EFFICIENCY CONSIDERATIONS ON BUILT-IN TAXONONIC REASONING IN PROLOG

Giorgio Montini
Laboratorio Intelligenza Artificiale
CSI-Piemonte
C.so U.Sovietica 216 - 10134 TORINO (ITALY)

ABSTRACT

The issue of integrating inheritance rules into Prolog has recently received some attention in literature. A Prolog-based interpreter extended with a special, built-in mechanism for handling is_a taxonomies has been built. The motivation for this work stems from two observations: (1) is_a hierarchies are common in many domains, and (2) the issue of representing in Prolog is_a hierarchies, and the inheritance properties related to it, is not so straightforward as it would seem at a first glance, especially whether time and space efficiency are required. The model underlying the proposed extension is shown and compared with relevant literature. We describe how some new capabilities have been added to a standard Prolog interpreter in order to implement the extended interpreter. In particular, the unification algorithm and the management of some data areas have been modified. The space requirements and time performance of the extended interpreter are compared with those of the original standard Prolog interpreter, and the results of a series of tests are discussed.

I INTRODUCTION

Set inclusion and membership are quite general and intuitive relations in many applicative domains. Some special built-in mechanisms for handling them were successfully applied to automatic theorem proving (5,8,14), and ultimately in logic programming (1,2). The concept of property inheritance is the basis of such mechanisms. The programmer defines a property as a feature of every element in a class, instead of defining the property for each single element. Classes are organized in hierarchies, often called is_a taxonomies. The inheritance mechanism makes use of the taxonomies and reconstructs the original property.

At a first glance, it seems that the Prolog resolution rule is suitable for this kind of taxonomic reasoning. A straightforward way of describing taxonomies in Prolog is the following. A class is represented as an one-argument predicate symbol. For each is_a arc (i.e. membership) assert in the database a clause:

```
cla88_name(object_name).
```

where class_name and object_name are the two elements connected by the is_a arc. For each s_s arc (i.e. set inclusion) assert a clause:

```
classjiame(A) :- subclassjiame(A).
```

Clauses for generic properties may refer to taxonomic predicates as follows:

```
generic_property(..,A,..) :- ... class_narae(A),..
```

This approach, called top-down, is quite natural, but not efficient if we want to prove a property for an instantiated object. The reason is that the top-down approach is generative. Prolog searches through the taxonomy until it finds the object, rather than <u>directly</u> access to its class and verify the consistence with the required class. Thus if we are using the top-down approach for this kind of goals, the search time depends on the order in which classes and objects are generated. Moreover, if the goal fails, search time is required in order to scan the whole taxonomy.

As an alternative approach - let's call it bottom-up - we introduce two (or more) clauses of the form:

```
claasjnaae     (classjnaae)     •      -1-
clasa_narae(A) :- superclass_name(A).   -2-
```

with the following (informal) semantics.
-1- class_name is a subclass of itself. -2- class_name is a subclass of every class A provided that 8uperclass_name is a subclass of A. The top class of the hierarchy is only subset of itself:

```
topclass_name(topclass_name).
```

For each object a single clause must be given:

```
objectjnarae(A) :- object_classjiame(A).
```

that is, objectjname is an instantiation of class A, if object^class^name is a subclass of A.

Membership is tested by asking a goal like:

```
?- object_name(te8t_clasB).
```

This representation is efficient for this kind of goal. Unfortunately, generative goals are quite difficult to formulate in the bottom-up approach.

A mixed approach is useful, provided that a technique is employed in order to distinguish the top-down and the bottom-up clauses. The disadvantage in the mixed approach is that a lot of memory is required, because of the two different sets of clauses.

It is widely accepted in literature that the inheritance mechanism should be embodied directly into the unification algorithm, rather than into resolution. The resulting system is hopefully more efficient than the standard Prolog, when handling taxonomies. An ad-hoc mechanism is also a viable solution for improving declarativeness of logic programs, at least as regards the taxonomy mechanism.

In order to support this assertion with some experimental evidence, we have built Taxlog, a Prolog interpreter which uses restricted variables for reasoning about classes and objects. Taxlog finds a description of both of them in a user-defined taxonomy. Taxlog implements a rather different model than LOGIN (1), in order to allow the programmer to define taxonomies having a class-metaclass structure (6,7). We compared the performance of Taxlog with standard Prolog in a series of tests in order to measure:

. the overhead involved by the taxonomic reasoning machinery,
. the time and space advantages that result in an intensive usage of the mechanism.

Section II of this paper describes our informal model of taxonomic reasoning, and Section III describes the relevant literature and discusses the differences between Taxlog and other approaches. Section IV contains the main implementation techniques we adopted for extending the capabilities of a standard Prolog interpreter and for building Taxlog. Finally, in Section V we report and discuss the results we obtained in some experiments, and compare the performance of the standard Prolog interpreter with that of Taxlog.

II TAXONOMIES AND OBJECTS IN TAXLOG

Taxlog taxonomies are built up from nodes, which are denoted by constants, and two kinds of arcs, is a for membership and s_s for set inclusion. Three special primitives allow the programmer to define taxonomies.

mkcateg(Class,Level) declares a new Class to the system, and relates it with a Level. The system actually uses the class level as a simple heuristics when an ss-relation must be proved.

mkisa(Atom,Class) builds an is_a arc between the Atom and the Class. The Class is not required to be a leaf of the s_s network; moreover, if any previous is_a arc has been declared, the old edge will be deleted and substituted with the new one.

mkss(Subclass,Class) builds an s_s arc between Subclass and Class, assuming that Subclass is a specialization of Class.

Here follows an example of taxonomy definition.

```
?- mkcateg(european,100),
   mkcateg(Citalian,french,german,englishJ,90),
   mkss(Citalian,french,german,english,europeanl\
   mkisa(me,Italian).
```

Appendix 1 shows a user-friendly syntax for defining taxonomies. Actually, this syntax is described by Prolog clauses, and is loaded at initialization time.

Note that the same constant may be considered both as an object and a class; as an object it can be classified in a class by means of an is_a edge. Thus, in Taxlog we can build taxonomies where superclasses are distinct from metaclasses. In the foregoing example, a metaclass may be defined:

```
?- mkcateg(nationality,100),
    mkisa(Citalian,french,german,englishJ,
           nationality).
```

The issue of extending the Prolog unification to an efficient use of taxonomies has been solved by introducing restricted variables, that is by allowing variables to be quantified over classes. In Prolog terms, an atom and a variable restricted to a clasB are unifiable only if the atom is a member of the class. An atom is a member of each class linked with it by a chain of exactly one is_a and zero or more s_s arcs. This inheritance rule avoids some typical problems arising when no distinction is made between isa and ss arcs (in our example, "me" is not a "nationality"!).

Taxlog interprets logic programs whose clauses are quite similar to Prolog ones, except that variables can be restricted to the classes defined in the taxonomy. As an example of Taxlog clause, consider the following, taken from (1):

```
happy(X) :- like(X,Y), got(X,Y)
          X isa person.
```

The constant person has been declared as a class. In the clause, X is a variable restricted to person, and Y is unrestricted. The ':' binary operator separates a clause from the restriction declarations over the variables. The parser can recognize the precondition part of a clause from the restriction part, and build an internal structure for representing restricted variables. See Appendix 2 for a brief description of the read and write primitives.

The unification algorithm has been extended, and actually covers three new cases:

. unify an atom and a restricted variable: this operation succeeds if the atom has boen declared as member of either the restriction class or a specialization of it; in every other case, including if the restriction class is a specialzation of the atom class, the unification fails;
. unify an unbound variable and a restricted variable: the two variables share, and the same restriction constrains the unified variable;
. unify two restricted variables; they are unified in the intersection of their restrictions.

The user may submit Taxlog any goal containing restricted variables; analogously a Taxlog answer can be an atom, a structure, a restricted variable, or an unrestricted variable. This fact distinguishes Taxlog from traditional Prolog interpreters, because Taxlog answers can refer to classes, not purely to single objects, by means of restricted variables. Note that this fact can avoid some confusing answers, that arise from an uncorrect mixing of classes and objects. Thus even in answers we can distinguish properties of each object in a class from properties of a class as an object. For example, if we declared:

```
  ?- mkisa(doggy1,dog),
     mkisa(doggy2,dog),
     mkisa(dog,animal_species).
```

and have the following clauses:

```
  barks(A) : A isa dog.
  contains(A,zoology_book) : A isa animalj3pecies.
```

the goal barks(X) produces the answer:

```
  X ■ _1 : _1 isa dog.
```

while the equivalent Prolog answer is:

```
  X = doggy1;          (ask an other solution)
  X = doggy2
```

The Taxlog goal

```
  ?- contains(dog,zoology book).
```

succeeds, while the goal

```
  ?- contains(X.zoologybook) : X isa dog.
```

fails, because the property "contained into the zoology_book" refers to the class of dogs as a whole, not to individual dogs.

There exists a suitable instantiation predicate, which allows the user to take Taxlog answers back to Prolog-style. This predicate extends DECSystem-10 Prolog current atom(A) predicate (11). The goal is successively satisfied by instantiating the variable A to the atoms in a system-dependent order. If A is a restricted variable, the goal will be satisfied exclusively by the atoms whose class satisfies the given restriction. We can use current_atom in order to enumerate the atoms which are members of a class and of its specializations. In the "barks" example, a Taxlog goal like:

```
  ?- barks(X), current_atorn(X).
```

produces the very same sequence as Prolog,

There is a discrepancy between Taxlog and Prolog answers when some class at the lowest level of the taxonomy is empty. The discrepancy may be eliminated by applying the current_atom predicate. For example, if no object has been declared member of the "dog" class, the Taxlog goal barks(X) produces the answer:

```
  X » 1 : _1 isa dog.
```

but the Taxlog goal:

```
  ?- barks(X), current_atom(X).
```

fails, as well as barks(X) submitted to Prolog.

III RELEVANT LITERATURE

McSkimin and Minker (8) introduce the TT-a clause representation for first order predicate calculus, where a variable quantification method over subsets of the universe of objects is used. Variable restrictions are managed by a semantic unification algorithm. In a dictionary, semantic categories are related to constants, function domains and ranges, and predicates by means of membership relations. Moreover, three kinds of relation interconnect semantic categories, that is superset, equality and disjointness.

Walther (13,14) extends the Markgraf Karl Refutation Procedure to a many-sorted theorem prover on the basis of its many-sorted calculus, where variables can be declared quantified over a sort. Some experimental results are available. Cohn (5) uses a sorting function to describe sort restrictions deriving from the argument positions of the function and predicate symbols that it occupies.

Finally, Ait-Kaci and Nasr (1) propose psi-term unification as a built-in mechanism for handling is_a taxonomies within Prolog.

Our proposal differs from Ait-Kaci and Nasr for a number of reasons. The main reason is that we distinguish is_a and s_s arcs, and thus enable metaclasses to be represented. In LOGIN there is no way of distinguish the class level from the object level, because both are considered as subtypes.

An other difference between the two approaches is that LOGIN allows functor symbols to be unified according to the taxonomy, while Taxlog forbids it. It is possible to perform this kind of taxonomic reasoning in Taxlog by representing the taxonomic factor as an adjunctive argument of a general functor symbol, thus enabling the difference between type-constructor and functor being used in order to preserve the efficiency in unification. The functor unification is performed in Taxlog as address comparison, while the inheritance property is entrusted to the simpler unification algorithm for constants.

We think that the Taxlog proposal may be considered, in a sense, complementary to LOGIN. Many of the features introduced in LOGIN are quite interesting for supplying Prolog with a suitable support for programming. In particular, we think that the three main representation improvements, non-fixed arity for terms with explicit labeling, tagged arguments, and compile time enforcement of taxonomic consistency should be effectively used in the design of Prolog compilers. Nevertheless, as a run time support, we think that Taxlog is quite efficient, and provides the user with a clearer and more intuitive semantics.

IV INTEGRATING TAXONOMIES INTO A PROLOG INTERPRETER

A prototype version of the Taxlog interpreter has been built by suitably extending an existing Prolog interpreter (9), fully compatible with the standard described in (4). The prototype is written in C language and currently runs on an AT&T

3B2 microcomputer under the UNIX(TM)* System V as operating system.

The Taxlog prototype interpreter currently implements all the functionalities described in the previous sections, except for the following restriction to the structure of taxonomies: each class is allowed to have only a single father. This (actual) limitation causes taxonomies to have a tree structure.

The following modifications to the Prolog interpreter have been done.

A new system area has been defined for classes, besides the traditional execution stacks (local, global, trail stack), and the database. The category area contains the class descriptors and a representation for is_a and s_s arcs. For each class record the following information is stored:
. a pointer to the generalization class,
. a pointer to the list of specialization classes,
. a pointer to the class member list,
. the class level,
. a pointer to the atom denoting the class.

Two fields have been added to the atom internal structure, a pointer to the class denoted by the same string as the atom, and a pointer to the class the atom is a member of.

Restrictions on variables are actually represented as pointers to the category area. The resolution algorithm is substantially unchanged, but the unification algorithm has been extended as described in Section II. The stack record internal structure has been left unchanged. The trail stack mechanism only has been modified: when a restricted variable is instantiated, or is restricted to a more specific class, the old restriction is saved, together with the address of the variable. Much care has been taken in order to use as little as possible space for the trail records. Actually Taxlog saves space for unrestricted variables, because it does not push on the trail the "undefined" value for them. During the backtracking phase, a test is performed on the second element on the top of the trail: if it is a restriction (i.e. a pointer to the category area), then this is the old restriction for the variable whose address is stored on the top of the trail.

Finally, the clause internal data structure contains a representation for restrictions on clause variables. Actually, for a number of reasons restricted variables are classified as global. The clause representation consists thus of an array of initial values for global variables only. When a clause is selected for matching, the system builds two new contexts in the local and global stacks, and initializes the global stack record to the restrictions descripted in the clause. If no restricted variable appears, the clause representation is almost identical to Prolog but a flag indicating whether or not an initial global stack record exists in the clause representation.

♦UNIX is a trademark of AT&T Bell Laboratories.

V PERFORMANCE COMPARISON BETWEEN PROLOG AND TAXLOG

Some considerations and experiments have been made in order to compare the performance of Taxlog and Prolog and to measure, in terms both of required space and computation time, the overhead and the advantages involved by the extensions.

A* Time efficiency

We carried out two series of experiments for measuring time efficiency. Much care has been taken in order to exclude data pollutions due to the presence of other users in the system. We took the tms_utime datum, and obtained it by means of the times system call. It represents the CPU time used by the calling process in order to execute operations in the user space (12). The experiments took place with the operating system in single-user mode.

1. Time overhead due to the taxonomy mechanism

In the first series of tests the programs do not make use of taxonomic reasoning at all. Note that Taxlog is an extension of Prolog, thus we could use the very same source programs in our experiments for both the interpreters.

Table 1 - Average execution times (100 runs) (data in lOOths sec.)

Procedure(Datum)	Prolog	Taxlog	Ratio
nreverse(1ist30)	58.51	60.62	1.0361
qsort(list50)	91 .90	94.56	1.0289
deriv(timeslO)	5.54	5.64	1.0181
deriv(dividelO)	5.73	6.02	1.0506
deriv(loglO)	4.17	4.36	1.0456
deriv(ops8)	4.01	4.20	1.0474
serialise(palin25)	51.58	53.39	1.0351
dbquery**	127,33	133.41	1.0477
dbquery	743.16	778.71	1.0478

We took the test programs from Warren (15):
• "reverse, which inverts the terms of a list, applied to list30, a 30 element list;
. qsort, a version of the quicksort algorithm, applied to the Warren list50 list;
• deriv, a program for computing symbolic derivatives, applied on four different arithmetic expressions, named timeslO, dividelO, loglO, and ops8;
. serialise, which translates a given string into a list of numbers which represent the character

**In the dbquery example, the system gives five answers. We reported in the table, respectively, the time for the first answer and the time for the refutation.

codes, applied to the string palin25;

. dbquery, which represents a simple query application on a little geographic database.

Table 1 summarizes the results of this series of tests. For the Warren's examples, the overhead falls into the interval between 1.80% and 5.06% of the total execution time.

The overhead does not depend on the total execution time. It is due exclusively to the higher complexity involved in the unification, because Taxlog considers the case of restricted variables even if in these examples they do not appear.

2. Time advantages in using taxonomies

In this second series of tests we measured the improvement which derives from an extensive use of the taxonomic reasoning capability. The testing program represents a simple query on a database. Some properties in the database are defined by using objects and classes declared in a taxonomy. In the query some variables are restricted.

Two different versions have been built for each test program. In the standard Prolog version, no restricted variable appears, while the Taxlog version contains a description of the taxonomy in terms *of* the primitives we introduced. A number of difficulties has been faced in order to make significant the comparison between the two systems. First, even though the two systems interpret the same example, they give a different number of answers, because Prolog instantiates variables to the constants which satisfy the goal, but Taxlog generalizes answers as far as possible to categories. In order to have comparable results two kinds of measures have been taken: the first-answer time, that is the time the system takes in order to find the first solution for the goal, and the refutation time, i.e. the time the interpreter takes to generate all solutions and then answer "no".

Second, first-answer times in a Prolog interpreter are greatly dependent on the clause ordering: in a choice point the interpreter examines directly the correct branch only if it is the first in the database. First-answer times were taken with the database ordered in the most favourable way. Note that the Taxlog interpreter is very little sensitive to the order of the various parts of a taxonomy. Thus, the first-answer time is the most favourable measure for Prolog. On the contrary, the refutation time is fully independent from the clause ordering in the database for both systems, provided that no extra-logical feature, such as cut, assert, etc., appears. In general, Taxlog capabilities are exploited for refutation problems.

Third, two different kinds of query have been taken into consideration. In the first case there is a two-variable query: the interpreter satisfies the goal by finding a suitable combination of values for the two variables. In the second case two constants appear instead of the variables: the interpreter proves that the query is satisfied by the two constants. Two-variable queries have been measured, for Taxlog, in two different ways: in the first, an answer is given in terms of restricted variables, in the second way the answer is made uniform to standard Prolog by filtering it through the current_atom primitive.

Fourth, the test programs for the two systems are not equal: it is therefore possible that different implementation techniques for the same program require different answer times. With respect to standard Prolog, the two solutions, top-down and bottom-up, have been considered for the problem of searching through a taxonomy. Note that for first-answer times with optimal ordering of clauses top-down and bottom-up are almost equivalent, because no true search is required. Thus, only results for top-down are reported. The difference between the two approaches may be better appreciated for refutation times.

Finally, first-answer times for random clause ordering range from the first-answer time for optimal clause ordering (including it) to the refutation time (excluding it). Thus the comparison of the two measures gives an idea of the answer times for increasingly more non-deterministic problems.

In Appendix 3 we report the test program for each version.

First-answer times

The test has been repeated for increasingly deeper taxonomies: the results (see Table 2) show that first-order times linearly depend on the taxonomy depth. As the rate is much higher for Prolog than for Taxlog, we have proved that a single complex unification step is far more efficient than a taxonomic deduction with a simple unification algorithm. Note that Taxlog results for two-variable queries do not depend on the taxonomy size, because only one clause is used, and taxonomic information is not required (see Appendix 3).

Refutation times

Three groups of tests have been done, in order to measure the refutation time rate resulting in increasing (uniformly):

. the number of objects per class,

. the number of subclasses per class,

. the depth of the taxonomy.

All experiments for each group of tests make use of balanced taxonomies with a uniform number of objects per bottom-level class. Table 3 shows the results for these tests.

Table 2 - Average first-answer times

Taxonomy depth	Two constants: Prolog top-down	Two constants: Taxlog	Two variables: Prolog top-down	Two variables: Taxlog	Two variables: Taxlog «
2	0.749	0.220	0.662	0.198	0.441
3	1.063	0.234	0.879	0.198	0.457
4	1.362	0.251	1.099	0.198	0.484
5	1.656	0.269	1.311	0.199	0.504
6	1.954	0.272	1.528	0.194	0.531
7	2.251	0.292	1.747	0.197	0.553
8	2.559	0.302	1.963	0.197	0.570
9	2.856	0.317	2.182	0.199	0.595
10	3.155	0.326	2.396	0.201	0.617
11	3.457	0.345	2.617	0.199	0.640
12	3.758	0.352	2.835	0.197	0.657
13	4.063	0.367	3.051	0.198	0.686
14	4.358	0.376	3.267	0.200	0.705
15	4.665	0.395	3.484	0.195	0.726
16	4.963	0.405	3.701	0.197	0.753
17	5.241	0.424	3.890	0.197	0.770

Table 3 - Average refutation times

Increasing the no. of objects per class.

Number of objects per class	Two variables: Prolog top down	Two variables: Taxlog	Two variables: Taxlog	Two constants: Prolog top down	Two constants: Prolog bottom up	Two constants: Taxlog
1	0.322	0.201	0.626	0.299	1.205	0.223
2	0.701	0.197	1.224	0.398	1.195	0.223
3	1.282	0,201	2.063	0.444	1.193	0.222
4	2.074	0.199	3.163	0.487	1.207	0.218

Increasing the no. of subclasses per class.

Number of subclasses per class	Two variables: Prolog top down	Two variables: Taxlog	Two variables: Taxlog ♦	Two constants: Prolog top down	Two constants: Prolog bottom up	Two constants: Taxlog
2	1.017	0.202	1..282	0.619	1.140	0.230
3	1.941	0.201	2,.193	0.841	1.136	0.229
4	3.198	0.203	3,.362	1.069	1.134	0.229
5	4.779	0.201	4.805	1.292	1.133	0.227

Increasing the depth of the taxonomy.

Depth of the taxonomy	Two variables: Prolog top down	Two variables: Taxlog	Two variables: Taxlog #	Two constants: Prolog top down	Two constants: Prolog bottom up	Two constants: Taxlog
1	0.322	0.201	0,.626	0.299	1.205	0.223
2	1.078	0.200	1..285	0.643	1.365	0.231
3	3.989	0.199	3,.507	1.334	1.535	0.237
4	15.459	0.201	11..628	2.711	1.700	0.246

♦The results in these columns refer to answer times for queries filtered through the current_atom primitive, in order to take Taxlog answers back to Prolog style.

B. Memory requirements

Some considerations arise when the space requirements for the two systems, Prolog and Taxlog, are compared. We can distinguish between static and dynamic space requirements.

1. Static space requirements

Actually, Taxlog imposes a space overhead of two words per atom, in order to store the two additional pointers (see Section III). This overhead must be payed for each atom in the system, thus included all atoms which are members of no class.

In order to store taxonomic relations, we need some space for classes, and for s_s and is_a arcs. Table 4 summarizes space requirements for Taxlog, and for the two approaches, top-down and bottom-up, in Prolog. In these measures we do not consider the space for storing atoms. Classes are represented in Taxlog by entities in the category area, while in Prolog they are represented as unary functors. The bottom-up approach, moreover, requires some space for storing a clause for each category entry. In Prolog more space is needed in order to store s_s and is_a arcs than in Taxlog because the arcs are represented as normal clauses, and thus a representation for head, tail, and control information must be provided.

Table 4 - Static space requirements for storing a taxonomy (in words).

Stored object	Taxlog	Prolog top-down	Prolog bottom-up
Class	5	7	15
s s arc	2	10	10
is a arc	2	8	10

Clauses not involving restricted variables need the very same space for both systems. When restricted variables appear, instead, the space difference between Prolog and Taxlog depends on two parameters: T, that is the number of taxonomic restrictions in a clause (and their equivalent representation as predicates in standard Prolog), and V, that is the number of global variables in the Taxlog clause. For our implementation, a formula gives the difference in number of words:

$$(\text{Prolog\_space} - \text{Taxlog\_\_space}) \approx 5 * T - V$$

Note that an intensive use of taxonomic restrictions favours Taxlog, because Prolog requires space in order to store the predicate form of the taxonomic restriction and the comma separator, while Taxlog stores directly the restriction as a pointer to the category area in the initial environment. But restricted variables are considered global, and, actually, the initial environment contains a word for each global variable. This fact explains why the corrective factor appears in the formula.

2. Dynamic space requirements

It is worth noting that Taxlog requires the very same dynamic space as Prolog, if the taxonomy mechanism is not used. The only stack space waste that Taxlog imposes, respect with Prolog, is in trail management, because the old variable restrictions must be stored together with the variable addresses, during the unification. But the careful trail management algorithm we used saves this space when unrestricted variables are instantiated.

Surprisingly, even if taxonomies are used, much dynamic space is gained, as the data in Table 5 confirm, because many stack records, that Prolog pushes on the local stack for handling the clauses describing the taxonomy, are simply not used (and thus not stored) by Taxlog. Consequently, more space is gained in the trail stack, because there is no need of saving the addresses of the taxonomic clauses; moreover, less variables exist in the system. The increase in the global stack size is due to the fact that restricted variables are classified as global, and thus are stored in the global stack instead of the local stack.

Table 5 refers to the same examples used for first-answer time queries with two variables. Data are relative to the smallest taxonomy (depth = 2). Bigger taxonomies give the same results for Taxlog, but a higher space usage for Prolog, due to the stack records for the taxonomy clauses.

Table 5 - Dynamic space requirements for an example (in bytes).

Memory area	Taxlog	Prolog top-down
Local Stack	S6	244
Global Stack	16	0
Trail Stack	32	52

VI CONCLUSIONS

The efficiency of a prototype of the Taxlog interpreter has been valued with respect to standard Prolog by means of a series of measurements. The maximum overhead, due to the higher degree of complexity of the extended interpreter, is about 5% of the execution times, and is fully independent from the size of the test problem. An intensive use of taxonomies involves considerable improvements in terms of execution times. The taxonomic reasoning model employed in Taxlog is quite general: this system feels a good candidate for many practical applications (3).

ACKNOWLEDGEMENTS

Most work on the model has been accomplished by the author and Cristina Bena, whose contribution is invaluable. Useful suggestions came to this work from Franco Sirovich, Gianfranco Rossi and Alberto Martelli at the Comp. Science Dept. of University of Turin. The author wishes to thank his colleagues at Laboratorio di Intelligenza Artificiale for their support and helpful discussions.

APPENDIX 1 - USER-FRIENDLY SYNTAX FOR DEFINING TAXONOMIES

A simple, user-friendly syntax has been defined for helping the programmer to introduce taxonomies into the Taxlog system. A BNF description of the grammar is given, together with a Prolog description of its semantics and an example.

BNF description:
```
  Taxonomy^declaration ::= declare Tree.
  Tree ::= Atom
         | Atom :=s List_of_elements
         | Tree -> Tree { , Tree i
  List of elements  ::= CAtom { , Atom
```
Note: {J are the usual Kleene closure operators, and indicate zero or more occurrences of a pattern.

Prolog interpretation:
```
  ?- op(255,fx,declare),
     op(254,xfy.'->'),
     op(200txfx,•:=•).
  declare Tree :- declare(Tree,1000000,).
  declare((A->B),N,Top) :- !,  declare(A,N,Top),
                 NmlO is N - 10,
                 declaress(Top,B,NmlO).
  declare((A := L),N,A) :- !, mkcateg(A,N),
                 mkisa(L,A).
  declare(A,N,A) :- mkcateg(A,N).
  declaress(T,(B,C),N) :- !, declare(B,N,TopB),
                 mkss(TopB,T),
                 declaress(T,C,N).
  dec1areas(T,B,N) :- declare(B,N,TopB),
                 mkss(TopB,T).
```
Example:
```
  ?- declare european -> italian :* fmej,
                           french, german, english.
  ?- declare nationality :« Citalian, french,
                                german, engli sh 3.
```

APPENDIX 2 - READING AND WRITING TAXLOG TERMS

From the syntax point of view, the Taxlog interpreter accepts exactly the Prolog constructs. The read primitive first builds the deep structure of the input expression, considering the special ^{1}:• symbol as a normal syntactic operator*, then interprets it and considers its second argument as a succession of restrictions to be applied to the variables appearing in the first argument. The read primitive makes therefore the symbol ':' and the syntactical structure of the restrictions

•Note that in the bootstrapping phase the operators are currently defined as follows:
```
  ?- op(255fxfx,':»), op(254,xfx,•:-•),
     op(253,xfy,•,•), etc.
```

transparent to the rest of the system. For example if the goal:

```
  ?- read(A), functor(AfF,Arity).
```

is given the input sequence:

```
  structure(Varl,Var2,Var3) : Varl isa catl.
```

the Taxlog answer is the following:

```
  A « structure(_l,_2,_3) : _1 isa catl
  F « structure
  Arity *= 3
```

while the goal:

```
  ?- read(A), arg(l,A,V).
```

with the same input structure as before produces the following answer:

```
  A = structure(_l,_2,_3) : _l isa catl
  V = _1 : _1 isa catl
```

The write primitive writes its argument with the same syntax as the answers: if in its argument any restricted variable appears, the write primitive prints the argument denoting each variable with a symbol. Then the special •:• operator is printed, followed by the restrictions on the variables in the format:

<variable> isa <class>

The symmetry between read and write has been maintained: structures written on a file by Taxlog are in the format accepted by the read primitive.

APPENDIX 3 - TEST PROGRAMS

Here follows an example of the test programs and taxonomies for time measurement.

```
Taxlog
  ?- mkcateg(zero,100), mkcateg([a,b3,90),
       mkss(£a,b2,zero),
       mkisa(oa,a)f mkisa(ob,b).
  relation(A,B) : A isa a, B isa b.
Prolog, top-down
  zero(A) :- a(A).              zero(A) :- b(A).
  a(oa).                        b(ob).
  relation(A.B) :- a(A), b(B).
Prolog, bottom-up
  zero(zero).
  a(a).                         b(b).
  a(A) :- zero(A).              b(A) :- zero(A).
  oa(A) :- a(A).                ob(A) :- b(A).
  relation(A.B) :- isa(A,a), isa(B,b).
  isa(0fC) :- I = ..C 0,C] , I.
```

```
Measurements for first-answer times
  measure :- cpu_time(Tl),
             relation*A,B),          /* (1) */
             cpu_time(T2),
             Tdiff is T2 - Tl,
             write(Tdiff).
For two-constant queries, instead of (1) use:
  relation(oa,ob),
For Taxlog two-variable queries with current_atom,
instead of (1) use:
  relation(A,B), current_atom(A), current_atom(B),
```

```
Measurements for refutation times
  measure :- cpu_time(T), refuted).
  refute(_) :- relation(A.B), fail.       /* (2) */
  refute(Tl) :- cpu_time(T2), Tdiff is T2 - Tl,
                write(Tdiff).
For two-constant queries, instead of (2) use:
  refute(_) :- relation(oafob), fall.
For Taxlog two-variable queries with current_atom,
instead of (2) use:
  refute(_) :- relation(A.B), current_atom(A),
               current_atom(B), fail.
```

REFERENCES

1 Ait-Kaci H., R. Nasr "LOGIN: A Logic Programming Language With Built-in Inheritance" J. Logic Program. 1:3 (1986) 185-215.

2 Bena C., G. Montini "Analysis and proposals for using logic languages in consultation systems", In Proc. 1st Italian Nat. Conf. on Logic Progr. (in Italian), Genova, March 1986, 152-159.

3 Bena C, G. Montini and F.Sirovich "Planning and Executing Office Procedures in Project ASPERA" In Proc. 10th 1JCAI, Milano, 1987.

4 Clocksin W.F., C.S. Mellish Programming In Prolog, Springer-Verlag, 1981.

5 Cohn A.G. "On the Solution of Schubert's Steamroller *in* Many-Sorted Logic", In Proc. 9th IJCAI, Los Angeles 1985, 1169-1174.

6 Goldberg A., D.Robson Smalltalk-80. The language and its implementation, Addison-Wesley, 1983.

7 Levesque H., J. Mylopoulos "A Procedural Semantics For Semantic Networks" in Findler, Associative Networks, Academic Press, 1979, 93-120.

8 McSkimin J.R., J. Minkcr "A Predicate Calculus Based Semantic Network For Deductive Searching", in Findler, Associative Networks, Academic Press, 1979, 205-238.

9 Montini G. "GioLog User's Manual", Int. Report -PQ0786-9, LIA, CSI-Piemonte, Torino, July 1986.

10 Montini G. "Efficiency of taxonomic reasoning and Prolog interpreters", In Proc. 2nd Italian Nat. Conf. on Logic Progr. (in Italian), Torino, May 1987.

11 Pereira L.M. User's Guide to DECsystem-10 Prolog, Divisao de Informatica, Lab. Nac. de Eng. Civ., Lisboa, 1977.

12 UNIX(TM) System V User Reference Manual, Release 2.0, October 1984, AT&T Bell Laboratories.

13 Walther C. "A Many-Sorted Calculus Based on Resolution and Paramodulation", In Proc. 8th IJCAI, Karlsruhe, 1983.

14 Walther C. "A Mechanical Solution of Schubert's Steamroller by Many-Sorted Resolution" Artif. Intel1. 26:2 (1985) 217-224.

15 Warren D.H.D. Applied Logic - Its Use and Implementation as a Programming Tool, PhD Thesis, Edinburgh, 1977.