only write the genes it adds to its parents.)

Bit-vector encoding effectively embeds the hierarchy in the lattice of subsets of $\{1, \ldots, k\}$. It is always possible to do so by setting $k = n$ and in letting vec_a be the row of the BM which corresponds to a. A simple counting argument shows that k must depend on the size of the hierarchy. Hence, bit-vector encoding is non-constant time, but it is uniform. For efficiency reasons, the implicit loop in (10) is unrolled, giving rise to a non-constant instruction count.

The challenge is in finding the minimal k for which such an embedding of the hierarchy in a lattice is possible. The problem is NP-hard [20], but several good heuristics were proposed, including Kaci, Boyer, Lincoln, and Nasr [21] work, Caseau's *Compact Hierarchical Encoding* [9], later improved by Habib, Caseau, Nourine and Raynaud [28]. Currently, *Near Optimal Hierarchical Encoding* (NHE) due to Krall, Vitek and Hoorspool [26], is the best bit vector encoding.

It is only natural to ask then whether it is possible to promise *constant encoding length*, while maintaining uniformity and "almost constant" time. An affirmative answer to this question was given by Agrawal, Borgida and Jagadish [1] in their *range-compression* encoding which generalizes relative numbering. Range compression encodes each type b as an integer id_b, with its ordinal in a postorder scan of a certain spanning forest of the hierarchy. Then,

the set $\Phi(b)$ of id's of the descendants of b,

$$\Phi(b) = \{\text{id}_a \mid a \in \text{descendants}(b)\}, \tag{11}$$

is represented by an array of consecutive disjoint intervals

$$[l_b@1, r_b@1], [l_b@2, r_b@2], \ldots, [l_b@k(b), r_b@k(b)].$$

Thus, $a \prec b$ iff

$$\#l_b@i \leq \text{id}_a \leq \#r_b@i \tag{12}$$

holds for *some* i, $1 \leq i \leq k(b)$. In SI, all descendants of a type are assigned consecutive numbering in a postorder traversal, and therefore the set (11) can be represented using a single interval. The encoding then degenerates to relative numbering.

Figure 6 gives a range-compression encoding of the hierarchy of Figure 1. We have for example

$$\Phi(\mathsf{B}) = \{1, 2, 3, 5, 6, 7, 8, 9\}$$

which can be represented as two intervals $[1, 3]$ and $[5, 9]$. Thus, $l_{\mathsf{B}} = \{1, 5\}$, $r_{\mathsf{B}} = \{3, 9\}$ and $k(\mathsf{B}) = 2$.

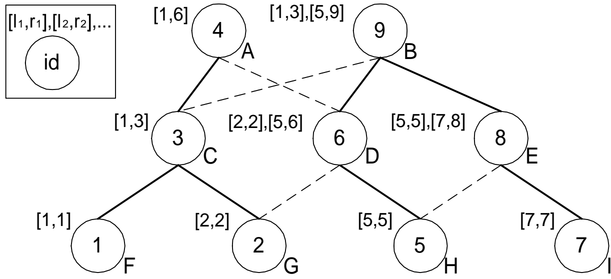

Figure 6: Range-compression encoding of the hierarchy of Figure 1. (Edges of the spanning forest are in bold.)

Examining (12) we see that only id_a has to be stored for a type a, since everything else is specialized into the subtyping test site. The specialization reduces the encoding length to $\lceil \log n \rceil$, but at a price of increasing the instruction count from constant to $k(b)$, which can be in the order of n. In all of our hierarchies however, the average $k(b)$ was always less than 2. The maximal $k(b) = 55$ was found in the Geode hierarchy.

The usual straightforward implementation of range compression requires $O(k(b))$ time. If $k(b)$ is large then a binary search on (12) reduces the time to $O(\log k(b))$. Note that this faster implementation does nothing to improve the instruction count in the specialized implementation which remains $\Omega(k(b))$.

Other non-constant encoding techniques were used in large data- and knowledge-bases, e.g., modulation techniques [21, 15], sparse terms encoding [16], and representation using union of interval orders [7]. The common objective is a small average, rather than worst-case, time for testing, which may be considered unsuitable for object oriented applications.

5. PQ-Encoding

This section describes PQE, an encoding scheme which achieves the smallest space requirements among all previously published algorithms. PQE combines the ideas of relative numbering with slicing as used in PE and BPE. Each type a belongs in a slice s_a. Also, for each type a we store an interval $[l_a, r_a]$ and a pseudo array id_a, such that $a \prec b$ iff

$$\#l_b \leq \text{id}_a@s_b \leq \#r_b. \tag{13}$$

Since b is known at compile time, testing (13) requires exactly the same number of RISC instructions as relative numbering (6). Also, since (13) is similar to boundaries check in an array access, it may be further optimized on architecture with dedicated instructions for this kind of check.

Figure 7 describes a PQE representation of our running example. Note that the encoding uses only one slice.

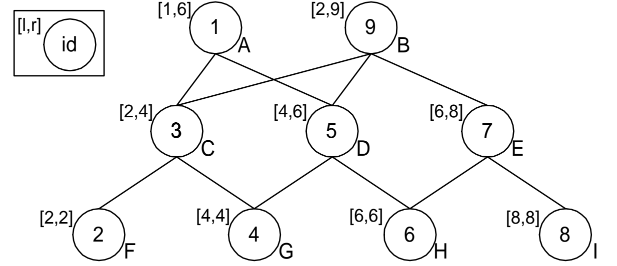

Figure 7: PQ-encoding of the hierarchy of Figure 1

A slice $S \subseteq T$ in PQE is a set of types, maximal with respect to the property that there is an ordering π of T such that all descendants of any $a \in S$ occur *consecutively* in π. This property makes is possible to represent the set of all descendants of a using merely two integers, as required by (13).

The name PQ-encoding is derived from the main tool used in finding these maximal slices: PQ-trees. This data structure is the invention of Booth and Leuker [6] who used it to test for the *consecutive 1*'s property in binary matrices of size $r \times s$ and in time $O(k+r+s)$ where k is the number of 1's in the matrix. Their result gave rise to the first linear time algorithm for recognizing interval graphs. Later, PQ-trees were used for other graph-theoretical problems, such as on-line planarity testing [3, 4] and maximum planar embeddings [5, 22, 23].

There are three kinds of a nodes in a PQ-tree: a *leaf* which represents a member of a given set U, a *Q-node* which represents the constraint that all of its children must occur in the order they occur in the tree or in reverse order, and a *P-node* which specifies that its children must occur together, but in any order. As a whole, a PQ-tree $\mathcal{P}$ represents a subset of the orderings of U, denoted by $\text{consistent}(\mathcal{P})$. The ordering of U obtained by a DFS traversal of $\mathcal{P}$, is denoted $\text{frontier}(\mathcal{P})$. Two transformations of $\mathcal{P}$ preserve $\text{consistent}(\mathcal{P})$: swapping any two children of a P-node, and reversing the order of the children of a Q-node. PQ-trees $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent ($\mathcal{P}_1 \equiv \mathcal{P}_2$) if $\mathcal{P}_2$ can be reached from $\mathcal{P}_2$ by a series of these transformations. Thus,

$$\text{consistent}(\mathcal{P}) = \{\text{frontier}(\mathcal{P}') \mid \mathcal{P}' \equiv \mathcal{P}\} \tag{14}$$

The *universal PQ-tree*, denoted $\mathcal{P}^{\emptyset}$, has a P-node as a root and a leaf for every member of U.

Let φ be a collection of subsets of a set U, i.e., $\varphi \subseteq 2^U$, and let $\Pi(\varphi)$ be the collection (which might be empty) of all orderings π of U such that the members of each $I \in \varphi$ occur consecutively in π.

THEOREM 5.1 (BOOTH-LEUKER (1976)). *For every collection of constraints φ there exists a PQ-tree $\mathcal{P}$, and for every tree $\mathcal{P}$ there exists a collection of constraints φ such that*

$$\Pi(\varphi) = \text{consistent}(\mathcal{P}).$$

Constructively, the tree $\mathcal{P}$ can be generated from φ by letting

$$\mathcal{P} \leftarrow \mathcal{P}^{\emptyset}$$

and making the procedure call

$$\mathsf{reduce}(\mathcal{P}, I)$$

for each $I \in \varphi$. Procedure $\mathsf{reduce}(\mathcal{P}, I)$ first checks whether there is a $\mathcal{P}'$, $\mathcal{P}' \equiv \mathcal{P}$, such that the elements of I appear consecutively in $\text{frontier}(\mathcal{P}')$. The procedure aborts if no such $\mathcal{P}'$ is found. Procedure reduce then conducts a bottom up traversal of the nodes of $\mathcal{P}$. At each step, one of standard eleven PQ-tree transformations is applied, until all elements of I appear consecutively in all consistent orderings. The time complexity is $O(|I|)$.

Algorithm 1 shows how the actual construction of the encoding is carried out. At the termination of the first outer loop (lines 3–9), the PQ-trees $\mathbf{S}[1], \ldots, \mathbf{S}[\ell - 1]$ represent the $\ell - 1$ slices generated by the algorithm. Then, all that remains is to assign unique id's within each slice, and to compute the id's of the right-most and left-most id's among the descendants of each type within each slice. The time complexity is $O\left((\ell - 1)|\prec|\right)$.

The order at which types are inserted into PQ-trees in line 3 is unspecified. This line is the main source of non-determinism in our algorithm. After having tried several traversal orders, including a random one, we concluded that the differences in the encoding length is small. It appears however that the best results are obtained by a reverse topological-order in which the leaves with the largest number of ancestors are visited first.

1: $\mathbf{S}[1] \leftarrow \mathcal{P}^{\emptyset}$
   // **S** *is an array of the slices (represented as PQ-trees)*
   // *created so far.*
2: $\ell \leftarrow 1$ // *The index of the first unused PQ-tree in* **S**.
3: **For all** $a \in T$ **do** // *Find a PQ-tree consistent with type* a.
4: **For** $s = 1, \ldots, \ell$ **do**
5: $\mathsf{reduce}(\mathbf{S}[s], \text{descendants}(a))$
6: **exit loop if** reduce succeeded
7: $s_a \leftarrow s$
8: **If** s $= \ell$ **then** // *Start a new universal PQ-tree*
9: $\ell \leftarrow \ell + 1$; $\mathbf{S}[\ell] \leftarrow \mathcal{P}^{\emptyset}$
10: **For** $s = 1, \ldots, \ell - 1$ **do** // *Assign unique* id*'s*
11: id $\leftarrow 1$ // *The first unused* id *in the slice* $\mathbf{S}[s]$.
12: **For all** $a \in \text{frontier}(\mathbf{S}[s])$ **do**
13: $\text{id}_a @ s \leftarrow \text{id}$; $\text{id} \leftarrow \text{id} + 1$
14: **For all** $a \in T$ **do** // *Assign an interval to each type* a
15: $\mathbf{D} \leftarrow \{\text{id}_b @ s_a \mid b \in \text{descendants}(a)\}$;
16: $l_a \leftarrow \min(\mathbf{D})$; $r_a \leftarrow \max(\mathbf{D})$

Algorithm 1: PQE algorithm for a hierarchy $(T, \prec)$

We next describe several optimizations techniques targeted at improving various complexity measures of the encoding.

The first such optimization reduces the encoding length as generated by Algorithm 1. Let

$$D = \{b \mid \exists a \in S \bullet b \prec a\},$$

be the set of descendants of a slice S. Then, $|S| \leq |D|$. For some of the smaller slices we might even have that $|D|$ is close to n. The *length optimization* relies on the observation that in these cases it is possible to reuse id's while numbering the types in D. The critical point to note is that two types $b_1, b_2 \in D$ need to be assigned distinct identifiers only if there is a type $a \in S$, such that $b_1 \in \text{descendants}(a)$, while $b_2 \notin \text{descendants}(a)$ (or vice versa). Phrased differently, S partitions T into equivalence classes, such that types b_1 and b_2 are in the same equivalence class if

$$\text{ancestors}(b_1) \cap S = \text{ancestors}(b_2) \cap S. \qquad (15)$$

We call this the *S-partitioning* of T. Note that $E_0 \equiv T \setminus D$ is a single equivalence class, which can be assigned the special id 0, which is not contained in any interval. The minimal range (number of different id's) needed to encode a slice S is exactly the number of equivalence classes in S-partitioning of T. We next show that the minimal range is always lower than twice the size of a slice.

Now, PQE ensures that for every $a \in S$ there is an interval I_a which consists of $\text{descendants}(a)$. These $|S|$ intervals partition the types in D into at most $2|S| - 1$ segments, such that all types in the same segment can receive the same id.

Consider for example Figure 8 in which the types in D were initially numbered $3, \ldots, 15$. Intervals I_1, I_2 and I_3 drawn in the figure partition D into $5 = 2 \cdot 3 - 1$ segments. This is the maximal possible number of segments, since every type in D must belong to at least one interval.

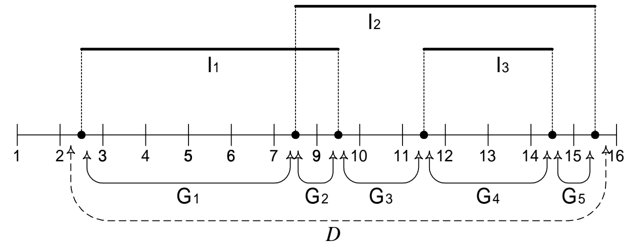

Figure 8: Reducing the range needed for PQE

We have then

THEOREM 5.2. *Let $S \subseteq T$ be a slice obtained by PQE. Then, the integral range required for numbering T is at most*

$$\min(2|S|, |T|).$$

Every equivalence class, except E_0, is a collection of segments, which proved Theorem 5.2. For example, $E_0 = \{1, 2, 16\}$, $E_1 = G_1$, $E_2 = G_2$, $E_3 = G_3 \cup G_5$, and $E_4 = G_4$.

In all hierarchies in the data set, we found that all slices, except the first, were of size 128 or less. Thus, thanks to Theorem 5.2, id_a can be represented as a byte array, with each slice adding a single byte to the encoding length. The first slice receives some special handling as will be described below in Section 6.

It is possible to modify the PQE algorithm to ensure that all but one slice has their range bounded by 255. An application of reduce (line 5 in Algorithm 1) to a PQ-tree (other than the first) is simply revoked if the range required for numbering exceeds 255.[9] Storing the current required numbering range of a PQ-tree, and updating it with each reduce is straightforward. One can also manage the equivalence classes of all slices incrementally in $O(|\prec|)$ total time.

The second optimization reduces the time of running Algorithm 1 by pruning in a preprocessing stage all bottom-trees (see Section 3); a lighter machinery is then used to produce the encoding of those. After the PQ-encoding of the core is generated, relative numbering of each bottom-tree is inserted into the interval of its root, after an appropriate expansion of this interval. (See Figure 9 for an illustration of this process.).

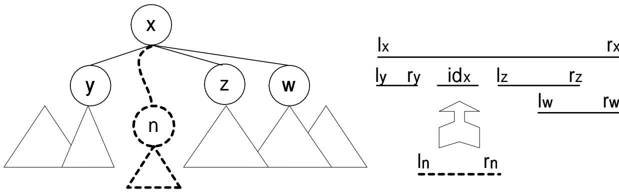

Figure 9: Inserting a bottom-tree into an existing PQ-encoding

The third optimization, *heterogeneous encoding*, is an encoding-length optimization technique. Recall that in BM-encoding each type adds exactly one bit to the encoding of all other types. The PQ-encoding of a small slice with $k < 8$ types adds a byte to the array id_a of each other type a, which is less efficient than using BM-encoding for types in this slice. In heterogeneous encoding, subtyping tests $a \preceq b$, where b belongs in such a small slice, are implemented using BM-encoding. Since b is known at compile time, the compiler can choose the appropriate code to plant at the subtyping test. We found that heterogeneous encoding may give rise to significant improvement to the encoding length. On the other hand, the total number of types in small slices is negligible, and therefore we do not expect a noticeable impact on the instruction count and test time.

6. Inlining and Coalescing Optimizations

So far it was tacitly assumed that in the test $a \preceq b$ that the type a is given at run time. In reality, however the type a must be computed from a certain object o. The traditional object model [17] stores with each object o a pointer p_a to a memory block with run time representation of the type a of o. The various encoding schemes store their auxiliary information in this area, called RTTI (run time type information) in the C++ jargon. The object-oriented paradigm mandates other uses to the RTTI records, including dispatching, downcasting, and garbage collection.

Inlining optimization makes use of the degree of freedom in that the compiler has in placing these records in memory. The simplest application of inlining is in relative numbering: The RTTI records are placed in memory in the same order as postorder traversal. In doing so, p_a can play the same role as r_a. As a result, the encoding length is reduced to zero and one load instruction in the subtyping test is saved.

[9]Note that this does not necessarily happen when the slice size hits 128.

Similarly, in range-compression (12), we can use p_a instead of the global id_a. If specialization is used then we obtain an encoding scheme with zero encoding length, but non-constant test time or instruction count.

In PQE, inlining is used to represent the id's of types with respect to the first slice, specifically, id_a @1 for all types a. In other words, the first entry in the pseudo array id_a, is encoded as p_a. The saving is significant since the first slice occupies the largest number of bits. Inlining also saves one load instruction in the case that type b belongs in the first slice. Since the first slice constitutes around 90% of the types, we expect this saving to lead to a noticeable saving in the average test time.

Thus, in PQE, there are three kinds of slices. The first slice is inlined. Heterogeneous encoding based on binary matrix representation is used for the small slices (with fewer than 8 types). Each of the remaining slices occupies a single byte in the array id.

We could think of no straightforward way for applying inlining neither for bit-vector encodings nor for Cohen's algorithm and its generalizations, PE and BPE.

Another intricacy which is often overlooked in subtyping tests is that even fetching the pointer p_a from the object o, may not be trivial. In the presence of multiple inheritance, traditional object layout schemes are inclined to store several pointers in an object to various RTTI records. The reason is a phenomenon sometimes called ***this** adjustment* occurring in upcasts in a multiple inheritance hierarchy. After such a cast, a pointer to an object does not necessarily point to its start.

For the sake of concreteness, let us use the object layout model of C++. In this model, there are various pointers, called *VPTR*s to the dispatching tables, called *VTBL*s. No matter what **this**-adjustment took place, it is always possible to fetch a VPTR from an object. The difficulty in applying inlining in this model, is that there is no unique value p_a for a type a.

Observe that the subtype tests of relative numbering (6), range compression (12), and PQE (13), check for inequality rather than equality. Thus, inlining can be done even if p_a is not unique by allocating a range of memory addresses, rather than a single address, as the value r_a (as in (6)) or the id (as in (12) and (13)).

Yet another intricacy posed by C++ is the location in memory of the subtype data as generated by the encoding algorithm. If there are several VTBLs, then an implementer faces the dilemma of either duplicating this data in each VTBL, or using another level of indirection from all the VTBLs of a certain class to a shared subtyping encoding. Not only sharing incurs a run time penalty, it also increases the memory overhead by a pointer for each VTBL of each class. With the reduction in encoding length, as offered by algorithms such as PQE, the alternative of duplication seems more appealing. In considering the tradeoff, it should be remembered that the number of VTBLs may be in the order of the number of ancestors [17]. Empirical data on the number of VTBLs can be found in [14].

If the sharing alternative is chosen, then another reduction in the encoding length can be achieved by coalescing the id arrays of PQE.

We now turn to describing *Coalesced PQ-Encoding* (CPQE).

Recall that the first entry of the pseudo array id_a has an implicit representation due to inlining. Let id'_a denote the array id_a after truncating its first entry, and let p'_a denote the pointer to id'_a (which is stored in all VTBLs of type a). If several id' arrays belonging to different types are identical, they can be stored only once. All the distinct arrays id' are stored in one large array denoted Z. We also note that if the number of different id' arrays is small, then p'_a can be replaced by the index of id'_a in the large array Z.

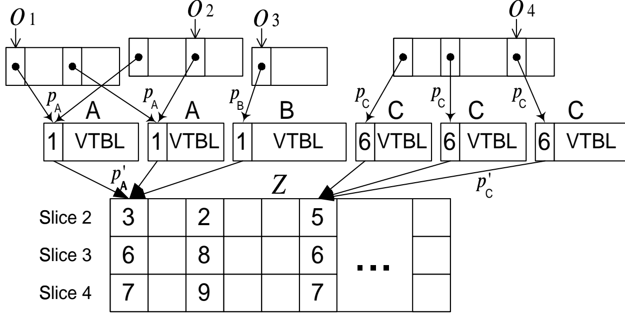

Figure 10: An example of the memory layout in CPQE

An example of the memory layout in CPQE is in Figure 10. In the figure, we see two instances o_1 and o_2 of type A. Each of these objects has two pointers to the two VTBLs of class A. Each of these VTBLs stores p'_{A} which points to the array id'_{A}. Since the total number of different id' arrays is small, instead of storing a pointer to id'_{A}, the VTBLs store the index of id'_{A} in the larger array Z.

Object o_3 of type B has a single VPTR to the VTBL of B. Arrays id'_{B} and id'_{A} are identical, and hence the VTBLs of both types store a reference to the same entry of array Z, i.e., $p'_{\mathsf{A}} = p'_{\mathsf{B}}$. Finally, object o_4 of type C has three VPTRs. The VTBL of C stores the index of the array id'_{C} in Z. We also see that o_2 and o_4 have undergone `this` adjustment.

In the same fashion we inlined the first slice by using p_a instead of id_a @1, we can now inline the second slice by using p'_a instead of id_a @2. The second inlining is possible since there is again a degree of freedom in the order in which arrays id' are stored in Z. In the test $a \preceq b$, if it is found that b belongs in slice S_2, then instead of using id_a @2 in the test (13), the compiler emits code for comparing p'_a with the values l_b and r_b, which are, as usual, specialized into the test code. The entries in array Z are then the arrays id'' produced by truncating the first two entries of array id.

Figure 11 depicts CPQE after the second inlining. Note that the arrays id'' are not necessarily distinct. The entries in array Z where reordered, so Slice 2 could be inlined into the VTBLs. For example, the first entry in Figure 10 is now in position 3.

Finding all coalescing opportunities among the arrays id' can be done using bucket sort (in time linear in the total size of all arrays). We can also show that the number of distinct arrays id' is exactly the number of equivalence classes in G-partitioning of T, where $G = T \setminus S_1$. Furthermore, this number is always smaller or equal to the size of the core.

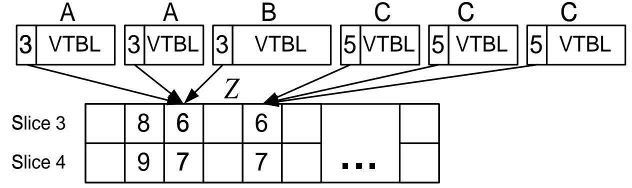

Figure 11: An example of CPQE of Figure 10 after the second inlining

We finally note that CPQE is not limited to C++, nor to the VTBLs implementation. It is possible to store the encoding of each type using this two levels scheme. In doing so, we presume, just as in all previously published algorithms, that the unique pointer p_a can be produced *deus ex machina* from the object o. Then p_a points to p'_a which points to array id''_a. This two levels scheme is similar to the one depicted in Figure 11, only each type has a single VTBL.

It should be stressed that the two-level structure makes CPQE slower than PQE. The slowdown is justified in the C++ sharing alternative. In other object layout models, the time penalty of the extra indirection has to be weighed against the potential saving in encoding length of CPQE.

7. Results

Table 2 compares the encoding length in bits of PQE and CPQE with that of other algorithms. PQE, and its variant CPQE, are presented with all optimizations, including length-optimization, heterogeneous encoding and the newly suggested inlining optimization.

We see that PQE encoding length improves on all previously published algorithms. As explained above, the memory requirements of PQE is zero for all SI hierarchies. As can be seen in the table, zero memory footprint occurs even in IDL, which is MI. The median improvement with the next best algorithm, NHE, is by 37%, while the average improvement is 50%.

In addition, PQE test time is constant, whereas NHE, which is based on bit-vector encoding, is non-constant. In the Eiffel4 hierarchy the constant BPE has a total space requirements of 39KB, compared to 16KB in PQE. These differences are significant since subtyping tests are frequent, so the encoding data should be cached.

If PQE is not optimized by inlining, then its encoding length will increase by 16 bits. Without this optimization, PQE is better than NHE in 8 out of the 13 hierarchies. In two hierarchies (LOV and Geode), the encoding length of NHE is 1 bit shorter than PQE, in two hierarchy (Self and JDK 1.18) it is 2 bits shorter, and in one hierarchy (Eiffel4) it is 9 bits shorter.

We see that in the first four hierarchies CPQE is not as efficient as PQE. The reason is that its two-level structure imposes an encoding length of 8 bits even in very degenerate hierarchies.

Let m denote the number of distinct arrays id' in CPQE. Then, we found that with the exception of Eiffel4, Geode, and JDK 1.30, m was less than 256. Thus, in all these hierarchies, p'_a can be represented as a single byte. We see that CPQE improves the encoding

Hierarchy	**CPQE**	**PQE**	NHE	BPE	PE	DAG[a]	Closure[b]	BM
IDL	8	0	17	32	96	7	27	66
Laure	8	6	23	63	128	10	74	295
Unidraw	8	2	30	63	96	8	31	613
JDK 1.1	8	1	19	32	64	9	26	225
Self	9	39	53	126	344	12	329	1801
Ed	17	36	54	94	216	15	72	434
LOV	21	42	57	94	216	16	77	436
Eiffel4	27	65	72	157	312	15	97	1999
Geode	39	80	95	157	408	21	154	1318
JDK 1.18	9	25	39	94	128	13	48	1704
JDK 1.22	10	36	62	157	184	16	57	4339
JDK 1.30	18	41	65	188	216	16	57	5438
Cecil	10	22	58	94	192	13	65	932

[a] $(|\prec_d|\lceil\log n\rceil)/n$
[b] $(|\prec|\lceil\log n\rceil)/n$

Table 2: The encoding length of different algorithms

length of PQE by factors ranging between 2 and 4.3.

Table 3 compares the encoding creation time of CPQE with that of NHE and BPE. The creation time of PQE and PE is the same as CPQE and BPE, respectively.

Hierarchy	(C)PQE [a]	NHE [b]	(B)PE [c]
IDL	1	-	5
Laure	4	21	9
Unidraw	1	93	10
JDK 1.1	1	19	10
Self	122	1367	22
Ed	77	136	12
LOV	95	168	10
Eiffel4	299	-	29
Geode	668	1902	28
JDK 1.18	29	-	26
JDK 1.22	140	-	77
JDK 1.30	187	-	90
Cecil	50	-	13

[a] 266 Mhz Pentium II
[b] 500 Mhz 21164 Alpha
[c] 750 Mhz Pentium III, user time in Linux

Table 3: Encoding creation time in milliseconds of different algorithms

The comparison is not easy, since the algorithms were run on different machines. Algorithm 1 was written in C++ based on the PQ-tree implementation of Leipert [27]. More experimentation is required before a faithful and fair comparison is possible. It appears as if PQE, which is based on a linear algorithm, outperforms the quadratic NHE-algorithm. PE and BPE, which use a fast implementation of set unions and intersections using bit-vector operations, seem to be the fastest. The Geode hierarchy is toughest for PQE and NHE. In this hierarchy, the average time for processing a type is less than one millisecond in PQE. In all benchmarks the time for computing the PQ-encoding is less than a second.

Table 4 presents values of some internal parameters in the execution of PQE and CPQE. The total number of distinct slices is k. The number of types in the first slice is denoted by n_1, while n_2 is the number of types in slices whose size is smaller than 8. The value $k' < k$ is the number of slices not which do not fall in these two categories.

Hierarchy	k	n_1/n	n_2/n	n_2	m	k'
IDL	1	100.0%	0.0%	0	0	0
Laure	2	98.0%	2.0%	6	7	0
Unidraw	2	99.7%	0.3%	2	2	0
JDK 1.1	2	99.6%	0.4%	1	1	0
Self	13	97.2%	1.7%	31	63	1
Ed	10	87.8%	4.6%	20	145	2
LOV	12	86.2%	6.0%	26	164	2
Eiffel4	11	89.1%	0.5%	9	376	7
Geode	16	86.0%	1.8%	24	419	7
JDK 1.18	6	97.5%	0.5%	9	74	2
JDK 1.22	8	97.6%	0.3%	12	235	3
JDK 1.30	8	97.7%	0.3%	17	286	3

Table 4: Internal parameters of PQE and CPQE

We see in the table that a very significant portion of all types fall in the first slice; very small fraction of types fall into the small slices. The encoding length of PQE is $8k' + n_2$. In CPQE the total size of array Z is $(8(k'-1)+n_2) \times m$ (recall that a second inlining was performed), and the encoding length is therefore

$$(8(k'-1)+n_2)\times m/n+\lceil\log m\rceil.$$

8. Conclusions and Future Research

The PQE algorithm improves the encoding length, creation time, test time and instruction count of NHE, the most space efficient previously published encoding algorithm. The CPQE variant, especially tailored for object layout like the one in C++, reduces the encoding length even further.

The main problem which this paper leaves open is an incremental algorithm for the subtyping problem, as required by languages such as JAVA, in which types may be added as leaves at run time. It turns out that the PQ-data structure is not susceptible to efficient updates of this sort.

On the theoretical side, it would be very interesting to see any non-trivial lower bound for the encoding length.

Acknowledgements We are truly indebted to Jan Vitek for his help in making the initial data set available to us, for contributing the three additional JAVA hierarchies, for help in running the experiments, and for many stimulating discussions!

9. References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 253–262, Portland, Oregon, 31 May–2 June 1989.

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

[3] G. D. Battista and R. Tamassia. On-line planarity testing. Technical Report CS-89-31, Brown University - Department of Computer Science, May 1989.

[4] G. D. Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In M. S. Paterson, editor, *Automata, Languages and Programming, 17th International Colloquium*, volume 443 of *Lecture Notes in Computer Science*, pages 598–611, Warwick University, England, 16–20 July 1990. Springer-Verlag.

[5] G. D. Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, Apr. 1996.

[6] 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.

[7] C. Capelle. Representation of an order as union of interval orders. In *ORDAL'94 (Orders, Algorithms and Applications), LNCS 831*, pages 143–162. Springer, 1994.

[8] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *16th Simposium on Principles of Programming Languages, POPL'89*, pages 202–212, Austin, Texas, Jan. 1989. ACM SIGPLAN — SIGACT, ACM Press.

[9] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In A. Paepcke, editor, *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 271–287, Washington, DC, USA, Sept. 26 - Oct. 1 1993. OOPSLA'93, ACM SIGPLAN Notices 28(10) Oct. 1993.

[10] C. Chambers. The Cecil language, specification and rationale. Technical report, University of Washington, Seattle, 1993.

[11] N. H. Cohen. Type-extension tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13:626–629, 1991.

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

[13] E. W. Dijkstra. Recursive programming. *Numerische Mathematik*, 2:312–318, 1960.

[14] N. Eckel and J. Y. Gil. Empirical study of object-layout strategies and optimization techniques. In E. Bertino, editor, *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.

[15] A. Fall. Heterogeneous encoding. In *Proceedings of International KRUSE'95 Conference : Knowledge Use, Retrieval and Storage for Efficiency*, pages 162–167, Aug. 1995.

[16] 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.

[17] J. Y. Gil and P. Sweeney. Space- and time-efficient memory layout for multiple inheritance. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 256–275, Denver, Colorado, Nov.1–5 1999. OOPSLA'99, ACM SIGPLAN Notices 34(10) Nov. 1999.

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

[19] R. Grafl. CACAO: Ein 64bit JavaVM just-in-time compiler. Master's thesis, University of Vienna, 1996.

[20] M. Habib and L. Nourine. Bit-vector encoding for partially ordered sets. In *ORDAL'94 (Orders, Algorithms and Applications), LNCS 831*, pages 1–12. Springer, 1994.

[21] P. L. Hassan Kaci, Robert Boyer and R. Nasr. Efficient implementation of lattice operation. *ACM Transactions on Programming Languages and Systems*, 11:115–146, 1989.

[22] M. Junger, S. Leipert, and P. Mutzel. On computing a maximal planar subgraph using PQ-trees. Technical report, Informatik, Universität zu Köln, 1996.

[23] M. Junger, S. Leipert, and P. Mutzel. A note on computing a maximal planar subgraph using PQ-trees. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(7), 1998.

[24] A. Krall and R. Grafl. CACAO – a 64 bit JavaVM just-in-time compiler. In G. C. Fox and W. Li, editors, *PPoPP'97 Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997.

[25] A. Krall, J. Vitek, and R. N. Horspool. Efficient type inclusion tests. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 142–157, Atlanta, Georgia, Oct. 5-9 1997. OOPSLA'97, ACM SIGPLAN Notices 32(10) Oct. 1997.

[26] A. Krall, J. Vitek, and R. N. Horspool. Near optimal hierarchical encoding of types. In M. Akşit and S. Matsuoka, editors, *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.

[27] S. Leipert. PQ-trees, an implementation as template class in C++. Technical report, Informatik, Universität zu Köln, 1997.

[28] L. N. M. Habib, Y. Caseau and O. Raynaud. Encoding of multiple inheritance hierarchie and partial orders. In *Computational Intelligence*, volume 15, pages 50–62, 1999.

[29] B. Meyer. *EIFFEL the Language*. Object-Oriented Series. Prentice-Hall, 1992.

[30] O. Raynaud and E. Thierry. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Budapest, Hungary, June18–22 2001. ECOOP 2001, Springer Verlag.

[31] M. A. Schubert, P. L.K., and J. Taugher. Determining type, part, colour, and time relationships. *Computer*, 16 (special issue on Knowledge Representation):53–60, Oct. 1983.

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

[33] 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.