Efficient Subtyping Tests with PQ-Encoding

JOSEPH (YOSSI) GIL[1] and YOAV ZIBIN
Technion—Israel Institute of Technology

Given a type hierarchy, a *subtyping test* determines whether one type is a direct or indirect descendant of another type. Such tests are a frequent operation during the execution of object-oriented programs. The implementation challenge is in a space-efficient encoding of the type hierarchy that simultaneously permits efficient subtyping tests. We present a new scheme for encoding multiple and single inheritance hierarchies, which, in the standard benchmark hierarchies, reduces the footprint of all previously published schemes. Our scheme is called *PQ-encoding* (PQE) 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 PQE subtyping tests are constant time, and use only two comparisons. The encoding creation time of PQE also compares favorably with previous results. It is less than a second on all standard benchmarks on a contemporary architecture, while the average time for processing a type is less than one millisecond. However, PQE is not an incremental algorithm. Other than PQ-trees, PQE employs several novel optimization techniques. These techniques are applicable also in improving the performance of other, previously published, encoding schemes.

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

General Terms: Algorithms, Design, Measurement, Performance, Theory

Additional Key Words and Phrases: Casting, Encoding, Hierarchy, Inheritance, Partially Ordered Sets, PQ, PQE, Subtyping, Type inclusion

1. INTRODUCTION

One of the basic operations in the runtime environment of object-oriented (OO) programs is a *subtyping test*. Given an object o and a type b, a subtype test determines whether a, the runtime type of o, is a subtype of b, i.e., a is a direct or indirect descendant of b in the inheritance hierarchy. These subtyping tests (also known as *type inclusion* tests) are carried out at runtime, and are distinct from static subtyping tests done by the language type checker.

[1]Research supported in part by the generous funding of the Israel Science Foundation, grant No. 128/02.
Authors' address: Technion City, Haifa 32000, Israel.
Contact the authors for patent information.
A preliminary version of this paper was published in the proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01).

1.1 Subtyping Tests in OO Languages

A programmer may apply a subtyping test explicitly using dedicated constructs such as JAVA's [Arnold and Gosling 1996] **`instanceof`**, and SMALLTALK's [Goldberg 1984] **`isKindOf:`** method. In addition, several other language constructs are implemented using these tests. For example, subtyping tests are implicit in the execution of type cast operations, e.g., **`?=`** in the EIFFEL programming language [Meyer 1992] and **`dynamic_cast`** in C++ [Stroustrup 1997]. In JAVA for example, the lack of parametric polymorphism is a common source of such casts. When extracting an element *o* from a collection class, e.g., **`Vector`**, the type of *o* is **`Object`**. Therefore, it is typically necessary to cast *o* to the expected type.

Also, the covariant nature of array subtyping in JAVA renders subtyping tests necessary in assignments to elements of arrays whose dynamic type is unknown. Consider for example the following code fragment

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

It may be a bit surprising that the assignment to `x[1]` in `f` requires a subtyping test. To understand why, suppose that function `f` was invoked with a value of type `B[]` (an array of elements of type `B`) as an actual argument `x`, e.g.,

```
f(new B[3]);
```

This invocation is correct because type `B[]` 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.

Yet another construct which is implemented using subtyping tests is covariant overriding of arguments in EIFFEL. The compiler is inclined to make subtyping tests in conjunction with calls to methods that use this feature in order to ensure type-safety[2].

Finally, we note that subtyping tests may also be part of the implementation of exception handling in JAVA, C++, and other languages. The following code excerpt is an example of a **`try`** block, followed by a **`catch`** clause and block.

```
    try { f(); ...}
    catch(B b) {...}
```

Suppose that an object *o* is **`throw`**n from the **`try`** block, e.g., from within function `f()`. Then, the program should execute the **`catch`** block, but only if type `B` is a supertype of the thrown object's type. Thus, the **`catch`** clause can be implemented with a subtyping test. In JAVA, the subtyping test is with respect to the dynamic type of *o*, while in C++ it is with respect to the static type of *o*. However, in both cases, this test must be carried out at runtime.

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

1.2 Problem Definition

The problem we deal with is defined formally as follows. A *hierarchy* is a partially ordered set $(\mathcal{T},\preceq)$ where $\mathcal{T}$ is a set of types[3] 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 they are *comparable*, that a is a *subtype* of b and that b is a *supertype* of a. Given a hierarchy $(\mathcal{T},\preceq)$, $|\mathcal{T}| = n$, the *subtyping problem* is to build a data structure supporting queries of the sort $a \preceq b$. This data structure is called an *encoding* of the hierarchy.

The subtyping problem has enjoyed considerable attention recently (see e.g., [Kaci et al. 1989; Agrawal et al. 1989; Caseau 1993; Habib and Nourine 1994; Capelle 1994; Fall 1995; 1996; Krall et al. 1997; Vitek et al. 1997; Habib et al. 1999; van Bommel and Beck 2000; Raynaud and Thierry 2001; Filman 2002; Palacz and Vitek 2003; Corsaro and Cytron 2003]). The challenge of implementing subtyping tests is to simultaneously optimize its four complexity measures:

(1) *Space.* Encoding methods associate certain data with each type. We measure the average number of bits per type, also called the *encoding length*.
Note that we do not include in the measure the space consumed by each object. Although the space overhead per object depends on the object layout model, it is usually assumed that each object includes a pointer to a type information record.

(2) *Instruction count.* This is the number of machine instructions in the *test code*, on a certain hardware architecture. There are indications [Vitek et al. 1997] that the space consumed by the test code, which can appear many times in a program, can dominate the encoding length. An encoding is said to be *uniform*[4] 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.

(3) *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 encodings with constant-time tests (which are always uniform). To improve timing performance, loops of not constant-time tests may be unrolled, giving rise to non-constant instruction count, without violating the uniformity condition. (Bit-vector encoding, presented in Sec. 4, is an example of a uniform encoding which is not constant-time.)
In explicit subtyping tests and in type casts, the supertype b is known at compile time. Therefore, the generated subtyping test code can be *specialized*, by pre-computing values that depend only on b and using them in the test code. Specialization thus benefits both instruction count and test time, and may even reduce the encoding length.

(4) *Encoding creation time.* Another important complexity measure is the time for generating the actual encoding, which can be large. It is essential that a compiler will be able to finish its computation in a reasonable time. In many cases, this is not possible. For example, the problem of finding an optimal bit-vector encoding was proved to be intractable [Habib and Nourine 1994]. Heuristics of bit vector encoding [Kaci et al.

[3]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.
[4]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.

1989; Caseau 1993; Habib et al. 1999; Krall et al. 1997; Raynaud and Thierry 2001; Filman 2002] offer a tradeoff between creation time and encoding length.
Most of the previous work assumed, as we shall do here, that the entire type hierarchy is supplied at compile time. JAVA, E [Hollander et al. 2001] and many other languages allow types to be dynamically loaded at runtime. If the encoding creation time is sufficiently small, then the encoding can be *recomputed* whenever such a load occurs. An active research topic is to find truly incremental algorithms, which can quickly *update* the encoding.

1.3 Naïve Solutions

The most obvious (uniform) representation as a *binary matrix* gives constant subtyping tests, but the encoding length is n. This method is useful for small hierarchies and is used, for example, for encoding the JAVA *interfaces* hierarchy [Krall 2001] in CACAO 64-bit JIT compiler [Grafl 1996; Krall and Grafl 1997]. The quadratic space overhead becomes very noticeable in large hierarchies. For example, one of the hierarchies in our data set has 5500 types giving rise to 3.8MB binary matrix. The binary matrix encoding can be (non-uniformly) implemented using a zero encoding length and $O(n)$ instruction count: relying on specialization, the test code for $a \preceq b$ 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 binary matrix representation is *in practice* very sparse, and therefore susceptible to optimization. Nevertheless, for arbitrary hierarchies the encoding length is $\Omega(n)$, and thus the performance of the (not optimized) binary matrix implementation is asymptotically optimal. (The representation of some posets requires $\Omega(n^2)$ bits[5] since the number of partially ordered sets (*posets*) with n elements is $2^{\Theta(n^2)}$.)

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

Fig. 1.1 depicts a directed acyclic graph (DAG) representation of a hierarchy which will serve as the running example of this paper.

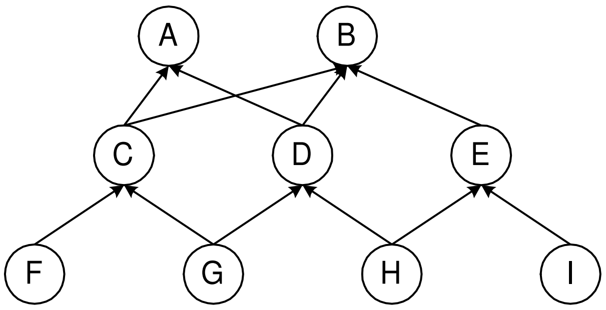

Fig. 1.1. A small example of a multiple subtyping hierarchy

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

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, the figure specifies (for example) that $\mathsf{G} \prec_{\mathrm{d}} \mathsf{C}$ and $\mathsf{H} \preceq \mathsf{A}$. In total in this hierarchy, $n = 9$, $|\prec_{\mathrm{d}}| = 11$, and $|\preceq| = 27$.

Another obvious solution to the subtyping problem is *DAG encoding*, which is based on the DAG defined by types as nodes and edges from $\prec_{\mathrm{d}}$. In this encoding, a list of parents is stored with each type, resulting in total space of $|\prec_{\mathrm{d}}|\lceil\log n\rceil$ bits[6] in an idealized bit-efficient representation. The DAG encoding length is therefore

$$\frac{|\prec_{\mathrm{d}}|}{n} \cdot \lceil\log n\rceil.$$

The average number of parents, $|\prec_{\mathrm{d}}|/n$, tends to be small; We will see that it is less than 2 in all the standard benchmark hierarchies. Unfortunately, a subtyping test in DAG encoding is $O(n)$ time.

Closure encoding presents another obvious tradeoff between space and test time. In this encoding, with each type we store the list of *all* of its ancestors using a simple sorted array representation. A subtyping test is then implemented using a binary search in $O(\log n)$ time. Since each entry in this array requires $\lceil\log n\rceil$ bits, the encoding length is

$$\frac{|\preceq|}{n} \cdot \lceil\log n\rceil.$$

Theoretically superior representations of this list include *Q-fast tries* [Willard 1984], which achieve deterministic $O(\sqrt{\log n})$ time, or the randomized stratified trees (also called *van Emde Boas data structure*) [van Emde Boas et al. 1977; van Emde Boas 1977], which achieve $O(\log\log n)$ time. Another alternative is perfect hash tables [Fredman et al. 1984] which give $O(1)$ lookup time. In moderately sized tables we expect the simple binary search algorithm to outperform the asymptotically better competitors. Also, these sophisticated data structures increase the encoding length by factors which can be prohibitively large.

The binary matrix, DAG, and Closure encodings 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, which occurs when the hierarchy DAG takes a tree or forest topology as mandated by the rules of languages such as SMALLTALK [Goldberg 1984] and OBJECTIVE-C [Cox 1986]. The general case of multiple inheritance is more difficult, and will be our main concern here.

1.4 PQ Encoding

Our chief result is based on *PQ-trees* [Booth and Leuker 1976], a technique for searching an ordering satisfying prescribed constraints: *PQ-encoding* (PQE) 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 single inheritance hierarchies and even in some multiple inheritance hierarchies.

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

The encoding creation time of PQE also compares favorably with previous results. It is less than a second on all standard benchmarks on a contemporary architecture, while the average time for processing a type is less than one millisecond. However, PQE is not an incremental algorithm. We are unaware of any efficient algorithm for updating the PQ encoding as types are added to the hierarchy.

The remainder of this article is organized as follows. Sec. 2 makes some pertinent definitions. The data set of the 13 hierarchies used as benchmarks is presented in Sec. 3. A survey of prior research is the subject of Sec. 4. This section also 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 Sec. 5. Sec. 6 presents our new optimization techniques, improving instruction count, test time and encoding length. The penultimate Sec. 7 presents the results of running these algorithms on our benchmark. Finally, some open problems and directions for future research are mentioned in Sec. 8. Appendix A demonstrates the inner workings of the PQ-tree data structure.

2. DEFINITIONS

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

$$\begin{aligned}
\mathrm{descendants}(a) &\equiv \{b \in \mathcal{T} \mid b \preceq a\} \\
\mathrm{ancestors}(a) &\equiv \{b \in \mathcal{T} \mid a \preceq b\} \\
\mathrm{children}(a) &\equiv \{b \in \mathcal{T} \mid b \prec_{\mathrm{d}} a\} \\
\mathrm{parents}(a) &\equiv \{b \in \mathcal{T} \mid a \prec_{\mathrm{d}} b\}
\end{aligned} \tag{2.1}$$

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

$$\begin{aligned}
\mathrm{level}(a) &\equiv 1 + \max\{\mathrm{level}(b) \mid b \in \mathrm{parents}(a)\} \\
\mathrm{height}(\mathcal{T}) &\equiv \max\{\mathrm{level}(a) \mid a \in \mathcal{T}\}
\end{aligned} \tag{2.2}$$

(In the above definition of $\mathrm{level}(a)$, 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 Fig. 1.1 we have

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

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

The following definitions will also become pertinent:

$$\begin{aligned}
\mathrm{roots}(\mathcal{T}) &\equiv \{a \in \mathcal{T} \mid \mathrm{parents}(a) = \emptyset\} \\
\mathrm{leaves}(\mathcal{T}) &\equiv \{a \in \mathcal{T} \mid \mathrm{children}(a) = \emptyset\}
\end{aligned} \tag{2.3}$$

In Fig. 1.1 we have

$$\mathrm{roots}(\mathcal{T}) = \{\mathsf{A}, \mathsf{B}\}$$
$$\mathrm{leaves}(\mathcal{T}) = \{\mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}\}.$$

3. DATA SET

To benchmark the algorithms, we started from the 9 multiple inheritance hierarchies used by Eckel and Gil [2000] in their benchmark of object layout techniques. Three new JAVA hierarchies (newer versions of the JAVA runtime environment), as well as the Cecil compiler hierarchy [Chambers 1993] were added to this benchmark. In total, our data set comprises large hierarchies drawn from six different OO languages. In particular, the set includes all multiple inheritance hierarchies used in previous studies of encoding schemes [Habib et al. 1999; Vitek et al. 1997; Krall et al. 1997]. Eckel and Gil [2000] gave a detailed description of these hierarchies. One of their findings is that many topological properties of typical hierarchies are similar to those of balanced trees. This makes it possible to find more efficient encodings 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.

The hierarchies in the data set are enumerated in ascending order of size in Table I. We see that the number of types ranges between 66 and 5,438. In total the 13 hierarchies represent over 19,500 types.

Hierarchy	n	$\lvert\prec_{\mathrm{d}}\rvert/n$	$\lvert\preceq\rvert/n$	α[a]	β[b]	γ[c]	$\lvert\mathcal{T}'\rvert/n$
IDL	66	0.98	3.83	8	6	7	15%
JDK 1.1	225	1.04	3.17	7	6	8	15%
Laure	295	1.07	8.13	16	11	9	18%
Ed	434	1.66	7.99	23	10	9	61%
LOV	436	1.71	8.50	24	9	9	62%
Unidraw	613	0.78	3.02	9	8	10	4%
Cecil	932	1.21	6.47	23	12	10	33%
Geode	1,318	1.89	13.99	50	13	11	75%
JDK 1.18	1,704	1.10	4.35	16	9	11	18%
Self	1,801	1.02	29.89	40	16	11	9%
Eiffel4	1,999	1.28	8.78	39	17	11	46%
JDK 1.22	4,339	1.19	4.37	17	9	13	22%
JDK 1.30	5,438	1.17	4.37	19	9	13	21%

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

Table I. Topological properties of hierarchies in the data set

Table I gives also some of the topological properties of the hierarchies. Examining the third column in the table we see that the average number of parents, $|\prec_{\mathrm{d}}|/n$, is always less than 2. On the other hand, the average number of ancestors, $|\preceq|/n$, can be 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.

We can learn a bit more on the topology of inheritance hierarchy by considering the set $\mathcal{T}'$ that is the *multiple inheritance core* of the hierarchy. Formally, a type is in the core if it has a descendant with more than one parent. Conversely, the set $\mathcal{T} \setminus \mathcal{T}'$ is a collection of maximal subtrees discovered in a bottom-up traversal of the hierarchy. It was previously noticed [Vitek et al. 1997] that encoding is easier if the core is considered first, and the *bottom trees* of $\mathcal{T} \setminus \mathcal{T}'$ are added to the encoding later. In Table I 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 reduces the runtime of our encoding algorithm.

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.

4.1 Encoding of Single Inheritance Hierarchies

4.1.1 *Relative numbering.* Perhaps the most elegant encoding algorithm is *relative numbering* [Schubert et al. 1983] (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 single inheritance hierarchy. For a type $b \in \mathcal{T}$, let r_b denote its ordinal (i.e., an integer in the range $1, \ldots, n$) in a postorder traversal of $\mathcal{T}$. A basic property of postorder traversal is that

$$r_b = \max\{r_a \mid a \preceq b\}. \tag{4.1}$$

Let l_b be defined by

$$l_b = \min\{r_a \mid a \preceq b\}. \tag{4.2}$$

Combining (4.1) and (4.2) with the fact that in postorder traversal the descendants of any type are assigned consecutively, we find that $a \preceq b$ if and only if

$$l_b \leq r_a \leq r_b. \tag{4.3}$$

Thus, in relative numbering, each type a is encoded by an interval $[l_a, r_a]$ as exemplified by Fig. 4.1.

In the figure we have (for example) that $\mathrm{descendants}(\mathsf{D}) = \{\mathsf{D}, \mathsf{H}, \mathsf{I}\}$. The r-descriptor of each of these descendants, i.e., $r_{\mathsf{D}} = 8$, $r_{\mathsf{H}} = 6$ and $r_{\mathsf{I}} = 7$, fall within the interval $[6, 8]$ associated with type D. No other r-descriptor falls in this interval.

Recall that since b is known at compile time, values which depend only on b can be pre-computed. Henceforth, such values are marked by a "#" prefix. With this notation, we write (4.3) as

$$\#l_b \leq r_a \leq \#r_b. \tag{4.4}$$

We note that $\#l_b$ and $\#r_b$ are compile-time constants so test (4.4) can be specialized by eliminating the memory fetches of these. In doing so, we find that l_b is not part of the encoding, bringing down the encoding length of relative numbering to $\lceil \log n \rceil$.

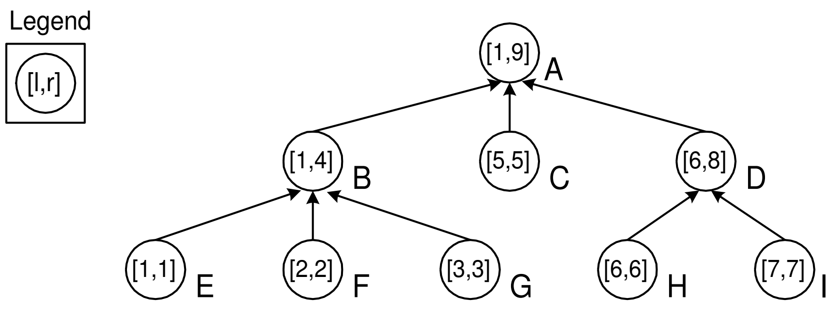

Fig. 4.1. Relative numbering in a tree hierarchy

Relative numbering is used in CACAO [Grafl 1996; Krall and Grafl 1997] to represent the JAVA class inheritance hierarchy [Krall 2001] (Recall that the binary matrix is used in CACAO for the interface hierarchy.) Range-compression [Agrawal et al. 1989], described below, is a generalization of relative numbering for multiple inheritance.

4.1.2 *Cohen's encoding.* A variant of Dijkstra's displays [1960] is Cohen's encoding [1991]. His encoding is yet another example of an algorithm initially designed for single inheritance, and later generalized to multiple inheritance. (The generalized algorithm, known as Packed Encoding [Vitek et al. 1997], will be described below.) Cohen's encoding relies on hierarchies being relatively shallow, and more so, on types having a small number of ancestors. As Table I shows, this is indeed the case in some of our multiple inheritance hierarchies. A type a is associated an array r_a of size

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

(in single inheritance, $\mathrm{level}(a) = |\mathrm{ancestors}(a)|$), with entries for each

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

Specifically, each type b, $b \succeq a$, is stored in location $\mathrm{level}(b)$ in array r_a. Thus, the test whether $b \succeq a$ is carried out by checking whether b indeed occurs in location $\mathrm{level}(b)$ of array r_a. The encoding is optimized by storing not 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 pseudo-array access. Pseudo-arrays are only used if the index i is always known at compile time. Therefore, a pseudo-array access is the same as record member selection, and is no slower than a non-pseudo array access. (If several pseudo-array entries are packed together in a single byte, then the required shift and mask operations may slow down this operation in comparison to normal array accesses.)

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

$$r_a @\ l_b = \#\,\mathrm{id}_b\,. \tag{4.5}$$

The test $a \preceq b$ is carried out by checking that $l_a \geq \#l_b$ and then that (4.5) holds. Note that l_b is known at compile time.

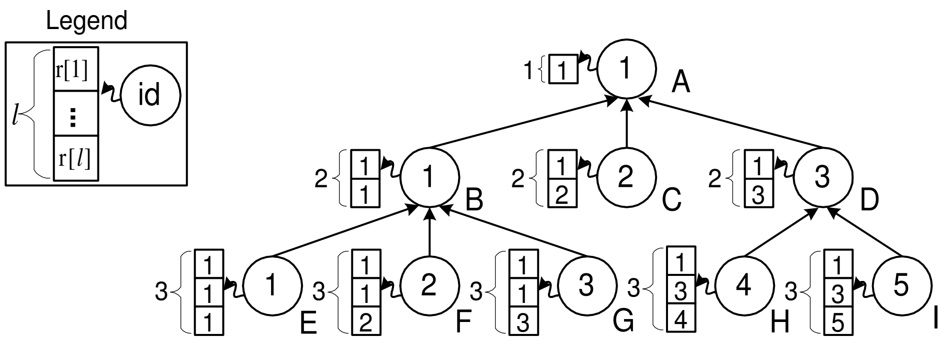

Fig. 4.2. Cohen's encoding of the tree hierarchy of Fig. 4.1

Consider for example the 3-level tree hierarchy depicted in Fig. 4.2. As shown in the figure, each type has a (pseudo-) array with at most 3-identifiers. The pseudo array of type H in the 3*rd* level has three entries: $r_{\mathsf{H}} @ \, 1 = 1$ since the ancestor of H at the 1*st* level is A, and $\mathrm{id}_{\mathsf{A}} = 1$. Similarly, the ancestor at the 2*nd* level is D, $\mathrm{id}_{\mathsf{D}} = 3$. The last entry of this array stores the id of H.

Also observe that in the figure how id's are reused at different levels. For example, for types C and F which are at different levels we have that $\mathrm{id}_{\mathsf{C}} = \mathrm{id}_{\mathsf{F}} = 2$.

The array boundary check $l_a \geq \#l_b$ in Cohen's encoding is inelegant. We observe that it can be eliminated at 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.

Jalapeño [7] [Alpern et al. 2001], IBM's implementation of the JAVA virtual machine (JVM), uses Cohen's algorithm for subtyping tests where the supertype is a class. The main reason is that this encoding is incremental, whereas vanilla relative numbering is not.

4.2 Encodings of Multiple Inheritance Hierarchies

4.2.1 *Packed encoding.* Cohen's algorithm was generalized to the multiple inheritance setting by Vitek et al. [1997] into what is called *Packed Encoding* (PE) and *Bit-Packed Encoding* (BPE), which are both constant-time methods. Cohen's algorithm, PE, BPE and our algorithm share a common theme: *slicing*, in which the set $\mathcal{T}$ 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 $a \preceq b$, $a \in \mathcal{T}$ 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 $a \in \mathcal{T}$ 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

[7]Jalapeño is now called Jikes RVM (Research Virtual Machine)

all $b \in \mathrm{ancestors}(a)$, index s_b stores id_b, i.e.,

$$r_a[s_b] = \#\,\mathrm{id}_b\,. \qquad (4.6)$$

A necessary and sufficient condition for $a \preceq b$ to hold is then (4.6). 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 set of ancestors. Checking the fifth column of Table I we see that some hierarchies require 40 slices or more.

Comparing (4.6) with (4.5), 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 $\mathrm{height}(\mathcal{T})$ anti-chains[8], while PE partitions the hierarchy into anti-chains where no two elements in an anti-chain have a common descendant. Fall [1995], 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 by Vitek et al. [1997] along with the constant time subtype test made PE viable. Based on this heuristic, Palacz and Vitek [2003] recently gave an incremental implementation of PE.

Vitek, Horspool and Krall's PE algorithm 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 multiple inheritance is heavily used. Thus, even though the general problem is intractable, their heuristics often finds an optimal solution.

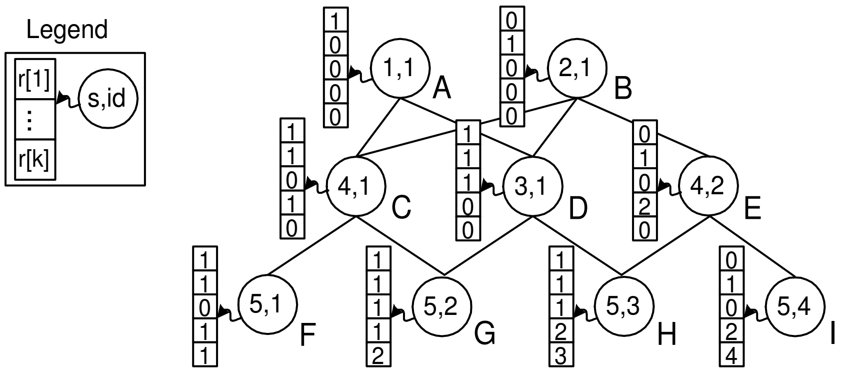

Fig. 4.3. PE representation of the hierarchy of Fig. 1.1

Consider Fig. 4.3 for an example of PE representation of the hierarchy of Fig. 1.1. 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 G (for example) has five ancestors.

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 (4.6) becomes a pseudo-array access:

$$r_a@\ s_b = \#\,\mathrm{id}_b\,. \qquad (4.7)$$

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

Starting from Fig. 4.3 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.

4.2.2 *Bit-vector encoding.* One of the most explored directions in prior art is *bit-vector encoding* [Habib and Nourine 1994; Kaci et al. 1989; Caseau 1993; Habib et al. 1999; Krall et al. 1997; Raynaud and Thierry 2001; Filman 2002]. In this scheme, each type a is encoded as a vector vec_a of β 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 $a \preceq b$ holds if and only if $\phi(a) \supseteq \phi(b)$, which can be easily checked by masking vec_b against vec_a, specifically, applying the test:

$$\mathrm{vec}_b \;\textbf{ and }\; \mathrm{vec}_a = \mathrm{vec}_b\,. \tag{4.8}$$

Fig. 4.4 gives an example of a bit-vector encoding of the hierarchy of Fig. 1.1 which uses 6 genes.

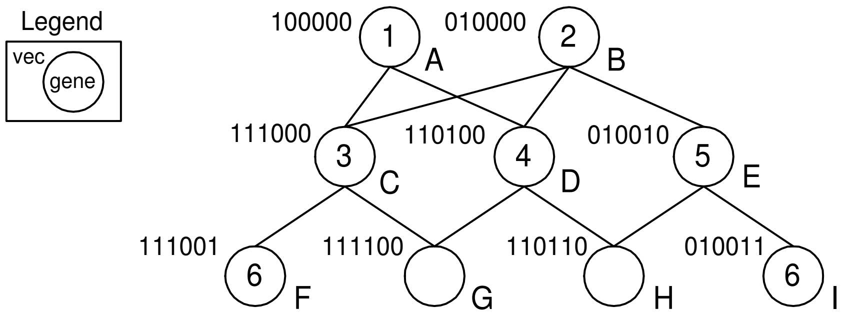

Fig. 4.4. Bit-vector encoding of the hierarchy of Fig. 1.1. (We only write the genes a type adds to its parents.)

The set of genes of type D (for example) is $\phi(\mathsf{D}) = \{1, 2, 4\}$, and thus $\mathrm{vec}_\mathsf{D} = 110100$. The genes of the ancestors of type D are contained in $\phi(\mathsf{D})$, and every other type has at least one gene not in $\phi(\mathsf{D})$.

Bit-vector encoding effectively embeds the hierarchy in the lattice of subsets of the set $\{1, \ldots, \beta\}$. It is always possible to do so by setting $\beta = n$ and in letting vec_a be the row of the binary matrix which corresponds to a. A simple counting argument shows that β must depend on the size of the hierarchy. Hence, bit-vector encoding is not constant-time, but it is uniform. For efficiency reasons, the implicit loop in (4.8) can be unrolled, giving rise to a non-constant instruction count.

The challenge is in finding the minimal β for which such an embedding of the hierarchy in a lattice is possible. Although the problem is NP-hard [Habib and Nourine 1994], several good heuristics were proposed, including Kaci et al. [1989] work, Caseau's *Compact Hierarchical Encoding* [1993], later improved by Habib et al. [1999]. Currently, *Near Optimal Hierarchical Encoding* (NHE), due to Krall et al. [1997], is the best general bit vector encoding. Better results can be obtained for the special case of single inheritance by *dichotomic encoding* [Raynaud and Thierry 2001] and its *polychotomic encoding* generalization [Filman 2002].

4.2.3 *Range compression.* 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 et al. [1989] 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,

$$\Phi(b) = \{\mathrm{id}_a \mid a \in \mathrm{descendants}(b)\}, \tag{4.9}$$

is represented by an array of consecutive disjoint intervals

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

Thus, $a \preceq b$ if and only if

$$\#l_b@\ i \leq \mathrm{id}_a \leq \#r_b@\ i \tag{4.10}$$

holds for *some* i, $1 \leq i \leq \gamma(b)$. In single inheritance, all descendants of a type are assigned consecutive numbering in a postorder traversal, and therefore the set (4.9) can be represented using a single interval. The encoding then degenerates to relative numbering.

Fig. 4.5 gives a range-compression encoding of the hierarchy of Fig. 1.1. 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, $l_{\mathsf{B}} = \langle 1, 5\rangle, r_{\mathsf{B}} = \langle 3, 9\rangle$ and $\gamma(\mathsf{B}) = 2$.

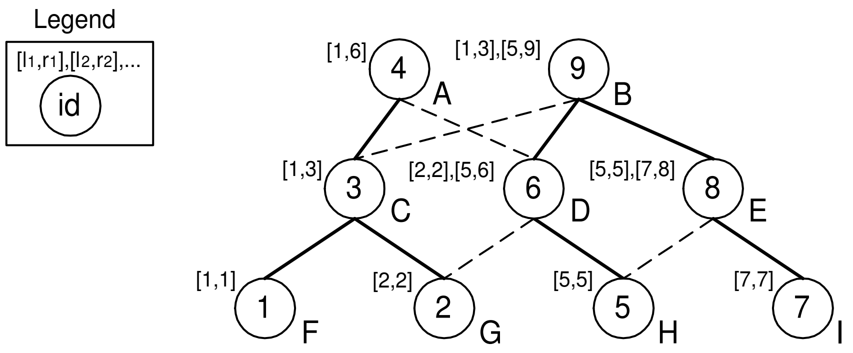

Fig. 4.5. Range-compression encoding of the hierarchy of Fig. 1.1. (Edges of the spanning forest are in bold.)

Examining (4.10) 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 $\gamma(b)$, which can be in the order of n. In all of our hierarchies however, the average of $\gamma(b)$ over all $b \in \mathcal{T}$ was always less than 2. The maximal $\gamma(b) = 55$ was found in the Geode hierarchy.

The usual straightforward implementation of range compression requires $O(\gamma(b))$ time. If $\gamma(b)$ is large then a binary search on (4.10) reduces the time to $O(\log \gamma(b))$. Note that this faster implementation does nothing to improve the instruction count in the specialized implementation which remains $\Omega(\gamma(b))$.

Other not -constant encoding techniques were used in large data- and knowledge-bases, e.g., modulation techniques [Kaci et al. 1989; Fall 1995], sparse terms encoding [Fall 1996], and representation using union of interval orders [Capelle 1994]. The common objective is a small average, rather than worst-case, time for testing, which may be considered unsuitable for an implementation of the runtime environment of OO languages.

5. PQ-ENCODING

This section describes PQ-encoding (PQE), our new encoding scheme, which achieves the smallest space requirements among all previously published encodings. In a nut-shell, PQE combines the ideas of *relative numbering* with *slicing* as used in PE and BPE.

The essence of relative numbering is in the (global) *consecutiveness property*, i.e., the requirement that the descendants of any given type are numbered consecutively; this property makes it possible to represent the entire set of descendants as a pair of two integers: the end points of the interval. In single inheritance, the consecutiveness property is satisfied by the numbering of a simple postorder visit. For multiple inheritance hierarchies, it is only natural to try to generalize relative-numbering by replacing the postorder visit by a DFS of the inheritance graph. Two issues must be addressed in order to make such a generalization work.

(1) The encoding must chose one DFS visit of the inheritance hierarchy from many different such visits, which may lead to *essentially* different orderings of the nodes. (Note that different DFS visits of a single inheritance hierarchy give rise to essentially the same relative numbering encoding.)
(2) In general, it is not guaranteed that there exists any single numbering which satisfies the consecutiveness property. Therefore, the generalization must handle hierarchies in which the consecutiveness property cannot be satisfied.

As explained in the previous section, the range-compression technique of Agrawal et al. [1989] addresses these issues by applying a heuristic for choosing a DFS. Also, if this heuristic fails, i.e., in case the set of descendants of a certain type does not fill up a single range, then this set is represented as a collection of ranges.

PQE uses two techniques in the generalization of relative numbering:

(1) Employing a sophisticated algorithmic tool, namely *PQ-trees*, for efficiently considering together even an exponential number of orderings. In particular, if there exists *any ordering* of the hierarchy which satisfies the global consecutiveness property, then the PQ-trees technique is guaranteed to find one in $O(|\preceq|)$ time.
(2) Using the slicing technique to make sure that subtyping tests require constant time, even if no ordering which satisfies the consecutiveness property exists.

We first (Sec. 5.1) explain data structure used by the encoding and the implementation of constant time subtyping tests. Sec. 5.2 explains the slicing technique in greater detail. In Sec. 5.3 we describe the PQ-trees data structure. Sec. 5.4 shows how it is used to find a PQ-encoding.

5.1 Subtyping Tests in the PQ-Encoding

The set of types is partitioned into disjoint *slices*, and each type has a distinct id with respect to *each* of the slices. Specifically, let k denote the number of slices. Then, for each type three pieces of data are stored:

(1) an integer s_a, $1 \leq s_a \leq k$, which is the number of the slice to which a belongs,
(2) a pseudo-array id_a of length k, such that $\mathrm{id}_a@\ i$ is the id of type a with respect to slice i, $1 \leq \mathrm{id}_a@\ i \leq n$ and
(3) an interval $[l_a, r_a]$, represented as a pair of integers, $1 \leq l_a \leq r_a \leq n$, which are the smallest and the largest id (with respect to slice s_a) of the descendants of a.

In total $k + 3$ integers are stored for each type. Our main effort, for which we will harness PQ-trees, is to minimize the number of slices k. Fine tuning of the representation as discussed below in Sec. 6 may make the encoding length less than $(k+3)\lceil \log n \rceil$.

The selection of id's and intervals is such that subtyping tests can be made using two comparisons. Specifically, $a \preceq b$ if and only if

$$\#l_b \leq \mathrm{id}_a@\ s_b \leq \#r_b. \tag{5.1}$$

Thus, subtyping tests begins with the pseudo-array access $\mathrm{id}_a@\ s_b$ which finds the id of a with respect to the slice of b. Then we check if this id is in the range $[l_b, r_b]$ of descendants of b.

Since b is known at compile time, testing (5.1) requires exactly the same number of RISC instructions as relative numbering (4.4). Note that the two comparisons in (5.1) are between integers of fixed size, which needs not to be longer than $\log_2 n$ bits. In each of the hierarchies in our data set, 16 bits comparisons are sufficient, and it is extremely unlikely that hierarchies will ever contain more than 2^{32} types. In contrast, subtyping tests in a bit vector encoding scheme may be implemented in only one comparison of bit vectors, but since the length of these vectors is not fixed (e.g., 95 bits for Geode in the NHE scheme), this comparison must be repeated several times.

Also note that the test (5.1) is similar to array bounds checking. Therefore, it may be possible to optimize the implementation on an architecture with dedicated instructions for this kind of check. Such architectures include the Intel 80186+ series (`bound` mnemonic) and the Motorola 680x0 series as well as Motorola 68300 (`chk2` mnemonic).

Palacz and Vitek [2003] explain that for reasonably sized hierarchies, including all hierarchies in standard benchmarks, it is possible to implement the check (5.1) using a single jump instruction instead of two. We now give a slightly improved version of the technique they describe. Consider the predicate

$$(x_1 > y_1) \wedge (x_2 > y_2) \tag{5.2}$$

where x_i and y_i, $i = 1, 2$ are 15-bit integers. Then we pack x_1 and x_2 (respectively, y_1 and y_2) together in a single 32-bit integer x (respectively, y). Let $x = 2^{16}x_1 + x_2$, and $y = 2^{16}y_1 + y_2 + M$, where $M = 2^{32} + 2^{16}$. Then, the expression

$$(y - x)\ \ \textbf{and}\ \ M \tag{5.3}$$

is zero if and only if (5.2) holds. Checking whether (5.3) is zero requires a subtraction, a bit mask operation and a jump.

5.2 Slicing

The essence of slicing is that when the global consecutiveness property cannot be satisfied, we maintain a weaker, local property. More specifically, given a slice $S \subseteq \mathcal{T}$, let $\varphi(S) \subseteq \wp(\mathcal{T})$ be the set of sets of descendants of types in this slice, i.e.,

$$\varphi(S) = \{\mathrm{descendants}(t) \mid t \in S\}.$$

DEFINITION 5.1. *A slice S satisfies the* local consecutiveness property *if there is an ordering of $\mathcal{T}$ in which all members of $\varphi(S)$ are consecutive.*

A partitioning of $\mathcal{T}$ into slices which satisfies the local consecutiveness property always exists, since this property trivially hold for singletons. The local consecutiveness property makes it possible to represent the set of all descendants of any type using merely two integers, and implement every type check as interval inclusion test, as done in (5.1).

Fig. 5.1 describes a PQE representation of the running example. The global consecutiveness property holds in this case— only one slice is used—and each type has a single id. To check whether G is a descendant of A, we only need to check whether $\mathrm{id}_{\mathsf{G}} = 4$ falls in the range $[l_{\mathsf{A}}, r_{\mathsf{A}}] = [1, 6]$.

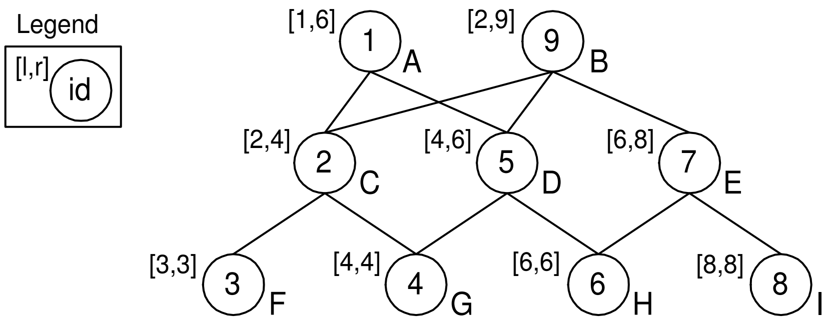

Fig. 5.1. PQ-encoding of the hierarchy of Fig. 1.1

The numbering of Fig. 5.1 was found using a PQ-tree, a data structure that maintains a set of *orderings* (permutations) of some *universe*. Initially, the PQ-tree represents the set of all $9!$ orderings of types $\mathsf{A}, \ldots, \mathsf{I}$. The tree is updated progressively, narrowing down this set, to reflect the constraints that the descendants of all types are consecutive. For each of the types, we try to update the PQ-tree so that it represents only the orderings in which the descendants of this type are consecutive.

In the running example, this update process never fails; we therefore ended in a PQ-tree representation of all orderings which satisfy the global consecutiveness property. The ordering depicted in Fig. 5.1 was obtained by picking one of these orderings.

If an ordering which satisfies the global consecutiveness property exists, then our algorithm is guaranteed to find it. In the general case, we use a greedy heuristic for minimizing the total number of slices, and hence the encoding length: "try to make the current slice as large as possible without violating the local consecutiveness property".

Fig. 5.2 shows our running example hierarchy augmented with a new type N, added as an additional ancestor of E.

There is no ordering of the types in this hierarchy which satisfies the global consecutiveness property. Therefore, PQ-encoding is inclined to use two slices:

$$\begin{aligned} S_1 &= \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}\}, \\ S_2 &= \{\mathsf{N}\}. \end{aligned} \tag{5.4}$$

We see that the greedy heuristic assigns all types but N to the first slice. In Fig. 5.2 the slice of each type is written to its left.

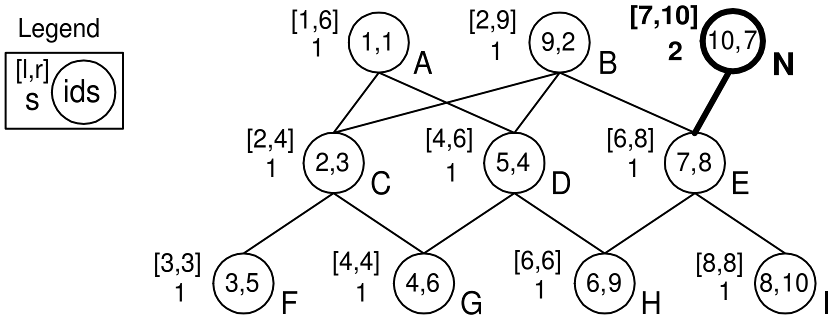

Fig. 5.2. A two slices PQ-encoding of the hierarchy of Fig. 1.1 augmented with a new type N

Comparing Fig. 5.2 with Fig. 5.1 we also see that each type has now two id's instead of one. To check whether G is a descendant of N, we first surmise that the slice of N is 2. We therefore use the *second* id of G, $\mathrm{id}_{\mathsf{G}}@\,2 = 6$ and check whether it falls in the range $[l_{\mathsf{N}}, r_{\mathsf{N}}] = [7, 10]$.

5.3 PQ-Trees

PQ-trees were invented by Booth and Leuker [1976] [9] who used them to test for the *consecutive 1*'s property in binary matrices of size $r \times s$, in $O(k + r + s)$ time, where k is the number of 1's in the matrix. Booth and Leuker's 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 [Battista and Tamassia 1989; 1990] and maximum planar embedding [Battista and Tamassia 1996; Junger et al. 1996; 1998].

DEFINITION 5.2. *A PQ-tree over a universe $\mathcal{T}$ is either a special $\perp$ symbol, or an ordered tree data-structure with a leaf for every member of $\mathcal{T}$, and such that each internal node is labelled as either a* Q-node *or a* P-node.

A PQ-tree represents a set of orderings of $\mathcal{T}$. The $\perp$ symbol represents an empty set of orderings. Otherwise, each Q-node in the data-structure represents the requirement that all children of the node must occur in the order they occur in the tree or in reverse order. A P-node represents the requirement that these children must occur together, but in no specific order.

The *universal PQ-tree*, denoted $\mathcal{P}^{\top}$ represents the set of all orderings; it has a P-node as a root and a leaf for every member of $\mathcal{T}$. A more interesting example is given by Fig. 5.3 which depicts a PQ-tree over the universe $\mathcal{T} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}$. This tree represents the requirement that A, B, and C must occur together, either in this order or in the reverse order $\langle \mathsf{C}, \mathsf{B}, \mathsf{A} \rangle$.

Let $\mathrm{consistent}(\mathcal{P})$ denote the subset of orderings of the universe $\mathcal{T}$ which is represented by a PQ-tree $\mathcal{P}$. The specific ordering of $\mathcal{T}$ obtained by a DFS traversal of $\mathcal{P}$, $\mathcal{P} \neq \perp$, is denoted $\mathrm{frontier}(\mathcal{P})$.

In Fig. 5.3 we have

$$\mathrm{frontier}(\mathcal{P}) = \langle \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E} \rangle, \tag{5.5}$$

[9] In fact, Lempel et al. [1967] were the first to coin the term *PQ-expressions*. PQ-trees are nothing more than an efficient representation of PQ-expressions.

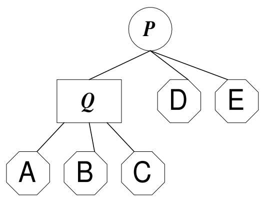

Fig. 5.3. A PQ-tree over the universe $\mathcal{T} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}$, with a single P-node (depicted as a circle), a Q-node (depicted as a rectangle) and five leaves (depicted as octagons)

and

$$\begin{aligned}\mathrm{consistent}(\mathcal{P}) = \{ &\langle \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E} \rangle, \langle \mathsf{C}, \mathsf{B}, \mathsf{A}, \mathsf{D}, \mathsf{E} \rangle, \langle \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{E}, \mathsf{D} \rangle, \langle \mathsf{C}, \mathsf{B}, \mathsf{A}, \mathsf{E}, \mathsf{D} \rangle, \\ &\langle \mathsf{E}, \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D} \rangle, \langle \mathsf{E}, \mathsf{C}, \mathsf{B}, \mathsf{A}, \mathsf{D} \rangle, \langle \mathsf{D}, \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{E} \rangle, \langle \mathsf{D}, \mathsf{C}, \mathsf{B}, \mathsf{A}, \mathsf{E} \rangle, \\ &\langle \mathsf{D}, \mathsf{E}, \mathsf{A}, \mathsf{B}, \mathsf{C} \rangle, \langle \mathsf{D}, \mathsf{E}, \mathsf{C}, \mathsf{B}, \mathsf{A} \rangle, \langle \mathsf{E}, \mathsf{D}, \mathsf{A}, \mathsf{B}, \mathsf{C} \rangle, \langle \mathsf{E}, \mathsf{D}, \mathsf{C}, \mathsf{B}, \mathsf{A} \rangle \}.\end{aligned} \tag{5.6}$$

There are two transformations of a PQ-tree $\mathcal{P}$ which preserve $\mathrm{consistent}(\mathcal{P})$: 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 $\mathcal{P}_1$ by a series of these transformations. Thus, $\mathrm{consistent}(\mathcal{P})$ can be more formally defined as

$$\mathrm{consistent}(\mathcal{P}) = \{\mathrm{frontier}(\mathcal{P}') \mid \mathcal{P}' \equiv \mathcal{P}\}, \tag{5.7}$$

and $\mathrm{consistent}(\perp) = \emptyset$.

A *constraint* (on orderings) is the requirement that certain elements of the universe occur together. A constraint is represented simply as a subset of the elements of the universe. (In our application, each constraint will be the set of descendants of a given type.) We denote the set of all orderings that satisfy constraint I as $\Pi(I)$.

Consider the special cases $I = \emptyset$, $|I| = 1$, or $I = \mathcal{T}$. Then, it is easy to see that all orderings satisfy I. Thus, in all these cases,

$$\Pi(I) = \mathrm{consistent}(\mathcal{P}^{\top}).$$

More generally,

FACT 5.3. *For every constraint I there exists a PQ-tree $\mathcal{P}$, $\mathcal{P} \neq \perp$, such that*

$$\Pi(I) = \mathrm{consistent}(\mathcal{P}).$$

PROOF. The root of $\mathcal{P}$ is a P-node whose children are the leaves $\mathcal{T} \setminus I$ and another P-node whose children are the leaves I. □

Let $\mathbf{I} \subseteq \wp(\mathcal{T})$ be a collection of constraints. Then, $\mathbf{\Pi}(\mathbf{I})$ is the set (which may be empty) of orderings that satisfy *all* constraints in $\mathbf{I}$, i.e.,

$$\mathbf{\Pi}(\mathbf{I}) = \bigcap_{I \in \mathbf{I}} \Pi(I).$$

(In our application, each slice S generates a collection of constraints $\varphi(S)$.)

For example, the requirement that types A and B are consecutive, and that types B and C are consecutive, is represented by

$$\mathbf{I} = \left\{ \{\mathsf{A}, \mathsf{B}\}, \{\mathsf{B}, \mathsf{C}\} \right\}.$$

It is easy to check that $\mathbf{\Pi}(\mathbf{I})$ is the set (5.6) of orderings consistent with the PQ-tree of Fig. 5.3. Another example is the empty set of constraints which is satisfied by all orderings, i.e., $\mathbf{\Pi}(\emptyset) = \mathrm{consistent}(\mathcal{P}^\top)$. More generally,

THEOREM 5.4. (BOOTH-LEUKER (1976)) *Suppose that $|\mathcal{T}| > 2$. Then, for every collection of constraints $\mathbf{I}$ there exists a PQ-tree $\mathcal{P}$, and for every PQ-tree $\mathcal{P}$ there exists a collection of constraints $\mathbf{I}$ such that $\mathbf{\Pi}(\mathbf{I}) = \mathrm{consistent}(\mathcal{P})$.*

Algorithm 1 Compute the PQ-tree of all orderings which satisfy a set of constraints $\mathbf{I}$

Given a universe $\mathcal{T}$, and a set of constraints $\mathbf{I} \subseteq \wp(\mathcal{T})$, return a PQ-tree of all orderings of $\mathcal{T}$ which satisfy $\mathbf{I}$.

```
Procedure genTree($\mathbf{I}$)
$\mathcal{P} \leftarrow \mathcal{P}^\top$ // $\mathcal{P}^\top$ is the universal PQ-tree.
For all $I \in \mathbf{I}$ do
    $\mathcal{P} \leftarrow$ reduce($\mathcal{P}, I$)
od
return $\mathcal{P}$
```

Constructively, the tree $\mathcal{P}$ is generated from $\mathbf{I}$ using the iterative process described in Alg. 1. The heart of the algorithm is the procedure reduce which "adds" a constraint to a PQ-tree in a time proportional to the size of the constraint. We here use Booth and Leuker [1976] clever implementation of reduce as a black box.[10] Formally,

THEOREM 5.5. (BOOTH-LEUKER (1976)) *Given a PQ-tree $\mathcal{P}$ and a constraint I, the call reduce$(\mathcal{P}, I)$ runs in $O(|I|)$ time, while the value it returns satisfies*

$$\mathrm{consistent}(\mathsf{reduce}(\mathcal{P}, I)) = \mathrm{consistent}(\mathcal{P}) \bigcap \Pi(I).$$

Note that the set $\mathrm{consistent}(\mathcal{P}) \bigcap \Pi(I)$ may be empty, in which case reduce returns $\perp$.

5.4 Finding a PQ-encoding

There are hierarchies for which Alg. 1 can be used to find a PQ-encoding. A case in point is our running example (see Fig. 5.1): Each of the nine types in this example imposes a constraint on the permissible orderings. Singleton constraints are not interesting since they are satisfied by any ordering. The remaining constraints are

$$\begin{aligned} I_\mathsf{C} &= \{\mathsf{C}, \mathsf{F}, \mathsf{G}\}, \\ I_\mathsf{D} &= \{\mathsf{G}, \mathsf{D}, \mathsf{H}\}, \\ I_\mathsf{E} &= \{\mathsf{H}, \mathsf{E}, \mathsf{I}\}, \\ I_\mathsf{A} &= \{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{A}\}, \\ I_\mathsf{B} &= \{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{E}, \mathsf{I}, \mathsf{B}\}. \end{aligned} \tag{5.8}$$

The constraint I_C is that the descendants of type C must occur consecutively, etc. The call genTree$(\{I_\mathsf{C}, I_\mathsf{D}, I_\mathsf{E}, I_\mathsf{A}, I_\mathsf{B}\})$ returns the PQ-tree of Fig. 5.4. (Appendix A shows and explains the intermediate trees generated in the computation process.)

[10] The curious reader may care to know that reduce conducts a bottom-up traversal of the input tree, applying one of eleven PQ-tree transformations at each step.

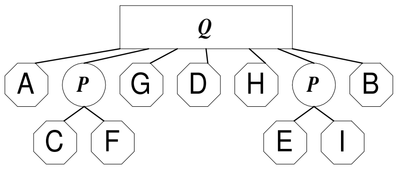

Fig. 5.4. The PQ-tree returned from the call genTree on the constraints (5.8)

The PQ-tree of the figure has one Q-node and two P-nodes with two children each. Therefore, this tree represents 8 different orderings, each satisfying the global consecutiveness property. The encoding of Fig. 5.1 uses the ordering represented by the tree's frontier $\langle$A, C, F, G, D, H, E, I, B$\rangle$.

In the general case, a PQ-encoding may require more than one slice. In such cases, the application of genTree to the set of all constraints $\varphi(\mathcal{T})$ returns $\perp$. (An example can be found in Appendix A.)

Alg. 2 generates a PQ-encoding for *any* given hierarchy $(\mathcal{T}, \preceq)$.

Algorithm 2 Compute the PQ-encoding of a hierarchy $(\mathcal{T}, \preceq)$

```
1:  $\mathbf{S} \leftarrow \emptyset$ // $\mathbf{S}$ is a set of the slices created so far. Each slice $\mu \in \mathbf{S}$ is represented as a
    // record $\langle \mathcal{P}, \mathrm{id} \rangle$, where $\mu.\mathcal{P}$ is the PQ-tree of the slice, and $\mu.\mathrm{id}$ is the id of the slice.
2:  For all $a \in \mathcal{T}$ do // Find a PQ-tree consistent with type $a$.
3:    For all $\mu \in \mathbf{S}$ do // Try to find a slice $\mu$ into which $a$ could be inserted
4:      $\mathcal{P}' \leftarrow$ reduce$(\mu.\mathcal{P}, \mathrm{descendants}(a))$
5:      If $\mathcal{P}' \neq \perp$ then // Type $a$ can be inserted into slice $\mu$
6:        $\mu.\mathcal{P} \leftarrow \mathcal{P}'$ // In the updated PQ-tree $\mathrm{descendants}(a)$ are consecutive
7:        $s_a \leftarrow \mu.\mathrm{id}$ // Type $a$ belongs to slice $\mu$
8:        next $a$ // Finished handling type $a$
9:      fi
10:   od
      // Type $a$ could not be inserted into any of the existing slices
11:   $\mu \leftarrow$ new Slice // Generate a new slice $\mu$
12:   $\mu.\mathcal{P} \leftarrow$ reduce$(\mathcal{P}^{\top}, \mathrm{descendants}(a))$ // By Fact 5.3 $\mu.\mathcal{P} \neq \perp$.
13:   $\mu.\mathrm{id} \leftarrow |\mathbf{S}| + 1$ // Slice id's are allocated in order $1, 2, \ldots$
14:   $\mathbf{S} \leftarrow \mathbf{S} \cup \{\mu\}$
15: od
16: For all $\mu \in \mathbf{S}$ do // Assign unique id's to types
17:   $\mathrm{id} \leftarrow 1$ // The first unused id in the slice $\mu$.
18:   For all $a \in \mathrm{frontier}(\mu.\mathcal{P})$ do // Assign id's to all types with respect to slice $\mu$
19:     $\mathrm{id}_a@(\mu.\mathrm{id}) \leftarrow \mathrm{id}$
20:     $\mathrm{id} \leftarrow \mathrm{id} + 1$
21:   od
22: od
23: For all $a \in \mathcal{T}$ do // Assign an interval to each type $a$
24:   $\mathbf{D} \leftarrow \{\mathrm{id}_b@\, s_a \mid b \in \mathrm{descendants}(a)\}$;
25:   $[l_a, r_a] \leftarrow [\min(\mathbf{D}), \max(\mathbf{D})]$
26: od
```

The main data structure used by the algorithm is the set $\mathbf{S}$, which is an internal representation of the set of slices. Each $\mu \in \mathbf{S}$ is a record $\langle \mathcal{P}, \mathrm{id} \rangle$, where $\mu.\mathcal{P}$ is the PQ-tree of the slice, and $\mu.\mathrm{id}$ is the id of the slice. The set $\mathbf{S}$ is discarded after the algorithm computes the encoding.

After a simple initialization (line 1), the algorithm comprises of three stages. The first outer loop (lines 2–15) finds the slices. In this loop we try to find an existing slice for each type, by trying to incorporate (line 4) the constraints that its descendants must lie consecutively, into each of the PQ-trees of the existing slices. If this should fail then we create a new slice (lines 11–14).

The second stage is the loop of lines 16–22, which assigns a unique id to each type with respect to each slice. The last stage (lines 23–26) is to find the interval of the id's of the descendants of each type $a \in \mathcal{T}$, i.e., the id's, with respect to the slice of a, of the right-most and left-most type among the descendants of a.

LEMMA 5.6. *Alg. 2 runs in $O\left(|\mathbf{S}| \cdot |\preceq|\right)$ time.*

PROOF. The first stage is the slowest. At this stage reduce is invoked at most $|\mathbf{S}|$ times for each of the types in the input. Using Thm 5.5 the total time of all such invocations is

$$O\Big(|\mathbf{S}| \cdot \sum_{a \in \mathcal{T}} |\mathrm{descendants}(a)|\Big) = O\Big(|\mathbf{S}| \cdot |\preceq|\Big).$$

The second stage runs in time

$$O\Big(|\mathbf{S}| \cdot |\mathcal{T}|\Big) \subseteq O\Big(|\mathbf{S}| \cdot |\preceq|\Big),$$

while the third stage time complexity is

$$O\Big(\sum_{a \in \mathcal{T}} |\mathrm{descendants}(a)|\Big) = O\Big(|\preceq|\Big) \subseteq O\Big(|\mathbf{S}| \cdot |\preceq|\Big). \quad \square$$

We do not know of any efficient algorithm for finding the optimal PQ-encoding, i.e., the encoding which achieves the minimal number of slices. This is the reason why Alg. 2 is non-deterministic in the following sense: The order at which types are inserted into PQ-trees (line 2) is unspecified. After having tried several traversal orders, including a random one, we concluded that the differences in the encoding length is small. Our empirical findings indicate that the best results are obtained by a reverse topological-order in which the leaves with the largest number of ancestors are visited first.

Similarly, the order at which we try to find the slices (line 3) is not specified by the algorithm. We found empirically that the best encoding is obtained by trying the slices in the order of decreasing size, i.e., trying the largest slice first, and the smallest one last. A heuristic which gives almost identical results is to try the slices at the order of their creation, with the oldest slice first.

6. OPTIMIZATIONS

In this section we describe how Alg. 2 can be further optimized. We have five different, non-language specific, optimization techniques targeted at improving the various complexity measures.

(1) *ID Range Compaction* (Sec. 6.1) reduces the space complexity measure, specifically by decreasing the memory footprint of the pseudo-arrays id_a. With this optimization,

borrowed from ideas originated by Vitek et al. [1997], it is possible to use byte-sized entries for all but the first entry of these arrays.

(2) *Pruning Bottom Trees* (Sec. 6.2) targets the encoding creation time measure. We show that the heavy-weight PQ-trees algorithm needs to be run only on the smaller core portion of the input hierarchy.

(3) *Reordering Type Records* (Sec. 6.3) is a novel technique which simultaneously improves three complexity measures: space, instruction count and test time. In this optimization, the type records of the runtime environment are pre-sorted in linear time by the first entry of the id pseudo-array. This makes it possible to eliminate this first entry which is (in a sense) encoded by the pointer stored in each object to its type record. A comparison of id's stored in the first entry is replaced by a comparison of these pointers. (The main cost is in the requirement that type records occur in a fixed order, which may be a burden to other parts of the computing environment.)

(4) *Heterogeneous Encoding* (Sec. 6.4) also reduces space complexity, by switching to binary matrix encoding in slices which contain no more than 8 types. This optimization which is similar to the one suggested by Vitek et al. [1997] may increase the instruction count and the test time complexity measures in subtyping tests involving these slices.

(5) *Coalescing ID-Arrays* (Sec. 6.5) is another novel technique which targets the space complexity while increasing the instruction count and the test time. The idea here is that if suffixes of the id pseudo-arrays are identical, they can be shared at the cost of an extra indirection.

6.1 ID-Range Compaction

ID-range compaction reduces the encoding length as generated by Alg. 2. Let D be the set of descendants of a slice S:

$$D = \bigcup_{a \in S} \mathrm{descendants}(a).$$

Clearly, $|S| \leq |D|$. However, it is often the case, especially with the smaller slices, that $|S| \ll |D|$, and that $|D|$ is close to n. ID-range compaction relies on the observation that in these cases id's can be reused while numbering the types in D. This reuse makes it possible to use fewer bits for the representation of each id.

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 $b_1 \in \mathrm{descendants}(a)$, while $b_2 \notin \mathrm{descendants}(a)$. Phrased differently, S partitions $\mathcal{T}$ into equivalence classes, such that types b_1 and b_2 are in the same equivalence class if and only if

$$\mathrm{ancestors}(b_1) \cap S = \mathrm{ancestors}(b_2) \cap S. \tag{6.1}$$

These equivalence classes are called the *S-partitioning* of $\mathcal{T}$.

The number of different id's needed to encode a slice S is exactly the number of equivalence classes in the S-partitioning of $\mathcal{T}$. We argue that this number is less than twice the slice size, specifically that there are at most

$$\min(2|S|, |\mathcal{T}|)$$

equivalence classes in the S-partitioning of $\mathcal{T}$. The reason is that the local consecutiveness property ensures that for every $a \in S$ there is an interval I_a which consists the id's of descendants of 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. The set $E_0 \equiv \mathcal{T} \setminus D$ defines an additional equivalence class, which is not contained in any interval.

Consider, for example, Fig. 6.1, in which types in D were initially numbered $3, \ldots, 15$.

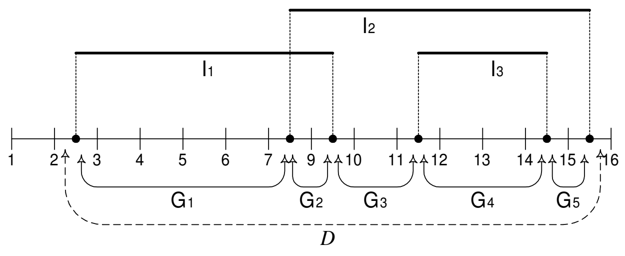

Fig. 6.1. Reducing the range needed for PQE

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. The equivalence classes in this example are $E_0 = \{1, 2, 16\}$, $E_1 = \mathsf{G}_1$, $E_2 = \mathsf{G}_2$, $E_3 = \mathsf{G}_3 \cup \mathsf{G}_5$, and $E_4 = \mathsf{G}_4$.

In all hierarchies in the data set, we found that all slices, except the first, were of size 128 or less. Thus the integral range required for numbering is at most 256 and 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 Sec. 6.3.

It is possible to modify Alg. 2 to ensure that all but one (the first) slice has their range bounded by 256. Specifically, line 5, must not only check $\mathcal{P}'$, the PQ-tree returned by the reduce routine, but also make sure that the range required for numbering does not exceed 256.[11] 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 $O(|\preceq|)$ total time.

6.2 Pruning Bottom Trees

Recall that in Sec. 3 we defined the core of a multiple inheritance hierarchy $\mathcal{T}' \subseteq \mathcal{T}$, such that $t \in \mathcal{T}'$ if t has a descendant with more than one parent. The set $\mathcal{T} \setminus \mathcal{T}'$ is a collection of bottom-trees discovered in a bottom-up traversal of the hierarchy. Intuitively, the core is where the intricacies of multiple inheritance occur. The bottom-trees are a forest of single inheritance hierarchies, hanging at the bottom of the core.

By pruning in a preprocessing stage all bottom-trees, we reduce the runtime of Alg. 2. A lighter machinery is then used to produce the encoding of the bottom-trees. Let $S_1, \ldots, S_k$ be the slices of $\mathcal{T}'$ found in the PQ-encoding of the pruned hierarchy, and $\pi'_1, \ldots, \pi'_k$ be the orderings of $\mathcal{T}'$ with respect to each slice. Thus, π'_i, $i = 1, \ldots, k$ is the ordering defined by the id's of all types with respect to slice S_i. Formally, π'_i satisfies the constraints $\varphi(S_i)$.

Next we describe how to extend π'_i of $\mathcal{T}'$ into an ordering π_i of $\mathcal{T}$ in such a way that it will satisfy the constraints $\varphi(S_i \cup (\mathcal{T} \setminus \mathcal{T}'))$. Consider an arbitrary bottom-tree whose root is t. Since t is not in the core, it has a single parent t', i.e., $\mathrm{parents}(t) = \{t'\}$.

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

Type t' must be in the core, otherwise t would not be the root of the bottom tree. (Note that t' might have several other children which are roots of other bottom trees.) When extending the ordering π'_i of $\mathcal{T}'$, we insert the relative numbering ordering of this bottom-tree immediately after (or before) $\pi'_i(t')$.

Fig. 6.2 gives an example of the insertion of relative-numbering orderings into the ordering of the core. Fig. 6.2a shows the core of the hierarchy of the running example, whereas the bottom-trees are highlighted in bold in Fig. 6.2b.

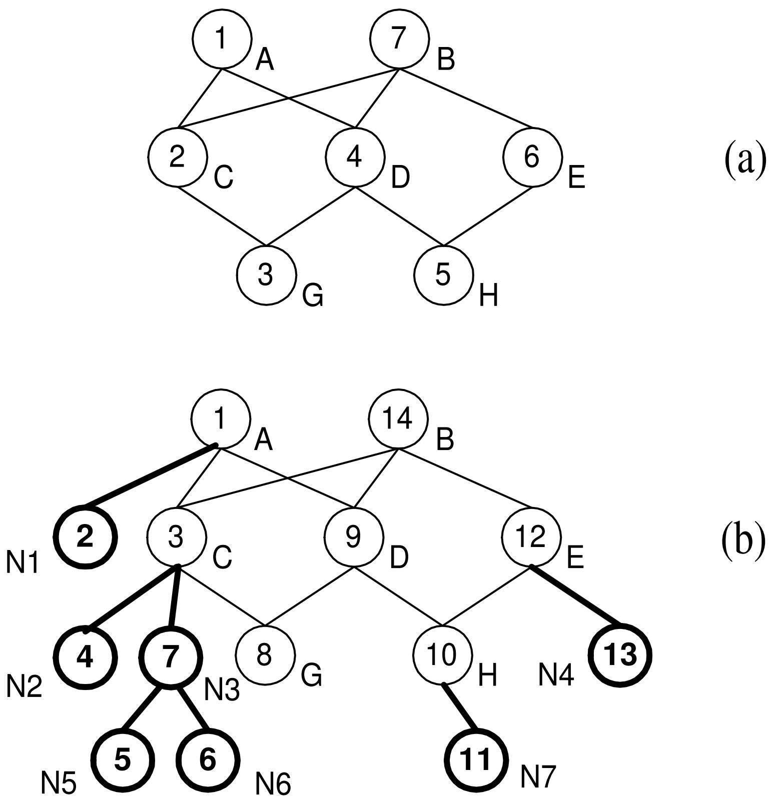

Fig. 6.2. PQE of the core of the running example (a) and PQE after inserting some bottom-trees (b)

Fig. 6.2a shows an ordering π' of the core $\mathcal{T}'$ which satisfies the constraints $\varphi(\mathcal{T}')$,

$$\pi' = \langle \mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{E}, \mathsf{B} \rangle.$$

Fig. 6.2b shows the extended ordering π of $\mathcal{T}$ which satisfy the constraints $\varphi(\mathcal{T})$:

$$\pi = \langle \mathsf{A}, \mathsf{N1}, \mathsf{C}, \mathsf{N2}, \mathsf{N5}, \mathsf{N6}, \mathsf{N3}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{N7}, \mathsf{E}, \mathsf{N4}, \mathsf{B} \rangle.$$

Note that the resulting ordering π_i of $\mathcal{T}$ satisfies the old constraints in $\varphi(S_i)$ (since descendants in a bottom-tree are adjacent to their parent in the core) and the new constraint in $\varphi(\mathcal{T} \setminus \mathcal{T}')$ (since relative-numbering ordering satisfies these constraints).

In order to complete the process of incorporating the bottom-trees into a PQ-encoding of the core, we must also assign each of the types in the bottom-trees into a slice. The fact that we inserted the relative numbering ordering of each bottom-tree makes it possible to chose any slice we want for each type in a bottom-tree. We chose to use the first slice for all these types since ID-range compaction works best when the first slice is much larger than the others. Another motivation for this choice is the “reordering of type records” optimization which, as we shall see below, makes it possible to eliminate all bits used for the first slice.

6.3 Reordering Type Records

Consider again the subtyping test $a \preceq b$. So far it was assumed that the type a is *given* at runtime. In reality, however, an object o is given and the runtime system must first infer its type a. Typically o stores a pointer p_a to its *type record*, a memory block with runtime representation of a. The various encoding schemes store their auxiliary information in this area. Many object-oriented language implementations mandate other uses to the type records, including dispatching, downcasting, serialization, and garbage collection.

The *reordering type records* optimization technique makes use of the degree of freedom the compiler has in placing type records in memory.[12] The simplest application of this technique is to relative numbering (Sec. 4): Type records are placed in memory in the same order as postorder traversal of the type hierarchy. In doing so, the pointer p_a plays the role as the ordinal in the postorder r_a. As a result, the encoding length is reduced to zero and one load instruction in the subtyping test is saved.

Similarly, in range-compression (4.10), p_a replaces the global id_a. If specialization is used then we obtain an encoding scheme with zero encoding length, but non-constant test time and instruction count.

We do not know whether the technique is applicable to either bit-vector encodings or to Cohen’s algorithm and its generalizations, PE and BPE. However, in PQE, the ability to reorder type records makes it possible to eliminate entirely the id’s of types with respect to the first slice. Specifically, $\mathrm{id}_a @\ 1$ of a type a is encoded in the pointer p_a. The saving is significant since the first slice occupies the largest number of bits. This technique also saves one load instruction when 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.

We finally note that this technique is applicable even with the unique C++ object layout. In this layout [Gil and Sweeney 1999] an object may contain several pointers to several *distinct* type records (VTBLs in the C++ jargon).

The reason that we can encode integers in pointers even though there is no unique value p_a for a type a is that the subtype tests of relative numbering (4.4), range compression (4.10), and PQE (5.1), all check for *inequality* rather than equality. We simply allocate a range of memory addresses to all type records of a given type, rather than a single address, as the value r_a (as in (4.4)) or the id (as in (4.10) and (5.1)).

6.4 Heterogeneous Encoding

Heterogeneous encoding is yet another optimization targeted at reducing the encoding length. Recall that in the binary matrix each type adds exactly one bit to the encoding

[12]We make the natural assumption that the location of the encoding tables is in a protected location of memory which is not subject to garbage collection. The reason is that these tables are generated as part of the compilation process and are not changed at runtime.

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 the binary matrix for types in this slice. In heterogeneous encoding, subtyping tests $a \preceq b$, where b belongs in such a small slice, are implemented using the binary matrix. 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.

6.5 Coalescing ID-Arrays

We now turn to describing *Coalesced PQ-Encoding* (CPQE). This memory optimization is based on the observation that the contents of the pseudo-arrays id_a tend to be similar. We rely on the fact that the first entry of these arrays is represented implicitly. Let id'_a denote the array obtained from id_a by truncating its first entry. Then, many of the arrays id'_a are identical, and need to be stored only once.

More specifically, we claim that the number of distinct arrays id' is exactly the number of equivalence classes in G-partitioning of $\mathcal{T}$, where $G = \mathcal{T} \setminus S_1$. In other words, two types a, b are in the same equivalence class if and only if $\mathrm{id}'_a = \mathrm{id}'_b$. Formally,

LEMMA 6.1. *Let a, b be two types, and $G = \mathcal{T} \setminus S_1$. Then*

$$\mathrm{ancestors}(a) \cap G = \mathrm{ancestors}(b) \cap G \Leftrightarrow \mathrm{id}_a@\ i = \mathrm{id}_b@\ i \text{ for } 2 \le i \le k.$$

PROOF. We previously showed (6.1) that two types can have the same identifiers if and only if they are in the same equivalence class, i.e.,

$$\mathrm{id}_a@\ i = \mathrm{id}_b@\ i \Leftrightarrow \mathrm{ancestors}(a) \cap S_i = \mathrm{ancestors}(b) \cap S_i.$$

Since $S_2 \dots S_k$ partition G we have that

$$\begin{aligned} \mathrm{ancestors}(a) \cap G &= \mathrm{ancestors}(b) \cap G \Leftrightarrow \\ \mathrm{ancestors}(a) \cap S_i &= \mathrm{ancestors}(b) \cap S_i \text{ for } 2 \le i \le k \Leftrightarrow \\ \mathrm{id}_a@\ i &= \mathrm{id}_b@\ i \text{ for } 2 \le i \le k. \quad \Box \end{aligned} \tag{6.2}$$

Furthermore, the number of distinct arrays id' is always smaller or equal to the size of the core. (The core is the set of types not belonging to a bottom tree; See Sec. 3.) Recall that the bottom trees were added to the first slice after they were pruned (see Sec. 6.2). Since each type in a bottom tree has the same ancestors set as the root of that tree, they are in the same equivalence class, and therefore can be coalesced together.

CPQE uses a bucket sort to find the distinct values of arrays id' in linear time, and then represents each type a as a pointer p'_a to one of these distinct values. The cost of the coalesced representation is in another level of indirection for subtyping tests involving the second or higher slice.

The pointer p'_a is not stored as an absolute memory address but rather as an index of an array Z, whose entries are the distinct id' arrays. Also the degree of freedom in placing entries in Z, is employed to encode $\mathrm{id}_a@\ 2$ (id's of the second slice) in p'_a in the same fashion that $\mathrm{id}_a@\ 1$ was encoded as p_a.

In the test $a \preceq b$, if it is found that b belongs in slice S_2, then instead of using $\mathrm{id}_a@\ 2$ in the test (5.1), 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 the arrays id.

A strong incentive to use CPQE is raised by languages such as C++, in which objects may contain *multiple* pointers to several *distinct* type records [Gil and Sweeney 1999]. Since these type records are similar, but not identical, the implementer must choose between (i) *replicating* the subtyping encoding data in each such record, or (ii) *sharing* at the cost of another level of indirection during subtyping tests. Coalescing optimization may tip the scale towards the sharing alternative.

7. RESULTS

Having described different optimization techniques we would like to appreciate the tradeoffs offered by these. To do so, we define (Sec. 7.1) variants of the main encoding scheme. We then show (Sec. 7.2) how the encoding length of these variants depends on the output of our main algorithm (Alg. 2), and in particular the number of slices and the distribution of their size. Sec. 7.3 compares the encoding length achieved by the different variants with the achievements of previous work. Sec. 7.4 gives the results of our timing of the algorithm for computing the encoding length.

7.1 Variants of the PQ-Encoding Scheme

There are many variants of PQ-encoding, depending on which of the optimizations described in the previous section are applied. The first two optimizations: ID range compaction and bottom tree elimination, which do not add to the main complexity measures are in fact incorporated to the main algorithm. We next define three encoding variants which successively apply the three other optimizations:

(1) Regular PQ-encoding, or RPQE for short is the variant in which reordering the type records is used to eliminate the representation of the first slice from the id arrays.
(2) The principal acronym PQE is reserved to the variant which also applies the heterogenous encoding optimization. As explained above, the cost is in longer subtyping tests in the rare cases involving the smaller slices.
Thus, in PQE, there are three kinds of slices: The first slice, whose representation is eliminated thanks to reordering of type-records. Heterogeneous encoding based on the binary matrix representation is used for slices whose size is smaller than 8. Each of the remaining slices occupies a single byte in the array id, which is used in the basic subtyping tests of PQE (5.1).
(3) CPQE is the encoding variant obtained from PQE by applying in addition the remaining fifth optimization: coalescing of ID-arrays, which adds to the cost of subtyping tests involving the third or higher slice.

7.2 Output of the PQ-Algorithm

Alg. 2, the main algorithm behind the PQ-encoding, returns a partitioning of the hierarchy into slices. It was mentioned before that the size of slices vary widely. Using the hierarchies in our data set we now turn to studying this variety in detail.

Table II displays some of the essential parameters of the slice size distribution. These parameters will become useful in appreciating the algorithm performance and the tradeoffs offered by the different optimizations. We can also use these to calculate the encoding length of the three encoding schemes described above.

Hierarchy	k[a]	n_1/n[b]	k_2[c]	n_2[d]	n_2/n	m[e]
IDL	1	100.0%	0	0	0.0%	0
JDK 1.1	2	99.6%	1	1	0.4%	1
Laure	2	98.0%	1	6	2.0%	7
Ed	10	87.8%	7	20	4.6%	145
LOV	12	86.2%	9	26	6.0%	164
Unidraw	2	99.7%	1	2	0.3%	2
Cecil	5	94.1%	2	6	0.6%	101
Geode	16	86.0%	8	24	1.8%	419
JDK 1.18	6	97.5%	3	9	0.5%	74
Self	13	97.2%	11	31	1.7%	63
Eiffel4	11	89.1%	3	9	0.5%	376
JDK 1.22	8	97.6%	4	12	0.3%	235
JDK 1.30	8	97.7%	4	17	0.3%	286

[a] number of slices
[b] fraction of types in the first slice
[c] number of small slices
[d] total number of types in small slices
[e] number of distinct id' arrays

Table II. Some characteristics of the slice partitioning of the PQ algorithm

Even though we do not have a non-trivial upper bound on the number of slices, the second column of the table shows that in actual hierarchies, k, the number of slices, is often small, and it does not increase as quickly as n. Thus, we have reasons to believe that $O(kn)$, the asymptotic space complexity of algorithm Alg. 2, is closer to linear than quadratic. Similar conclusions can be drawn on $O(k|\preceq|)$, the time complexity of the algorithm.

Integer k is also useful in computing the encoding length of RPQE. Recall that with the exception of the first slice, the id's with respect to each slice can be represented in a single byte. Therefore, the encoding length of RPQE is $8(k-1)$. (Also, consider a variant of RPQE in which type records are not reordered. Then, the encoding length in this variant is $16 + 8(k-1) = 8(k+1)$.)

The next column in the table gives the ratio n_1/n, where n_1 is the number of types in the first slice (which is also the largest slice). We see that in all hierarchies over 85% of the types fall in this slice. In fact, in more than half the hierarchies, this slice occupies at least 97.5% of all types. Thus, we expect that an overwhelming portion of the actual subtyping tests will use this slice. The test time of these will greatly benefit from reordering of type records.

Small slices, i.e., slices with no more than 8 types, receive special handling by PQE. The heterogeneous encoding optimization specifies that types in these slices use a binary matrix representation. The subtyping test then involves bit operations, and is not as simple as the range testing used for the other slices.

The fourth column of Table II shows k_2, the number of small slices. We see that most of the slices generated by the PQ-algorithm are small. However, examining the next column (the total number of types in the small slices n_2, $k_2 \leq n_2 \leq 8k_2$), we see that n_2 is small. The penultimate column of the table shows that the fraction of types in small slices is tiny, typically less than 1%. We are lead to hope that the frequency of the more complex tests will be equally negligible.

Interestingly, the values shown in Table II can be used to compute the encoding length of PQE. Since all slices except the first and small slices occupy a single byte in id-array, we have that this length is

$$8(k - k_2 - 1) + n_2.$$

To compute the encoding length of CPQE we need the final column of the table which shows m, the number of distinct id' arrays. We see that this number is much smaller than the number of types. In fact, $m \leq 256$ in all hierarchies except for Eiffel4, Geode, and JDK 1.30. The pointer p'_a can thus often be represented as a single byte. More generally, the precise encoding length of CPQE is

$$8 \left\lceil \frac{\log m}{8} \right\rceil + \frac{(8(k - k_2 - 2) + n_2) \times m}{n}.$$

7.3 Encoding Length in the Data Set

Table III compares the encoding length in bits of the three encoding variants with that of other encoding schemes.

Hierarchy	**CPQE**	**PQE**	**RPQE**	NHE	BPE	PE	DAG[a]	Closure [b]	Binary matrix
IDL	8	0	0	17	32	96	7	27	66
JDK 1.1	8	1	8	19	32	64	9	26	225
Laure	8	6	8	23	63	128	10	74	295
Ed	17	36	72	54	94	216	15	72	434
LOV	21	42	88	57	94	216	16	77	436
Unidraw	8	2	8	30	63	96	8	31	613
Cecil	10	22	32	58	94	192	13	65	932
Geode	39	80	120	95	157	408	21	154	1,318
JDK 1.18	9	25	40	39	94	128	13	48	1,704
Self	9	39	96	53	126	344	12	329	1,801
Eiffel4	27	65	80	72	157	312	15	97	1,999
JDK 1.22	10	36	56	62	157	184	16	57	4,339
JDK 1.30	18	41	56	65	188	216	16	57	5,438

[a]Computed idealistically as $(|\prec_{\mathrm{d}}| \cdot \lceil \log n \rceil)/n$.
[b]Computed idealistically as $(|\preceq| \cdot \lceil \log n \rceil)/n$

Table III. The encoding length of different algorithms

The most important conclusion to draw from the table is that in all hierarchies in the data set, the encoding length achieved by PQE is better than that of all other encoding schemes. The only exception to these is an idealistic DAG representation, in which, as mentioned above, test time can be $O(n)$.

We stress again that the memory requirements of PQE is zero for all single inheritance hierarchies. As can be seen in the table, zero memory footprint occurs even in IDL, which is multiple inheritance. The median improvement over the next best algorithm, NHE, is by 37%, while the average improvement is 50%.

PQE remains the shortest encoding even if it is not optimized by reordering type records (in which case the encoding length increases by 16): Without this optimization, PQE is better than NHE in 9 out of the 13 hierarchies. In one hierarchy (LOV), 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.

In comparing PQE with NHE we must also recall that the test time in the bit vector based NHE is non-constant. Thus, even if the two schemes use the same number of bits, subtyping tests in PQE are likely to be more efficient since they do not need to access all bits in the representation of the compared types.

The space reduction of PQE over BPE, the best previous *constant time encoding*, is even more impressive: In the Eiffel4 hierarchy BPE total space requirement is 39KB, compared with 16KB in PQE. These differences are significant since subtyping tests are very frequent. Vitek [Palacz and Vitek 2003] benchmarks give 320,000 tests in a second. Smaller encoding makes it possible to fit the entire representation in the cache.

Examining the second and third columns of Table III we see that coalescing of id records, employed by CPQE, shortens the encoding length of PQE, by factors ranging between 2 and 4.3. In fact, CPQE competes favorably even with the idealized DAG encoding!

Hierarchies IDL, Laure, Unidraw and JDK 1.1 are anomalous in the sense CPQE gives a longer encoding than PQE. This phenomenon is explained by the fact that the two-level structure employed by CPQE requires at least 8 bits for p'_a.

We finally note that even RPQE competes favorably with NHE, winning in 7 out of the 13 hierarchies in the data set.

7.4 Encoding Creation Time

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

Hierarchy	(R | C)PQE [a]	NHE [b]	(B)PE [c]
IDL	1	-	5
JDK 1.1	1	19	10
Laure	4	21	9
Ed	77	136	12
LOV	95	168	10
Unidraw	1	93	10
Cecil	50	-	13
Geode	668	1,902	28
JDK 1.18	29	-	26
Self	122	1,367	22
Eiffel4	299	-	29
JDK 1.22	140	-	77
JDK 1.30	187	-	90

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

Table IV. Encoding creation time in milliseconds of different algorithms

The comparison is not easy, since the algorithms were run on different machines. Alg. 2 was written in C++ based on the PQ-tree implementation of Leipert [1997]. 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, which use a fast implementation of set unions and intersections using bit-vector operations, seems 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 PQE is less than a second.

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 reduces the encoding length even further at the cost of an extra indirection in some, typically infrequent, subtyping tests.

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 runtime. 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.

An interesting instance of the subtyping problem occurs when the ordinary type hierarchy is compounded by an interplay with *genericity*, as in EIFFEL and in the proposed addition of generics to JAVA. In EIFFEL, a double ended queue of rectangles is a subtype of a queue of polygons ($\mathsf{DQueue}[\mathsf{Rectangle}] \preceq \mathsf{Queue}[\mathsf{Polygon}]$) since (i) $\mathsf{Rectangle} \preceq \mathsf{Polygon}$, and (ii) the generic class $\mathsf{DQueue}[T]$ inherited from $\mathsf{Queue}[T]$. EIFFEL has a default subtyping rule which can be written as

$$\forall a, b, A \bullet a \preceq b \Rightarrow A[a] \preceq A[b],$$

and the definition of generic classes which inherit from others adds other rules such as

$$\begin{aligned} \forall a \bullet A[a] &\preceq B[a], \\ \forall a, b \bullet C[a, b] &\preceq D[a[b]]. \end{aligned}$$

The research question is whether pre-processing of such rules can make it possible to decide subtyping more efficiently.

ACKNOWLEDGMENTS

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!

REFERENCES

AGRAWAL, R., BORGIDA, A., AND JAGADISH, H. V. 1989. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, J. Clifford, B. G. Lindsay, and D. Maier, Eds. ACM Press, Portland, Oregon, 253–262.

ALPERN, B., COCCHI, A., AND GROVE, D. 2001. Dynamic type checking in Jalapeño. In *Java Virtual Machine Research and Technology Symposium*, J. Clifford, B. G. Lindsay, and D. Maier, Eds. USENIX, Monterey, California.

ARNOLD, K. AND GOSLING, J. 1996. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, Massachusetts.

BATTISTA, G. D. AND TAMASSIA, R. 1989. On-line planarity testing. Technical Report CS-89-31, Brown University - Department of Computer Science. May.

BATTISTA, G. D. AND TAMASSIA, R. 1990. On-line graph algorithms with SPQR-trees. In *Automata, Languages and Programming, 17th International Colloquium,* M. S. Paterson, Ed. Lecture Notes in Computer Science, vol. 443. Springer-Verlag, Warwick University, England, 598–611.

BATTISTA, G. D. AND TAMASSIA, R. 1996. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica 15,* 4 (Apr.), 302–318.

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

BOUCHITTE, V. AND MORVAN, M., Eds. 1994. *International Workshop on Orders, Algorithms, and Applications (ORDAL'94).* Number 831 in Lecture Notes in Computer Science. Springer Verlag, Lyon, France.

CAPELLE, C. 1994. Representation of an order as union of interval orders. See Bouchitte and Morvan [1994], 143–162.

CASEAU, Y. 1993. Efficient handling of multiple inheritance hierarchies. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA'93, ACM SIGPLAN Notices 28(10) Oct. 1993, Washington, DC, USA, 271–287.

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

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

CORSARO, A. AND CYTRON, R. 2003. Efficient memory-reference checks for real-time java. In *Proceedings of 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03).* ACM, San Diego, California, USA, 51–58.

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

DIJKSTRA, E. W. 1960. Recursive programming. *Numerische Mathematik 2*, 312–318.

ECKEL, N. AND GIL, J. Y. 2000. 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. ECOOP 2000, Springer Verlag, Sophia Antipolis and Cannes, France, 394–421.

FALL, A. 1995. Heterogeneous encoding. In *Proceedings of International KRUSE'95 Conference: Knowledge Use, Retrieval and Storage for Efficiency*, G. Ellis, R. Levinson, A. Fall, and V. Dahl, Eds. Department of Computer Science, University of California at Santa Cruz, USA, Santa Cruz, California, 162–167.

FALL, A. 1996. Sparse term encoding for dynamic taxonomies. In *Proceedings of the Fourth International Conference on Conceptual Structures (ICCS-96): Knowlegde Representation as Interlingua*, P. W. Eklund, G. Ellis, and G. Mann, Eds. LNAI, vol. 1115. Springer, Berlin, 277–292.

FILMAN, R. E. 2002. Polychotomic encoding: A better quasi-optimal bit-vector encoding of tree hierarchies. In *Proceedings of the 16th European Conference on Object-Oriented Programming*. Number 2374 in Lecture Notes in Computer Science. ECOOP 2002, Springer Verlag, Malaga, Spain, 545–561.

FREDMAN, M. L., KOMLÓS, J., AND SZEMERÉDI, E. 1984. Storing a sparse table with $O(1)$ worst case access time. *J. ACM 31,* 3 (July), 538–544.

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

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

GRAFL, R. 1996. CACAO: Ein 64bit JavaVM just-in-time compiler. M.S. thesis, University of Vienna.

HABIB, M., CASEAU, Y., NOURINE, L., AND RAYNAUD, O. 1999. Encoding of multiple inheritance hierarchie and partial orders. *Computational Intelligence 15*, 50–62.

HABIB, M. AND NOURINE, L. 1994. Bit-vector encoding for partially ordered sets. See Bouchitte and Morvan [1994], 1–12.

HOLLANDER, Y., MORLEY, M., AND NOY, A. 2001. The e language: A fresh separation of concerns. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems*. TOOLS 2001 Europe Conference, Prentice-Hall, Zurich, Switzerland, 41–51.

JUNGER, M., LEIPERT, S., AND MUTZEL, P. 1996. On computing a maximal planar subgraph using PQ-trees. Tech. rep., Informatik, Universität zu Köln.

JUNGER, M., LEIPERT, S., AND MUTZEL, P. 1998. A note on computing a maximal planar subgraph using PQ-trees. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17,* 7 (July), 609–612.

KACI, H., BOYER, R., LINCOLN, P., AND NASR, R. 1989. Efficient implementation of lattice operation. *ACM Trans. Prog. Lang. Syst. 11*, 115–146.

KRALL, A. 2001. personal communication.

KRALL, A. AND GRAFL, R. 1997. CACAO – a 64 bit JavaVM just-in-time compiler. In *PPoPP'97 Workshop on Java for Science and Engineering Computation*, G. C. Fox and W. Li, Eds. ACM Press, Las Vegas.

KRALL, A., VITEK, J., AND HORSPOOL, R. N. 1997. Near optimal hierarchical encoding of types. In *Proceedings of the 11^{th} European Conference on Object-Oriented Programming*. Number 1241 in Lecture Notes in Computer Science. ECOOP'97, Springer Verlag, Jyväskylä, Finland, 128–145.

LEIPERT, S. 1997. PQ-trees, an implementation as template class in C++. Tech. rep., Informatik, Universität zu Köln.

LEMPEL, A., EVEN, S., AND CEDERBAUM, I. 1967. An algorithm for planarity testing of graphs. In *Theory of Graphs, International Symposium*. Gordon and Breach, New York, NY, 215–232.

MEYER, B. 1992. *EIFFEL the Language*. Object-Oriented Series. Prentice-Hall, Hemel Hempstead, Hertfordshire, UK.

PALACZ, K. AND VITEK, J. 2003. Java subtype tests in real-time. In *Proceedings of the 17^{th} European Conference on Object-Oriented Programming*. Number 2743 in Lecture Notes in Computer Science. ECOOP 2003, Springer Verlag, Darmstadt, Germany.

RAYNAUD, O. AND THIERRY, E. 2001. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. In *Proceedings of the 15^{th} European Conference on Object-Oriented Programming*. Number 1850 in Lecture Notes in Computer Science. ECOOP 2001, Springer Verlag, Budapest, Hungary, 165–181.

SCHUBERT, M. A., L.K., P., AND TAUGHER, J. 1983. Determining type, part, colour, and time relationships. *Computer 16 (special issue on Knowledge Representation)*, 53–60.

STROUSTRUP, B. 1997. *The C++ Programming Language*, 3^{rd} ed. Addison-Wesley, Reading, Massachusetts.

VAN BOMMEL, M. F. AND BECK, T. J. 2000. Incremental encoding of multiple inheritance hierarchies. In *Proceedings of the 8^{th} International Conference on Information Knowledgement (CIKM-99)*. ACM Press, N.Y., 507–513.

VAN EMDE BOAS, P. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett. 6(3),* 80–82.

VAN EMDE BOAS, P., KAAS, R., AND ZIJLSTRA, E. 1977. Design and implementation of an efficient priority queue. *Math. Systems Theory 10*, 99–127.

VITEK, J., HORSPOOL, R. N., AND KRALL, A. 1997. Efficient type inclusion tests. In *Proceedings of the 12^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA'97, ACM SIGPLAN Notices 32(10) Oct. 1997, Atlanta, Georgia, 142–157.

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

A. A DETAILED PQ-TREE EXAMPLE

The example below will shed some light on the "magic" behind Thm 5.5 and the implementation of reduce due to Booth and Leuker [1976].

We first trace the execution of genTree (Alg. 1) where the input is the constraints (5.8) of the running example. The algorithm starts with a universal PQ-tree $\mathcal{P}^\top$ over the universe

$$\mathcal{T} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}\},$$

and iteratively calls reduce for each of the input constraints in the order they appear in (5.8). The output of the algorithm is then the PQ-tree depicted in Fig. 5.4, which satisfies these constraints. (Using any other order would have resulted in an equivalent PQ-tree.)

Fig. A.1 shows the PQ-tree at each of the intermediate steps in this iterative process. Each sub-figure shows the next input constraint (variable I in Alg. 1), and the current PQ-tree (variable $\mathcal{P}$ in the algorithm), where the leaves corresponding to types constrained to appear together in the next iteration are highlighted. Thus, Fig. A.1b is the PQ-tree obtained by performing $\mathsf{reduce}(\mathcal{P}^\top, I_\mathsf{C})$, while figures A.1c, d, e, f show the PQ-tree after reducing it with constraints I_D, I_E, I_A, and I_B, respectively.

Imposing the constraint $I_\mathsf{C} = \{\mathsf{C}, \mathsf{F}, \mathsf{G}\}$ on the initial universal tree (Fig. A.1a) yields the tree of Fig. A.1b, which uses an extra P-node to ensure that these three types occur together. The next constraint to add is $I_\mathsf{D} = \{\mathsf{G}, \mathsf{D}, \mathsf{H}\}$. Since type G is common to both I_C and I_D we have that the permissible orderings must have a subsequence which matches one of the following two patterns:

(1) Types C and F occur together, in any order, then type G, and then types D and H together, but in any order.

(2) Types D and H occur together, in any order, then type G, and then types C and F together, but in any order.

These two patterns are captured by the PQ-tree of Fig. A.1c, in which one P-node forces C and F to occur together, while another P-node forces D and H to occur together. The Q-node of this tree makes sure that G falls between the pairs $\{\mathsf{C}, \mathsf{F}\}$ and $\{\mathsf{D}, \mathsf{H}\}$.

The transition between Fig. A.1c and Fig. A.1d is even more interesting. Let α_c be the subtree rooted at the Q-node of Fig. A.1c. Then, subtree α_c ensures that the five types C, F, G, D and H occur together. To this requirement we now must add the constraint $I_\mathsf{E} = \{\mathsf{H}, \mathsf{E}, \mathsf{I}\}$, which means that H must be adjacent to either E or I. Therefore, H must occur in a boundary position (either first or last) in the placement of the five types in α_c. The problem is that α_c allows D to take the place of H in this boundary position. The remedy is in "lifting" both H and D to the containing Q-node, making sure that if H is first, then D is second, while if H is last then D is in the penultimate position. After having guaranteed that H is in a boundary position, procedure reduce incorporates a P-node of types E and I into the boundary of α. The result is shown in Fig. A.1d.

The transition from Fig. A.1d to Fig. A.1e is rather simple. Let α_d be the subtree rooted at the Q-node of Fig. A.1d. Then, the constraint $I_\mathsf{A} = \{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{A}\}$ is almost satisfied by α_d; the only missing requirement is that α_d does not guarantee that A is adjacent to the others in the requirement. Procedure reduce then makes the leaf A a child of this Q-node. It is possible to do so, since the set $\{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}\}$ has a "free" boundary (the other boundary is constrained to be either E or I.

The transition from Fig. A.1e to Fig. A.1f follows the same lines as the previous transition. Again, the set $\{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{E}, \mathsf{I}\}$ has only one "free boundary" in the Q-node of Fig. A.1e. The constraint $\{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{E}, \mathsf{I}\}$ is realized by adding B in the Q-node at this free boundary. Fig. A.1f (which is the same as Fig. 5.4) is the final PQ-tree, representing the eight different orderings which satisfy the constraints in (5.8).

To see a situation in which Alg. 1 returns $\bot$, which will make it necessary to use more than one slice, consider the hierarchy depicted before in Fig. 5.2. This hierarchy is identical to the running example except that a new type N was added as a parent of type E. This new

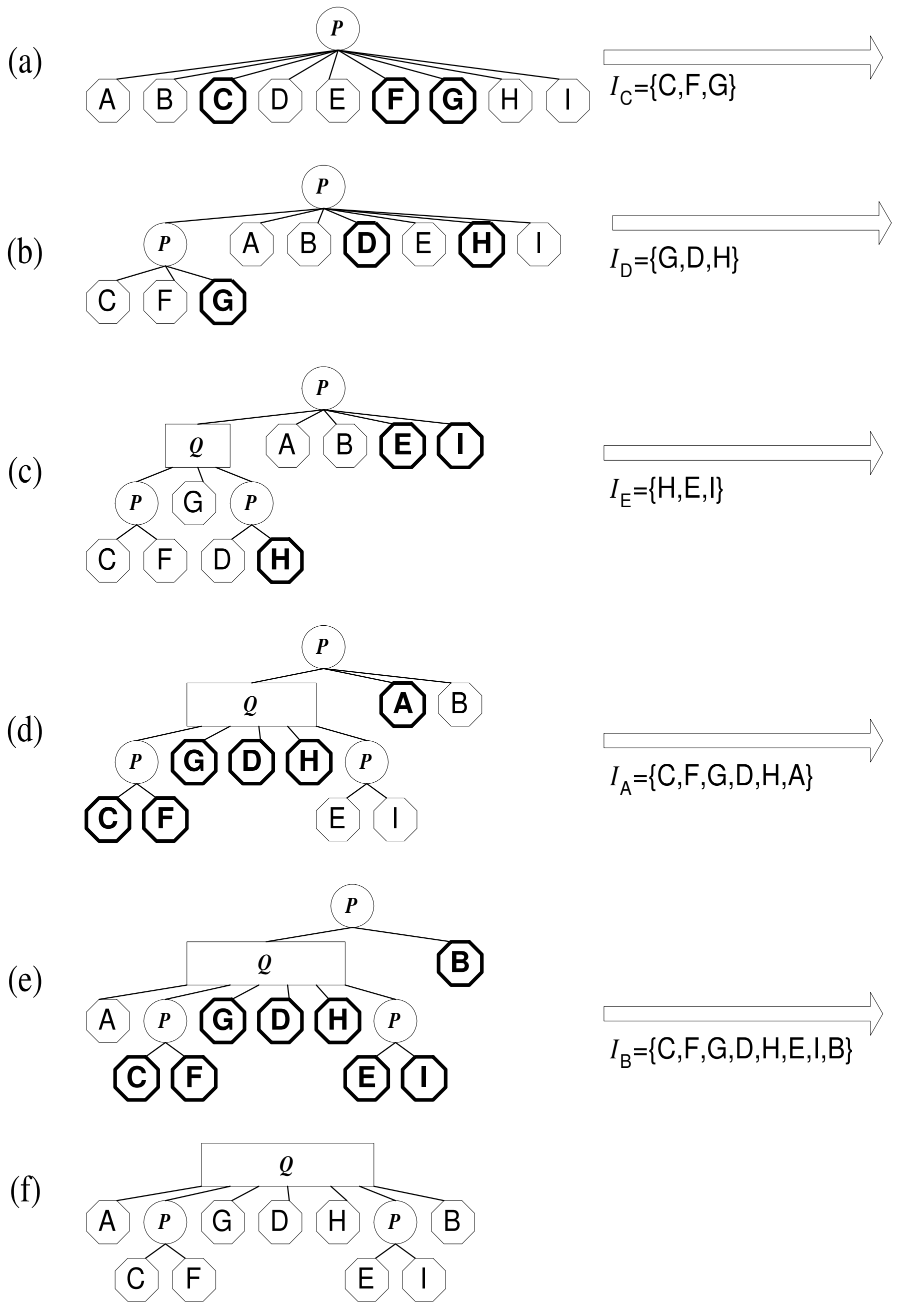

Fig. A.1. Intermediate PQ-trees in the invocation of genTree on the constraints of the hierarchy Fig. 1.1

node adds the constraint that all of its descendants must lie together, i.e., the constraint

$$I_{\mathsf{N}} = \{\mathsf{N}, \mathsf{E}, \mathsf{H}, \mathsf{I}\}, \tag{A.1}$$

is added to **I**.

Fig. A.2 shows the PQ-tree of the augmented hierarchy after all the *other* constraints in (5.8) were incorporated. (This tree is easily obtained by adding type N to the PQ-tree of Fig. A.1f.)

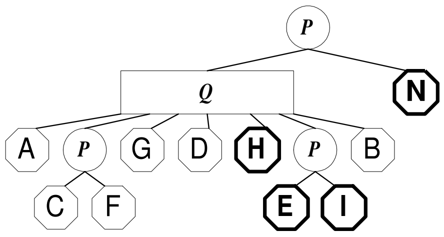

Fig. A.2. PQ-tree with a new configuration in which reduce will return $\perp$

Consider now the constraint (A.1), depicted by highlighting types N, E, H, I in Fig. A.2. By examining the figure, we see that N cannot be made adjacent to any of the types E, H, I. For example, N cannot be adjacent to H, because H lies between D, and one of E and I. In other words, the set $\{\mathsf{H}, \mathsf{E}, \mathsf{I}\}$ has no "free" boundaries. Therefore, calling reduce with the PQ-tree of Fig. A.2 and the constraint (A.1) returns $\perp$.

Received January 2001; accepted May 2004