Efficient Type Inclusion Tests

Jan Vitek
Object Systems Group, CUI,
Université de Genève,
Geneva, Switzerland
Jan.Vitek@cui.unige.ch

R. Nigel Horspool
Dept. of Computer Science,
University of Victoria,
Victoria, BC, Canada
nigelh@csr.uvic.ca

Andreas Krall
Institut für Computersprachen,
Technische Universität Wien,
Wien, Austria
andi@complang.tuwien.ac.at

Abstract

A type inclusion test determines whether one type is a subtype of another. Efficient type testing techniques exist for single subtyping, but not for languages with multiple subtyping. To date, the only fast constant-time technique relies on a binary matrix encoding of the subtype relation with quadratic space requirements. In this paper, we present three new encodings of the subtype relation, the *packed encoding*, the *bit-packed encoding* and the *compact encoding*. These encodings have different characteristics. The bit-packed encoding delivers the best compression rates: on average 85% for real life programs. The packed encoding performs type inclusion tests in only 4 machine instructions. We present a fast algorithm for computing these encoding which runs in less than 13 milliseconds for PE and BPE, and 23 milliseconds for CE on an Alpha processor. Finally, we compare our results with other constant-time type inclusion tests on a suite of 11 large benchmark hierarchies.

1 Introduction

Many modern programming languages, particularly object-oriented ones, have been built around the notion of type conformance to allow for a form of polymorphism and code reuse. The idea is that, if a type A conforms to a type B, then A can be used in any context where B is expected. This notion is essential for the code inheritance advocated by most object-oriented languages. Conformance is usually summarized by a transitive, reflexive, anti-symmetric subtype relation ($<:$) between the types of a hierarchy.

A type inclusion test determines if a pair of types is in the subtyping relation. Such tests are performed frequently during compilation. Most object-oriented language implementations are also able to perform tests at runtime. In SMALLTALK the *isKindOf:* method tests whether an object's class is a subclass of the class given as argument, OBERON provides type tests and type guards, JAVA *instanceof, etc.* Type tests need not always be explicitly requested by the programmer, they may also be inserted by the compiler, either as an optimization ([14]) or for safety. For example, in the JAVA code fragment shown below, the assignment to the local variable `b` is checked to ensure that the actual, runtime, type of the argument to the method is effectively a subtype of `B`:

```
class B extends A {
    void foo( A a ) {
        B b = (B) a ;
    }
}
```

Since the subtype relation is a partial order on the types of the program, type inclusion testing is more than the mere comparison of type tags. Depending on the implementation of the type test algorithm and on the dynamic frequency of tests, the cost of dynamic typechecking can strain the overall system performance.

This paper discusses the implementation of type inclusion tests in languages that allow multiple subtyping[1]. We present and compare different encodings of the subtype relation, as well as algorithms to compute these encodings and perform the type inclusion test. Our exploration of the design space of algorithms and encodings was driven by three requirements:

1. Runtime efficiency: Type tests should be fast. Our original motivation for this research was to optimize method dispatch ([15], [14]). To this end, the cost of testing for type inclusion had to be comparable to the cost of dispatch in statically typed languages (5 machine instructions, but see [8]). We also insist on constant-time tests[2] as we believe that the cost of language primitives should be predictable.

Proc. of the Object-Oriented Programming Languages, Systems and Applications, OOPSLA'97, Atlanta, October 1997.

[1] Note that we make a difference between subtyping and inheritance. JAVA is a single inheritance language with multiple subtyping.

[2] In this context, constant means *constant number of instructions*, we did not explore cache behavior.

2. Space efficiency: The runtime data structures that encode the subtype relation must be small. Furthermore, the code sequence emitted by the compiler for each static occurrence of a subtype test must be short.

3. Incremental hierarchy modifications: Support for runtime updates of the subtype relation. The concern here is that the cost in space and time of recomputing the encoding must not be prohibitive.

To the best of our knowledge no existing technique meets our requirements. Algorithms based on dynamic data structures such as linked lists and hash tables are slow and exhibit unpredictable behavior. Constant-time techniques either require large amounts of space, as for the bit matrix encoding, or are quite complex to compute, as for the hierarchical encoding [2], [12].

In this paper, we present three new encodings of the subtype relation, the *packed encoding*, the *bit-packed encoding* and the *compact encoding*. We describe how they are computed and how they are used to implement constant-time tests. The packed encoding extends to multiple subtyping an algorithm first described by Cohen [3] and rediscovered independently by Queinnec [13]. When multiple subtyping is not used our solution is the same as Cohen's. We improve on the runtime performance of tests by removing a bound check advocated by Cohen. The code sequence that implements the type test is short enough to be inlined and thus avoid the cost of an extra call. The computation of the packed encoding is very fast and requires little memory. Thus, it is well suited for on-the-fly updates of the hierarchy. Furthermore, there are categories of updates that do not require recomputing the encoding. The second new encoding, called bit-packed encoding, reduces further the space requirement of the packed encoding at the cost of slower type inclusion tests. The last encoding, compact encoding, adapts the compact dispatch table technique of Vitek and Horspool [16]. It is designed for very large hierarchies. For small and medium-sized ones, it is less efficient than the packed encoding. We compare the new encodings and algorithms to the bit matrix encoding and the near optimal hierarchical encoding of [12] and conclude with guidelines for choosing an encoding of the subtype relation.

The remainder of this paper is organized as follows. Section 2 introduces terminology, important definitions, and a running example. Section 3 briefly reviews previous work in the field, including the binary matrix encoding, Cohen's encoding and the near optimal hierarchical encoding. Section 4 presents the packed encoding, the type inclusion test and the encoding construction algorithm. Section 5 presents the bit-packed encoding. Section 6 presents the compact encoding. Section 7 compares time and space requirements of the techniques on a set of 11 benchmark programs. Finally, section 8 presents our conclusions.

2 Definitions and Example Hierarchy

A type hierarchy $H = \langle \mathcal{T}, <: \rangle$ is a set of types $\mathcal{T}$ and a reflexive, transitive, anti-symmetric subtype relation $<:$. If $A <: B$ holds, then we say that A is a subtype of B and B is a supertype of A. In class-based languages this hierarchy is defined explicitly by the programmer through the subclassing relationship between classes. In languages with structural subtyping, the subtype relation is derived automatically.

We also define an anti-reflexive, anti-symmetric direct subtype relation $<:_d$

$$
\begin{array}{lll}
<:_d & \equiv & \{\langle x \in \mathcal{T}, y \in \mathcal{T} \rangle | x <: y \; \wedge \\
 & & \quad (\nexists z \in \mathcal{T} | x \neq z \;\wedge\; y \neq z \;\wedge\; \; x <: z <: y)\}
\end{array}
$$

The subtype relation is represented by a directed acyclic graph, shown in fig. 1, with vertices for types and edges for the subtype relation. By convention, we draw supertypes above their subtypes and draw only edges in $<:_d$. We also need the following definitions:

$$
\begin{array}{lll}
roots(\mathcal{T}) & \equiv & \{x \in \mathcal{T} | \; \nexists y \in \mathcal{T} : x <: y\} \\
parents(x) & \equiv & \{y \in \mathcal{T} | x <:_d y\} \\
children(x) & \equiv & \{y \in \mathcal{T} | y <:_d x\} \\
ancestors(x) & \equiv & \{y \in \mathcal{T} | x <: y\} \\
descendants(x) & \equiv & \{y \in \mathcal{T} | y <: x\} \\
multis(\mathcal{T}) & \equiv & \{x \in \mathcal{T} | card(parents(x)) > 1\}
\end{array}
$$

where $card(\mathcal{S})$ is the cardinality of a set $\mathcal{S}$. *Roots* is the set of top level types. *Parents* and *children* are sets of direct supertypes and subtypes, respectively. *Ancestors* and *descendants* are sets of all subtypes and supertypes of a type. *Multis* is the set of all types with more than a single direct supertype.

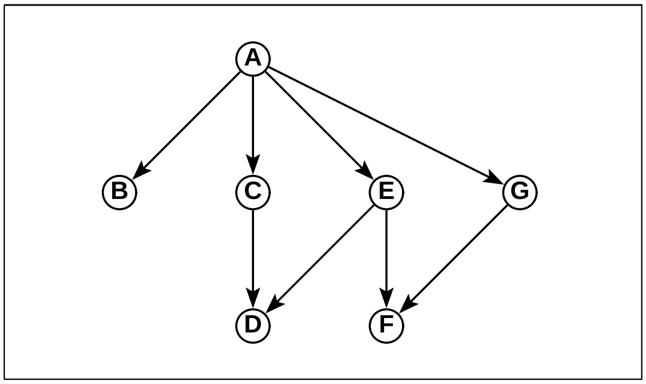


Figure 1: A small type hierarchy.

```
class Object_rep {
      Type_rep type_rep
      ...
}
```

Figure 2: Object runtime representation.

A *single subtyping* type hierarchy restricts the number of direct parents to one, $card(parents(x)) \leq 1$. We assume single rooted hierarchies, *i.e.* $card(roots(\mathcal{T})) = 1$. In practice, we fix hierarchies that do not fulfill this assumption by adding an extra root type R so that $children(R) = roots(\mathcal{T})$. The hierarchy of figure 1 is a multiple subtyping hierarchy, with $roots(\mathcal{T}) = \{A\}$, $multis(\mathcal{T}) = \{D, F\}$, $parents(F) = \{E, G\}$ and $ancestors(F) = \{A, E, G\}$.

We define the *level* of a type in a hierarchy as the length of its longest path to the root:

$$level(x) \equiv \begin{cases} 0 & \textbf{if } parents(x) = \{\} \\ max(L) + 1 & \textbf{otherwise} \end{cases}$$
$$\textbf{where} \quad L \equiv \{level(y) | y \in parents(x)\}$$

For the runtime representation of objects, we assume they are implemented by data structures with, as a common prefix, a reference to a type information data structure, the `Type_rep` field of fig. 2. In many implementations this field can be merged with the dispatch data structure (*e.g.* the `vtbl` of C++).

Unless explicitly stated, the type test instruction sequences check subtyping against a type known at compile-time. This corresponds to a test of the form

```
obj instanceof A
```

where `A` is a type constant. This is the most frequent use of a subtype test. We assume that the compiler or linker uses this information to fill in the values of the appropriate constants once the program is complete.

As a convention, we prefix compile- and link-time constants with a #.

3 Previous Work

3.1 Hierarchy Traversal Algorithms

Type inclusion tests for single subtyping are trivially implemented by traversing a linked list of types, as proposed by Wirth [17]. The linked list encoding requires little space and may be updated incrementally. Unfortunately, tests are slow, running in time proportional to the distance between the two types in $<:_d$. This led Wirth to switch to a constant-time scheme for OBERON [18]. Linked data structures for multiple subtyping only increase the cost of type tests. We have experimented with linked representations as well as with other non-constant-time schemes based on hashing while working on this paper. Non-constant-time techniques are much slower than the algorithms discussed in the remainder of the paper. We decided to concentrate on constant-time solutions.

3.2 Constant-time Algorithms

3.2.1 Binary Matrix (BM)

Type inclusion tests can be performed in constant-time if the subtype relation is encoded as a binary matrix. If $N = card(\mathcal{T})$, and $\gamma : \mathcal{T} \to [1 \ldots N]$ is a one-to-one mapping from types to indices, we build a $N \times N$ binary matrix M_{BM} such that:

$$M_{BM}[\gamma(x), \gamma(y)] \equiv \begin{cases} 1 & \textbf{if } x <: y \\ 0 & \textbf{otherwise} \end{cases}$$

The binary matrix encoding for the hierarchy of fig. 1 is shown in fig. 3(a).

The runtime representation of types decomposes the matrix into rows corresponding to a type and stores each row into a `Type_rep` data structure, fig. 3(b). Every type representation has the same layout. This data structure has two fields: a position, `pos`, and a sequence of $(N + 31)$ **mod** 32 words, `row`. The position field encodes γ and is used during type inclusion testing to compute a word index and a bit index. If we assume 32 bit words, the word index is `pos >> 5` and the bit index is `pos & 31`.

With BM, a type inclusion test is simply an array access, a bit shift and a comparison. Figure 3(c) tests whether the type of an object `obj` is a subtype of a type with known `word_pos` and `bit_pos`. The machine instruction sequence for this test is given in the appendix.

This encoding is trivial to compute. Its main drawback is that it has quadratic space requirements. For large programs, half megabyte matrices are easily conceivable. Nevertheless, the simplicity of the binary matrix has motivated its use in practice [11], [4].

The other constant-time algorithms presented in this paper use encodings which can be viewed as compressed forms of the binary matrix. The constraint on the compression is that very fast random access to elements must be guaranteed. In this view, the works on parse table optimization and dispatch table optimization are closely related, as, in both cases, their aim is to compress sparsely populated matrices. The parse table compression techniques discussed by Dencker, Dürre and

A	0
B	1
C	2
D	3
E	4
F	5
G	6

γ

A	1	0	0	0	0	0	0
B	1	1	0	0	0	0	0
C	1	0	1	0	0	0	0
D	1	0	1	1	1	0	0
E	1	0	0	0	1	0	0
F	1	1	0	0	0	1	1
G	1	0	0	0	0	0	1
γ	0	1	2	3	4	5	6

M_{BM}

(a) The encoding of figure 1.

```
class  Type_rep  {
     int32  pos
     array [1...N] of int32  row
}
```

(a) Runtime data structures.

```
Type_rep  type := obj.type_rep
int32  word := type.row[#word_pos]
if  (bit_extract(word, #bit_pos) = 1)
```

(c) Type inclusion test.

Figure 3: Binary Matrix (BM).

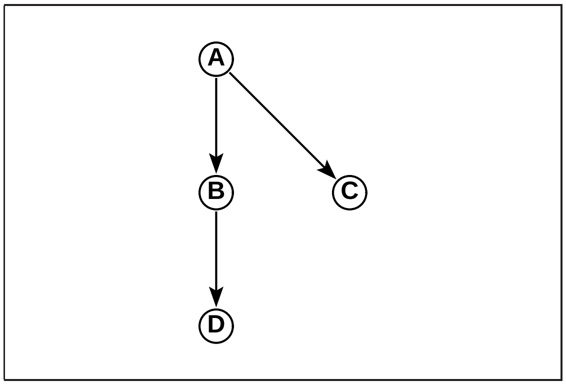
Heuft [5] have influenced works in the field of dispatch table compression [7], [15]. The compact encoding is in fact a straightforward adaptation of compact dispatch tables of [15].

3.2.2 Cohen's Algorithm

Cohen proposed the first practical algorithm for performing subtype tests in constant-time [3]. Cohen's idea is a variation of Dijkstra's "displays" [6]. Each type is identified by a unique type identifier, `tid`, which is simply a number. The runtime type information data structure also records each type's complete path to the root as a sequence of type identifiers. The key trick is to build, for each type x, an array of $card(ancestors(x))$ type identifiers so that for each ancestor y, the `tid` of y is stored at an offset equal to $level(y)$ in the array. The Cohen encoding for sample hierarchy of fig. 4(a) is given in fig. 4(b).

With this encoding, type inclusion tests reduce to a bound-checked array access and a comparison operation. The bound check is necessary as array sizes are not uniform. The runtime data structure, shown in fig. 4(c), consists of a level field, `level` and a sequence of L type identifiers, `row`, where L is equal to the value of the current type's `level`. Note that the type identifier of a type x is stored at `x.row[x.level]`. The code sequence that tests whether an object's type is a subtype of some known type is shown in fig. 4(d).

The advantages of Cohen's algorithm are that it is both easy to understand and easy to implement, it performs tests in constant-time and requires little space. The packed algorithm of section 4 extends the algorithm to multiple subtyping and proposes a type inclusion test that is faster than the one outlined above.



(a) A small single subtyping hierarchy.

	tid	*lvl*
A	1	0
B	2	1
C	3	1
D	4	2

A	1		
B	1	2	
C	1	3	
D	1	2	4

(b) The encoding of figure 4(a).

```
class  Type_rep  {
     int16  level
     array [0...L] of int16  row
}
```

(c) Runtime data structures.

```
Type_rep type := obj.type_rep
if   (type.level ≤ #level
    &&  type.row[#level] = #tid )
```

(d) Type inclusion test.

Figure 4: Cohen's encoding.

3.2.3 Hierarchical Encodings (NHE)

Hierarchical encoding represents each type with a set of integers. This set must be chosen so that

$$x <: y \quad \Leftrightarrow \quad \gamma(x) \subseteq \gamma(y)$$

where $\gamma(x)$ maps type x to its set representation. Thus, the set of a subtype has to be a superset of the set representing its parent. The sets have a natural representation as bit vectors; an example is shown in fig. 5(a). In the bit vector representation the test function becomes

$$x <: y \quad \Leftrightarrow \quad \gamma(x) \vee \gamma(y) = \gamma(x)$$

or alternatively

$$x <: y \quad \Leftrightarrow \quad \gamma(x) \wedge \gamma(y) = \gamma(y)$$

A simple, but inefficient way to construct the bit vectors is to map each type into the corresponding row of the binary matrix of section 3.2.1. The resulting bit vectors are extremely sparse as the number of ancestors of a type is usually much smaller than the total number of types. Better techniques have been proposed in the literature, in particular the modulation method[3] of Aït-Kaci *et al.* [1] and the gene encoding technique of Caseau [2], which try to minimize the range of integers used to construct the sets, thus shortening the corresponding bit vectors. It is well known that finding an optimal bit vector encoding for partial ordered sets is NP-hard [10] and that there exist classes of partial ordered sets (distributive and simplicial lattices) where an optimal encoding is as large as the number of types with only one supertype [10]. Fortunately, type hierarchies can be encoded much more compactly than distributive lattices. In a previous paper, we have developed a new and improved version of the Caseau approach [2] which we call Near Optimal Hierarchical Encoding (NHE) [12]. This version generalizes Caseau's algorithm by expressing it as a graph coloring problem. It is able to encode arbitrary partially ordered sets rather than just lattices [2]. Our algorithm generates the sets faster and generates much smaller sets (about 50% percent smaller than our implementation of [2]), thus making type inclusion tests more efficient.

A complete description of the algorithm can be found in [12], we will summarize it briefly here. A simple version of the technique would assign a set element (*i.e.* a position of a bit in the bit vector) to each node in the type hierarchy graph. This element distinguishes the node from other nodes. This distinguishing element is called a gene by Caseau. The set representation for a type is formed as the union of all its ancestor's sets of genes plus its own gene. However, if the set of ancestors of a type x with more than one immediate parent is not a subset of another ancestor set, then x does not need a gene. We can construct a conflict graph where the nodes represent types and the edges connect types which are not allowed to use the same gene. Graph coloring is then used to assign different genes to conflicting nodes. A crucial part of the technique, performed prior to computing the conflict graph, is inserting extra nodes into the hierarchy in order to balance the graph – the aim is to reduce the maximum number of children possessed by any node and that will tend to reduce the number of nodes that require distinct genes. Fig. 5(a) shows the NHE encoding of the example hierarchy (fig. 1). The algorithm uses only four genes as D and F are able to reuse the genes of their parents, the root A does not need a gene as it encodes the empty set. This technique yields the optimal encoding for single subtyping and near optimal encoding for multiple subtyping hierarchies.

A	0000
B	1000
C	0100
D	0110
E	0010
F	0011
G	0001

(a) The encoding fig. 1.

```
class Type_rep {
        array [1...H] of int32  row
}
```

(b) Runtime data structures.

```
Type_rep type := obj.type_rep
if  ( (type.row[i] &  #row_i)  =  #row_i)
  // repeated for i from 1 to H
```

(c) Type inclusion test.

Figure 5: Near Optimal Hierarchical Encoding (NHE).

The bit vector is of fixed size and can be stored at any fixed position inside the class object. The runtime data structure is shown in fig. 5(b), `row` is a sequence of H integers, H is the length of the bit string in words.

[3] The modulation method is an efficient encoding of lattices which is used to perform lattice operations such as finding the least upper bound or greatest lower bound, as well as relative complementation. Type hierarchies are not necessarily lattices.

The comparison part of the test function has to be replicated for each machine word used in the bit vector. This leads to the problem that with increasing code length both execution time and instruction space increase. The number of unrollings is only known at link time when the entire hierarchy is at hand, so the algorithm is constant-time at run-time but not constant-time at compile-time[4]. The implementation of the run-time test against a known bit vector (`#row`) is shown in fig. 5(c).

This implementation will be referred to as inline near optimal hierarchical encoding, INHE. It has three drawbacks: first it requires varying numbers of instructions, second, even in the best case, the instruction sequence is longer than for the other algorithms. This causes code bloat as discussed in sect. 7.3. A slightly slower alternative is to wrap the test in a function, we refer to this solution as the generic near optimal hierarchical encoding, or GNHE. GNHE is implemented by coding a number of similar type test functions, one for each unrolling factor. Then, depending on the length of the bit vector, the appropriate test function will be called. The third drawback of the method is that it is computationally intensive and that the full encoding must be regenerated after any change to the type hierarchy.

3.3 Relative Numbering

We mention briefly one last encoding of the subtype relation based on relative numbering of trees. In a tree it is possible to find out if a node is a child of another node as follows. For each node store two numbers, *left* and *right*. Traverse the tree in order, for each new node increment a counter c. When a node is first encountered, store c in *left*. When the traversal leaves the node store the current value of c in *right*. A node n_1 is a child of a node n_2 if

$$n_2.left \leq n_1.left \quad \wedge \quad n_1.left \leq n_2.right$$

A single subtyping hierarchy is a tree; relative numbering is therefore a very compact and elegant representation of the single subtyping relation. This scheme is used in the DEC SRC Modula-3 system. Unfortunately, there is no obvious way to extend the technique to multiple subtyping.

4 Packed Encoding (PE)

Experience with binary matrices shows that they are always sparse. It is therefore not surprising that they can be compressed. We propose a technique which works well in practice and manages to reduce the size of encodings of real type hierarchies.

In the binary matrix encoding, γ is a one-to-one mapping from types to matrix indices. Each type has a column and a row of the matrix. In the packed encoding, we propose to reuse columns for unrelated types. This reuse of columns is similar in spirit to the reuse of genes in hierarchical encoding and to the levels of Cohen's algorithm.

4.1 The encoding

For the packed encoding of a hierarchy $\langle \mathcal{T}, <: \rangle$ with N types, we construct a $N \times P$ bucket matrix M_{PE}

$$M_{PE} : \mathcal{T} \times [1 \ldots P] \rightarrow tid$$

so that

$$x <: y \quad \Leftrightarrow \quad M_{PE}[x, \gamma(y)] = M_{PE}[y, \gamma(y)]$$

where $\gamma : \mathcal{T} \rightarrow [1 \ldots P]$ maps types to columns indices (*N.B.* we call columns of M_{PE}, *buckets*), and $\tau : \mathcal{T} \rightarrow tid$ maps types to identifiers, which are simply small numbers. The number of columns P is computed by the bucket assignment algorithm of sec. 4.3. For an example of packed encoding, consider fig. 6 which encodes the hierarchy of fig. 1.

The type inclusion test to determine whether A is a subtype of B proceeds as follows:

$$\begin{array}{rcl} A <: B & \equiv & M_{PE}[A, \gamma(B)] = M_{PE}[B, \gamma(B)] \\ & & M_{PE}[A, 1] = M_{PE}[B, 1] \\ & & 1 = 1 \\ & & true \end{array}$$

Buckets partition the set of types according to a simple rule: no two types in the same bucket may have

A	1
B	2
C	2
D	3
E	4
F	3
G	2

γ

A	1
B	1
C	2
D	1
E	1
F	2
G	3

τ

A	1	0	0	0
B	1	1	0	0
C	1	2	0	0
D	1	2	1	0
E	1	0	0	1
F	1	3	2	1
G	1	3	0	0
γ	1	2	3	4

M_{PE}

Figure 6: Packed encoding of fig. 1.

[4] This use of "constant" is a slight abuse of language. In our set of benchmark programs the maximal number of unrollings is 3 as the longest bit vector length is 96, [12]. The longest test takes 18 machine instructions. The shortest test is performed in 8 machine instructions. Note also that the number of instructions is solely determined by the supertype. So, if the supertype is known at link-time (this accounts for the overwhelming majority of type tests in real programs) the number of instructions needed is also known statically.

common descendants. Thus a valid packed encoding must abide by the following *bucket assignment rule*.

Rule 1 *Bucket assignment. Types in the same bucket can not have common subtypes.*

$$\gamma(x) = \gamma(y) \quad \Rightarrow \quad \mathit{descendants}(x) \cap \mathit{descendants}(y) = \{\}$$

where $x \in \mathcal{T} \ \wedge \ y \in \mathcal{T} \ \wedge \ x \neq y$.

Clearly, this rule implies that in pathological cases the packed encoding may degenerate into a binary matrix. This occurs for a flat hierarchy with a bottom element that is a subtype of every other type. Fortunately, such a hierarchy is unlikely as it implies that some type has all the operations and attributes of all other types in the program.

Identifiers are assigned so as to ensure that two types in the same bucket will not have the same identifier. A valid encoding must abide by the following *identifier assignment rule*.

Rule 2 *Identifier assignment rule. Types in the same bucket have different identifiers.*

$$\gamma(x) = \gamma(y) \quad \Rightarrow \quad \tau(x) \neq \tau(y)$$

where $x \in \mathcal{T} \ \wedge \ y \in \mathcal{T} \ \wedge \ x \neq y$.

4.2 Implementing type inclusion tests

The runtime representation of a type assumed by the packed encoding is shown in fig. 7(a). It is composed of a short integer `bucket` which represents the bucket to which the type was assigned, *i.e.* the value of γ, and an array of bytes, `row`, which contains the identifiers of all ancestors of the type—each array is a row of M_{PE}. The type identifier (*i.e.* the value of τ) does not need to be stored explicitly as it can be fetched from `row`. Furthermore, type identifiers can be small numbers as the assignment rule (rule 2) does not require them to be globally unique. Identifiers need only be unique within a bucket. In our set of benchmarks, only a few buckets contain more than 255 types. So, we chose to limit identifiers to a byte and create additional buckets when necessary.

The type inclusion test for checking whether an object `obj` is a subtype of a type with identifier `#tid` and bucket `#bucket` is shown in fig. 7(b). The type test is faster than Cohen's encoding; it is not necessary to perform a bound check since all `row` arrays have the same length. The machine instruction sequence, shown in the appendix, is four instructions long. This is shorter than any known multiple inheritance dispatch sequence[5] and probably short enough to be inlined.

```
class  Type_rep  {
       int8   bucket
       array [1 ...P] of int8  row
}
```

(a) Runtime data structures.

```
Type_rep   type := obj.type_rep
if  (type.row[ #bucket ]  =  #tid )
```

(b) Type inclusion test.

Figure 7: Implementing the Packed Encoding (PE).

4.3 Computing the packed encoding

The bucket assignment rule can be turned into an algorithm without too much effort. It suffices to associate with every type the set of its descendants, and to maintain, for every bucket, a set that is the union of the descendant sets of all of the types it contains. The algorithm is then to build a list of types sorted by their level, to guarantee that we visit parents before children. Then, for each type in the list, the algorithm must find a bucket for which the intersection between the bucket's set of descendants and the type's set of descendants is empty. If no such bucket can be found, a new bucket is added. This is what we did in an earlier version of this paper. Unfortunately, the result is an extremely inefficient algorithm which spends most of its time performing intersections and unions of large sets—the sets are arbitrary subsets of $\mathcal{T}$.

We present a more sophisticated algorithm which is an order of magnitude faster and yet remains simple and easy to implement. The crucial idea is to separate the single subtyping portion of the hierarchy from the multiple subtyping portion and to use this to refine the bucket assignment rule. We start by defining three disjoint subsets of $\mathcal{T}$. The first subset is the set of *join* types. A join type is a type with multiple parents (*i.e.* direct supertypes) which has only single subtyping descendants.

$$\mathit{join}(\mathcal{T}) \ \equiv \ \{x \in \mathit{multis}(\mathcal{T}) | \ \nexists y \in \mathit{multis}(\mathcal{T}) : y <: x\}$$

The second subset is the set of *spine* types. Any ancestor of a join type belongs to this set.

$$\mathit{spine}(\mathcal{T}) \quad \equiv \quad \{x \in \mathit{ancestors}(y) | y \in \mathit{join}(\mathcal{T})\}$$

The last subset is the set of *plain* types, these are types which are neither in spine nor in join. A plain type is

[5] Single inheritance dispatch in a statically typed language can be done in three instructions [8]. Note also, that multiple subtyping dispatch in Java can be done in 3 instructions [11].

a type that has a single parent, and whose descendants are also plain types.

$$plain(\mathcal{T}) \equiv \mathcal{T} - (spine(\mathcal{T}) \cup join(\mathcal{T}))$$

We will also use two list building functions *level_order* and *rev_level_order*. Each of them returns a list of types sorted by their level.

$$\begin{aligned} level\_order(\mathcal{S}) \quad &\equiv \quad [x_1, \ldots, x_N] \\ &\quad\ \ \textbf{where } N \equiv card(\mathcal{S}), \\ &\quad\ \ \textbf{and } level(x_i) \leq level(x_{i+1}) \end{aligned}$$

$$\begin{aligned} rev\_level\_order(\mathcal{S}) \quad &\equiv \quad [x_1, \ldots, x_N] \\ &\quad\ \ \textbf{where } N \equiv card(\mathcal{S}), \\ &\quad\ \ \textbf{and } level(x_i) \geq level(x_{i+1}) \end{aligned}$$

Rule 3 *Bucket assignment (plain and join). Plain and join types may be assigned the same bucket only if they are not related by <:.*

$$\gamma(x) = \gamma(y) \Rightarrow x \notin ancestors(y) \qquad (a)$$

$$\gamma(x) = \gamma(y) \Rightarrow y \notin ancestors(x) \qquad (b)$$

where $x \in join(\mathcal{T}) \cup plain(\mathcal{T}) \wedge y \in join(\mathcal{T}) \cup plain(\mathcal{T} \wedge x \neq y$.

This rule is trivial since, for single subtyping, the only way for two types to have a common descendant is that either $x <: y$ or $y <: x$.

Rule 4 *Bucket assignment (spine). Two spine types may be assigned the same bucket only if they have no join type in common.*

$$\gamma(x) = \gamma(y) \quad \Rightarrow \quad joins(x) \cap joins(y) = \{\}$$

where $x \in \mathcal{T} \ \wedge \ y \in \mathcal{T} \ \wedge \ x \neq y \ \wedge$
$joins(z) \equiv descendants(z) \cap join(\mathcal{T})$.

This rule is equivalent to rule 1. By construction, every spine type has one or more join nodes in its descendants list. If $x <: y$ then $joins(x) \cap joins(y) = joins(x) \neq \{\}$. If $y <: x$ then $joins(x) \cap joins(y) = joins(y) \neq \{\}$. If $x \not<: y$ and $y :\not> y$ then if the types have common descendants at least one of them must be in $spine(\mathcal{T})$.

The bucket assignment algorithm, shown in fig. 8 starts by assigning buckets to spine types, as the other types depend on them. Spine types are visited in reverse topological order as the lower types are less likely to conflict with each other. A spine type is added to a bucket if the bucket is not full (fewer than 255 types) and if adding the type to the bucket does not violate rule 4. Checking the validity of the rule requires types and buckets to have a set of join types. The set of join types of the bucket is updated each time a type is added. Note that the size of these sets is limited by the number of join nodes in the hierarchy. If there is no bucket where to put the type, a new bucket must be created. Another reason for visiting join types in reverse level order is that we can build the join sets while

```
$\mathcal{T} := load\_hierarchy()$
$Buckets := \{\}$

foreach($x \in \mathcal{T}$)
    $x.joins := \{\}$
    $x.used := \{\}$
foreach($x \in join(\mathcal{T})$)
    foreach($y \in parents(x)$)
        $y.joins := y.joins \cup \{x\}$

foreach($x \in rev\_level\_order(spine(\mathcal{T}))$)
    $found :=$ false
    foreach($b \in Buckets$)
        if($card(b.elements) \leq 255$
          $\wedge\ x.joins \cap b.joins = \{\}$)
            $found :=$ true
            $b.elements := b.elements \cup \{x\}$
            $b.joins := b.joins \cup x.joins$
            break
    if($found =$ false)
        $b :=$ new$Bucket$
        $Buckets := Buckets \cup \{b\}$
        $b.elements := b.elements \cup \{x\}$
        $b.joins := x.joins$

    foreach($y \in parents(x)$)
        $y.joins := y.joins \cup x.joins$

foreach($x \in level\_order(plain(\mathcal{T}) \cup join(\mathcal{T}))$)
    $found :=$ false
    foreach($b \in Buckets$)
        if($card(b.elements) < 255 \wedge\ b \notin x.used$)
            $found :=$ true
            $b.elements := b.elements \cup \{x\}$
            $x.used := x.used \cup \{b\}$
            break
    if($found =$ false)
        $b :=$ new$Bucket$
        $Buckets := Buckets \cup \{b\}$
        $b.elements := b.elements \cup \{x\}$
        $x.used := x.used \cup \{b\}$
    foreach($y \in children(x)$)
        $y.used := y.used \cup x.used$
```

Figure 8: Bucket assignment algorithm.

assigning buckets. After assigning a bucket to a type, the join sets of the parents of the type are updated with the joins of the current type. The second part of the algorithm deals with non-spine types. These types are visited in level order to ensure that buckets are assigned to parents before children. All that needs to be done is to compute for every type, the set of buckets that have already been used by its ancestors. Any bucket not in this set can be used for the type. This implements rule 3.

Building the runtime data structures once the buckets have been assigned is merely a matter of traversing the bucket set in any order and creating `Type_rep` objects. We maintain a counter n that indicate the column index of the bucket, this is used for setting `bucket` (γ) and an intra bucket counter c which is used for type identifiers (τ). The size of the rows, P, is the cardinality of the set of buckets. The last stage of the algorithm is to traverse the hierarchy in level order and set the `row` fields of all types to their correct values.

```
$P = card(Buckets)$
$n := 0$
foreach($b \in Buckets$)
    $c := 0$
    $n := n + 1$
    foreach($x \in b.elements$)
        $c := c + 1$
        $x.type :=$ new $Type\_rep$
        $x.type.bucket := n$
        $x.type.row :=$
                new $Array$ $[1 \ldots P]$ of $int8$
        foreach($i \in [1 \ldots P]$)
            $x.type.row[i] := 0$
        $x.type.row[x.type.\gamma] := c$

foreach($x \in level\_order(\mathcal{T})$)
    foreach($y \in children(x)$)
        foreach($i \in [1 \ldots P]$)
            $y.type.row[i] :=$
                $y.type.row[i] \ | \ x.type.row[i]$
                // | is the logical-or operator
```

Figure 9: Building the PE type representation.

4.4 Discussion

The bucket construction algorithm is quite fast (see sec. 7.2), but does not guarantee an optimal bucket assignment. In some cases it may allocate too many buckets. Consider the type hierarchy of fig. 10, The optimal assignment is

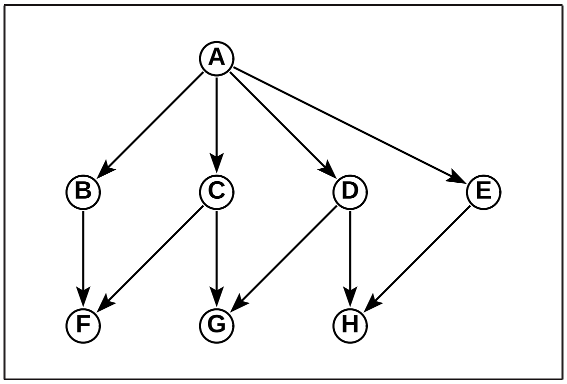


Figure 10: Type hierarchy.

Bucket	1	2	3	4
Types	A	B, D	C, E	F, G, H

Depending on the order in which level–1 types are visited the algorithm may return the following bucket assignment:

Bucket	1	2	3	4	5
Types	A	B, E	C, H	D, F	G

This assignment requires one extra bucket. Because B and E were put in the same bucket, C and D had to be placed in different buckets.

The obvious approach for finding the optimal assignment would require graph coloring, which we wanted to avoid, as one of the strong points of this algorithm is its speed. But, before looking for more complex solutions, it is a good idea to evaluate what there is to gain. One way to do this is to compute an approximation of the lower bound on the number of buckets needed in our set of benchmark programs and compare that with the number of buckets generated by the bucket assignment algorithm. A very simple lower bound is the largest value of $ancestors(x)$ for each hierarchy. It is guaranteed by the bucket assignment rules that the optimal encoding will have at least that many buckets. We have done that for our benchmark programs. The results are summarized in table 1. The only three programs where we actually lose are GEO, EDE and LOV; all three are the output of a code generator which makes extensive use of multiple subtyping—see 7.1 for a description

Hierarchy	VW2	DG3	NXT	ET+	UNI	SLF
max ancestors	15	14	8	9	10	41
comp. buckets	15	14	8	9	10	41

Hierarchy	GEO	LOV	EDE	LAU	JAV
max ancestors	50	24	23	16	7
comp. buckets	51	27	26	16	7

Table 1: Assessing the quality of bucket assignments.

of the benchmark suite. The difference in the case of GEO is one bucket and, for LOV and EDE, three buckets. Such small numbers do not warrant complicating the algorithm. We also believe that these examples are atypical in their heavy use of multiple subtyping.

5 Bit-Packed Encoding (BPE)

The choice of an uniform bucket length for the packed encoding was motivated by an emphasis on speed of type inclusion tests. If data size is the issue, the encoding can be compressed further by allowing variable bucket lengths. A length of 8 bits is used for PE which allows 255 types to share the same bucket. In practice, the number of types that actually share a bucket is much lower. In fact, for the multiple subtyping hierarchies of our benchmark suites, 33% of the buckets contain a single type. These buckets actually need a single bit. The *bit-packed encoding* (BPE) uses variable sized representations for buckets. With this simple change it improves the compression rate of all multiple subtyping examples of the benchmark suite (see section 7.3).

The BPE encoding is generated by an algorithm which is run after PE generation and which simply packs as many buckets as possible in a single word. Fig. 11(a) shows the result for the hierarchy of fig. 1. The value of γ is the offset in the bit string, τ is the type identifier bit string. For practical purposes, the BPE algorithm will not split type identifiers across word boundaries. Thus words may be padded to 32 bits if needed. In fig. 11(a), the identifier of type A requires single bit while those of all the other type require 2 bits.

The main differences between PE and BPE are their runtime data structures and type inclusion tests. With the bit-packed encoding, each `Type_rep` contains an array of B 32 bit words, `row`, where B is obtained by packing the PE encoding. Type identifiers are represented by numbers no larger than 8 bits at an arbitrary offset in a word. To be able to extract a type identifier, it is thus necessary to know its word, its position in a word and its length. Thus, a `Type_rep` contains a `bucket_word` field, a `bucket_pos` field and a `bucket_mask`. The last field is used to mask irrelevant bits out of a byte. The runtime data structure is shown in fig. 11(b). The type inclusion test, shown in fig. 11(c), extracts the type identifier by shifting by `bucket_pos` and masking with `bucket_mask`.

We refer to the machine instruction sequence for the BPE test of fig. 11(c) as *inline bit-packed encoding* (IBPE). The IBPE type test takes 6 machine instructions. Similarly to the INHE, long instruction sequences may lead to code bloat. This can be avoided by performing most of the type test out of line, in a separate procedure. This variant of BPE is called *generic bit-packed encoding* (GBPE). It reduces the per test site overhead to 3 instructions. The GBPE type test is given in the appendix.

The BPE has another advantage over PE. For the worst case scenario of a flat hierarchy described in section 4.1, the space needed for BPE is exactly the same as for the binary matrix. With PE's uniform bucket lengths, the encoding is 8 times as large.

A	1
B	2
C	2
D	4
E	6
F	4
G	2

γ

A	1
B	01
C	10
D	01
E	01
F	10
G	11

τ

A	100000
B	101000
C	110000
D	110010
E	100001
F	111101
G	111000
γ	123456

M_{BPE}

(a) Encoding of fig. 1.

```
class  Type_rep  {
     int8   bucket_word
     int5   bucket_pos
     int8   bucket_mask
     array [1 ...B] of int32  row
}
```

(b) Runtime data structures.

```
Type_rep type := obj.type_rep
int32 word := type.row[#bucket_word]
word := word >> #bucket_pos
word := word & #bucket_mask
if  (  word  =  #tid )
```

(c) Type inclusion test.

Figure 11: Bit-Packed Encoding (BPE).

6 Compact Encoding (CE)

A notable characteristic of all constant-time encodings is redundancy. In Cohen's encoding, a row differs from its parent in only one position. With multiple subtyping, more than one position may differ as each type may have more than one parent. Yet, in general, rows remain fairly constant from one generation to the next.

The compact encoding is a straightforward adaption of the compact dispatch table technique of Vitek and

Horspool [15]. It reduces repetition by introducing sharing between rows of a type matrix. The idea is simple, start with a $N \times M$ matrix (either a binary matrix or the packed encoding, in the following we take the packed encoding) and break it into a number, m, of chunks. Each chunk is composed of N rows and M_i columns. Then, for each chunk, compare all rows and merge equal rows. This yields a set of smaller, $N_i \times M_i$, matrices where $N_i \leq N$ and $M_i \leq M$ for each of the chunks. The choice of the chunk size and of the column in which to put in a chunk relies on heuristics as discussed in [15].

A	γ_1	1
B	γ_2	1
C	γ_2	1
D	γ_3	2
E	γ_2	1
F	γ_3	2
G	γ_2	1

γ_i

A	1
B	1
C	2
D	1
E	1
F	2
G	3

τ

A, B, C, D, E, F, G	1
γ_1	1

M_{1CE}

A, E	0
B	1
C, D	2
F, G	3
γ_2	1

M_{2CE}

A, B, C, G	0	0
D	1	0
E	0	1
F	2	1
γ_3	1	2

M_{3CE}

(a) The encoding of fig. 1.

```
class  Type_rep  {
       short   chunk
       short   bucket
       array [1...M]  of Row   row
}
class  Row  {
       array [1...m_i]  of int8   elem
}
```

(b) Runtime data structures.

```
Type_rep  type := obj.type_rep
Row  row := type.row[ #chunk ]
if  (row.elem[ #bucket ] = #tid )
```

(c) Type inclusion test.

Figure 12: Compact Encoding (CE).

The compact encoding for the small type hierarchy of fig. 1 is shown in fig. 12(a). In this encoding the packed matrix (7×5) is split into three chunks. So, with $m = 3$, the three chunks have dimensions 1×1, 4×1, and 4×2.

The runtime data structure for each `Type_rep` consists of a short integer `chunk` which indicates which γ_i to use, a second short integer `bucket` which is the value of γ and an array of rows, `row`. An element in this array of rows is a chunk, a portion of one of the rows of the original matrix. The actual `Row` objects are shared by multiple `Type_rep` objects. Fig. 12(b) shows these data structures. As before, the type identifier may be recovered from the type, so the identifier of type x is stored at x`.row[`x`.chunk].elem[`x`.bucket]`. The type inclusion test against a type with chunk `#chunk`, bucket `#bucket` and identifier `#tid` is shown in fig. 12(c).

7 Evaluation

This last section evaluates the different constant-time type inclusion test techniques according to four criteria: the runtime characteristics of the type test algorithms, space requirements of the associated encoding, generation time of the encoding and suitability for incremental hierarchy modifications.

We compare five algorithms: the binary matrix (BM) of section 3.2.1, the near optimal hierarchical encodings (NHE) of section 3.2.3, the packed encoding (PE) of section 4, the bit-packed encoding (BPE) of section 5, and the compact encoding (CE) of section 6. Type tests with NHE and BPE can be either performed inline (INHE and IBPE) or in a separate function (GNHE and GBPE). We refer to the algorithms by the above mentioned acronyms.

7.1 Benchmark data sets

Choosing data sets to compare encodings is a tricky task. While it is fairly easy to generate arbitrary directed acyclic graphs, they seldom resemble those of real programs. For example, the degree of multiple subtyping that humans seem to be comfortable with is usually quite low; the average number of direct supertypes is very close to 1 in all large programs we have been able to study. The encodings that we want to compare have been designed to be space efficient representation of type hierarchies, we thus feel that it is necessary to compare them on real-life data sets.

Library	Lang.	Types num.	Level max.	Parent (max./avg.)	Ancestor (max./avg.)
VW2	Smalltalk	1956	15	1 / 1	15 / 6.40
DG3	Smalltalk	1357	14	1 / 1	14 / 6.40
NXT	Obj.-C	311	8	1 / 1	8 / 3.94
ET+	C++	371	9	1 / 1	9 / 4.30
UNI	C++	614	10	2 / 1.01	10 / 4.02
SLF	Self	1802	18	9 / 1.05	41 / 30.88
GEO	Eiffel	1319	14	16 / 1.89	50 / 14
EDE	Eiffel	434	11	7 / 1.66	23 / 7.99
LOV	Eiffel	436	10	10 / 1.71	24 / 8.50
LAU	Laure	295	12	3 / 1.07	16 / 8.13
JAV	Java	225	7	3 / 1.04	7 / 3.43

Table 2: Benchmark type hierarchies.

Another consideration is whether to include single subtyping hierarchies. Since single subtyping is a special case of multiple subtyping, and it is fairly common to find single subtyping hierarchies in languages with multiple subtyping (*e.g.* ET++, see below), we must include single subtyping in this evaluation. Furthermore, as the packed encoding (PE) reduces to Cohen's encoding in the single subtyping case, it is interesting to compare its space requirements with those of the hierarchical encoding.

We use a collection of 11 medium to large type hierarchies to evaluate encodings[6] [9].

Some descriptive data about the hierarchies is given in table 2. *Level* indicates the depth of each hierarchy, *parent* gives both the largest and average number of direct supertypes, and, finally, *ancestors* gives largest and average number of supertypes for each hiearchy.

VW2 and DG3 are both large Smalltalk-80 class libraries, respectively VisualWork2 and Digitalk3. Each class corresponds to a type, the subtype relation is the inheritance relationship between classes. VW2 is our largest hierarchy with almost 2000 types. VW2 is also quite deep with 15 levels. NXT contains types extracted from the NeXTStep class library. ET+ is the ET++ graphical user interface library. UNI is the Unidraw C++ toolkit. SLF contains data extracted from the Self system[7]. This is our largest multiple subtyping example, it is also the deepest hierarchy (18 levels). Notice that the maximum number of parents is 9 which is rather high. The largest number of ancestors 41 and the average number of ancestors is more than 30. Both values are much larger than in class-based languages. GEO, EDE and LOV are Eiffel applications produced by a code generator. They exhibit very large amounts of multiple subtyping, up to 16 parents for GEO. Their average number of parents is also way higher than that of the other hierarchies. LAU is the Laure language of Caseau. Finally, JAV is the Java JDK 1.02 library. We refer to the data sets by their acronyms.

We consider these hierarchies to be fairly large, but expect to see much larger hierarchies for big systems. Another source of large hierarchies is the growing number of code generators that use object-oriented languages (Java for example) as their target. Generated code may use multiple subtyping more extensively as automatic tools are better at keeping track of complex hierarchies than human programmers.

7.2 Runtime behavior of type tests

Based on the machine code sequences given in the appendix, the different algorithms are compared with respect to their speed, instruction count and register usage. The comparison is based on a generic RISC architecture which executes one instruction every cycle with a load latency of 2 cycles and no penalty for correctly predicted branches. The variable H for INHE and GNHE is a factor of the length of the bit string encoding the hierarchy. If the word size is 32 bits and the encoding is n bits, $H = (n+31)\ \mathbf{mod}\ 32$. In our set of programs the largest H was 3. For GNHE, we count the number of instructions at the call site only. All algorithms under consideration guarantee constant-time type tests. In the case of the INHE and GNHE, the time is determined at link-time when the entire hierarchy is known. The, perhaps surprising, result of table 3 is that type tests with PE are as efficient as type tests that use a binary matrix. The other techniques are slower, require more registers and have higher instruction counts.

Resources	BM	INHE	GNHE
Cycles	6	$3 + 6H$	$5 + 6H$
Instructions	4	$3 + 5H$	4
Registers	1	4	5

Resources	PE	IBPE	GBPE	CE
Cycles	6	8	11	8
Instructions	4	6	3	5
Registers	1	1	3	1

Table 3: Comparing runtime characteristics.

7.3 Space requirements

Table 4 summarizes space requirements of the different encodings relative to the binary matrix encoding. Compression rates are computed as $1 - (sizeX/sizeBM)$. These measurements assume 32 bit pointers and 32 bit alignment of the data and do not include the size of the machine code sequences.

[6] We thank Yves Caseau (LAU) and Karel Driesen (VW2, DG3, ET+, UNI, SLF). The benchmark data set is available from `http://www.cs.ucsb.edu/oocsb/classhierarchies/`.

[7] In Self shared behavior is implemented by maps, for our purpose each map represents a type.

The space requirements of the naive approach (BM) can come close to 0.5MB and these get compressed down to 16 KB with NHE and 30 KB with PE and BPE. The size of BM depends on the number of types, we get equally large hierarchies with single (VW2) and multiple subtyping (SLF). NHE has consistently better compression rates. It performs slightly worse on inputs containing multiple inheritance like EDE, LOV and JAV, but interestingly enough performs very well on SLF and GEO. PE demonstrates good compression rates for single subtyping and only adequate compression rates multiple subtyping. BPE improves on PE for all multiple subtyping hierarchies. For instance for SLF, the encoding size drops from 77 KB to 28 KB. CE fails to improve on the PE, except for LOV and EDE where it performs slightly better. The reason for this poor performance is that gains due to sharing parts of bit vectors are offset by the cost of the additional pointers in each type data structure. These numbers suggest that CE needs larger hierarchies to become profitable.

Lib.	BM	NHE	PE	BPE	CE
VW2	485.3	16.0	30.5	30.5	39.3
		(96.7%)	(93.7%)	(93.7%)	(91.9%)
DG3	233.4	10.9	21.2	15.9	24.0
		(95.3%)	(90.9%)	(93.2%)	(89.7%)
NXT	12.4	1.2	2.4	2.4	3.7
		(90.3%)	(80.6%)	(80.6%)	(70.2%)
ET+	17.8	1.4	4.3	2.8	4.8
		(92.1%)	(75.8%)	(84.3%)	(73.0%)
UNI	49.1	2.4	7.2	4.8	8.5
		(95.1%)	(85.3%)	(90.2%)	(82.7%)
SLF	410.8	14.7	77.4	28.1	85.0
		(96.4%)	(81.2%)	(93.2%)	(79.3%)
GEO	221.5	15.9	66.9	25.7	67.1
		(92.8%)	(69.8%)	(88.4%)	(69.7%)
EDE	24.3	3.4	11.9	5.1	10.5
		(86.0%)	(51.0%)	(79.0%)	(56.8%)
LOV	24.4	3.4	11.9	5.1	10.8
		(86.1%)	(51.2%)	(79.1%)	(55.7%)
LAU	11.8	1.1	4.6	2.3	6.2
		(90.7%)	(61.0%)	(80.5%)	(47.5%)
JAV	7.2	0.9	1.8	0.9	2.7
		(87.5%)	(75.0%)	(87.5%)	(62.5%)

Table 4: Space requirements (KB/compression rate).

7.3.1 Considering instruction space

We were able to obtain the number of static type check calls (3861) for the Java library (JDK 1.0.2). If the space requirements for both the table and the instructions are considered, the rankings of the algorithms are completely reversed. The results are presented in table 5. The generic algorithms (GPBE and GNHE) win as they require fewer instructions per test site. The size of the tables actually is irrelevant, code space dominates the size requirements. Nevertheless, the code size measures should be taken with caution: (1) it is not clear how representative this data is, (2) many of these type tests will be inlined away by an optimizing compiler, and (3) the JDK1.0.2 hierarchy was quite small. These numbers should be considered as upper bounds on size requirements.

Space	BM	INHE	GNHEE
code only	60.3	123.4	60.3
data + code	67.6	124.3	61.2

Space	PE	IBPE	GBPE	CE
code only	60.3	90.5	45.2	77.1
data + code	62.1	91.4	46.1	79.6

Table 5: Space requirements with instructions (KB).

7.4 Encoding generation

The time needed to generate the encoding can not be neglected as it will lengthen the overall compile and link cycle time or even play a role at runtime in the case of incremental hiearchy updates.

We have measured the speed of all four algorithms on a 500 MHz 21164 Alpha processor. The running times in milliseconds are shown in table 6. These times were obtained by computing the encoding 100 times for each hierarchy.

The difference between BM, PE, BPE and CE is quite small, all three algorithms run fast. The worst time for BM is 10 msecs for VW2 which is the largest

	BM	(B)PE	CE	NHE
VW2	10	12	13	890
DG3	6	8	9	426
NXT	1	2	2	30
ET+	1	2	2	39
UNI	2	3	4	93
SLF	9	11	14	1367
GEO	8	13	23	1902
EDE	2	4	5	136
LOV	1	4	5	168
LAU	1	2	2	21
JAV	1	1	2	19

Table 6: Encoding generation times (msecs).

hierarchy. PE and CE take 13 and 23 msecs, respectively, for GEO which is large and features heavy multiple inheritance. NHE is slower, yet it still generates encodings in less than 2 seconds.

7.5 Incremental hierarchy updates

Dealing with changes in the subtyping relation is difficult. As for most table compression algorithms small changes in the input can result in widely different compressed outputs. Thus it is not always possible to avoid recomputing the entire encoding.

There are two kinds of changes to the subtype relation: *destructive changes*, changes that modify the type graph either by adding or removing edges between existing vertices, and *additive changes*, changes that only add new vertices and new edges to a type graph. The first kind is usually restricted to programming environments during software development. The second kind may actually occur at runtime when new software components are dynamically linked. In class-based languages, such as JAVA, new classes and interfaces can be loaded at arbitrary points during program execution. The new types thus created are always subtypes of already existing types. In languages with structural subtyping, new types may also be supertypes of existing types.

Supporting dynamic changes to the subtype relation implies that the information dependent on a particular encoding must be localized to some well defined portion of the program and easy to change or update to reflect the new situation. This comes at a cost in efficiency. For one, compile- or link-time constants can not be updated. In general that would prove too costly. Thus type inclusion tests must be wrapped in function calls to a generic test function that expects two `Type_rep` objects and is able to extract the necessary information for a type test out of their fields.

Another trick to speed up recomputation of the encodings is not to recompute them. Or, at least, to wait until the last possible time before doing so. The motivation is that changes often come in batches. As it is economical to recompute for as many types as possible, we must try to wait until all the types in the batch have been added before starting the update. What is the latest time? It is either the first subtype test, or, if we want to be more precise, the first subtype test that involves a new type. So, we can either modify—by overwriting code—the type test function to trigger recomputation, or add extra information to type representations to indicate whether they have been initialized and add an extra check to each type test to verify that both types already have been installed.

In any case, the next question is what to do when recomputing is necessary. Assume that we have batched a group of updates. If the batch contains destructive updates the encoding will have to be recomputed. If the batch contains no destructive updates, the binary matrix does not have to be updated. For a new type, each row has to be extended by an entry and a new row must be added. The cost of extension can be reduced by pre-allocating longer rows with some unused entries. In the case of the hierarchical encoding, recomputing can not be avoided easily. For the packed and compact encodings, adding new subtypes does not necessarily mean recomputing the encoding. Recomputing is only necessary if we add new join types of previously existing types. Otherwise the update can be performed by extending rows. For the bit-packed encoding, the same comments as for PE apply, except that the encoding must also be recomputed if the number of bits required to represent a bucket changes.

When the encoding has to be recomputed, generation time and memory requirements become important. BM, PE and BPE have the fastest generation times. CE follows close behind PE. Finally, NHE is most computationally intensive algorithm and thus less suited to frequent encoding generation.

8 Conclusions

In this paper we have looked at the problem of testing for type inclusion in object-oriented programming languages with multiple subtyping. We evaluated five main techniques for computing type inclusion with different trade-offs. Which is the best type test method? If run-time speed is the primary concern, the Packed Encoding is a clear winner. It ties with the Binary Matrix as achieving the fastest type test times, it is almost as fast to compute, yet it requires much less storage for tables. The packed encoding is thus suited for statically compiled programming languages as well as to environments that permits dynamic addition of new types (as with SMALLTALK and JAVA). If space and speed of tests are equal concerns, the Bit-Packed Encoding is the best choice as it is consistently more compact than the Packed Encoding, yet it is fast to compute and guarantees constant time type inclusion tests. If space is the major concern, our generic Near Optimal Hierarchical Encoding method will give the best results. Finally, we believe that the Compact Encoding may compress some very large hierarchies better than the other encodings but we were not able to substantiate this hypothesis with the data at our disposal.

Source code for the algorithms described in this paper is available from:
http://www.complang.tuwien.ac.at/andi/typecheck/
http://cuiwww.unige.ch/~jvitek/fcttit/

Acknowledgments

The authors wish to thank Ole Agesen, Laurent Dami, Karel Driesen and Manuel Serrano for thoughtful comments on earlier versions of this paper; Christian Queinnec for interesting discussions of alternative techniques and uses of type inclusion tests; and the OOPSLA reviewers as their technical comments helped us improve this paper.

References

[1] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

[2] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *Proc. Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA'93*, Published as SIGPLAN Notices 28(10), pages 271–287. ACM Press, September 1993.

[3] N. H. Cohen. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4):626–629, 1991.

[4] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proc. Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA'96*. ACM Press, October 1996.

[5] P. Dencker, K. Dürre, and J. Heuft. Optimization of parser tables for portable compilers. *ACM Transaction on Programming Languages and Systems*, 6(4):546–572, October 1984.

[6] E. W. Dijkstra. Recursive programming. *Numer. Programming*, (2):312–318, 1960.

[7] K. Driesen. Selector table indexing and sparse arrays. In *Proc. Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA'93*, Published as SIGPLAN Notices 28(10), pages 259–270. ACM Press, September 1993.

[8] K. Driesen, U. Hölzle, and J. Vitek. Message dispatch on pipelined processors. In *Proc. European Conference on Object-Oriented Programming, ECOOP'95*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[9] K. Driesen, U. Hölzle, and J. Vitek. The OOCSB class heterarchy benchmark suite. Technical Report TRCS97-09, Dept. of Computer Science, University of California, Santa Barbara, July 1997.

[10] M. Habib and L. Nourine. Tree structure for distributive lattices and its applications. *Theoretical Computer Science*, 165:391–405, 1996.

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

[12] A. Krall, J. Vitek, and R. N. Horspool. Near optimal hierarchical encoding of types. In *Proc. European Conference on Object-Oriented Programming, ECOOP'97*, Lecture Notes in Computer Science. Springer-Verlag, June 1997.

[13] C. Queinnec. Designing MEROON v3. In C. Rathke, J. Kopp, H. Hohl, and H. Bretthauer, editors, *Object-Oriented Programming in Lisp: Languages and Applications. A report on the ECOOP'93 Workshop*, September 1993.

[14] C. Queinnec. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters (accepted for publication)*, 1997.

[15] J. Vitek. Compact dispatch tables for dynamically-typed object-oriented languages. M.sc. thesis, University of Victoria, April 1995.

[16] J. Vitek and R. N. Horspool. Taming message passing: Efficient method look-up for dynamically-typed languages. In *Proc. European Conference on Object Oriented Programming, ECOOP'94*, Lecture Notes in Computer Science. Springer-Verlag, 1994.

[17] N. Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, 1988.

[18] N. Wirth. Reply to "type-extension type tests can be performed in constant time". *ACM Transactions on Programming Languages and Systems*, 13(4):630, 1991.

Appendix: Implementations in Generic RISC Assembly Code

In all four code sequences below, control transfers to the label FAIL if the type inclusion test fails and drops through to the following instruction if it succeeds.

Binary Matrix

```
    load   [object + #type_rep], type_rep
    load   [type_rep + #word_pos], bit
    lshift bit, 31 - #bit_pos,     bit
    bgez   bit, #FAIL
```

Packed Encoding

```
    load  [object + #type_rep], type_rep
    load  [type_rep + #bucket],    tid
    cmp   tid, #tid
    bne   #FAIL
```

Inline Bit-Packed Encoding

```
    load   [object + #type_rep], type_rep
    load   [type_rep + #bucket_word], tid
    rshift tid, #bucket_pos, tid
    and    tid, #bucket_mask, tid
    cmp    tid, #tid
    bne   #FAIL
```

Generic Bit-Packed Encoding

```
    load   [object + #type_rep], type_rep
    add    #0, #par_tid, par_tid
    call   check_n
check_n:
    load   [type_rep + #bucket_word], tid
    rshift tid, #bucket_pos, tid
    and    tid, #bucket_mask, tid
    cmp    tid, par_tid
    bne   #FAIL
    ret
```

Compact Encoding

```
    load  [object + #type_rep], type_rep
    load  [type_rep + #chunk],  chunk
    load  [chunk + #bucket],    tid
    cmp   tid, #tid
    bne   #FAIL
```

Inline Near Optimal Hierarchical Encoding

```
    load  [object + #type_rep], type_rep
    sethi high(#parent_type), parent
    setlo low(#parent_type),  parent
    // repeated H times:
    load  [type_rep], this_tid
    load  [parent],   parent_tid
    and   this_tid,   parent_tid, this_tid
    cmp   this_tid,   parent_tid
    bne   #FAIL
```

Generic Near Optimal Hierarchical Encoding

```
    load  [object + #type_rep ], type_rep
    sethi high(#parent_type),    parent
    setlo low(#parent_type),     parent
    call  GNHE_H

GNHE_H:
    // comparison of one machine word
    load  [type_rep], this_tid
    load  [parent],   parent_tid
    and   this_tid,   parent_tid, this_tid
    cmp   this_tid,   parent_tid
    bne   #FAIL
    // repeated H times:
    ret
```