Near Optimal Hierarchical Encoding of Types *

Andreas Krall[1], Jan Vitek[2] and R. Nigel Horspool[3]

[1] Institut für Computersprachen, Technische Universität Wien
Argentinierstraße 8, A-1040 Wien, Austria
andi@complang.tuwien.ac.at

[2] Object Systems Group, Centre Universitaire d'Informatique
Université de Genève,
24 rue Général-Dufour, CH-1211 Geneva, Switzerland
jvitek@cui.unige.ch

[3] Department of Computer Science, University of Victoria,
P.O. Box 3055, Victoria, BC, Canada V8W 3P6
nigelh@csr.uvic.ca

Abstract. A type inclusion test is a procedure to decide whether two types are related by a given subtyping relationship. An efficient implementation of the type inclusion test plays an important role in the performance of object oriented programming languages with multiple subtyping like C++, Eiffel or Java. There are well-known methods for performing fast constant time type inclusion tests that use a hierarchical bit vector encoding of the partial ordered set representing the type hierarchy. The number of instructions required by the type inclusion test is proportional to the length of those bit vectors. We present a new algorithm based on graph coloring which computes a near optimal hierarchical encoding of type hierarchies. The new algorithm improves significantly on previous results – it is faster, simpler and generates smaller bit vectors.

1 Introduction

Checking the type of a value is a common operation in typed programming languages. In many cases this requires little more than a comparison. But, modern languages – those which allow types to be extended – complicate matters slightly. Type tests must check for inclusion of types, that is, whether a given type is an extension (or a subtype) of another type. The subtyping relation, a partial order on types, written $<:$, is the transitive and reflexive closure of the direct subtype relation $<:_d$. The common practice for object-oriented programming languages is to derive $<:_d$ directly from the inheritance structure of a program. Thus, each class A defines a type A, and A is a subtype of B either if A = B, or if A inherits from B.

* To appear at ECOOP'97

Type inclusion tests can occur so frequently in programs, particularly object-oriented programs, as to put a strain on the overall system performance. It is important to have type inclusion testing techniques which are both fast and constant-time. However, these techniques should also be economical in space.

The techniques developed in this paper are based on a scheme called *hierarchical encoding*. This scheme represents each type as a set of natural numbers. The sets must be chosen so that either

$$x <: y \Leftrightarrow \gamma(x) \supseteq \gamma(y) \quad \text{(top down encoding)}$$

or

$$x <: y \Leftrightarrow \gamma(x) \subseteq \gamma(y) \quad \text{(bottom up encoding)}$$

where $\gamma(x)$ maps type x to its set representation. Thus, the set used for a subtype has to be a superset of the set representing its parent. The sets have a natural representation as bit vectors. An example for a small hierarchy is shown in figure 1 (top down encoding) and figure 2 (bottom up encoding). In the bit vector representation the test function for hierarchical top down encoding becomes

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

or alternatively

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

which would be implemented in C code as

```
if ((type->code & parenttype->code) == parenttype->code)
    /* it is a subtype */
```

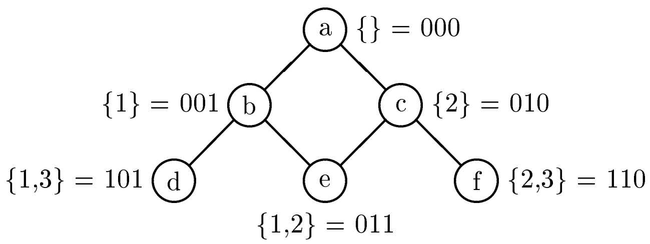

Fig. 1. Hierarchical encoding (top down)

The following sections briefly discuss previous work on type inclusion tests. Subsequently, we describe our new method which uses graph coloring techniques to find nearly optimal set representations for types in a multiple inheritance hierarchy. Finally, we present experimental results which show that our new

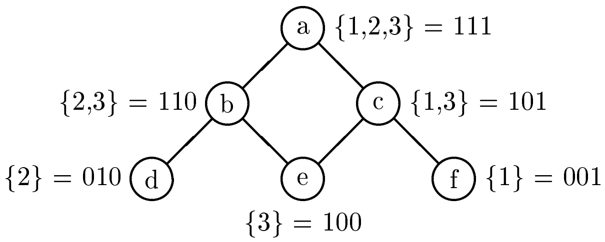

Fig. 2. Hierarchical encoding (bottom up)

method is significantly better than the main competing method on three counts. It generates significantly shorter bit vectors, it computes the vectors faster, and it requires less working storage.

2 Previous work

One 'obvious' algorithm for implementing the type inclusion test is that described by Wirth[Wir88]. To test if $x <: y$, the algorithm proceeds up the inheritance hierarchy starting from x to see if y is an ancestor. However, the algorithm does not run in constant time, which is a problem if the hierarchy becomes large, and the basic algorithm works only for single inheritance hierarchies. Generalizing the method to work with multiple inheritance, either by using backtracking or by constructing sets of parents, makes it slower still.

Another 'obvious' algorithm, and one which achieves a fast constant time test, is to use a precomputed matrix that records all possible relationships. An element M[x,y] in the binary matrix holds a 1 if $x <: y$ and 0 otherwise. Although this implementation is used by some O-O languages, it has the drawback that the matrix can be very large. If there are 2000 types, the matrix will consume nearly 500 KB. (There are a number of schemes for compacting the matrix at the expense of making a look-up in the matrix much slower [DDH84].)

Cohen showed how the type inclusion test can be implemented in constant time using the concept of displays to precompute paths through the inheritance hierarchy[Coh91]. However, Cohen's method uses more memory than Wirth's and, in its original form, is applicable only to single inheritance hierarchies.

Caseau took a different path based on hierarchical top down encoding. He was inspired by a method originally developed for fast implementation of lattice operations [ABLN89] based on hierarchical bottom up encoding and adapted it to the type inclusion problem[Cas93]. Caseau's scheme computes a bit vector for each type. The bit vector represents a set of *genes*, where a gene is represented by a natural number. Each type that has only one parent in the hierarchy has an associated gene. A type with multiple parents has no associated gene. The bit vector for a type T is computed as the set of all genes associated with itself and with all ancestors of T. Testing if $x <: y$ is implemented as a test to see if the set of genes for type x is a superset of the set for y. Caseau's method requires that the type hierarchy be a lattice. This requirement may force extra nodes to be added to the hierarchy. Caseau gave an incremental algorithm for maintaining the lattice property and gave a backtracking technique for finding sets of genes and for updating previously computed sets of genes as the hierarchy is constructed in a top-down manner.

Problems with implementing Caseau's algorithm inspired us to develop our own method for finding sets of genes. We encountered situations where the Caseau algorithm produces incorrect results. Such an example is shown in figure 3. Even if we assume that the error can be corrected, Caseau's method for maintaining the lattice property may force the addition of an exponential number of additional nodes (and therefore also require exponential running time). The worst case is unlikely to occur in practice, but this is nevertheless undesirable behavior. We also discovered that the number of distinct genes used by Caseau's algorithm may be considerably more, sometimes by a factor of 4, than the optimal number. Since the number of genes determines the sizes of the bit vectors (and therefore determines the running time of the set inclusion test too), it is important to minimize the number.

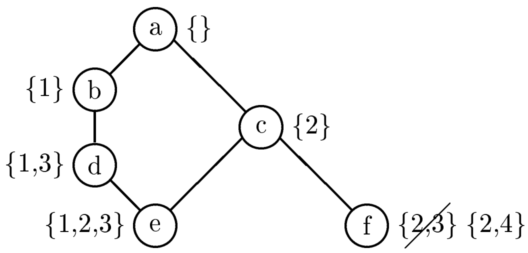

Fig. 3. An incorrect encoding produced by Caseau's algorithm

Habib and Nourine showed that constructing an optimal bit vector encoding for partially ordered sets is NP-hard [HN94]. They also showed in [HN94] and [HN96] that there exist some classes of lattices (distributive and simplicial lattices) where, for an optimal solution, all genes have to be different. For these classes of lattices, therefore, an optimal solution can be constructed in linear time. Partially ordered sets resulting from type hierarchies tend to be very different from distributive lattices, so their encodings are correspondingly an order of magnitude more compact.

3 Near optimal hierarchical encoding

Our near optimal hierarchical encoding algorithm is similar to Caseau's because it also relies on a top down encoding. But, unlike Caseau's algorithm, our algorithm does not require the hierarchy to have a lattice structure – it can encode any partially ordered set. We rely on balancing the height of the hierarchy and use graph coloring to find a near optimal solution. The algorithm was designed for fast execution (it has worst case quadratic run time complexity) for integration into compilers for object oriented programming languages with multiple inheritance or multiple subtyping. Instead of performing a full and slow search for optimal encodings, we have used simple heuristics to find a near optimal solution in a matter of seconds.

3.1 The basic algorithm

To make hierarchical encoding of partially ordered sets practical, we must avoid any restriction to lattice structures and thereby avoid the exponential behavior of lattice completion. We can easily eliminate such a restriction if we associate a gene (i.e. a distinguishing bit) with all nodes in the hierarchy. In contrast, Caseau's method associates a gene only with nodes that have a single parent. However, a better solution is to determine which nodes actually need a gene.

To find a correct hierarchical top down encoding, the following equation must be fulfilled in both directions:

$$x <: y \Leftrightarrow \gamma(x) \supseteq \gamma(y)$$

If a type with only one parent gets a gene and every type inherits all the genes of its super types, the left to right direction of the equation is fulfilled. The opposite direction is more difficult to achieve if the hierarchy is not a lattice. Consider the example hierarchy of figure 4. Types e and f both have more than one direct super type. Type e needs its own gene (4), otherwise its encoding would be included in the encoding of f – which wrongly would state that f is a subtype of e.

The solution to the problem is to give all types with multiple super types a gene if they would violate the above equation. So our algorithm just checks the above equation to determine which types require a gene. For the purpose of describing our algorithm, we first give some definitions:

$parents(x)$	// all nodes which are a direct supertype of x
$children(x)$	// all nodes which are a direct subtype of x
$ancestors(x)$	// all nodes which are a supertype of x
$descendants(x)$	// all nodes which are a subtype of x
$singles$	// all nodes in the hierarchy with a single parent
$multis$	// all nodes with more than one parent
$needgenes$	// all nodes which need a gene

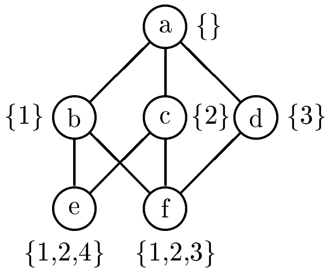

Fig. 4. Multiple inheritance type (e) needs a gene (4)

All nodes $m \in singles$ need a gene and $needgenes$ becomes $singles$. All nodes $m \in multis$ for which $\exists n \in multis$ and not $n <: m$ need a gene (and are added to $needgenes$) if

$$ancestors(m) \cap needgenes \subseteq ancestors(n).$$

For a correct hierarchical encoding it is not necessary for the genes to be distinct. Two genes can be the same if other genes ensure different encodings. For an example just take a hierarchy with two chains. The genes in one chain can be the same as in the other chain. Only the topmost node in each chain must be different to ensure correct encoding. In the hierarchy in figure 1, the genes of d and f can be the same. The different genes for b and c ensure a correct encoding.

Our algorithm determines which nodes cannot use the same genes. For each node, the set of conflicting nodes is determined and a conflict graph is constructed. An edge in the conflict graph means that two nodes are not allowed to use the same gene.

The conflict graph is constructed as follows:

- Every node conflicts with all descendants of its parents.
- In addition, a node N conflicts with all ancestors of any descendants of N's parents if these descendants are not descendants of N.

A correctness proof for the conflicting genes can be found in [Cas93]. It has to be modified slightly, since Caseau missed some cases for the second class of conflicts. The following pseudocode gives a more formal description of conflict graph computation.

for each $x \in hierarchy$ **do**
    $parx := parents(x)$

```
if parx = {} then parx := {x}
for each y ∈ descendants(p), y ≠ x, ∀p ∈ parx do
    enter conflict between x and y in conflict graph
    if y ∈ multis, ¬(y <: x) then
        ∀anc ∈ ancestors(y), anc ≠ y, enter conflict between
            x and anc in conflict graph
```

After the conflict graph has been constructed, graph coloring is used to find a solution to the gene assignment problem. The hierarchical code for a node is then computed as the union of the genes for all its ancestors and for itself.

A better, near optimal, solution can be found if sets of children are subdivided and the hierarchy is *balanced* before the conflict sets are computed. The next two subsections describe both coloring and balancing in some detail. The main steps of the encoding algorithm are as follows (complete pseudocode can be found in the appendix).

```
mark all nodes in hierarchy which need a gene
split children lists and balance the hierarchy
compute conflict graph
color the conflict graph
compute code
```

3.2 Coloring the conflict graph

Computing the chromatic number of a graph (determining the minimal number of colors needed to color vertices of the graph) is a NP-complete problem. There exist backtracking algorithms which can compute the chromatic number for very small graphs (up to 100 vertices), there are probabilistic algorithms with almost polynomial run time [EL89] and there are genetic, tabu and hybrid algorithms for graph coloring [FF95]. But all these algorithms are unusable for the large conflict graphs which we must construct for type hierarchies. The graphs may have 2000 vertices and 200000 edges (see table 6).

There is, however, a class of very fast heuristic algorithms which give good results on most graphs and are used, for example, in graph coloring register allocators [BCKT93]. These sequential vertex coloring algorithms [MMI72] have a run time which is linear in the number of vertices plus the number of edges in the conflict graph [MB83]. All these algorithms order the vertices according to some predetermined criteria and color the vertices in this order. If no color, out of those used so far, can be reused for the current vertex, the number of colors is increased by one and the vertex is assigned the new color. Otherwise, one of the existing colors, one which does not cause a conflict for the current vertex, is selected.

[MMI72] presents two algorithms which give the best results: *largest degree first* ordering and *smallest degree last* ordering. *Largest degree first* ordering sorts the vertices by the vertex degree (number of edges from the vertex) and starts coloring with the vertex with the largest degree. *Smallest degree last* ordering recursively removes the vertex with the smallest degree together with all its edges from the graph and colors the vertices in reverse order of removal. Often the *smallest degree last* algorithm gives the best results.

Another possibility is to construct a vertex order from the structure of the hierarchy. The simplest order is generated by a top down, depth-first, traversal of the hierarchy. A different order is based on a topologically sorted order. Here, the top down traversal is modified so that it descends to a node N in the hierarchy only if all parents of N have already been visited. This traversal visits the nodes in an order similar to that assumed by Caseau in his algorithm. We will refer to this order as the *Caseau order*. An evaluation of all these algorithms shows that the *smallest degree last* algorithm gives the best results (see section 4 table 4). For many hierarchies, this algorithm finds an optimal result.

There are different strategies for choosing which color to reuse for the current vertex. If the colors are numbered in order of first use, two simple strategies are to use the color with (1) the smallest number or (2) the largest number. Another strategy is to choose the most heavily used color which does not cause a conflict. Table 5 in section 4 shows some results using these strategies. The strategy that selects the most used color weighted by the degree of the node often gave the best results in our experiments. Since there is no consistent winner, a mixed strategy which tries more than one method and then picks the best result might be appropriate.

In [MB83], an improvement to sequential vertex coloring is presented. If there is no unused color available, an color exchange is tried. First all conflicting colors are collected which conflict only once with the vertex to color. Then there is a search for a vertex which is not in conflict with one of these collected vertices and the new vertex. If such a vertex can be found, the colors can be exchanged and the new vertex can be colored. Unfortunately, we found that this color exchange strategy fails with the conflict graphs constructed for our type hierarchies. Our graphs tend to have so many edges that there are no nodes which can be exchanged. We assume the reason is that nodes near the top of the hierarchy conflict with nearly all nodes.

3.3 Splitting and balancing the hierarchy

Caseau noted in [Cas93] that the number of bits needed for hierarchical encoding is greatly influenced by the number of children at a node. If a node has k children, then k distinct genes are immediately needed to distinguish these children. To reduce this number when k is large, we can either use more than one gene to distinguish the different children or we can split the children into smaller groups by adding additional nodes to the hierarchy. Using more bits to identify a type complicates the algorithm and makes it difficult to find a near optimal solution. Therefore, whenever a node had more than 8 children, Caseau split them into two groups and introduced two additional nodes as parents for those groups. Repeatedly applying this technique reduces the total number of genes needed, but it far from being an optimal strategy.

We also use the idea of splitting children into groups but we attempt to balance the hierarchy when inserting new nodes. A lower bound on the number of genes needed for hierarchical encoding may be constructed as maximum over all weighted path lengths from the root node to a leaf node. The path length for a leaf node is

$$\sum |children(N)|$$

where $children(N)$ is the set of child nodes for node N, and the sum is made over all nodes N on the longest path from the root node to the leaf node. Only child nodes which need a gene are counted for the path length. For hierarchies which are trees, the largest path length also provides the optimal solution. An optimal solution for the hierarchical encoding of trees can be constructed by splitting children lists and generating a balanced binary tree which minimizes the path length. A bottom-up algorithm can be used to balance the tree. The example in figure 5 shows the number of genes needed being reduced from 5 to 4 by balancing.

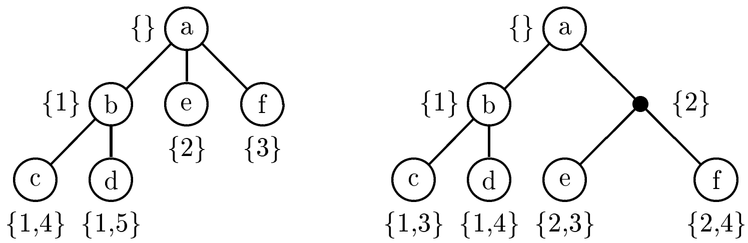

Fig. 5. Balancing a tree

An optimal balancing algorithm appears to be feasible only for tree-structured hierarchies. With multiple subtyping, the hierarchy has to be balanced to generate the minimal chromatic number for its conflict graph. Since computing the minimal chromatic number is NP-complete, the balancing problem is very likely to be NP-complete too. We therefore looked for a heuristic solution. In practice, most multiple subtyping hierarchies deviate only slightly from a tree structure. Heuristics based on the tree balancing method work satisfactorily when taking into account the characteristics of multiple inheritance hierarchies.

If we are balancing a tree, splitting the children into two groups can be performed arbitrarily. In the multiple subtyping case, children which share some common descendants should be assigned to the same group. If we did not do that, coloring is made harder because these common descendants would gain an additional parent node.

The splitting process is faster if it is performed in two stages. A 'presplitting' pass repeatedly performs a heuristic split into two groups and adds two parent nodes until the groups are smaller than a certain limit (currently 14 nodes) using precomputed path lengths. The second pass recomputes the path lengths after every split and does a more complicated split inserting one or two nodes.

The presplitting pass computes an optimistic path length for every leaf node. These optimistic path lengths are computed assuming fewer than three children per node. It is assumed that the hierarchy can be balanced without introducing nodes on the critical path. A leaf node's path length is propagated together with an unique number to all ancestors of the leaf node. During the propagation, larger path lengths overwrite smaller ones. Furthermore the set of all descendants of a node are computed as a bit vector. Using these sets, children which are detected to have overlapping descendant sets are placed in the same bucket. All children lists are sorted according to three criteria. The primary criterion is by bucket, the secondary criterion is by leaf nodes, and the third by the size of the path length. Then every list of children which is longer than the limit is split into two parts so that the lengths of both lists are smaller than the largest power of two which is smaller than the original length of the list.

The second splitting pass precomputes the correct path length after every split, and uses the sum of all children which need a gene on the path from the root to a leaf. The leaf's path length is again propagated to all ancestors. Then the ancestors of the leaf node with the largest path length are checked for a children list to split. This splitting takes care that ancestors of the leaf node are in the same list after splitting. The path lengths of the nodes are also taken into account and, depending on the circumstances, either one or two new nodes are inserted.

3.4 Space and time complexity

A careful implementation of the algorithm needs 19 milliseconds for the smallest hierarchy and 2 seconds for the largest hierarchy when encoding the hierarchy on an Alpha workstation with a 500MHz 21164a processor. The worst-case time complexity of the algorithm is quadratic. The average complexity is lower and depends on the number of edges in the conflict graph. The marking part is quadratic in the number of nodes that have more than one parent (i.e. the size of *multis*). Each splitting step during balancing is linear in the number of nodes, but since the number of nodes can be doubled this also implies quadratic complexity. Coloring is linear in the sum of nodes and edges in the conflict graph

[MB83]. The number of edges is limited by the number of nodes squared, but usually is about twice as large as the average number of ancestors times the number of nodes. Table 1 shows the proportion of the total run time spent on each of the algorithm's subtasks (encoding the Geode hierarchy).

input management	marking	splitting		conflict graph	graph coloring
		pre	final		
6.6%	6.2%	3.1%	58.4%	21.3%	4.4%

Table 1. Execution profile of the encoding algorithm

The space cost is dominated by the storage needed for the conflict graph. The graph is stored in two representations. One is a bit vector to provide a fast check to see if a conflict has already been entered in the graph. The second is a list representation that allows fast sequential access to conflicting nodes. If space is a concern, computation time can be traded for space. It is not necessary to store the conflict graph – it can be computed twice. Initially, only the degree for each node is stored, and then the nodes are sorted according to decreasing degree. Subsequently, the conflicts are computed for each node and immediately colored. This increases the time, but reduces space requirements.

3.5 Incremental algorithm

The algorithm as presented above is not suited for incremental computation of the encoding bit vector. But if slightly worse encodings are accepted, it can be modified for incremental computation. An incremental algorithm can only be implemented in a top down manner where all super types of an added type have to belong to the hierarchy already. The main difficulties are that the size of the encoding could grow from one machine word to two (or from two words to three, and so on), as well as the space consumption and execution time consumption caused by a recomputation of the encoding, if the balancing or encoding changes.

The problem caused by increasing the number of machine words can be solved by linking at run time with different type checking subroutines which work for one, two, three or more machine words.

The current algorithm stores the complete bit matrix for fast computation of type inclusion tests. Additionally, ancestors sets and descendants sets are stored for faster determination of which nodes need a gene and for faster balancing. In an incremental algorithm, fast type inclusion can be performed using the bit vector encodings. Also the test whether a type with more than one super type needs its own gene can be performed using bit vector encodings instead of ancestor sets. Balancing could be replaced by a simpler splitting process which ignores the depth of the tree. Coloring could be carried out using an algorithm similar to the one proposed by Caseau.

4 Results

This last section evaluates different aspects of the algorithm and compares the performance of the algorithm with other approaches. As test data, we used a collection of class libraries compiled by Karel Driesen. We also obtained the Laure type hierarchy from Yves Caseau [Cas93] and the Java API library from Sun [GYT96]. Table 2 presents the relevant characteristics of those libraries. The number of classes varies from 225 to 1956, representing both big applications and libraries. The depth of the hierarchy ranges from 7 to 18. The first four libraries use single inheritance only; the others use multiple inheritance with up to 16 parents per class. Except for the three programs written in LOV (a language similar to Eiffel), the average number of parents is close to one. For the three LOV programs the average number of parents is close to two.

library name	language	classes	depth	max parents	avg. parents
Visualworks2	Smalltalk-80	1956	15	1	1
digitalk3	Smalltalk-80	1357	14	1	1
NeXTStep	Objective-C	311	8	1	1
ET++	C++	371	9	1	1
Unidraw	C++	614	10	2	1.01
Self	Self	1802	18	9	1.05
Geode	LOV(Eiffel)	1319	14	16	1.89
Ed	LOV(Eiffel)	434	11	7	1.66
LOV	LOV(Eiffel)	436	10	10	1.71
Laure	Laure	295	12	3	1.07
Java	Java	225	7	3	1.04

Table 2. Hierarchy characteristics

Table 3 shows the main result, the number of bits needed for the encoding using three different splitting strategies combined with two different coloring strategies. The first two columns show the number of genes needed for encoding the original hierarchy. The next two columns show the genes needed for a hierarchy where all classes with more than 8 children have been replaced by a class that has two new classes as children, each having one half of the children of the original class. The last two columns show the results for a balanced hierarchy using the balancing algorithm described in the previous section. The two sequential coloring techniques use an ordering similar to that used by Caseau (top down after all parents of a class have been colored) and the smallest degree last ordering. Note that Caseau's algorithm cannot directly encode all our hierarchies because it requires every hierarchy to be a lattice; we only color the classes in a sequence which is similar to the ordering of his algorithm. To compare Caseau's results with ours, it is necessary to compare the column *Caseau* of *max 8 children* with the last column. Our algorithm can reduce the sizes of the encodings to one quarter of those produced by Caseau's algorithm.

	original hierarchy		max 8 children		balanced hierarchy	
benchmark	Caseau	smallest last	Caseau	smallest last	Caseau	smallest last
Visualworks2	420	420	124	124	50	50
digitalk3	325	325	116	116	36	36
NeXTStep	177	177	92	92	23	23
ET++	181	181	61	61	30	30
Unidraw	227	227	96	96	30	30
Self	297	297	180	180	55	53
Geode	404	403	231	228	110	95
Ed	128	126	90	80	62	54
LOV	130	127	92	86	68	57
Laure	34	33	34	33	23	23
Java	97	97	50	50	22	19

Table 3. Bit count of Caseau and near optimal coloring for different balanced hierarchies

Table 4 gives the performance using six different sequential coloring techniques. The first column (smallest first) is the worst ordering; it starts with the class which has the smallest degree (the smallest number of conflicting classes). Random ordering takes the classes in the order they are read in. Top down ordering traverses the hierarchy in a depth first manner from the root node down to the leaf nodes. The Caseau ordering also traverses the hierarchy top down, but it colors a class only after all parent classes have been colored. Largest degree first and smallest degree last are the orderings suggested by Matula [MMI72] and give the best results for our conflict graphs. The *lower bound* column gives an estimate for the lower bound using the largest path length as described in the previous section. This estimate is quite accurate for tree-like hierarchies but is only approximate for other hierarchies. In many cases, coloring needs the same number of colors as estimated by the lower bound and this shows that an optimal solution has been found. It is evident that conflict graphs resulting from single inheritance hierarchies can be colored optimally regardless of the algorithm used.

The quality of a sequential coloring algorithm not only depends on the or-

benchmark	smallest first	random	top down	Caseau	largest first	smallest last	lower bound
Visualworks2	50	50	50	50	50	50	50
digitalk3	36	36	36	36	36	36	36
NeXTStep	23	23	23	23	23	23	23
ET++	30	30	30	30	30	30	30
Unidraw	30	30	30	30	30	30	30
Self	60	57	56	55	53	53	47
Geode	140	122	120	110	99	95	42
Ed	84	72	68	62	57	54	30
LOV	86	73	79	68	59	57	31
Laure	24	25	23	23	23	23	23
Java	22	22	22	22	19	19	19

Table 4. Bit count of different coloring techniques

dering of the vertices but also on the color chosen if there is a choice of more than one non-conflicting color to reuse. The *last use coloring method* sorts the colors by their last uses and takes the first used color which does not conflict. The *largest coloring method* selects the color with the largest number while the *smallest coloring method* selects the color with the smallest number. The best color selection algorithms are based on an assumption that preferring a color which is heavily used should produce fewer conflicts later on. The *max use coloring method* counts the number of uses of each color and takes the most used one. The last two algorithms weight the use by the degree of the class. The *max sdl coloring method* weights the use count by the removal degree obtained by the *smallest degree last* ordering, and the *max ldf coloring method* weights the use count by the unmodified degree. The *smallest* coloring method and the three *max use* methods sometimes give different best results. Because the computation time for a coloring is small compared to the time needed to construct the conflict graph, it makes sense to try all four algorithms and take the best result.

Table 6 gives more data on the characteristics of the different type hierarchies with respect to the algorithm. It is evident that in most hierarchies the number of types which need their own gene is small compared to the number of types with multiple super types. The only exceptions are the three LOV hierarchies, where half the types need their own gene. The column *balancing nodes* shows also that the most added balancing nodes were needed for the Geode hierarchy. The number of conflict edges increases if there is a higher use of multiple inheritance. Computations of the encodings have been performed on an Alpha workstation with a 500MHz 21164a processor. All computation times are in milliseconds.

We compared the size of the tables resulting from a bit matrix representation of the transitive closure of the subtype relation with our encoding (table 7). The

benchmark	last use color	largest color	smallest color	max use color	max sdl color	max ldf color
Visualworks2	50	50	50	50	50	50
digitalk3	36	36	36	36	36	36
NeXTStep	23	23	23	23	23	23
ET++	30	30	30	30	30	30
Unidraw	30	30	30	30	30	30
Self	54	53	54	54	53	54
Geode	97	97	95	95	97	95
Ed	56	56	55	56	56	54
LOV	60	62	59	61	62	57
Laure	23	23	23	23	23	23
Java	19	19	19	19	19	19

Table 5. Bit count of different color choosing techniques

benchmark	type number	singles	multis	need gene	balancing nodes	conflict nodes	conflict edges	computaion time (ms)
Visualworks2	1965	1965	0	0	388	2353	62394	890
digitalk3	1357	1357	0	0	298	1655	37871	426
NeXTStep	311	311	0	0	103	414	6141	30
ET++	371	371	0	0	94	465	7997	39
Unidraw	614	604	10	4	164	772	13541	93
Self	1802	1741	61	22	465	2228	113489	1367
Geode	1319	614	705	384	796	1794	149052	1902
Ed	434	272	162	68	198	538	26885	136
LOV	436	271	165	70	217	558	30428	168
Laure	295	275	20	0	29	304	4823	21
Java	225	216	9	1	63	280	3509	19

Table 6. Complexity data of hierarchies

size of the table can be reduced by a factor of up to 31 for the test hierarchies. The size of the bit matrix encoding increases by n^2 with the number of types. The size of the bit vector encoding (for a hierarchy which is a balanced binary tree) encreases by $2 * n \log n$. If the multiple inheritance portion is low, our algorithm comes close to the logarithmic size increase.

benchmark	size of bit matrix	size of codes	reduction factor
Visualworks2	485.3	16.0	31
digitalk3	233.4	11.0	21
NeXTStep	12.4	1.2	10
ET++	17.8	1.4	12
Unidraw	49.1	2.4	20
Self	410.8	14.7	28
Geode	221.5	15.9	14
Ed	24.3	3.4	7
LOV	24.4	3.4	7
Laure	11.8	1.1	10
Java	7.2	0.9	8

Table 7. Table sizes and reduction factor

5 Conclusion

We have presented a near optimal algorithm for finding hierarchical encodings for type hierarchies. Our algorithm produces encodings which are up to four times shorter than encodings generated by a previous algorithm (Caseau) and therefore provide a faster type inclusion check for object oriented languages with multiple subtyping. The algorithm is also an order of magnitude faster than the previous algorithm which makes it practical for use in compilers. To evaluate our algorithm, the complete source code can be obtained via world wide web at `http://www.complang.tuwien.ac.at/andi/typecheck/`.

References

[ABLN89] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

[BCKT93] Preston Briggs, Keith Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *ACM Conference on Programming Language Design and Implementation*, pages 275–284, Portland, June 1993. ACM.

[Cas93] Yves Caseau. Efficient handling of multiple inheritance hierarchies. In *Conference on Object Oriented Programming Systems, Languages & Applications*, pages 271–287, Washington, October 1993. ACM.

[Coh91] Norman H. Cohen. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4):626–629, 1991.

[DDH84] Peter Dencker, Karl Dürre, and Johannes Heuft. Optimization of parser tables for portable compilers. *ACM Transactions on Programming Languages and Systems*, 6(6):546–572, 1984.

[EL89] J. A. Ellis and P. M. Lepolesa. A Las Vegas graph coloring algorithm. *The Computer Journal*, 32(5):474–476, 1989.

[FF95] Charles Fleurent and Jacques A. Ferland. Genetic and hybrid algorithms for graph coloring. *Annals of Operations Research*, page to appear, 1995.

[GYT96] James Gosling, Frank Yellin, and The Java Team. *The Java Application Programming Interface*. Addison-Weley, 1996.

[HN94] Michel Habib and Lhouari Nourine. Bit-vector encoding for partially ordered sets. In *ORDAL'94*, LNCS 831, pages 1–12. Springer, 1994.

[HN96] Michel Habib and Lhouari Nourine. Tree structure for distributive lattices and its applications. *Theoretical Computer Science*, 165:391–405, 1996.

[MB83] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 30(3):417–427, July 1983.

[MMI72] David W. Matula, George Marble, and Joel D. Isaacson. Graph coloring algorithms. In R. C. Read, editor, *Graph Theory and Computing*, pages 109–122. Academic Press, 1972.

[Wir88] Niklaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, 1988.

Appendix: the encoding algorithm

// definitions

$parents(x)$	// all nodes which are a direct supertype of x
$children(x)$	// all nodes which are a direct subtype of x
$ancestors(x)$	// all nodes which are a supertype of x
$descendants(x)$	// all nodes which are a subtype of x
$mark(x)$	// flag, is 1, if x need a distinguishing gene, 0 otherwise
$length(x)$	// longest path length between x and a leaf node
$leaf(x)$	// leaf node of the longest path which includes x
$gene(x)$	// gene number, bit position in bit vector
$code(x)$	// the bit vector of class x
$singles$	// all nodes in the hierarchy with a single parent
$multis$	// all nodes with more than one parent
$needgenes$	// all nodes which need a gene

// mark all nodes of hierarchy which need a bit
$mark(s) := 1 \; \forall s \in singles$
$needgenes := singles$
for each $m \in multis$ **do**
    if $\exists n \in multis, \neg(n <: m), ancestors(m) \cap needgenes \subseteq ancestors(n)$
    then $mark(m) := 1, needgenes := needgenes \cup \{m\}$
    else $mark(m) := 0$

```
// balance the hierarchy
define compute_length(l ∈ Integer, leaf ∈ hierarchy, x ∈ hierarchy) as
    l := l + ∑ mark(childx), ∀childx ∈ children(x)
    for each parentx ∈ parents(x) do
        if length(parentx) < l then
            length(parentx) := l
            leaf(parentx) := leaf
            compute_length(l, leaf, parentx)
length(x) := −1 ∀x ∈ hierarchy
for each leaf ∈ hierarchy, children(leaf) = {} do
    length(leaf) := 0
    leaf(leaf) := leaf
    compute_length(0, leaf, leaf)
for each x ∈ hierarchy, size(children(x)) > 2 do
    split children(x) and add one or two nodes to hierarchy
    if this is possible without increasing length(y) for any y ∈ hierarchy

// compute conflict graph
for each x ∈ hierarchy do
    parx := parents(x)
    if parx = {} then parx := {x}
    for each y ∈ descendants(p), y ≠ x, ∀p ∈ parx do
        enter conflict between x and y in conflict graph
        if y ∈ multis, ¬(y <: x) then
            ∀anc ∈ ancestors(y), anc ≠ y, enter conflict between
                x and anc in conflict graph

// color the conflict graph
for each x ∈ hierarchy in smallest degree last order do
    if mark(x) = 1 then gene(x) := the most used non conflicting gene

// compute code
for each x ∈ hierarchy do
    code(x) := ∪gene(ancx), ∀ancx ∈ ancestors(x)
```

This article was processed using the LaTeX macro package with LLNCS style