Parallel Processing for Artificial Intelligence 1
L.N. Kanal, V. Kumar, H. Kitano and C.B. Suttner (Editors)
© 1994 Elsevier Science B.V. All rights reserved.

Chapter 5

Inheritance Operations in Massively Parallel Knowledge Representation

James Geller

Institute for Integrated Systems, CIS Department, New Jersey Institute of Technology, Newark, NJ 07102

This chapter elaborates on an approach to knowledge representation that combines the use of a limited inference strategy with massive parallelism. Conceptually, a class hierarchy is used. The nodes in this hierarchy are augmented by a preorder numbering scheme. The augmented hierarchy is then transformed into a pointer-free *linear tree representation.* This tree representation is implemented on a CM-2 Connection Machine, such that every node resides on its own processor. This chapter discusses in detail an algorithm for inheritance in the linear tree representation. It also introduces an algorithm for *upward-inductive inheritance* for the linear tree representation. Experimental data from a Connection Machine CM-2 implementation show that for a given machine size downward inheritance can be performed in constant time. The height of the class tree has no influence on the run time. Upward-inductive inheritance run times grow very moderately with the number of nodes in the tree.[1]

1. Massively Parallel Knowledge Representation

The development of the Connection Machine and of Massively Parallel Computing in general was heavily influenced by problems from Artificial Intelligence. Hillis [1] used examples from Computer Vision and Knowledge Representation to motivate its design. One of these examples was Fahlman's [2] famous NETL system. NETL was the first attempt in the history of Artificial Intelligence to create a Knowledge Representation model

[1]This work was partially supported by the National Science Foundation, Grant #IRI-9204655

This work was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by and operates under contract to DARPA and the Air Force Systems Command, Rome Air Development Center (RADC), Griffiss Air Force Base, NY, under contract# F306002-88-C-0031.

that could be translated naturally into hardware.

Knowledge Representation is the heart of many areas of Artificial Intelligence. Unfortunately, Knowledge Representation implementations are also notorious for being slow. Shastri [3, 4] has pointed out that "To be deemed intelligent, a system must be capable of action within a specified time frame..." [3] (p. 3). We concur with his opinion that "A possible solution of the computational effectiveness problem lies in a synthesis of the limited inference approach and massive parallelism." [3] (p. 4).

Waltz [5] has pointed out that AI has made little use of massively parallel processing. Given the history of the Connection Machine this is a rather surprising fact. However, there are a small number of researchers that are working on what we refer to as Massively Parallel Knowledge Representation (MaPKR, pronounced "mapcar"). Evett, Hendler, and Spector [6, 7] have been working on PARKA, an implementation of a Knowledge Representation system on the Connection Machine. Work on massively parallel inference has been reported, e.g., by Kurfess [8].

Geller [9] and Geller and Du [10] have presented an alternative approach to parallelizing the operations in an IS-A hierarchy, which is the backbone of many Knowledge Representation systems. In this paper we first review the basics of this approach and show an implementation of it. Then we discuss two types of inheritance operations, standard (downward) inheritance and upward-inductive inheritance. It will be shown that in this implementation, for a given machine size, downward inheritance in a tree is a constant time operation, even if the height of the tree is varied.

Section 2 discusses an encoding of IS-A hierarchies which permits the fast evaluation of IS-A queries. Section 3 introduces a *linear tree representation* of this encoding. Section 4 explains the parallel algorithm for updating the linear tree representation efficiently. Sections 5 – 8 discuss parallel downward inheritance and parallel inductive-upward inheritance. Section 9 reports experimental results, and Section 10 contains our conclusions. This chapter is a considerably extended and revised version of [11] which appeared in [12].

2. Schubert's Tree Encoding of IS-A Hierarchies

Most Knowledge Representation systems, as well as all object-oriented languages, databases, and systems, use a so called IS-A hierarchy or class hierarchy as their backbone. In the simplest possible case this hierarchy is a tree. It consists of nodes, which stand for classes, and connecting arcs, which stand for the IS-A relation. In fig. 1 an example of such an IS-A hierarchy is shown. One of the nodes in this tree has the label Bird, which means that it stands for the class of all Birds. Below the Bird node there

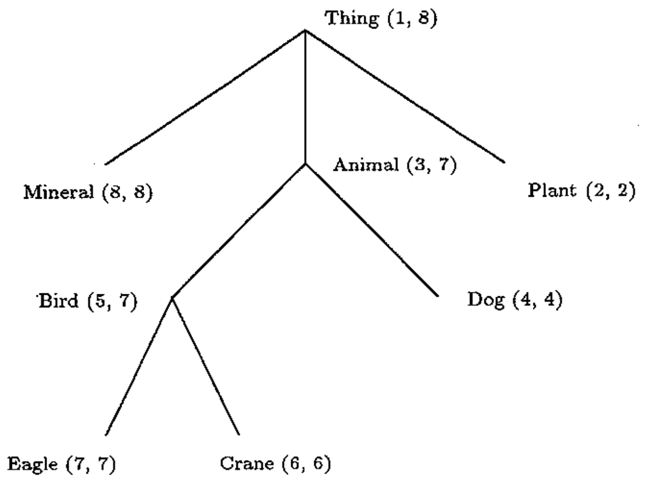

Fig. 1. A Class Hierarchy with Schubert's Numbering

are two nodes, the Eagle node and the Crane node. These two nodes are connected to the Bird node by two arcs. Therefore, every member of the class Eagle is a member of the class Bird, and every member of the class Crane is also a member of the class Bird.

The IS-A relation is transitive. The node Bird is itself connected by one arc to the node Animal, which means that every bird is an animal. Due to the transitivity of the IS-A relation, every eagle is also an animal, etc. One method of reasoning which is commonly associated with IS-A hierarchies is inheritance. It is assumed that each node in the hierarchy has certain attributes associated with it. If an attribute is associated with a node A and the user wants to know whether a node B has this attribute, then it is possible to answer this question in the positive, if A is above B in the hierarchy and if there exists a path of IS-A arcs between A and B.

While it is possible to follow a chain of pointers to verify the existence of a subclass relation, this becomes inefficient for large class hierarchies. To overcome this efficiency problem, Schubert [13, 14] introduced a special purpose class reasoner. Whenever necessary, this class reasoner is invoked by a general purpose resolution theorem prover. The two together function as a hybrid reasoning system [15–17]. In order to achieve the necessary speed of the special purpose class reasoner, Schubert used a coding scheme that can be applied to the nodes of any IS-A hierarchy of mutually exclusive classes.

In fig. 1 every node is followed by a pair of numbers. The first number is the result of a preorder right-to-left tree traversal of the class hierarchy.

In other words, the nodes are numbered according to a depth first, right-to-left search. Such a search would visit the nodes in fig. 1 in the order Thing, Plant, Animal, Dog, Bird, Crane, Eagle, Mineral. Hereafter, we will call this first number the *preorder number* of a node.

The second number of every node is the largest preorder number that occurs under this node in the hierarchy. For example, under the node Animal the numbers (4, 5, 6, 7) occur as preorder numbers of the nodes (Dog, Bird, Eagle, Crane) respectively. Because the largest of these is 7, the second number of Animal is 7, too.

Leaf nodes have no nodes under them. However, if we define every node to be under itself, then we can maintain the above rule for selecting the second number of a pair. A leaf node is assigned the largest first number of any node under it. Because it has only itself under it, its second number is identical to its first number. In the balance of this paper we will call the second number the *maximum number* of a node.

The decision whether a node B is under a node A can then be made very easily by comparing the number pair assigned to B with the number pair assigned to A. If and only if B is a subclass of A, then the number pair assigned to B is a subinterval of the number pair assigned to A. In our example, Bird is a subclass of Animal because (5, 7) is a subinterval of (3, 7). On the other hand, Bird is not a Plant because (5, 7) is not a subinterval of (2, 2). By representing the nodes and their associated number pairs in a hash table, it can be rapidly decided whether a subclass relation exists.

Schubert's technique can be extended from trees to directed acyclic graphs, but this requires the assignment of more than one number pair to some of the nodes. Agrawal, Borgida and Jagadish [18] have presented methods to minimize the number of additional pairs needed. In the AI literature interest in such transitive closure techniques has been limited to the IS-A relationship. This is not the case in the database literature where similar techniques have been used for recursive query evaluation [18, 19]. Efficient compile-time techniques for an IS-A hierarchy encoding have also been introduced in the theory of object-oriented languages [20]. We feel that all these areas can benefit from the techniques developed in this chapter.

3. How to Achieve the Same Effect Without Trees

Schubert's method for the representation of class hierarchies has proven to be efficient for subclass verification. However, any attempt to update the tree requires the recomputation of the number pairs of many of the nodes. This difficulty can be overcome if one makes use of the following two observations.

(i) The number pairs actually make the tree redundant. Instead of a tree one can use a list of nodes with number pairs associated.
(ii) It is possible to update all the number pairs in parallel, making the Connection Machine a viable tool for this problem.

We will now explain the intuitions behind (i) and the simpler of two algorithms for (ii). Detailed proofs of the viability of this approach can be found in [10].

The list representation is obtained from the tree representation by a left-to-right preorder traversal. For our example in fig. 1 this would result in the list (Thing (1, 8) Mineral (8, 8) Animal (3, 7) Bird (5, 7) Eagle (7, 7) Crane (6, 6) Dog (4, 4) Plant (2, 2)).

The verification of the subclass relation between any two nodes is not affected by the use of the linear tree representation. As before, two hashing operations and a comparison of two intervals are needed. The "only" additional requirement is an update operation for adding a new node to this list. This update operation should have the same effect as if we would have inserted that new node into the tree and would have recreated the linear representation by a left-to-right preorder traversal.

It is possible to find such an update operation if one assumes, without loss of generality, that nodes in the tree are always inserted at the leftmost possible position. Remember that the order of siblings at any one level in the class tree did not contain any information. Therefore, our restriction is feasible. With this assumption we can insert this new node into the list immediately after its parent, and we get the required effect.

In other words, inserting a node at the leftmost position under a parent node in a tree and transforming the tree into a list (as described above) results in the same list as inserting the node immediately after its parent in the list. For example, assume that the node Robin is to be inserted under Bird in fig. 1. Without loss of generality we may add Robin to the left of Eagle. A new left-to-right preorder traversal would then result in the list (Thing (1, 8) Mineral (8, 8) Animal (3, 7) Bird (5, 7) Robin () Eagle (7, 7) Crane (6, 6) Dog (4, 4) Plant (2, 2)). This result is identical to just inserting Robin into the list immediately after the Bird node.

The next step is to update the number pairs of all the nodes and to assign a number pair to the node Robin itself. Figure 2 shows that when one inserts a node into a tree as a new leaf, this operation divides the tree into three parts. The most important part is a path of IS-A arcs that leads from the root node to the newly inserted node. This path cuts the tree into a left part and a right part. There are three simple (but different) rules for updating the number pairs of the nodes residing in these three parts.

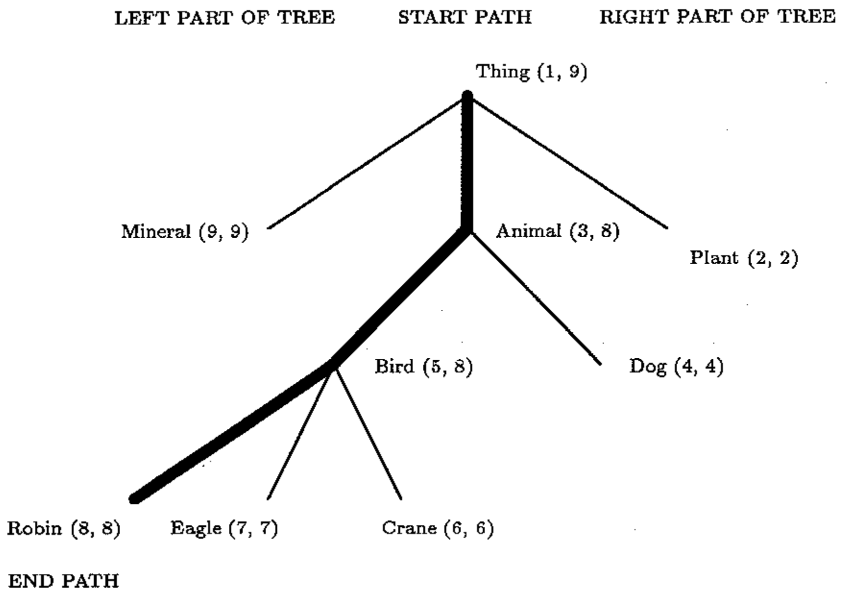

Fig. 2. Updating the three Parts of a Class Tree

All the nodes in the right part have unchanged number pairs. The reason for this is that they will all be visited by the numbering operation (a right-to-left traversal) before the new node is encountered.

All the nodes in the path will have their preorder number unchanged, because they will be visited before the new node is encountered. They will, however, have their maximum number incremented (by 1) because they have the new node under themselves.

All the nodes in the left part will have both numbers incremented. The preorder number is incremented because the right-to-left traversal will now be delayed by one position. The maximum number will be incremented because the node providing this maximum number is guaranteed to have its preorder number incremented by 1. (Recall that we define a node as being under itself. A proof for these update rules is given in [10].)

One final question remains in order to decide whether the suggested list representation can actually be used, namely whether the update rules can be translated into similar simple rules that are valid for the list representation of such a class hierarchy. The answer is yes. It is easy to see that the nodes in the right subtree will be exactly the nodes that occur in the list *after* the newly inserted node. Therefore, a serial algorithm can simply stop processing the list after it has inserted the new node and updated all the number pairs up to the new node.

However, the nodes of the left part and the path are intermixed in the list before the newly inserted node. Is there a simple way to decide whether a node in this list segment belongs to the path or the left part of the tree? The answer is yes, again. All nodes in the path will have a preorder number which is smaller than the preorder number of the parent node of the newly inserted node. All nodes in the left part will have a preorder number which is larger than the preorder number of the parent node.

In summary, the linear representation has three corresponding update rules which maintain the numbering of the tree correctly. Returning to our example, we see that the nodes Plant, Dog, Crane, and Eagle do not require any changes to their number pairs. The nodes Thing, Animal, and Bird belong to the path and therefore have their maximum numbers incremented. The node Mineral is the only node of the left part and has both its numbers incremented. The resulting list representation is therefore (Thing (1, 9) Mineral (9, 9) Animal (3, 8) Bird (5, 8) Robin () Eagle (7, 7) Crane (6, 6) Dog (4, 4) Plant (2, 2)).

The new node Robin is a leaf, and as such has a pair of identical numbers. It comes last in the right-to-left traversal of the subtree of its parent, therefore its preorder number has to be the largest preorder number under its parent. Because of that, the preorder number of Robin has to be identical to the maximum number of Bird which has already been updated. This concludes the update operation, giving (Thing (1, 9) Mineral (9, 9) Animal (3, 8) Bird (5, 8) Robin (8, 8) Eagle (7, 7) Crane (6, 6) Dog (4, 4) Plant (2, 2)). The reader is invited to compute the new list representation directly from the updated tree, and should end up with the same result.

Formally speaking, the insert operation is a mapping from a triple into a sequence of three dimensional vectors. The elements of the triple are an old sequence of three dimensional vectors, a parent index into that sequence, and a class name for a new child node. The indexed symbol c_i stands for the *ith* class name. Specifically, c_c is the child class to be inserted into the class tree. The symbol π_i is used for the *ith* preorder number, and the symbol μ_i is used for the *ith* maximum number. The length of the sequence before inserting the new child node is n. The upper index m describes values before the insert operation, and the upper index m+1 describes the same values after the insert operation.

$$\text{INSERT:} \begin{pmatrix} c_k \\ \pi_k \\ \mu_k \end{pmatrix} \times p \times c_c \rightarrow \begin{pmatrix} c_j \\ \pi_j \\ \mu_j \end{pmatrix}$$

$1 \leq k \leq n$, $1 \leq j \leq n{+}1$, and $1 \leq p \leq n$.

$$c_j^{m+1} = \begin{cases} c_{j-1}^m & n+1 \geq j > p+1 \\ c_c & j = p+1 \\ c_j^m & \text{otherwise} \end{cases}$$

$$\pi_j^{m+1} = \begin{cases} \pi_{j-1}^m & n+1 \geq j > p+1 \\ \pi_j^m + 1 & \pi_j > \pi_p \\ \mu_p^m + 1 & j = p+1 \\ \pi_j^m & \text{otherwise} \end{cases}$$

$$\mu_j^{m+1} = \begin{cases} \mu_{j-1}^m & n+1 \geq j > p+1 \\ \mu_p^m + 1 & j = p+1 \\ \mu_j^m + 1 & \text{otherwise} \end{cases}$$

Note that in the above, the term "–1" always appears in the index expression, while the term "+1" appears after the indexed expression. Note also that the update of π_{p+1}^{m+1} requires the use of μ_p^m. Therefore, in this formalization, the update of μ_j has to happen strictly after the update of π_j.

4. Parallelizing the Update Algorithm

A basic familiarity with the principles of massively parallel computation is assumed. However, a few comments about programming the Connection Machine are necessary. The Connection Machine [21, 22] is programmed in *LISP, a dialect of Common LISP, by manipulating parallel variables (*pvar*). A *pvar* is a variable with one name that exists on every single processor and that is created with one single declaration.

In semantic network implementations on massively parallel hardware, every node in a concept network is assigned to one processor (e.g., [1]). Attributes are usually represented as *pvars* and attribute values as values of these *pvars*. As opposed to work in connectionism on distributed representations, this is a localized representation of attributes which we share with Evett *et al.* [6].

The most important feature of the previously described update algorithm is that the computation of all the new number pairs does not contain any mutual dependencies. In other words, all these computations may be performed in parallel. For this purpose, we assign every node to one processor on the Connection Machine. The *pvar* SELF-ADDRESS!! contains on every processor the linear position number of that processor. The *pvar* PRENUM!! contains the preorder number of every number pair, while the *pvar* MAXNUM!! contains the maximum number of every number pair. (The suffix !! is a mnemonic indicating that the variables in question are *pvars.*)

Adding a node requires that all the nodes to the right of the parent node should be moved one more position to the right. This can conveniently be done in parallel. However, the heart of the updating algorithm is the following piece of code which takes care of updating the number pairs. The notation used should be self-explanatory.

```
UPDATE (p: PROCESSOR-NUMBER)
   ACTIVATE-PROCESSORS-WITH
      (SELF-ADDRESS!! >!! 0!!)  AND!!
      (SELF-ADDRESS!! <=!! (!! p))
   DO BEGIN
      MAXNUM!! :=!! 1!! +!! MAXNUM!!
      IF!! PRENUM!!  >!! (!! PRENUM(Np)) THEN!!
           PRENUM!! :=!! 1!! +!! PRENUM!!
   END
```

In this algorithm Np stands for the parent node, and p for the position number of the processor containing the parent node. PRENUM(Np) is a (serial) function that returns the PRENUM of a node.

In practice, the implementation is considerably more complicated. It is possible that a whole subtree has to be inserted, not just a single node. It is also possible that the system has to maintain a number of different unconnected trees (a forest) two of which may be connected at any time by a new IS-A assertion. When the latter occurs, the parent node may be in the left or in the right tree, and there may be any number of intermediate trees that are not involved in this update operation.

Tests of this extended algorithm were performed on a CM-2, using half of its 16k processors [10]. Experiments with sets of several thousand nodes show that verification of a subclass relation can be done in practically constant time in under 0.5 seconds. Update times grow very slowly from 0.4 seconds for a problem with 12 IS-A assertions to about 0.9 seconds for a problem with over 4000 IS-A assertions.

5. Inheritance Terminology

Definition 1: An *entry point* in an inheritance problem is a class or an individual for which we want to derive the value of an attribute. This value may or may not be stored directly at the entry point.

Definition 2: A *source point* in an inheritance problem is a class or an individual at which an attribute value is available which can solve the inheritance problem for an entry point, or contribute to its solution.

Definition 3: The *required value* in an inheritance situation is the attribute value whose retrieval is attempted at the entry point.

Definition 4: The *direction of search* in an inheritance problem is defined by the position of the entry point relative to the source point(s) in relation to their common hierarchy. If a search from the entry point to the source point(s) leads upwards in the common hierarchy, then the direction of search is upwards. If the search from the entry point to the source points(s) leads downwards, the direction of search is downwards.

Definition 5: *Downward inheritance* (standard inheritance) is defined as an algorithm that is invoked when a required value is not available at the entry point. The algorithm derives a solution by an upward search.

There is, of course, nothing original about Definition 5, but we present it to show that it is parallel to the following definition.

Definition 6: *Upward-inductive inheritance* is defined as an algorithm that is invoked when a required value is not available at the entry point. The algorithm derives a solution by a downward search.

6. Upward-Inductive Inheritance

Downward inheritance is a well established and accepted technique. The contribution of this paper to the research in downward inheritance consists solely of experimental verification that it can be done in a massively parallel environment in constant time. This assumes a massively parallel computer of constant size that is sufficiently large to represent every concept by one processor. This issue will be discussed in more detail in Section 7 and Section 9.

On the other hand, upward-inductive inheritance is a new technique which needs some justification. A reasoner might be in a situation where he would like to obtain, through inheritance, an attribute value for an instance of a class. However, the class itself and all its superclasses might not have an appropriate default value for the attribute. On the other hand, the reasoner might have previously stored attribute values for a large number of instances of that class. It would be a considerable waste of knowledge if the reasoner did not avail himself of these values. E.g., if a reasoner has seen 50 horses, and 49 had four legs, it would be unintelligent if he would refuse an answer to the question "How many legs does a horse have"? Instead, he should apply upward-inductive inheritance and reply that horses have four legs. In this evidential approach to reasoning we are again influenced by Shastri [3].

Formally, we introduce upward-inductive inheritance as the solution to the following problem:

Preliminary Notation: The letter X, possibly with integer subscripts, represents nodes which may be individuals or classes. Capital Greek letters represent variables ranging over nodes. The letters S and T represent sets of nodes. The capital letter A represents an attribute, and the lower case letter a represents its corresponding value. When we say that an attribute A is *not defined at a node* X, that means that there is no valid value a associated with A.

Situation: Assume a set of n nodes $S = \{X_1, X_2, \ldots, X_n\}$ which are all subclasses of a class X, or individuals belonging to a class X. Assume further a set T of m nodes, such that $T \subset S$, i.e., $m < n$. Assume an attribute A, that is defined for every $\Phi \in T$. However, a_i of Φ_i and a_j Φ_j are not necessarily the same, unless $i = j$. Finally, assume that A is undefined for X and cannot be derived by standard inheritance. In addition, A is undefined for all $\Psi \in (S - T)$.

Problem: Find an algorithm that assigns a correct value a to the attribute A of X. It is interesting also to consider the subproblem where every $X_i \in S$ is an individual node.

This is an induction problem, and induction is generally considered to be an unsolved problem. In this paper we do not claim any research on solving the induction problem in general. However, we claim that a reasoner may find itself in a situation where no answer would be more disastrous than an incorrect answer. If there is any probability for a given answer to be correct, then such an answer is certainly preferred to no answer at all.[2]

7. Downward Inheritance Algorithm

The downward inheritance algorithm is extremely elegant and simple and relies on the fact that in our linear tree representation all the ancestors of a node are to its left. In addition, the number pairs of all ancestors are super-intervals of the number pair of a given node. Therefore, using the Connection Machine technique of first activating all processors, followed by stepwise deactivation of processors that are irrelevant, the algorithm is as follows:

```
DOWNWARD-INHERIT(N: ENTRY-POINT;
                 ATTRIBUTE!!: ATTRIBUTE)
   ACTIVATE-PROCESSORS-WITH
      PRENUM!!    <=!! (!! PRENUM(N))  AND!!
```

[2]This situation occurs for instance at written midterm examinations.

```
    MAXNUM!!    >=!! (!! MAXNUM(N))  AND!!
    ATTRIBUTE!! <>!! (!! -1)
  DO BEGIN
    (RETRIEVE-PVAR ATTRIBUTE!!
                   (*MAX SELF-ADDR!!))
  END
```

We first activate all the processors that have a number pair including the given number pair. These are the possible ancestors. Of these we want only the nodes that have a value for the given attribute. We are marking an undefined attribute in a class by the value −1. Of these, we want the rightmost node, which will be lowest in the hierarchy. This is achieved by finding the node with the highest address. PRENUM(N) and MAXNUM(N) contain the number pair for the entry point. ATTRIBUTE!! has to be replaced by the *pvar* that maintains the required attribute. *MAX finds, in parallel, the largest value of a *pvar.*

Note the elegance of this algorithm. There is no iterated search from the parent to its parent and its ancestors. All ancestors are activated in parallel. The lowest ancestor is found by one single operation. (*MAX itself is of logarithmic complexity. However, this does not change the complexity of retrieval for a constant machine size.)

There is one complication which we have completely solved, but the description of which was omitted in this algorithm. As can be seen in the earlier parts of this paper, an update of the class hierarchy leads to the necessity of moving around whole subtrees. The time required for this operation depends on the number of *pvars* that are needed to describe the state of one node. If the number of *pvars* grows, then the response times would increase dramatically. Unfortunately, we need a large number of *pvars*, one for every attribute.

The solution of this problem consists of maintaining attribute information and class information in separate processors, with one pointer *pvar* leading from "hierarchy space" to "attribute space". While the hierarchical part of the presentation of a class is potentially moved around at every insertion of a new class, the attribute information is never moved. Every processor therefore does double duty. It maintains the hierarchy information of one class, and the attribute information of another class. In exceptional cases, one processor might maintain hierarchy information and attribute information for the same class; however, this is a coincidence which may be destroyed by the next class insertion.

8. Upward-Inductive Inheritance Algorithm

The algorithm for upward-inductive inheritance is somewhat more involved, but the basic idea is similar. We want to retrieve attribute values from all nodes that are under a given node. All the nodes that are under the entry point are activated. Of these nodes we need to select all the nodes that are defined for the given attribute.

However, the similarity ends here. There might be many different attribute values, and it is necessary to loop over them. At this point, the algorithm selects the leftmost active node. It retrieves its attribute value and then deactivates temporarily all the nodes that do *not* have the same value. It counts the number of nodes that have the same attribute value and then permanently deactivates all those nodes. It now repeats the same step, if necessary several times. It finds the leftmost active node again, which will have a different attribute value than before, etc. In the end, it reports every found attribute value together with the number of its occurrences. It is left to the user or the general purpose reasoner calling the class reasoner to decide whether he wants to use the most commonly occurring value, or not.

Clearly, this algorithm has a serial component, and the worst case retrieval time depends on the number of different existing attribute values. This might result in very long response times for some attributes, but for many practically useful attributes, such as color, there is a limited number of possible attribute values.

Our implementation is recursive and sorts the results according to the number of occurrences of each attribute value, but we present an iterative algorithm, analogous to the previous description.

```
UPWARD-INHERIT(N: ENTRY-POINT;
               ATTRIBUTE!!: ATTRIBUTE)
   ACTIVATE-PROCESSORS-WITH
      PRENUM!!    >=!! (!! PRENUM(N))  AND!!
      MAXNUM!!    <=!! (!! MAXNUM(N))  AND!!
      ATTRIBUTE!! <>!! (!! -1)
   DO BEGIN
      WHILE some processors are active DO BEGIN
         ATT := (RETRIEVE-PVAR ATTRIBUTE!!
                               (*MIN SELF-ADDR!!))
         CNT := (*COUNT processors with
                        ATTRIBUTE!! =!! (!! ATT))
         DEACTIVATE processors with
```

Table 1.
Influence of Tree Size on Run Times for Downward Inheritance

# of Sons	Times for Downward Inheritance
1054	0.44
1534	0.46 0.47
2046	0.44 0.46
2558	0.46 0.47 0.48
3006	0.46 0.45 0.47
3582	0.47 0.47 0.46
4030	0.52 0.48 0.47
4350	0.47 0.45 0.45
5070	0.45 0.45 0.47
5630	0.46 0.46 0.48

```
                      ATTRIBUTE!! =!! (!! ATT)
       (PRINT ATT CNT)
    END
END
```

We emphasize that no claim is made that this algorithm correctly solves the induction problem. It computes a reasonable set of values that might be preferred by a reasoner to a complete absence of information. This comes across in the fact that the algorithm reports all found attribute values and leaves the decision how to use them to a general purpose reasoner.

9. Experimental Results

We will now describe a set of six experiments that analyze the temporal behavior of downward inheritance and upward-inductive inheritance. In the first experiment 10 test sets are created. Every one of them consists of a single class tree of the same height (11 levels); however the number of nodes increases from set to set. In every test the root node is assigned an attribute value which is queried at one of the leaf nodes.

The purpose of this experiment is to show that the presented inheritance algorithm is independent of the number of sons (= nodes − 1) in the tree. The first column shows the number of nodes in the tree. The second column shows the run time for the inheritance operation. For most of the experiments several runs were performed, and then all the times are shown in the second column. Times for experiments interrupted by dynamic garbage collections are omitted. All times are in seconds.

Clearly the speed of inheritance is independent of the number of nodes

Table 2.
Influence of Tree Size on Run Times for Upward-Inductive Inheritance

# of Sons	Times for Upward Inheritance
1054	0.43
1534	0.46 0.47 0.46
2046	0.54 0.55 0.51
2558	0.54 0.57
3006	0.59 0.60 0.59
3582	0.68 0.64
4030	0.69 0.69
4350	0.74 0.72 0.72
5070	0.76 0.75
5630	0.83 0.81 0.79

Table 3.
Influence of Tree Height on Run Times for Downward Inheritance

Levels	# of Sons	Times for Downward Inheritance
3	1032	0.47 0.47 0.46
5	1016	0.44 0.45
7	1047	0.47 0.46 0.45
9	1006	0.46 0.45 0.46
11	1026	0.46 0.46 0.46
13	1052	0.47 0.46 0.45
15	1068	0.47 0.48 0.47
17	1102	0.49 0.47 0.48
19	1089	0.46 0.47
21	1131	0.46 0.47 0.46

in the tree. The next experiment uses the same trees, but the operation is inverted. One of the leaf nodes receives an attribute value, and that value is queried at the root node.

As can be seen, the times required for upward-inductive inheritance grow very modestly with the size of the tree.

In the next set of experiments 10 trees of approximately constant number of nodes are created, with each of a different height. The first column contains the number of levels in the tree, the second the number of sons (nodes − 1), and the third the observed runtimes.

It can be clearly seen that the height of the tree has no influence on the runtime of downward inheritance reasoning. The following experiment displays the use of the same set of class trees with the previously described operation of retrieving an attribute value at the root that was set at a leaf.

Table 4.
Influence of Tree Height on Run Times for Upward Inheritance

Levels	# of Sons	Times for Upward Inheritance
3	1032	0.43 0.42 0.42
5	1016	0.41 0.42 0.41
7	1047	0.42 0.42
9	1006	0.42 0.43
11	1026	0.42 0.42 0.42
13	1052	0.44 0.43 0.44
15	1068	0.45 0.42 0.44
17	1102	0.44 0.45 0.45
19	1089	0.45 0.43 0.46
21	1131	0.45 0.44 0.44

Table 5.
Influence of Branching Factor on Run Times for Downward Inheritance

Branching Factor	# of Sons	Times for Downward Inheritance
1	3	0.65 0.60 0.61
2	14	0.62 0.60 0.63
3	39	0.63 0.64 0.53
4	84	0.64 0.63 0.58
5	155	0.60 0.52 0.52
6	258	0.60 0.53 0.52
7	399	0.65 0.51 0.54
8	584	0.54 0.45 0.46
9	819	0.63 0.60 0.61
10	1110	0.61 0.59 0.55

Again, tree height has no influence on the time for this operation. The following set of experiments creates 10 trees with 4 levels each. Every tree has a different branching factor used throughout. The first column shows branching factors, the second the number of sons in the tree, and the third run times. (Important note: these experiments were performed at the Pittsburgh Super Computing Center. Therefore, the times are not identical to the times of the previous experiments which were run at the NPAC Center in Syracuse.)

The times are clearly independent of the branching factor. As before, the same set of trees is now used for upward-inductive inheritance.

Times are growing slightly, as we would expect from our previous results with growing numbers of nodes.

In summary, we have presented three experiments for downward inheritance and three corresponding experiments for upward-inductive inheri-

Table 6.
Influence of Branching Factor on Run Times for Upward Inheritance

Branching Factor	# of Sons	Times for Upward Inheritance
1	3	0.40 0.40 0.41
2	14	0.48 0.52 0.48
3	39	0.49 0.50 0.49
4	84	0.48 0.43 0.45
5	155	0.46 0.46 0.45
6	258	0.45 0.49 0.44
7	399	0.47 0.54 0.56
8	584	0.56 0.49
9	819	0.59 0.59
10	1110	0.52 0.59 0.57

tance. The experiments show clearly that downward inheritance is independent of the number of nodes in the tree, of the height of the tree, and of the branching factor of the tree.

Upward-inductive inheritance shows a very moderate growth of processing time with the number of nodes in the tree; however, it is independent of the height and the branching factor of the tree.

10. Conclusions

We have shown algorithms for standard (downward) inheritance, and for a newly introduced operation that we call upward-inductive inheritance. Both these algorithms are based on a parallel class tree representation that uses a list in combination with a preorder numbering scheme. An implementation of the two algorithms on the CM-2 Connection Machine was described. Every element of the class list is maintained on its own processor; this arrangement permits efficient updates and constant time downward inheritance. A constant machine size is assumed in this context. The constant time results can be observed, even for varying height of the class tree. For upward-inductive inheritance, response times are growing very moderately with the size of the class tree. Our current research deals with extending the described approach to Massively Parallel Knowledge Representation from strict hierarchies (trees) to directed acyclic graphs that permit multiple inheritance [23, 24].

Acknowledgement: Mike Halper has considerably improved the English of this manuscript.

References

[1] D. Hillis, *The Connection Machine.* Cambridge, MA: MIT Press, 1985.

[2] S. Fahlman, *NETL: A System for Representing and Using Real-World Knowledge.* Cambridge, MA: MIT Press, 1979.

[3] L. Shastri, *Semantic Networks: An Evidential Formalization and its Connectionist Realization.* San Mateo, CA: Morgan Kaufmann Publishers, 1988.

[4] L. Shastri, "Default reasoning in semantic networks: A formalization of recognition and inheritance," *Artificial Intelligence*, vol. 39, no. 3, pp. 283–356, 1989.

[5] D. Waltz, "Massively parallel AI," in *Eighth National Conference on Artificial Intelligence*, pp. 1117–1122, San Mateo, CA: Morgan Kaufmann, 1990.

[6] M. Evett, J. Hendler, and L. Spector, "PARKA: Parallel knowledge representation on the Connection Machine," Tech. Rep. UMIACS-TR-90-22, Department of Computer Science, 1990.

[7] M. Evett, L. Spector, and J. Hendler, "Knowledge representation on the Connection Machine," in *Supercomputing '89*, (Reno, Nevada), pp. 283–293, 1989.

[8] F. Kurfess, "Massive parallelism in inference systems," in *Proceedings of the Workshop on Parallel Processing for AI at IJCAI-91*, (Sydney, Australia), 106-109, 1991.

[9] J. Geller, "A theoretical foundation for massively parallel knowledge representation," *Parallel Computing News*, vol. 3, no. 11, pp. 4–8, 1990.

[10] J. Geller and C. Du, "Parallel implementation of a class reasoner," *Journal for Experimental and Theoretical Artificial Intelligence*, vol. 3, pp. 109–127, 1991.

[11] J. Geller, "Upward-inductive inheritance and constant time downward inheritance in massively parallel knowledge representation," in *Proceedings of the Workshop on Parallel Processing for AI at IJCAI-91*, (Sydney, Australia), pp. 63–68, 1991.

[12] L. Kanal and C. Suttner, *Proceedings for Workshop W.1 on Parallel Processing for AI at IJCAI-91.* 1991.

[13] L. Schubert, M. Papalaskaris, and J. Taugher, "Determining type, part, color, and time relationships," *Computer*, vol. 16, no. 10, 1983.

[14] L. Schubert, M. Papalaskaris, and J. Taugher, "Accelerating deductive inference: special methods for taxonomies, colors and times," in *The Knowledge Frontier* (N. Cercone and G. McCalla, eds.), pp. 187–220, New York, NY: Springer Verlag, 1987.

[15] A. Frisch, "The substitutional framework for sorted deduction: fundamental results on hybrid reasoning," *Artificial Intelligence*, vol. 49, no. 1, pp. 161–198, 1991.

[16] A. Frisch and A. Cohn, "Thoughts and afterthoughts on the 1988 workshop on principles of hybrid reasoning," *AI Magazine*, vol. 11, no. 5, pp. 77–83, 1991.

[17] A. Frisch and R. Scherl, "A bibliography on hybrid reasoning," *AI Magazine*,

vol. 11, no. 5, pp. 84–87, 1991.

[18] R. Agrawal, A. Borgida, and H. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, (Portland, OR), pp. 253–262, 1989.

[19] R. Agrawal, S. Dar, and H. Jagadish, "Direct transitive closure algorithms: Design and performance evaluation," *ACM Transactions on Database Systems*, vol. 15, no. 3, pp. 428–458, 1990.

[20] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr, "Efficient implementation of lattice operations," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 115–146, 1989.

[21] Thinking Machines Corporation, **LISP Reference Manual Version 5.0 edition.* Cambridge, MA: Thinking Machines Corporation, 1988.

[22] Thinking Machines Corporation, *The Connection Machine System, *Lisp Dictionary.* 1990.

[23] R. Czech, "Advanced operations on a parallel class reasoner," tech. rep., MS Project, New Jersey Institute of Technology, CIS Dept., 1991.

[24] J. Geller, "Advanced update operations in massively parallel knowledge representation," tech. rep., Research Report CIS-91-28, New Jersey Institute of Technology, CIS Dept., 1991.