Acquiring object-knowledge

LUC DE RAEDT¹, JOHAN FEYAERTS and MAURICE
BRUYNOOGHE¹

*Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
email: luc.DeRaedt@cs.kuleuven.ac.be*

Abstract. A novel approach to interactively acquire knowledge about new objects in a logic environment is presented. When the user supplies an unknown fact containing unknown objects (constants), the system will ask interesting membership and existential queries about the objects. The answers to these questions allow the system to update its knowledge base. Two basic strategies are implemented: one that examines existing Horn clauses for the predicate and another one that uses types. Furthermore, a powerful heuristic based on analogy, to pose the most interesting questions first, is presented.

Keywords: learning by being told, knowledge acquisition, knowledge base updating, analogy

Received 2 April 1991; revised 2 January 1992

1. Introduction

It is a well known cliché that the power of artificial intelligence systems is—to a large degree—due to the knowledge they possess. The acquisition of the knowledge, necessary to build intelligent systems, is a difficult and time-consuming process. These difficulties explain the current interest in knowledge acquisition tools (Gaines and Boose, since 1989) which assist the knowledge engineer in his task. Related research issues are concerned with the so-called learning system (Michalski *et al.* 1983, 1986, Kodratoff and Michalski 1990, Kodratoff 1988), which promise to contribute to the widening of the knowledge acquisition bottleneck. The knowledge about objects and relations that hold among them constitutes the main part of the domain specific knowledge in intelligent systems. Therefore, an important subtask in knowledge acquisition as well as in learning systems is the acquisition of knowledge regarding objects. In learning systems this knowledge is often a prerequisite for success. Indeed, many systems require that the domains of variables are completely known (Michalski 1983, Quinlan 1986). Other approaches, like (De Raedt and Bruynooghe 1989, 1992b, Wirth

---

¹Supported by the Belgian National Fund for Scientific Research.

1988, Quinlan 1990) cannot learn if they are not provided with basic knowledge about the different objects. In learning apprentice systems such as **Disciple** (Kodratoff and Tecuci 1987, Tecuci and Kodratoff 1990), some knowledge about objects is acquired as a side-effect of the learning, while in other systems such as **Blip** (Morik 1989, Wrobel 1988), knowledge about objects is derived by the system from the facts which are input.

We take a different approach. When the user inputs a true fact containing an unknown object, an attempt is made to gather as much knowledge about the object as possible. To this purpose, the knowledge base is examined and interesting membership and existential queries (Angluin 1987, Shapiro 1983) are formulated to the user. The answers to these questions allow updating of the knowledge base. Two main algorithms are employed: analysing existing definitions (Horn clauses) for the predicate and using types or sorts of predicates. Furthermore a powerful heuristic based on analogy is employed to generate the most interesting questions first. The novel method is completely implemented in **BIMprolog**. It can easily be integrated in other systems, for example **Disciple** (Kodratoff and Tecuci 1987, Tecuci and Kodratoff 1990), **Clint** (De Raedt and Bruynooghe 1989, 1992b), **LFP** (Wirth 1988), **Cigol** (Muggleton and Buntine 1988) and **Blip** (Morik 1989, Wrobel 1988). It is also feasible to regard the approach as a form of learning by being told, and in this sense, there is some resemblance with Haas and Hendrix (1983).

This paper is organized as follows: in Section 2, we define the basic notions and a problem specification; in Section 3, the algorithms are presented while in Section 4, an example of the algorithms at work is shown; and finally, in Section 5 we conclude and touch briefly upon related work. Before going through Section 2 and/or 3, it may be useful to have a look at the example session in Section 4.

2. Problem specification

We use a logical framework (Genesereth and Nilsson 1987). Within this framework, a predicate-definition is a set of Horn clauses for the predicate (as in PROLOG (Sterling and Shapiro 1986, Bratko 1986)). For the sake of simplicity, we restrict ourselves to function-free predicates and Horn clauses without recursion. We distinguish different kinds of predicates: on the one hand, *stable* and *modifiable* predicates and on the other hand, *base*, *view* and *mixed* predicates.

Definition 2.1: *A stable predicate is assumed to be correct. It may not be modified.*

Definition 2.2: *A modifiable predicate may contain errors. It may be modified.*

Definition 2.3: *For a predicate declared as base, only ground facts are allowed in the definition.*

Definition 2.4: *For a predicate declared as view, only non-unit clauses are allowed in the definition.*

Definition 2.5: *For a predicate declared as mixed, all possible Horn clauses are allowed in the definition.*

The definition of a mixed predicate may contain facts as well as non-unit clauses. Examples of base, view and mixed predicates can be found in example 1.

We assume that the system has access to an oracle that is willing to answer membership and existential questions. In practice, the oracle is the user who answers the questions generated by the system.

Definition 2.6 (Shapiro 1983, Angluin 1987): A membership question queries an oracle about the truth in the intended interpretation of a ground atom $p(a_1, \dots, a_n)$, where p is a predicate and the a_i are constants (objects). The oracle must answer 'yes' if the relation p holds among the constants $a_1, \dots, a_n$ in the intended interpretation; otherwise, it must answer 'no'.

Definition 2.7 (Shapiro 1983, Angluin 1987): An existential question queries an oracle about the truth in the intended interpretation of a non-ground atom $p(x_1, \dots, x_n)$, where p is a predicate, at least one of the x_i is a variable and at least one of the x_i is a constant (an object). The oracle must return all ground substitutions θ such that $p(x_1, \dots, x_n)\theta$ is true in the intended interpretation or if there are no such substitutions, it must answer 'no'.

The intended interpretation denotes the meaning of the desired target knowledge base.

Asking for the truthvalue of `owns(john, rolls-royce)` is a membership question whereas asking whether there is a `X` such that `owns(john,X)` holds is an existential one.

When the oracle answers 'no' to a question q , the question q is added to the set of facts F . F contains all the facts that are known to be false in the intended interpretation. This set is used to prune the search and to prevent the assertion of false facts. The set F is potentially large, but it can be used in an efficient way when using indexing techniques.

By now, the problem can be specified more formally as:

Given:

- a knowledge base KB of predicate definitions
- a set of facts F , such that $\forall f \in F$ there is no substitution θ for which $f\theta$ is true in the intended interpretation
- a fact $p(o_1, \dots, o_n)$ such that $KB \not\models p(o_1, \dots, o_n)$ and at least one of the o_i is a constant, not occurring in the knowledge base KB , i.e. o_i does not occur in the Herbrand domain of KB .
- an oracle willing to answer existential and membership questions.

Find: a set S of facts about modifiable predicates, true in the intended interpretation, such that $KB \cup S \models p(o_1, \dots, o_n)$ and there is no $s \in S$ for which there is an $f \in F$ and a substitution θ such that $f\theta = s$; the set S will be called the update.

In this problem specification, we have a fact $p(o_1, \dots, o_n)$, that is not implied by the knowledge base KB , but which should be. Furthermore, there is the set of known false facts F . The aim is then to find a set of true facts S such that $p(o_1, \dots, o_n)$ is implied by the knowledge base $KB \cup S$. We will sometimes refer to this problem as the *update* problem; furthermore, if the predicate p is a view predicate it will be referred to as the *view update* problem; if p is a base predicate, the problem becomes the *base update* problem. These notions are illustrated in Example 1.

In Example 1, it is rather clear which questions should be posed to the oracle as `drives` is a view predicate. However, suppose we want to add a new type of car (a base predicate). Then it would be harder to decide which questions to generate. Of course, one could assert the fact itself. This is not desirable as the amount of knowledge acquired in this way is minimal. We will present two

Suppose we have the following definitions in our knowledge base :

```

declare owns/2 : base & modifiable
declare car/1 base : & modifiable
declare bicycle/1 : base & modifiable
declare vehicle/1 : view & modifiable
declare drives/2 : view & modifiable
declare has_license/2 : base & modifiable

```

```

owns(yves, peugeot) ←
owns(katharina, mercedes) ←
owns(maurice, renault) ←
owns(luc, plume) ←

```

```

vehicle(X) ← car(X)
vehicle(X) ← bicycle(X)

```

```

car(peugeot) ←
car(mercedes) ←
car(renault) ←
bicycle(plume) ←

```

```

has_license(katharina) ←
has_license(yves) ←
has_license(maurice) ←

```

```

drives(X,Y) ← owns(X,Y) ∧ car(Y) ∧ has_license(X)
drives(X,Y) ← owns(X,Y) ∧ bicycle(Y)

```

In this knowledge base, the predicates **drives** and **vehicle** are view predicates, the others are base predicates. Suppose we want to make the fact **drives(luc, citroen)** true. There are several possibilities to realize this :

```

S1 = { drives(luc, citroen) }
S2 = { owns(luc, citroen), has_license(luc), car(citroen) }
S3 = { owns(luc, citroen), bicycle(citroen) }

```

Given the available information, we only know that S_1 is correct w.r.t. the intended interpretation. However, if one of the other possible updates is correct, it is not interesting to assert S_1 in the knowledge base. The reason is very obvious : by asserting S_1 , much potential information would be lost. Indeed, it would not be possible to derive, for example, **vehicle(citroen)** from the knowledge base. Furthermore, update S_1 is not allowed as an update as the predicate **drives** is view and would become mixed. In order to be sure that one of the other possible updates is correct we have to query the oracle for the truth value of the facts in the updates S_2 and S_3 . Also, update S_3 could have been excluded if **bicycle** was declared stable.

Example 1: illustrating the problem specification.

approaches to generate *interesting* questions to the oracle. In these approaches, we attempt to maximize the acquired knowledge while keeping the number of questions to the user acceptable. The first approach is essentially an abductive one and realizes view updates (Section 3.1). The second one relies on the type graph of the knowledge base and updates base predicates (Section 3.2). Both of these approaches rely on a heuristic that is based on analogy and that allows the

most interesting questions to be posed first. This heuristic is described in Section 3.3.

Observe that the above-sketched problem can be regarded as an elementary *machine learning* problem: it is a form of learning by being told (Carbonell *et al.* 1983), where new information is to be transformed before it can be added to the knowledge base. The technique employed in updates S_2 and S_3 is clearly also a form of abduction. Furthermore, the problem relates to intensional knowledge base updating as described in Guessoum and Lloyd (1990), Bry (1990), De Raedt and Bruynooghe (1992a).

It is also essential that inductive concept-learners address the sketched problem. Without addressing it, they are often unable to learn. Consider for example, the concept-learner **Clint** (De Raedt and Bruynooghe 1989). In order to process the true fact (positive example) **drives(luc,citroen)**, it first derives a starting clause for **drives** by looking for the relations that hold among the objects (**luc** and **citroen**). Clearly if nothing is known about these objects, **Clint** will fail to derive the starting clause and nothing will be learned. In this situation, **Clint** cannot make the necessary modifications to its knowledge base to make the fact **drives(luc, citroen)** implied. Similar problems hold for related concept-learners such as **Disciple** (Tecuci and Kodratoff 1990) and **Blip** (Morik 1989, Wrobel 1989). Actually, the technique described in this paper was developed to overcome these problems with concept learners. We also wish to stress that it is important to see our technique as a component of a concept-learning system as it addresses only how to extend a presumably correct knowledge base and not how to remove incorrect knowledge from the knowledge base. In machine learning terminology, our approach copes with (part of) the incomplete theory problem, i.e. a true fact not implied by the knowledge base. It addresses only *part of* that (inductive) problem as we do not allow for general clauses to be added to the knowledge base. Our approach does not deal with the inconsistency problem, i.e. a false fact implied by the knowledge base. The other issues are typically also addressed in learning systems such as **Clint** (De Raedt and Bruynooghe 1992b).

3. The novel method

The top level of the novel method is specified in Algorithm 1. When a true fact that contains new objects (cf. Definition 3.1) is received, the novel method is called. When the true fact does not contain new objects, a general learning procedure such as **Clint** can be called to induce new clauses for insertion in the knowledge base. The latter is motivated by the assumption that the knowledge base already contains enough knowledge about the objects in the fact.

Definition 3.1: *A new object is an object that does not yet occur in the knowledge base, or one that is currently being added to the knowledge base by one of the algorithms presented below.*

Formulated more precisely, a new object does not occur in the Herbrand domain of the current knowledge base. The constants **rolls-royce** and **citroen** are new objects for the knowledge base given in Example 1.

As said earlier, the method to acquire knowledge about novel objects consists of three components: the abductive algorithm to update view predicates (**Handle_View**, see Section 3.1), the algorithm that uses types to update base and mixed predicates (**Handle_Base**, Section 3.2) and the heuristic to select the most interesting questions first (Section 3.3). The procedure **Handle_fact_with_new_object**

```

procedure Addfact( $p(o_1, \dots, o_n) : \text{true fact}$ )
  if one of the  $o_i$  is a new object
  then call Handle_fact_with_new_object( $p(o_1, \dots, o_n)$ )
    Reconfigure the type-graph
  else call some learning procedure, e.g. Clint (De Raedt and Bruynooghe 1992b)
  endif
endproc

procedure Handle_fact_with_new_object( $p(o_1, \dots, o_n) : \text{true fact}$ )
  case  $p$  is declared view do
    call Handle_View( $p(o_1, \dots, o_n)$ )
  case  $p$  is declared mixed do
    call Handle_View( $p(o_1, \dots, o_n)$ )
    if  $\text{KB} \not\models p(o_1, \dots, o_n)$ 
    then call Handle_Base( $p(o_1, \dots, o_n)$ )
    endif
  case  $p$  is declared base do
    call Handle_Base( $p(o_1, \dots, o_n)$ )
  endcase
endproc

```

Algorithm 1: updating the knowledge base.

shows that for mixed predicates, the algorithm first attempts to use the abductive method, and if this fails, it calls the type method.

3.1. Updating view predicates

We first present the assumption that underlies the abductive algorithm (Section 3.1.1) and then the abductive algorithm itself is described (Section 3.1.2).

3.1.1. *The assumption* Our algorithm to handle view updates relies on the following notion of dependency:

Definition 3.2: A literal l of a modifiable predicate in a goal g depends on a constant c iff (1) c occurs in l or (2) there is a variable v occurring in l , that also occurs in a literal l' of g , such that l' depends on the constant c .
If l does not depend on a constant c , it is independent of c .

For example, a literal $l(\dots, c, \dots)$ in a goal depends on the constant c but also $l_0(\dots, v_1, \dots)$ depends on the constant c if $l_0(\dots, v_1, \dots) \wedge l_1(\dots, v_1, \dots, v_2, \dots) \wedge \dots \wedge l_{n-1}(\dots, v_{n-1}, \dots, v_n, \dots) \wedge l_n(\dots, v_n, \dots, c, \dots)$ occur in the goal where $v_i, i = 1..n$ are variables. Further examples will be given in Example 2.

The set of literals of a goal g that are independent of new objects in the goal will be written as $I(g)$; whereas the set of literals that depend on new objects is $D(g)$.

The notion of dependency is essential for our approach because our algorithm is based on the following assumption:

Assumption 1: *The success or failure of query $q = \leftarrow g_1 \wedge g_2 \wedge \dots \wedge g_k$ can be determined by applying Algorithm 2 on the goal.*

This assumption implies that for the independent literals, the knowledge base is supposed to be correct. This means that the intended interpretation is assumed to be the same as the one according to the knowledge base. Therefore independent subgoals are solved by querying the knowledge base and dependent ones by querying the user. The algorithm **Solve** (Algorithm 2) works as follows: it adds the original goal to its list of goals; then it repeatedly deletes a goal from the list and attempts to prove a literal in the goal. In order to prove the literal, the algorithm tests whether the literal occurs in the dependent or the independent part of the goal. Independent literals are run on the knowledge base and dependent literals are asked to the oracle. In case of success, the remaining subgoals are added—after instantiation—to the list of goals. The process terminates with success when the list of goals contains the empty goal. It terminates with failure when the list of goals is empty. The algorithm as such will not be used in what follows; it will be fine-tuned to reduce the number of oracle questions. Example 2 makes the involved notions clearer.

3.1.2. The abductive algorithm In Algorithm 3, we summarize how to realize view updates under our assumption. The algorithm starts from a fact $p(o_1, \dots, o_n)$ for a view (or mixed) predicate that contains a new object and that is not implied by the knowledge base. The algorithm will look for facts making $p(o_1, \dots, o_n)$

```

procedure Solve( $\leftarrow g_1 \wedge \dots \wedge g_n$ : goal) returns : success or failure
   $L = \{ \leftarrow g_1 \wedge \dots \wedge g_n \}$ 
  repeat
    delete a goal  $g$  from  $L$ 
     $g' := g - \{l\}$  such that  $l \in I(g)$  if possible
     $S := \text{Query}(l, g)$ 
    for all  $\theta \in S$  do
      add  $g'\theta$  to  $L$ 
    endfor
  until  $L$  contains the empty goal or  $L = \emptyset$ 
  if  $L$  contains the empty goal
  then return success
  else return failure
  endif
endproc

procedure Query( $l$  : literal;  $g$  : goal) returns : set of substitutions
  if  $l \in I(g)$ 
  then return  $\{ \theta \mid KB \models l\theta \}$ 
  else ask  $l$  to the oracle
    return set of all  $\theta$  returned by the oracle
  endif
endproc

```

Algorithm 2: solving a goal.

Assume we have the goal $\leftarrow \text{person}(\text{jeff}) \wedge \text{has\_license}(\text{jeff}) \wedge \text{works\_for}(\text{jeff}, C) \wedge \text{air\_line\_company}(C) \wedge \text{owner}(C, \text{concorde} - 7) \wedge \text{plane}(\text{concorde} - 7)$ where `concorde-7` is a new object. Assuming the independent literals `person(jeff)` and `has_license(jeff)` to be implied by the knowledge base, the algorithm continues with the goal $\leftarrow \text{works\_for}(\text{jeff}, C) \wedge \text{air\_line\_company}(C) \wedge \text{owner}(C, \text{concorde} - 7) \wedge \text{plane}(\text{concorde} - 7)$. Now, the independent part is empty and one of the literals containing the constant `concorde-7`, say `owner(C,concorde-7)` will be posed to the oracle. Assume the oracle returns 'yes' with the single substitution $\theta = \{ C = \text{air\_france} \}$, where `air_france` is not a new object. The new goal is $\leftarrow \text{air\_line\_company}(\text{air\_france}) \wedge \text{works\_for}(\text{jeff}, \text{air\_france}) \wedge \text{plane}(\text{concorde} - 7)$. Assuming the independent literals, `air_line_company(air_france)` and `works_for(jeff, air_france)` are implied by the knowledge base, the remaining goal is $\leftarrow \text{plane}(\text{concorde} - 7)$, which is then passed to the oracle. The initial query succeeds if the oracle returns 'yes'; it fails otherwise.

Example 2: illustrating dependency.

implied. During the session, the system also acquires knowledge about other objects and predicates.

In Algorithm 3, L contains the list of goals g fulfilling the property that if g is true in the intended interpretation, there is a clause in the knowledge base that covers the fact $p(o_1, \dots, o_n)$.

Definition 3.3: A clause c covers a fact f if there is a substitution θ such that $\text{head}(c)\theta = f$ and $\text{body}(c)\theta$ is true in the intended interpretation.

For example, $\text{vehicle}(X) \leftarrow \text{car}(X)$ covers `vehicle(rolls-royce)` provided that `car(rolls-royce)` is true in the intended interpretation.

F is the set of facts, that are false in the intended interpretation. These facts are obtained by answers to previous oracle questions. The procedure `Handle_View` first initializes the list L with the goals corresponding to the instantiated bodies of clauses in the knowledge base.

In the repeat-loop, the most interesting goal (see Section 3.3. and below) is deleted from L . The inner part of the repeat loop verifies whether the goal contributes to the coverage of the fact. Clearly, if one of the literals in such a goal is false in the intended interpretation (i.e. belongs to the set F), then it cannot contribute to the coverage of the fact. As in Example 2 and Algorithm 2, to determine the truthvalue of the goal, the independent part of the goal is solved by querying the knowledge base. If there are no independent subgoals in the goal, a dependent literal is used to generate a question to the oracle. If such a literal concerns a stable predicate, it is assumed to be false in the intended interpretation. Otherwise the user is queried for the substitutions θ making the subgoals true in the intended interpretation. These substitutions are then propagated to the remainder of the original goal, which is then added to L . Notice that in case there is more than one true substitution for the selected subgoal(s), more than one goal will be added to L .

The heuristic to delete the most interesting goal from L will always select goals according to the clauses from which they are derived. In this way, the heuristic considers the goal corresponding to the body of one clause, and continues to


```

procedure Handle_View ( $p(o_1, \dots, o_n) : \text{true fact containing a new object } o_i$ )
   $L := \{ \text{body}(c) \mid c \text{ is a clause for } p \text{ in KB and}$ 
     $\text{most general unifier}(\text{head}(c), p(o_1, \dots, o_n)) = p \}$ 
   $M := \emptyset$ 
  repeat
    delete the most interesting goal  $g$  from  $L$ 
    if not  $(\exists l' \in g : \exists \tau, f \in F : l' = f\tau)$ 
    then  $g' := g - \{l\}$  such that  $l \in I(g)$  if possible
       $S := \text{Query}(l, g)$ 
      for all  $\theta \in S$  do
        add  $g'\theta$  to  $L$ 
        add  $l\theta$  to  $M$ 
      endfor
    endif
  until  $L$  contains the empty goal or  $L = \emptyset$ 
  for all  $l \in M$  do
    call Addfact( $l$ )
  endfor
endproc

procedure Query( $l : \text{literal}, g : \text{goal}$ ) returns : set of substitutions
  if  $l \in I(g)$ 
  then return  $\{ \theta \mid \text{KB} \models l\theta \}$ 
  else if  $l$  corresponds to a stable predicate
  then return  $\emptyset$ 
  else ask  $l$  to the oracle
    if oracle answers 'no'
    then add  $l$  to  $F$ 
    else return  $\{ \theta \mid \theta \text{ returned by the oracle} \}$ 
    endif
  endif
endif
endproc

```

Algorithm 3: view updates.

work on subgoals of this clause until a solution has been found or it is certain that the clause does not cover the fact. At that point, a new clause will be selected according to the heuristic presented in Section 3.3. So, within one clause, a depth-first strategy is realized and in between different clauses, a best-first strategy.

If the repeat loop stops on the condition $L = \emptyset$, it means that there is no clause in the knowledge base covering the fact. At that point, the system will fail when the predicate p was declared view and call the procedure **Handle_Base** when the predicate is mixed. Alternatively, instead of failing for a view predicate, one could also make the predicate a mixed predicate and call the procedure **Handle_Base**. Notice that after the repeat loop **Addfact** is called to assure that the facts obtained by querying the user will also be implied:

3.2. Updating base predicates

3.2.1. The type assumption We first define the very important notion of a type. Types in our approach are treated as in **Blip** (Kietz 1988, Morik 1989, Wrobel 1989).

Definition 3.4.: *The type T_p^i , the type corresponding to the i th argument position of the predicate p is defined as follows:*

$T_p^i = \{o \text{ is an object} \mid \exists x^1, \dots, x_{i-1}, x_{i+1}, \dots, x_n \text{ for which } KB \models p(x_1, \dots, x_{i-1}, o, x_{i+1}, \dots, x_n)\}$. We will assume that all types are finite!

So, the type T_p^i is the set of all objects occurring at the i th position in a fact for p , implied by the knowledge base. Since types are sets of objects, they are partially ordered by the subset relation. Hence, they can be represented in a type-graph with at the top the universal type, containing all possible objects and at the bottom the empty type, containing no objects. We will assume here that types are represented in a graph structure, as in **Blip**. The explicit representation of types from a knowledge base can be automatically computed using the techniques of Kietz (1988). Furthermore, as in **Blip**'s approach, we assume that non-empty intersection types are explicitly represented in the graph.

Definition 3.5: *An intersection-type T is a type such that $T = T_1 \cap T_2$ where T_1 and T_2 are either intersection-types or types corresponding to an argument position of a predicate.*

Definition 3.6: *The type-graph of a knowledge base represents all types corresponding to an argument position of a predicate, the empty type, the universal type and all resulting intersection-types.*

The advantage of explicitly representing intersection-types is that it is easy to determine the supertype(s) and subtype(s) of a certain type and to decide whether two types are disjoint or overlapping. Different types representing the same set of objects are called equivalent; they label the same node of the type graph.

Definition 3.7: *A supertype S of a type T w.r.t. a certain type-graph G is a type occurring in G such that S is a proper superset of T .*

Definition 3.8: *A subtype S of a type T w.r.t. a certain type-graph G is a type occurring in G such that S is a proper subset of T .*

In Example 3 and Figure 1, we illustrate types.

The main assumption underlying Algorithm 4 is the following:

Assumption 2: *If a new object o is inserted in type T of the type-graph, there is a high probability that it should also be inserted in the supertypes of T .*

Notice that this assumption is a plausible one. Given the type-graph of Figure 1, it states for instance that when a new car is encountered, it is likely that it is also an element of T_{owns}^2 . So, it is interesting to ask for the owner of the car. The assumption will be satisfied when the type-graph is stable enough to represent the true relations among types of the domain. Clearly, the more stable the type-graph, the more clever questions Algorithm 4 will generate. The assumption explains why Algorithm 4 queries the oracle for the truthvalue of $q(x_1, \dots, x_{j-1},$

The types for the knowledge base of Example 1 are the following:

$$T_{\text{owns}}^1 = \{ \text{yves, katharina, maurice, luc} \}$$

$$T_{\text{owns}}^2 = \{ \text{peugeot, mercedes, renault, plume} \}$$

$$T_{\text{car}}^1 = \{ \text{peugeot, mercedes, renault} \}$$

$$T_{\text{bicycle}}^1 = \{ \text{plume} \}$$

$$T_{\text{vehicle}}^1 = T_{\text{owns}}^2$$

$$T_{\text{drives}}^1 = T_{\text{owns}}^1$$

$$T_{\text{drives}}^2 = T_{\text{owns}}^2$$

$$T_{\text{has\_license}}^1 = \{ \text{yves, katharina, maurice} \}$$

Furthermore, we have a universal type and the empty type:

$$T_{\text{universal}} = \{ \text{yves, katharina, maurice, luc, peugeot, mercedes, renault, plume} \}$$

$$T_{\text{empty}} = \{ \}$$

There are also some intersection types. However, these do not give rise to additional types. Indeed, all intersection types as e.g. $\{ \text{plume} \}$ already occur as a regular type. The resulting type-graph is shown in Figure 1.

Example 3: types.

Figure 1. The type-graph of the knowledge base in Example 2.

$o, x_{j+1}, \dots, x_n$) if o has to be inserted in a type $T \subset T_q$. Algorithm 4 is more complicated than this because different kinds of predicates and types have to be taken into account.

3.2.2. The type algorithm Algorithm 4 works as follows: for each new object o_i occurring in the given fact, the type where the object has to be inserted is located. Following the assumption, the algorithm verifies whether the object also has to be inserted in its supertypes and later also in its subtypes. In order to do so, the procedures **Supertypes** and **Subtypes** (see Algorithm 5) are invoked. Also, after these procedures terminate, the algorithm checks the equivalent types (types representing the same set of objects).

Let us first discuss the procedure **Supertypes**. This procedure moves higher and higher in the type-graph and generates questions about those types for which it is not known whether the considered object should belong to these types or not.

```

procedure Handle_Base( $p(o_1, \dots, o_n)$  : true fact containing a new object  $o_i$ )
  assert  $p(o_1, \dots, o_n)$ 
  for all positions  $i$  for which  $o_i$  is a new object in  $p(o_1, \dots, o_n)$  do
    call Supertypes( $T_p^i, o_i$ )
    call Subtypes( $T_p^i, o_i$ )
    for all  $T_q^i$  such that  $T_q^i$  is equivalent with  $T_p^i$  and  $T_q^i$  is not an intersection-type and
    Unknown( $T_q^i, o$ ) and modifiable( $q$ ) do
      ask  $q(X_1, \dots, X_{j-1}, o, X_{j+1}, \dots, X_k)$  to the oracle
      if oracle returns 'no'
        then add  $q(X_1, \dots, X_{j-1}, o, X_{j+1}, \dots, X_k)$  to  $F$ 
      else for all  $\theta$  returned by the oracle do
        assert  $q(X_1, \dots, X_{j-1}, o, X_{j+1}, \dots, X_k)\theta$ 
      endfor
    endif
  endfor
endfor
endproc

```

Algorithm 4: realizing base updates.

The answers of the oracle to the questions are treated as usual. For intersection types no queries are generated about the intersection type itself but rather about its ancestors. It is known (cf. Algorithm 6) whether an object belongs to a type, when:

- the object has already been considered for membership of the type;
- the type corresponds to a view predicate; the reason for this is that questions concerning view predicates are postponed as much as possible as facts for these predicates may become true after asserting facts for base predicates;
- the corresponding facts are known to be false in the intended interpretations (using the set F); or
- the current knowledge base already implies that the object should belong to the type.

Notice that the graph traversal is not pursued for supertypes of R_q^i for which the oracle has returned answers. The point is that this will be done by the new call to **Handle_Base**. (**Unknown** immediately returns false when revisiting a node; so, the overhead of starting another graph traversal is small.)

The procedure **Subtypes** is analogous to **Supertypes**, except that it moves downwards in the type-graph and for the intersection types. Also, upon a negative answer from the oracle, the system will not consider the subtypes of the considered type. Indeed, if an object does not belong to a type and the type-graph is stable, it will not belong to its subtypes. Types that are disjoint from a type that is known to contain the novel object are pruned from further consideration. This prevents the system from asking whether someone is male, when the system knows that the person is female. Intersection types are handled as follows: if an object is not yet in the intersection type, the minimal supertypes of the intersection type are


```

procedure Supertypes(T : type, o : object)
  L := { S | S is a minimal supertype of T }
  while L ≠ { Universal Type } do
    delete the most interesting type  $R_q^i$  ( $\neq$  Universal Type )
    if  $R_q^i$  is no intersection-type and Unknown( $R_q^i$ , o) and modifiable(q)
    then ask q(X1, ..., Xj-1, o, Xj+1, ..., Xn) to the oracle
      if oracle returns 'no'
      then add q(X1, ..., Xj-1, o, Xj+1, ..., Xn) to F
        add the minimal supertypes of  $R_q^i$  to L
      else for all  $\theta$  returned by the oracle do
        call Handle_fact_with_new_object(q(X1, ..., Xj-1, o, Xj+1, ..., Xn) $\theta$ )
      endfor
    endif
  else add the minimal supertypes of  $R_q^i$  to L
  endif
endwhile
endproc

procedure Subtypes(T : type, o : object)
  L := { S | S is a maximal subtype of T }
  while L ≠ { Empty Type } do
    delete the most interesting type  $R_q^i$  ( $\neq$  Empty Type ) from L
    if  $R_q^i$  is no intersection-type and Unknown( $R_q^i$ , o) and modifiable(q)
    then ask q(X1, ..., Xj-1, o, Xj+1, ..., Xn) to the oracle
      if oracle returns 'no'
      then add q(X1, ..., Xj-1, o, Xj+1, ..., Xn) to F
      else for all  $\theta$  returned by the oracle do
        call Handle_fact_with_new_object(q(X1, ..., Xj-1, o, Xj+1, ..., Xn) $\theta$ )
      endfor
      Remove types from L that are disjoint with  $R_q^i$ 
    endif
    else if  $R_q^i$  is intersection-type
    then if o ∈  $R_q^i$ 
      then add the maximal subtypes of  $R_q^i$  to L
      else add the minimal supertypes of  $R_q^i$  to L
      endif
    else add the maximal subtypes of  $R_q^i$  to L
    endif
  endwhile
endproc

```

Algorithm 5: subroutines.

visited (i.e. the overlapping types). Addition of the object to one of these minimal supertypes will trigger another visit to the intersection type through a new call to **Handle_fact_with_new_object**. Eventually, when the object belongs to all these minimal supertypes, it will also belong to the intersection types and the subtypes of the intersection types will be visited.

```

procedure Unknown( $T_q^i$  : type,  $o$  : object) returns : boolean
  if  $o$  has already been considered for membership of type  $T_q^i$ 
    or  $q$  is view
  then return false
endif
if  $\exists \tau, f \in F : q(X_1, \dots, X_{j-1}, o, X_{j+1}, \dots, X_n) = f\tau$ 
then return false
endif
if  $\exists \theta : KB \models q(X_1, \dots, X_{j-1}, o, X_{j+1}, \dots, X_n)\theta$ 
then return false
endif
return true
endproc

```

Algorithm 6: subroutines.

3.3. A heuristic to select the most interesting candidates first

To be able to select the most interesting candidates first in the above algorithms, the system keeps track of a set $analog(n)$ of analogous objects o for each new object n . Furthermore, for each object $o \in analog(n)$ it computes a score, consisting of two numbers: the number of tests t on analogy in which the analogous object was involved and the number of times st a test has been successful. For each answer to an oracle question (the initial new term is treated as an answer to an oracle question) $q(x_1, \dots, x_{j-1}, n, x_{j+1}, \dots, x_k)$ involving the new object n , the set $analog(n)$ is modified as follows:

If the answer was ‘yes’ for a substitution θ , then (1) the set $analog(n)$ is augmented with all objects o , that did not yet occur on the set and for which $KB \models q(x_1, \dots, x_{j-1}, o, x_{j+1}, \dots, x_k)\theta$ holds. The number of tests t for these objects o is initialized to 0. (2) The number of successful tests st of all objects o on the new set $analog(n)$ satisfying $KB \models q(x_1, \dots, x_{j-1}, n, x_{j+1}, \dots, x_k)\theta$, is increased by 1. (3) The number of tests t of all objects on the set is increased by 1.

If the oracle’s answer was ‘no’, then the number of tests t of all members of the set is increased by 1 and the number of successful tests st of all objects o for which there is no θ such that $KB \models q(x_1, \dots, x_{j-1}, x_{j+1}, \dots, x_k)\theta$, is increased by 1.

By taking subsets of the set of analogous objects, it becomes possible to detect the most interesting candidate. In order to do so, one takes a subset corresponding to each candidate and calculates the average ratio st/t for each considered candidate. The most interesting candidate is then the one whose average ratio is maximal.

In Algorithm 3, the heuristic can be used to choose among the clauses. This is done by associating to each clause the subset of objects from $analog(n)$ that are provable using that clause.

For the selection of the most interesting subtype in Algorithm 5, the same method is used. The subset is then the intersection of the set $analog(n)$ with the type itself. The heuristic is only applied when the candidate subtypes are disjoint. The reason is that when types overlap, the system will consider the other alternatives anyway (cf. procedure **Supertypes**).

The heuristic works very well in practice. Its only drawback is that when there are many elements in the set, it requires much computation. Of course, the number of elements in the set could be kept within certain bounds, by storing only a representative sample of the analogous objects. (See Example 4.)

4. An example session with the system.

The algorithms specified in Section 3, are implemented in BIMprolog. Below, we show a trace of a session with the implemented system. Part of the knowledge base used in the session is shown in Example 5 and part of the type-graph is shown in Figure 2. Comments are in *italic*, and user input is in **bold**.

?-Addfact(is_allowed_to_fly(john,concorde)).

Reconsider Example 1 and assume that we wish to insert the fact **drives(yves, rolls-royce)**. *analog(rolls-royce)* is initialized with the objects in T_{drives}^2 , i.e. **peugeot**, **mercedes**, **renault** and **plume**. Initially, both *t* and *st* are 0 for these objects. Because **drives(yves, rolls-royce)** holds, *t* is increased by 1, while *st* is increased by 1 only for **peugeot** (**drives(yves, ○)** only holds for $\bigcirc = \text{peugeot}$). So, *analog(rolls-royce)* becomes { **peugeot (t:1, st:1)**, **mercedes (t:1, st:0)**, **renault (t:1, st:0)**, **plume(t:1, st:0)** }. To choose among the clauses in Algorithm 2, the system assigns { **peugeot (t:1, st:1)**, **mercedes (t:1, st:0)**, **renault (t:1, st:0)** } to the first clause for **drives** and { **plume (t:1, st:0)** } to the second one. The resulting score for the first clause is 1/4 and for the second one 0. This means that the system would proceed with the first clause. As **has_license(yves)** holds, the system still needs to verify **owns(yves, rolls-royce)** and **car(rolls-royce)**. Now, it can first ask **car(rolls-royce)** and then **owns(yves, rolls-royce)**. The quality of the recommendations made by the heuristic increases with the number of tests.

Example 4: illustrating the heuristic.

```

is_allowed_to_fly(X,Y) ← person(X) ∧ owner(X,Y) ∧ has_license(X) ∧
    plane(Y)
is_allowed_to_fly(X,Y) ← person(X) ∧ has_license(X) ∧ works_for(X,C)
    ∧ air_line_company(C) ∧ owner(C,Y) ∧ plane(Y)
is_allowed_to_fly(X,Y) ← person(X) ∧ works_for(X,C) ∧ air_force(C) ∧
    owner(C,Y) ∧ military_plane(Y)

person(X) ← man(X)
person(X) ← woman(X)

plane(X) ← passengers_plane(X)
plane(X) ← military_plane(X)
plane(X) ← carrier(X)

```

The other relevant part of the knowledge base for this session, contains only facts.

Example 5: part of the knowledge base used in the session.

Figure 13. The type-graph of the knowledge base in Example 2.

?-Addfact(is_allowed_to_fly(john,concorde)).

is_allowed_to_fly(john,concorde) is a fact with new objects - john - concorde

level 0 : Handle_View(is_allowed_to_fly(john,concorde)) - clause 2

The analogy heuristic identifies the second clause as the most interesting one, and the system proceeds.

Is person(john) true? **yes**.

person(john) is a fact with new object - john
added to M0

Is has_license(john) true? **yes**.

has_license(john) is a fact with new object - john
added to M0

Is works_for(john,X) true for some X? **X = air_france**.

Is works_for(john,X) true for some other X? **no**.

works_for(john,air_france) is a fact with new object - john

air_france is a known object

added to M0

air_line_company(air_france) implied

Is owner(air_france,concorde) true ? **yes**.

owner(air_france,concorde) is a fact with new object - concorde
added to M0

Is plane(concorde) true ? **yes**.

added to M0

clause 2 correctly covers is_allowed_to_fly(john,concorde)

Removing goals from M0 :

level 1 : Handle_View(person(john)) - clause 1

Is man(john) true ? **yes**.

man(john) is a fact with new object - john
added to M1

clause 1 correctly covers person(john)

Removing goals from M1:

level 2 : Handle_Base(man(john))

man(john) asserted ...

No unknown supertypes (T_{person}^1 known)
 Subtypes are intersection-types
 Intersection types overlap with $T_{is\_allowed\_to\_fly}^1$, $T_{has\_license}^1$
It is assumed here that there is no overlap with $T_{works\_for}^1$; this is due to the knowledge in the knowledge base.
As it is known that john belongs to all these types, it also belongs to the intersection-types.
However, the only subtype of these intersection-types is the empty type.
So, procedure Subtypes terminates.
 john belongs to intersection-types $T_{is\_allowed\_to\_fly}^1$, $T_{has\_license}^1$
 No unknown subtypes
 end level 2
 M1 = empty
 level 1 : Handle_Base(has license(john))
 has_license(john) asserted ...
 No unknown supertypes (T_{person}^1 known)
 No unknown subtypes
 end level 1
 level 1 : Handle_Base(works_for(john,air_france))
 works_for(john,air_france) asserted ...
 No unknown supertypes (T_{person}^1 known)
 No unknown subtypes
 end level 1
 level 1 : Handle_Base(owner(air_france,concorde))
 owner(air_france,concorde) asserted ...
 No unknown supertypes
 Known subtype T_{plane}^1
 Subtypes are $T_{passengers\_plane}^1$, $T_{military\_plane}^1$ and $T_{carrier}^1$
 Is passengers_plane(concorde) true ? **yes**.
 passengers_plane(concorde) is a fact with new object - concorde
 level 2 : Handle_Base(passengers_plane(concorde))
 passengers_plane(concorde) asserted ...
 No unknown supertypes (T_{plane}^1 , T_{owner}^2 known)
 Subtypes are intersection-types
 Intersection types overlap with $T_{subsonic}^1$, T_{jet}^1
 Unknown whether concorde belongs to $T_{subsonic}^1$, T_{jet}^1
 Is subsonic(concorde) true ? **no**.
 Is jet(concorde) true ? **yes**.
 jet(concorde) is a fact with new object - concorde
 level 3 : Handle_Base(jet(concorde))
 jet(concorde) asserted ...
 No unknown supertypes
 Supertypes T_{plane}^1 and T_{owner}^2 known
 Subtype $T_{supersonic}^1$ unknown
 Is supersonic(concorde) true ? **yes**.
 supersonic(concorde) is a fact with new object - concorde
 level 4 : Handle_Base(supersonic(concorde))
 supersonic(concorde) asserted ...
 No unknown supertypes (T_{jet}^1 , T_{jet}^1 , and T_{owner}^2 known)


```

    No unknown subtypes
  end level 4
  No further subtypes considered
end level 3
  No further subtypes considered
end level 2
  Removing disjoint types  $T_{\text{military\_plane}}^1$  and  $T_{\text{carrier}}^1$ 
  No further subtypes considered
end level 1
plane(concorde) implied
This goal was still on M0.
M0 = empty
end level 0

```

5. Conclusions and related work

The presented work is related to work in learning apprentices such as **Disciple** (Tecuci and Kodratoff 1990, Kodratoff and Tecuci 1987) and **Blip** (Morik 1989, Wrobel 1989), learning by being told (Haas and Hendrix 1983) and knowledge base revision (Tomasic 1988).

The learning apprentices **Disciple** and **Blip** have some facilities to help the user to input new facts. Basically these facilities are such that the system shows the user some situation and that the user is required to enter some new facts about this situation. In **Disciple** this occurs when the system is in its weak theory mode and that part of an explanation is needed by the system. If the system is unable to generate this explanation the user is supposed to help the system by inputting some facts. In **Blip** the user inputs facts and gets immediate feedback about the consequences of the input facts. If the user notices that not all desired facts concerning a certain object are consequences she can enter additional facts.

The type formalism used in this paper is the same as in **Blip** (Kietz 1988). In **Blip**, the type graph is automatically modified when the user enters new facts. Also, if the type structure should be modified, the user is told so. In this way the user is provided with additional hints about possibly missing knowledge (Kietz 1988).

Haas and Hendrix (1983) present an approach to interactively acquire knowledge about a certain domain. Their approach gathers knowledge in a very similar way to ours. However, there are also some important differences: they have a natural language interface, determine the structure of the type graph by asking questions to the user and use a rather specific knowledge representation formalism. Furthermore it is not clear how to integrate their approach in concept-learning systems because of the different formalism.

The approaches to knowledge base updating such as Tomasic (1988) attempt to automatically find the transactions needed to accommodate new facts in the data base. However, they do not try to obtain as much knowledge as possible about the new objects.

As a conclusion, our technique is different from the above-mentioned ones in the sense that it attempts to acquire in a systematic way as much knowledge as possible about new objects. In this respect, we go further than the knowledge base updating and learning apprentice approaches. Also because we automatically acquire the type graph as in Kietz (1988) and show how to integrate our methods

in existing concept-learners, we believe that the presented method integrated with a system as *Clint* is very useful for learning apprentices.

Acknowledgements

This work was inspired during Luc's stay in Katharina Morik's group at the GMD in the summer of 1989.

References

- Angluin, D. (1987) Queries and concept-learning, *Machine Learning*, 2:319–342.
- Bratko, I. (1986) *Prolog Programming for Artificial Intelligence* (Wokingham: Addison-Wesley)
- Bry, F. (1990) Intensional updates: abduction via deduction. In D. Warren and P. Szeredi (eds). *Proceedings of the 7th International Conference on Logic Programming*, 561–578.
- Carbonell, J. G., Michalski, R. S., and Mitchell, T. M. (1983) An overview of machine learning. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (eds) *Machine Learning: An Artificial Intelligence Approach*, volume 1 (Palo Alto, CA: Morgan Kaufmann) pp. 3–24.
- De Raedt, L. and Bruynooghe, M. (1989) Towards friendly concept-learners. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 849–856.
- De Raedt, L. and Bruynooghe, M. (1992) Belief updating from integrity constraints and queries, *Artificial Intelligence*, 53: 291–307.
- De Raedt, L. and Bruynooghe, M. (1992) Interactive concept-learning and constructive induction by analogy. *Machine Learning*, 8:107–150.
- Gaines, B. and Boose, J. (Editors) (since 1989) *The knowledge acquisition journal*. (London: Academic Press).
- Genesereth, M. and Nilsson, N. (1987) *Logical Foundations of Artificial Intelligence* (Los Altos, CA: Morgan Kaufmann).
- Guessoum, A. and Lloyd, J. W. (1990) Updating knowledge bases, *New Generation Computing*, 8:71–88.
- Haas, N. and Hendrix, G. (1983) Learning by being told : acquiring knowledge for information management. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (eds) *Machine Learning : An Artificial Intelligence Approach*, volume 1 (Los Alto, CA: Morgan Kaufmann) pp. 405–428.
- Kietz, J. U. (1988) Incremental and reversible acquisition of taxonomies. In M. Linster, B. Gaines, and J. Boose (eds) *Proceedings of the 2nd European Knowledge Acquisition for Knowledge Based Systems Workshops*, 1988, Paris.
- Kodratoff, Y. (1988) *Introduction to Machine Learning* (London: Pitman).
- Kodratoff Y. and Michalski, R. S. editors (1990) *Machine Learning: An Artificial Intelligence Approach*, Volume 3 (San Mateo, CA: Morgan Kaufmann).
- Kodratoff Y. and G. Tecuci (1987) Disciple-1: interactive system in weak theory fields. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pp. 272–273.
- Michalski, R. S. (1983) A theory and methodology of inductive learning. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (eds) *Machine Learning: An Artificial Intelligence Approach*, Volume 1 (Palo Alto, CA: Morgan Kaufmann). pp. 83–134.
- Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (1983) *Machine Learning: An Artificial Intelligence Approach*, Volume 1 (Palo Alto, CA: Morgan Kaufmann).
- Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (1986) *Machine Learning: An Artificial Intelligence Approach*, Volume 2 (Los Alto, CA: Morgan Kaufmann).
- Morik, K. (1989) Sloppy modeling. In K. Morik (ed) *Knowledge Representation and Organization in Machine Learning*, volume 347 of *Lecture Notes in Artificial Intelligence*, (Berlin: Springer-Verlag). pp. 107–134.
- Muggleton, S. and Buntine W. (1988) Machine invention of first order predicates by inverting resolution. In *Proceedings of the 5th International Conference on Machine Learning*, 339–351.
- Quinlan, J. R. (1986) Induction of decision trees, *Machine Learning*, 1:81–106.
- Quinlan, J. R. (1990) Learning logical definitions from relations, *Machine Learning*, 5:239–266.
- Shapiro, E. Y. (1986) *Algorithmic Program Debugging* (Cambridge, MA: The MIT Press).
- Sterling, L. and Shapiro, E. (1986) *The Art of Prolog* (Cambridge, CA: The MIT Press)
- Tecuci, G. and Kodratoff, Y. (1990) Apprenticeship learning in non-homogeneous domain theories. In Y. Kodratoff and R. S. Michalski (eds) *Machine Learning: An Artificial Intelligence Approach*, volume 3 (San Malto, CA: Morgan Kaufmann). pp. 514–551.
- Tomasic, A. (1988) View update translation via deduction and annotation. In *Proceedings 2nd International Conference on Database Theory*, volume 326 of *Lecture Notes in Computer Science*, 338–351 (Berlin: Springer-Verlag). pp. 338–357.

- Wirth, R. (1988) Learning by failure to prove. In D. Sleeman (ed) *Proceedings of the 3rd European Working Session on Learning* (London: Pitman) pp. 237–252.
- Wrobel, S. (1988) Automatic representation adjustment in an observational discovery system. In Sleeman D. (ed) *Proceedings of the 3rd European Working Session on Learning*, (London: Pitman) pp. 253–261.
- Wrobel, S. (1989) Demand driven concept-formation. In K. Morik (ed) *Knowledge Representation and Organization in Machine Learning*, volume 347 of *Lecture Notes in Artificial Intelligence* (Berlin: Springer-Verlag) pp. 289–318.