Machine Learning, 8, 107-150 (1992)

Interactive Concept-Learning and Constructive Induction by Analogy

LUC DE RAEDT (LUCDR@CS.KULEUVEN.AC.BE)
MAURICE BRUYNOOGHE
Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium

Editor: Katharina Morik

Abstract. The available concept-learners only partially fulfill the needs imposed by the learning apprentice generation of learners. We present a novel approach to interactive concept-learning and constructive induction that better fits the requirements imposed by the learning apprentice paradigm. The approach is incorporated in the system Clint-Cia, which integrates several user-friendly features into one working whole: it is interactive, generates examples, shifts its bias, identifies concepts in the limit, copes with indirect relevance, recovers from errors, performs constructive induction and invents new concepts by analogy to previously learned ones.

Keywords. Inductive logic programming, concept-learning, constructive induction, experimentation

1. Introduction

One of the achievements of early work in machine learning (Michalski, Carbonell & Mitchell, 1983) was the development of concept-learners that operated in isolation, relied heavily on the user and addressed only well defined and restricted tasks. Today, the need for intelligent knowledge acquisition tools forms the main motivation for the development of a new generation of learning systems, the so called learning apprentices. Systems that fall in this category are for example Disciple (Kodratoff & Tecuci, 1987; Tecuci & Kodratoff, 1990) learning preconditions of actions and object models, Blip (Morik, 1989; Wrobel, 1988; Emde, 1989; Thieme, 1989) modeling the world by observation, Prodigy (Minton, 1988) and Leap (Mitchell, Mahadevan & Steinberg, 1985) improving their problem-solving performance. According to Smith, Mitchell, Winston and Buchanan (1985) a *learning apprentice* is an interactive aid for building and refining a knowledge base during normal operation of the system. Concept-acquisition is often a major subtask in knowledge acquisition and therefore also in learning apprentices. In order to meet the requirements imposed by the learning apprentice approach we developed a novel approach to concept-learning, which will be presented here. More in particular, we shall present the interactive concept-learner Clint (Concept-Learning in an INTeractive way) (De Raedt & Bruynooghe, 1988; De Raedt & Bruynooghe, 1990a; De Raedt & Bruynooghe, 1989c; De Raedt & Bruynooghe, 1989d) together with its companion system Cia (Constructive Induction by Analogy) (De Raedt & Bruynooghe, 1989a; De Raedt & Bruynooghe, 1989b).

The two major requirements for concept-learners in the learning apprentice context are (1) user-friendliness and (2) opportunism. A system is *user-friendly* if it can be used in a simple manner by users that are not familiar with the underlying techniques and *opportunistic* if it takes the initiative to learn whenever there is a possibility to do so. Also, the opportunities should be exploited as much as possible in order to maximize the gained knowledge. To realize these characteristics, we attempted to integrate many user-friendly features into Clint. Clint is an incremental closed-loop system that learns from examples, uses knowledge, generates examples, copes with indirect relevance, shifts its bias, recovers from errors and identifies concepts in the limit. Although Clint is able to change its representation from one description language to another, it only learns given concepts from examples. This is a weakness, shared with most inductive concept-learners, because it cannot identify relevant new concepts that are of use to the user and the apprentice. The problem of automatically identifying relevant new concepts is often referred to as the problem of constructive induction. Systems that only learn given concepts assume that an important part of the concept learning task, i.e., the choice of the representation and vocabulary of the learner, has already been solved. It is often argued that constructive induction is the main problem with current approaches to concept learning. Recently, some authors have begun to address it (Muggleton, 1987; Muggleton & Buntine, 1988; Rendell; 1989; Mehra, Rendell & Wah, 1989; Wrobel, 1988). Especially in the learning apprentice context, it is important to let the system suggest useful new concepts to the user. We studied the problem of constructive induction in the framework of Clint and have enhanced Clint with the Cia technique. In a sense, Cia can be seen as a (concept-description) language learning assistant, which interactively proposes new concepts for inclusion in the knowledge base and the concept-description language. As such Cia's aim is different from the usual constructive induction techniques as advocated by the Rendell-school (Rendell, 1989; Mehra, Rendell & Wah, 1989). The Rendell-like approaches attempt to derive new concepts from existing ones with the aim of better being able to learn one given concept. The Cia approach tries to invent new concepts that are of general interest to the user, without reference to the goal of learning one specific concept. In this respect, Cia is closer to the work of Lenat (1983). A second issue which is addressed by our Cia technique is the application of machine learning techniques on the machine learning process itself (Stepp, Whitehall & Lawrence, 1988). This research investigates the development of learning learning machines, i.e., machines that adapt their learning behavior over time. The key idea behind the Cia-technique is very simple: when concept-learning is viewed as a process that transforms an input (some examples) into an output (a concept-description) then one can use analogy to solve future concept-learning problems. The analogous method analyzes the result of the source concept-learning problem by deriving a concept-schema from it and tries to apply the schema in future concept-learning situations. The schema is not only used to solve concept-learning problems by analogy but also to invent new concepts.

This paper is organized as follows: in Section 2, a framework for the learning task is defined; in Section 3, the algorithms of Clint are presented; an experiment with Clint is discussed in Section 4; in Section 5, the constructive induction by analogy is introduced; in Section 6, we report on some experiments and finally, in Section 7, we conclude and discuss related work.

2. A framework for the learning task

2.1. Logic and concept-learning

Concept-learning refers to the process of generating a concept-description from examples. We use a logical framework (Genesereth & Nilsson, 1987):

Definition 1. *A concept is a predicate.*

Definition 2. *A concept-description is a set of (definite) Horn-Clauses defining a predicate.*

Definition 3. *Examples are ground instances of a predicate. Positive examples are true and negative examples are false.*

We restrict ourselves to function free predicates (see (De Raedt, 1991) for a more general setting). Moreover, literals are of the form $p(X_1, \ldots, X_n)$ where the X_i are distinct variables. This is not an additional restriction (see (Rouveirol & Puget, 1989)) because constants are allowed in the knowledge-base and an equality predicate `eq` is also available.[1] For example the clause `q(a, X, X)` ← rewrites to `q(X, Y, Z)` ← `A(X)` ∧ `eq(Y, Z)` where `A(a)` ← is in the knowledge-base.

Definition 4. *A concept-description covers an example if the example logically follows from the concept-description and the knowledge base.*

In practice the concept-learning process is not only driven by training-examples but also by bias (Utgoff & Mitchell, 1982), which is—following Utgoff and Mitchell—anything which influences how the concept-learner draws inductive inferences based on the training-examples. The significance of bias for concept-learning is widely recognized since Utgoff and Mitchell (1982) and Utgoff (1986). Although bias may be incorporated in a concept-learning system in a variety of ways, we will only consider bias in the form of restrictions on the description language. In our framework, we have that

Definition 5. *A bias is correct for learning a predicate from a knowledge base if there exists a set of Horn-clauses for the predicate in the description language associated with the bias, such that all positive and no negative examples are covered by the set of Horn-clauses and the knowledge base.*

Definition 6. *A concept is conjunctive if it can be defined with one Horn-clause. Otherwise it is disjunctive.*

2.2. Notions used by Clint-Cia

The knowledge-base, employed by Clint-Cia, contains two different types of predicates:

- *Basic* predicates are defined by the user and assumed to be correct. They are specified by arbitrary Horn-clauses.
- *Learned* predicates are defined by the learner and derived from opportunities.

We distinguish two kinds of opportunities:

- *example-driven* opportunities occur when the user inputs an uncovered positive or a covered negative example; they are handled by Clint
- *concept-driven* opportunities occur when the learner is able to propose interesting new predicate definitions to the user; they are handled by Cia

By processing the opportunities, the learner's knowledge base should gradually converge to the desired one.

2.2.1. Clint's framework

For each example-driven opportunity, the learner analyzes and augments its knowledge by asking membership questions to the user.

Definition 7. *A membership question asks for the classification of an example for a learned predicate.*

All questions generated by Clint and Cia have to be answered by an oracle that is supposed to answer these queries correctly. In practice, the oracle is the user. We will use the terms oracle and user as synonyms.

Horn-clauses for learned predicates satisfy one of the language-definitions in the learner's knowledge-base.

Definition 8. *A language-definition describes a set of syntactic restrictions, i.e., a bias, which must be satisfied by a Horn-clause in order to belong to the language.*

One such definition describes the restriction that all variables that occur in the body of a clause also have to occur in the head of the clause. Our concept-learner, Clint, uses a possibly infinite and ordered series of language-definitions $L_1, \ldots, L_n, \ldots$. For all languages L_i we have that $d(L_i) \subset d(L_{i+1})$, where $d(L)$ is the set of clauses that are expressible in L. This means that all clauses that can be formulated in L_i are also expressible in L_{i+1}. Furthermore, with a finite set of predicates in the knowledge base, each language has only a finite number of clauses and all clauses have a finite number of literals. In this way an infinite language (a subset of Prolog) is divided in an infinite number of finite languages.

The languages are dynamically computed from an example and the predicates in the knowledge base. Assume for instance that we have as a positive example `is_allowed_to_drive (katharina, mercedes)` and that the knowledge base is as shown in example 1. Because the knowledge base contains only the predicates `license`, `isa`, `owns` and `eq`, all literals in the body of clauses are renamings of

```
{ isa(X, Y), license(U, V), owns(W, Z), eq(Q, R) }
```

Example 1. A simple knowledge base

```
license(katharina, car) ←
license(yves, truck) ←
license(yves, car) ←
license(stephan, car) ←

isa(mercedes, car) ←
isa(dyane, car) ←
isa(eddy_merckx_bike, bike) ←
isa(peugeot, car) ←

owns(katharina, mercedes) ←
owns(stephan, eddy_merckx_bike) ←
owns(yves, peugeot) ←
owns(luc, dyane) ←
```

□

Because the example belongs to the predicate `is_allowed_to_drive`, we have that clauses in L_i are of the form:

`is_allowed_to_drive(X, Y)` ← *body*; where

body $\subset \bigcup_{j=0}^{j=i} S(j)$ for language L_i such that

- literals l in $S(0)$ satisfy $var(l) \subset$ {`X, Y`}, i.e., all variables occurring in the body of a clause of L_0 also occur in the head of the clause. Given our knowledge base, we have:

  $S(0)$ = { `license(X, Y), isa(X, Y), owns (X, Y), license(Y, X), isa(Y, X), owns(Y, X), eq(X, Y)` }

- literals in $S(1)$ satisfy the restriction that all variables except one occur in the head. The latter one is not allowed to occur elsewhere, i.e., the body of clauses in L_1 can have existentially quantified variables, however such variables only have a single occurrence. Given our example, we have (up to a renaming of the variables `Zi`):

  $S(1)$ = { `license(X, Z1), isa(X, Z2), owns(X, Z3), license(Y, Z4), isa(Y, Z5), owns(Y, Z6), license(Z7, X), isa(Z8, X), owns(Z9, X), license(Z10, Y), isa(Z11, Y), owns(Z12, Y)` }

  It does not make sense to consider literals of the form `eq(X, Zi)`, because the `Zi` may only have one occurrence anywhere else.

- literals in the $S(2)$ part of the clause are relations between the variables that occur in the union of the $S(0)$ and $S(1)$ part of that clause. Having the variables `Zi` in $S(1)$ this yields for our example

  $S(2)$ = { `license(A, B)` } ∪ { `isa(A, B)` } ∪ { `owns(A, B)` } ∪ { `eq(A, B)` }

  where `A` and `B` have to be either `X` or `Y` or one of the `Zi`.

  These languages will be further illustrated in example 3.

Observe that there are basically two operations in order to obtain the next language in the series: either allow for more existentially quantified variables or allow for relations between previously introduced variables. These two operations can be used to define many interesting series of languages. For more details and a full formal presentation of the language-series that can be used by Clint, we refer to De Raedt and Bruynooghe (1990a), De Raedt (1991), and De Raedt and Bruynooghe (1989c).

2.2.2. *Cia's framework*

For each concept-driven opportunity, a number of predicate definitions are proposed to the user who has to name the predicates if they are relevant for the domain and otherwise reject them. In order to propose interesting new concepts to the user, the system uses second order schemata:

Definition 9. *A second order schema is an abstracted Horn-clause, where predicate names are interpreted as existentially quantified predicate variables.*

One such abstracted Horn-clause or second order schema is, e.g.,

$$S = (\exists p, q, r: p(X, Y) \leftarrow q(X, XW) \wedge q(YW, Y) \wedge r(XW, YW))$$

Second order schemata were introduced in the learning literature by the Blip-system and its precursors (Morik, 1989; Wrobel, 1989; Thieme, 1989; Emde, Habel & Rollinger, 1983).

Definition 10. *A second order substitution is a substitution that replaces predicate-variables by predicate names.*

For schema S, $\Theta = \{p = \texttt{grandparent}, q = \texttt{parent}, r = \texttt{eq}\}$ is a second order substitution. The instantiated schema $S\Theta$ yields:

$$\texttt{grandparent(X, Y)} \leftarrow \texttt{parent(X, XW)} \wedge \texttt{parent(YW, Y)} \wedge \texttt{eq(XW, YW)}$$

The Cia-technique will propose a number of predicate definitions to the user. For example, it will ask the user whether p defined as $\exists p: p(X, Y) \leftarrow \texttt{parent}(X, Y) \wedge \texttt{female}(X)$ is an interesting predicate. If the user believes the predicate is of interest, she/he can name it as, e.g., mother and the resulting clause $\texttt{mother}(X, Y) \leftarrow \texttt{parent}(X, Y) \wedge \texttt{female}(X)$ will be asserted in the knowledge base. Another type of question that is generated by Cia is a question whether a certain clause is correct. As an example, Cia might ask the user, while attempting to learn the definition of grandparent, whether $\texttt{grandparent}(X, Y) \leftarrow \texttt{parent}(X, Z) \wedge \texttt{parent}(Z, Y)$ is correct. If the user confirms that the clause is correct, it will be asserted in the knowledge base. So, Cia uses two types of questions:

Definition 11. *A new term question asks to name a predicate p in a schema $\exists p: p(t_1, \ldots, t_n) \leftarrow q_1 \wedge \ldots \wedge q_k$, where p is the only predicate-variable in the schema. When the oracle returns the substitution $\Lambda = \{p = P\}$, where P is a predicate-name, the clause $P(t_1, \ldots, t_n) \leftarrow q_1 \wedge \ldots \wedge q_k$ is asserted in the knowledge base.*

Definition 12. *A clause question asks the oracle whether a clause c is correct. If the clause is correct it is asserted in the knowledge base.*

To generate interesting clause and new term questions, Cia matches schemata with clauses:

Definition 13. *A second-order schema S matches a clause c, notation match(S, c), if* $\exists\Theta$, ρ: $head(S)\Theta\rho = head(c)$ *and* $body(S)\Theta\rho \subset body(c)$, *where* Θ *is a second order substitution and* ρ *is a substitution that renames variables in S.*

Definition 14. *A second-order schema S partially matches a clause c, notation pmatch(S, c), if* $\exists\Theta$, ρ: $body(S)\Theta\rho \subset body(c)$, *where* Θ *is a second order substitution and* ρ *is a substitution that renames variables in S.*

In the future, we will not always write the substitutions ρ, which rename variables, explicitly. The reason is that these renamings are straightforward.

In the Cia technique, partial matches give rise to new term questions whereas complete matches result in clause questions. These definitions are illustrated in example 2.

Example 2. Matching schemata

The schema S matches the clause:

```
grandparent(F, C) ← male(F) ∧ male(C) ∧ parent(F, M1)
    ∧ parent(M2, C) ∧ eq(M1, M2)
```

with substitutions

$$\Theta = \{p = \texttt{grandparent}, q = \texttt{parent}, r = \texttt{eq}\}$$
$$\rho = \{\texttt{X} = \texttt{F}, \texttt{Y} = \texttt{C}, \texttt{XW} = \texttt{M1}, \texttt{YW} = \texttt{M2}\}.$$

The instantiated schema $S\Theta\rho$ is:

```
grandparent(F, C) ← parent(F, M1) ∧ parent(M2, C) ∧ eq(M1, M2)
```

The same clause partially matches the schema T:

$$T = (\exists p, q, r: p(\texttt{X}, \texttt{Y}) \leftarrow q(\texttt{X}) \wedge r(\texttt{X}, \texttt{Y}))$$

with substitution $\Lambda = \{q = \texttt{male}, r = \texttt{parent}\}$.

The partially instantiated schema $T\Lambda$ is:

$$\exists p: p(\texttt{F}, \texttt{M1}) \leftarrow \texttt{male}(\texttt{F}) \wedge \texttt{parent}(\texttt{F}, \texttt{M1})$$

As said before, we omit the substitution ρ that renames the variables. □

3. Interactive concept-learning

A realistic concept-learner for learning apprentice systems should at least have an incremental way of processing examples, an easy way to formulate examples, be able to formulate interesting questions and offer some convergence guarantees. This motivates the following design decisions for Clint:

- Examples are represented as ground facts. It is unnecessary to state explicitly, as in, e.g., Marvin (Sammut & Banerji, 1986), which properties of an example are important for making the example an instance of the concept. These properties should be determined by the learner although the oracle should be allowed to help the system by identifying some irrelevant properties for the concept.
- The description languages of the system are built-in and hidden to the user. She/he does not have to bother about the adequacy of a particular language since the learner is able to automatically shift the bias when necessary.
- In order to gain as much knowledge as possible from an example-driven opportunity, the system formulates membership questions. This has the advantage that the user does not need to choose appropriate training-sets. Also the number of needed examples is (partially) determined by the system.

The problem, solved by Clint, can be defined formally as:

- **Given:**
  - — a knowledge-base of basic and learned predicates
  - — for learned predicates, a set of positive and negative examples
  - — a series of language-definitions $L_1, \ldots, L_n, \ldots$
  - — an example-driven opportunity to learn, indicated by the oracle
  - — an oracle, willing to answer membership questions
- **Find:** an adapted knowledge-base such that all positive examples and no negative examples are covered

The two classes of example-driven opportunities, uncovered positive examples and covered negative examples, are processed in a different way (see algorithm 1). A session with Clint is shown in example 8. The reader may already want to have a look at it.

Algorithm 1. Clint-Cia

```
procedure Clint-Cia
    while the user wants to continue do
        receive an example e for a predicate p from the user
        add e to the set of examples
        call handle_example(p, e)
    endwhile
endproc
```

```
procedure handle_example(p: predicate, e: example)
    if e is positive and uncovered
    then call handle_uncovered(p, e)
    endif
    if e is negative and covered
    then call handle_covered(p, e)
    endif
endproc
```

3.1. Processing a positive example

When the learner receives an uncovered positive example, it constructs a new clause covering the example and adds it to the knowledge-base. To compute the clause, the learner constructs a starting clause, and generalizes it by asking membership questions. The algorithm to process an uncovered positive example is shown in handle_uncovered of algorithm 2 and is explained in the following subsections.

Algorithm 2. An uncovered positive example

```
procedure handle_uncovered(p: predicate, e: uncovered positive example)
    bias := 0
    repeat
        bias := bias + 1
        repeat
            find a starting clause sc in L_bias that has not been considered yet
        until all starting clauses in L_bias have been considered or
            sc does not cover negative examples
    until sc does not cover negative examples
    c := cia_part_1(sc); see algorithm 4;
    if c = void; cia did not find a correct concept-description
    then new_negatives := ∅
        c := sc
        repeat
            find a clause c' such that:
                (1) c' is derived from c by deleting
                    a set of elements S from body(c)
                (2) c' covers an example e'
                (3) c does not cover e'
                (4) c' does not cover negative examples
                (5) there is no c'' satisfying (1), (2), (3) and (4)
                    for which body(c') ⊂ body(c'')
            ask the oracle to classify e'
            if e' is positive
            then c := c'
```

```
            else e' is negative
            add e' to new_negatives
            endif
        until all possible clauses c' have been considered
        add c to the knowledge base
        derive a second-order schema from c and store it
        if there are clauses for p, different from c
        then for all examples e'' ∈ new_negatives do
                call handle_example(p, e'')
            endfor
        endif
    else add c to the knowledge base
    endif
    call cia_part_2(sc); see algorithm 4
endproc
```

3.1.1. Finding the right starting-clause

Relevance

As examples are ground facts, they only specify the objects which are directly involved and not the relevant relations among them, i.e., the properties that make the example an instance of the concept. All relations in the knowledge-base are potentially relevant. However, considering all these relations simultaneously is not feasible since this would very rapidly lead to combinatorial problems. Therefore, our learner considers only a certain set of relations at a time. Such a set of relations is a justification, which corresponds to an instance of the body of a starting clause.

Definition 15. *A starting clause c for an example e w.r.t. a language L_i and a knowledge base KB is a maximal specific clause expressible in L_i such that c covers e given the knowledge base KB.*

Definition 16. *A justification X of an example e w.r.t. a language L_i and a knowledge base KB is a set of relations such that $X = body(c)\theta$ and $e = head(c)\theta$ for a starting clause c for e in L_i and a substitution θ grounding c such that $body(c)\theta$ is implied by the knowledge base KB.*

Example 3 illustrates these notions.

Example 3. Justification and starting clauses

Suppose we have the knowledge base of example 1 and the example `is_allowed_to_drive(katharina, mercedes)`. Then we have the following justifications and starting clauses:

in L_0, starting clause:

is_allowed_to_drive(K, M) ← owns(K, M)

justification: { owns(katharina, mercedes) }

in L_1, starting clause:

is_allowed_to_drive(K, M) ← owns(K, M) ∧ isa(M, Z)
∧ license(K, W)

justification: { owns(katharina, mercedes),
isa(mercedes, car), license(katharina, car) }

in L_2, starting clause:

is_allowed_to_drive(K, M) ← owns(K, M) ∧ isa(M, Z) ∧
license(K, W) ∧ eq(W, Z)

justification: { owns(katharina, mercedes), isa(mercedes, car),
license(katharina, car), eq(car, car) }

□

Observe that at least one of the starting clauses for a language and a positive example is true, provided that the concept can be described in terms of the knowledge base and the language. The reason is that a starting clause is a most specific clause covering the example with respect to the knowledge base and the language-definition.

For some languages and knowledge bases the starting-clause is unique. However, in general this property does not hold: when existential quantified variables are introduced, there may be several different instantiations corresponding to different objects of these existentially quantified variables. At a deeper level, these different objects may give rise to different relations in the justification. From a theoretical point of view, all starting clauses should be considered. From a practical point of view it is often plausible to consider only one starting clause for each language because starting clauses for the same language are quite similar (De Raedt & Bruynooghe, 1990a; De Raedt & Bruynooghe, 1989c).

Language bias

In the previous paragraph, we stated that Clint focuses its search on the relations in a starting clause. However, as a starting clause (and the corresponding justification) contains only part of the potentially relevant relations, it may not be sufficient to correctly describe the concept. The starting clause does not suffice when it covers negative examples. Then it has to be rejected because Clint only considers generalizations of the starting clause. Because of these difficulties, Clint orders the starting clauses and selects the first starting clause that does not cover negative examples. The selected clause is then passed to the generalization procedure. The starting clauses are ordered according to the language they belong to, i.e., the starting clauses for language L_i are considered before the ones for L_{i+1}.

In example 3, suppose the negative example is_allowed_to_drive(luc, dyane) is known. Then the starting clause from L_1 would be selected because the starting clause for L_0 covers the negative example.

Querying the user to remove irrelevant relations

It is sometimes interesting to query the user in order to remove irrelevant relations from the starting clause. Removing irrelevant relations from the starting clauses reduces the search space. The technique used in Clint to remove relations from starting clauses is based on a similar technique which is employed in the system Disciple (Kodratoff & Tecuci, 1987; Tecuci & Kodratoff, 1990). There are two strategies that can be followed: removing irrelevant relations by querying the user for the (ir)relevance of relations in a justification or asking the user for the (ir)relevance of the arguments in a justification. This is briefly illustrated in example 4.

Example 4. Query the user

Suppose we have the following starting clause and justification for the positive example `grandfather(jeffrey, peter)`:

```
grandfather(X, Y) ← father(X, Z) ∧ parent(Z, Y)
                    ∧ taller(X, W) ∧ nephew(X, W)

justification: { father(jeffrey, tom) ∧ parent(tom, peter) ∧
                 taller(jeffrey, steve) ∧ nephew(jeffrey, steve) }
```

The irrelevant relations can be located by presenting the justification to the user and asking him/her which relations are irrelevant. Alternatively, the user could tell the system which objects (arguments) in the justification are irrelevant. The system would then present the objects `tom` and `steve` to the user and ask him/her which of them are irrelevant. (Notice that `peter` and `jeffrey` are assumed to be relevant because they occur in the example). When the user states that `steve` is irrelevant for `grandfather(jeffrey, peter)`, all relations containing `steve` would be removed from the justification (and all corresponding relations from the starting clause).

□

3.1.2. Generalizing the starting clause

Once a consistent starting clause has been determined, the starting clause is carefully generalized using a specific to general search which drops multiple conditions (this part constitutes the second repeat loop). This is realized in the next step of handle_uncovered. The generalization step computes a new clause c' obtained from the old clause (initially the starting clause), by deleting a set of elements S from the body of the old clause, such that the new clause covers an example e' that is not covered by the old clause, the new clause does not cover negative examples and c' is maximal. The example e' is then presented to the user for classification if its classification is not yet known. If the example is positive, the old clause is replaced by the new one, and the process is repeated. Otherwise, the learner considers other generalizations. If there are no more generalizations fulfilling the conditions, the generalization process terminates and the clause is added to the knowledge base.

During generalization, the system may collect new negative examples. The algorithm tests whether these are covered by previously learned rules, and if necessary, it takes appropriate action in order to recover from possible errors.

At this point we also want to stress that the sets S, to be removed from $body(c)$, in handle_uncovered can efficiently be computed using a breadth-first search, pruned by knowledge and guided by heuristics (for more details on the implementation, see De Raedt and Bruynooghe (1988) and De Raedt (1991)).

The example e' in algorithm 2 is obtained by querying the knowledge base. This is achieved by generating the query $\leftarrow body(c') \wedge \neg S$. If θ is an answer to this query, a suitable choice for e' is $head(c')\theta$. For complex knowledge bases, these queries may be computationally expensive.

The motivation for choosing this generalization operator is that it is complete for our languages. This means that any semantically significant element in the language can be obtained by applying it. When there is a set of clauses that are semantically equivalent (i.e., cover the same set of examples; such descriptions are sometimes called syntactic variants), at least one of them can be generated by our generalization operator.

3.2. Processing a negative example

The algorithm to handle covered negative examples is shown in handle-covered of algorithm 3. When the learner receives a covered negative example, there must be an incorrect clause in its knowledge base (cf. (Shapiro, 1983)).

Algorithm 3. A covered negative example

```
procedure handle_covered(p: predicate, e: covered negative example)
    build the proof tree for e
    analyze the proof tree in order to obtain an example e' and
    an incorrect clause c such that:
        (1) e' is negative
        (2) there is a substitution θ such that head(c)θ = e'
        (3) body(c)θ is true
    retract c
    for all positive examples e'' of p that are covered by c do
        call handle_example(p, e'')
    endfor
    for all predicates q learned after p and using p do
        for all examples e'' of q do
            call handle_example(q, e'')
        endfor
    endfor
endproc
```

Definition 17. *A clause c is incorrect if there exists a substitution θ such that head(c)θ is false and body(c)θ is true.*

To locate incorrect clauses, the learner constructs the proof tree for the negative example and analyses it. If necessary, it asks intelligent questions to the user. These questions are also membership questions. They ask for the classification of examples that occur as nodes in the proof tree. The method to locate incorrect clauses is very similar to Shapiro's debugging method used in the Model Inference System (Shapiro, 1983). When an incorrect clause is located, it is retracted from the knowledge base. To maintain consistency, the learner must verify whether the positive examples that were covered by the retracted clause, are still covered. If not, the learner will generate and process an example-driven opportunity for each uncovered positive example. Also, if a predicate was learned after the predicate for the incorrect clause and if the latter one was involved in learning the former, then its positive and negative examples have to be verified. In order to avoid these interactions between different predicates as much as possible, it is advised that predicates are thoroughly tested before being used to learn other predicates.

The problem of handling covered negative examples is sometimes referred to as the knowledge revision problem. Some approaches such as Inde (Aben & Van Someren, 1990) and Blip (Wrobel, 1989; Morik, 1989) try to solve it without generating questions to the user.

3.3. An evaluation of Clint

The amount of redundancy in the starting clause(s) has a great influence on the overall efficiency of the algorithm and on the number of questions posed to the user. The overall efficiency is also influenced by the size of the knowledge base while the number of relations in the starting clause(s) also has a substantial impact on the number of questions. For most of the problems we tried, the number of questions needed and the overall efficiency are acceptable (see (De Raedt & Bruynooghe, 1988; De Raedt, 1991)).

The algorithm has several interesting features:

- It copes with indirect relevance (Michalski, 1983; De Raedt & Bruynooghe, 1990a) because the user only has to supply the directly involved objects in an example and not the relevant relations holding among them. This is a more natural and less demanding way to describe examples than the approach taken in systems such as Marvin (Sammut & Banerji, 1986) and Alvin (Krawchuk & Witten, 1988). For use in learning apprentice systems this feature is indispensible.
- Disjunctive as well as conjunctive predicates can be learned.
- The system can shift its bias when it discovers that its language is not sufficient to describe the predicate. This situation occurs when all starting clauses covering a positive example in the language L_i cover also negative examples (see also example 8). Then the language of the clause will be shifted to L_{i+1}. So, Clint shifts the bias at the clause-level.
- The system generates most of the examples it needs. The user has to specify at most one positive example to learn a clause.

- The system is a closed-loop system. This means that newly learned predicates are integrated (assimilated) in the knowledge base and that once learned, they are used like any other predicate.
- The system provides a kind of error-recovery. An incorrect clause is localized and retracted when there is an opportunity to do so.
- Clint identifies concepts in the limit, provided that the predicate can be described in one of the languages and the current knowledge. This limiting behavior is proven in De Raedt and Bruynooghe (1988), De Raedt and Bruynooghe (1989c), and De Raedt (1991). For conjunctive predicates and simple languages there is even finite identification.

There are still some remaining problems with the algorithm:

- Recovering from an error in an assimilated predicate may also affect other predicates. Therefore, recovering from such errors may involve much work.
- The order in which predicates are learned is important. Ideally, the easier predicates should be learned first, and the more difficult ones, which use the easier ones in their definition should only be learned afterwards. However, if the user presents the predicates in a different order, this does not necessarily lead to problems, because of Clint's error-recovery ability. In fact, problems only arise when there is a positive and a negative example such that, given the knowledge and the languages, there is no clause which covers the positive example and which does not cover the negative one. In that case, Clint will continue to shift its bias.

  In a recent but rather complicated variant of Clint, this problem has been solved. The idea is to postpone the learning of a positive example when there is no consistent starting clause covering the example in the first M languages. Later when the knowledge base is modified, the postponed examples are reconsidered. This allows to alleviate the problem. For more details, we refer to De Raedt (1991).
- The learned clauses depend very much on the knowledge Clint possesses. If Clint knows all relevant predicates for the concept to be learned, then Clint will learn a small number of clauses in easy languages. On the other hand, if Clint knows only predicates that are not so relevant, Clint will learn a larger number of clauses or the bias of the clauses will be more complex.

  This problem seems to be inherent to concept-learners.
- Clint does not take into account the relationship between different clauses for the same concept. As a consequence, Clint does not always learn the concept-description with the minimal number of clauses. This problem also arises in Marvin (Sammut & Banerji, 1986) and the Model Inference System (Shapiro, 1983).
- For some language-definitions, knowledge-bases and examples there may be many justifications (and starting clauses). If the number of possible justifications is large, the algorithm may require much computation. As said before, in practice it is often feasible to consider only one starting clause since they are often quite similar. For a further discussion of these issues, we refer to De Raedt and Bruynooghe (1990a) and De Raedt (1991).
- Clint could in principle also be used for logic program synthesis. Functors can be handled by Clint, if we add for each functor f with n arguments a predicate F with the following fact as definition: $F(X_1, \ldots, X_n, f(X_1, \ldots, X_n)) \leftarrow$. Using these definitions, Clint's

languages allow the use of functors (e.g., `member(X, [X|Z]) ←` would be rewritten as `member(X, Y) ← list(X, Z, Y)` with as definition for `list` the fact `list(X, Y, [X|Y]) ←`). For practical purposes, it may be desirable to slightly refine the languages of Clint for handling functors. Recursion can also be handled if Clint first learns the base cases and only later the recursive ones. To learn the well-known `member` predicate, the user could first supply the positive example `member(a, [a])` to Clint and only in a later phase present more complex examples such as `member(a, [b, a])`. So, in principle the framework and algorithms of Clint are sufficient and suitable to synthesize logic programs. However, using Clint for logic program synthesis does not work in practice. The reason is that Clint has to query the knowledge base (e.g., in order to construct examples) and that problems arise with nonterminating queries. The conclusion is that given a decidable procedure for answering queries, Clint could synthesize logic programs. However, for general logic programs, it has been proven that there exists no such procedure. For a further discussion of these issues, we refer to De Raedt (1991).

4. An experiment with Clint

Dietterich and Michalski (1985) and Quinlan (1990) describe a simple card game named Eleusis, which has been used to evaluate the learning systems Sparc/e and Foil. In Eleusis, the dealer invents a secret rule specifying the conditions under which a card can be added to a sequence of cards. The player can add a card to the current sequence and is told whether the played card is a legal successor of the sequence or not. This allows to derive positive and negative examples of the rule. In the experiments with Sparc/e and Foil, certain sequences containing legal as well as illegal successors were transformed into examples and an attempt was made to guess the secret rule by learning a rule from the examples. Sparc/e and Foil did not actually play the game since they did not propose successor cards for a given sequence.

As Clint is able to generate examples it seems well suited to play Eleusis. This could be realized by learning a predicate such as `can_follow(A, B, C, D, E, F)`, defined by Quinlan (1990). The predicate succeeds when card `A`, suit `B` can follow a sequence ending with card `C` (suit `D`) and `E` consecutive cards of suit `D` and `F` consecutive cards of the same color. However, legal examples from the point of view of Eleusis have the property that `C`, `D`, `E` and `F` are fixed by the current sequence of cards. Although Clint could learn `can_follow` it should be modified in order to satisfy this constraint.

Instead of modifying Clint, we considered a variant of Eleusis in which the sequential aspect is omitted. In this variant, the dealer invents a secret rule specifying the conditions under which an ordered tuple of cards is legal. The player can then ask whether certain tuples of cards are legal or illegal. On this variant of Eleusis, Clint was equipped with the knowledge base specified in example 5 and used to learn the rules specified in example 6. Concept 1 and 2 in example 6 correspond to the nonsequential variants of the secret rules used in Quinlan's second and third experiment. The results are summarized in table 1. Notice that Clint learned the intended rules; whereas Foil finds only one rule corresponding to our Concept 2. The discovery of the intended rules by Clint is quite normal because Clint identifies concepts in the limit provided that it can describe the concepts in one of

its languages and the given knowledge base. However, if Clint cannot describe the concept it would continue to shift its bias. This explains why we did not attempt to learn the concept that corresponds to Quinlan's first experiment. In Quinlan's first experiment, knowledge about the sex of cards is required to learn the correct rules. Foil was able to learn an approximation of the inteded rules without this knowledge.

Example 5. A knowledge base to play the variant of Eleusis

The following knowledge base is adapted from Quinlan (1990) towards Clint:

- `precedes_rank(X, Y)`, `precedes_suit(X, Y)`: rank (resp. suit) `X` precedes rank (resp. suit) `Y`
- `lower_rank(X, Y)`, `lower_suit(X, Y)`: rank (resp. suit) `X` is lower than rank (resp. suit) `Y`
- `face(X)`: rank `X` is a face card
- `same_color(X, Y)`: suit `X` and suit `Y` are the same color
- `odd_rank(X)`, `odd_suit(X)`: rank (resp. suit) `X` is odd
- `rank(X)`, `suit(X)`: `X` is a rank (resp. suit)
- the negation of all these predicates was also available to Clint. This was done because the basic Clint system cannot cope with negation as failure. There is however an extension of Clint that is able to cope with a constructive form of negation and to represent the unknown by means of a three valued logic (see (De Raedt & Bruynooghe 1990b)).

□

The aim was to learn the predicate `legal_tuple(X, Y, Z, W)` which succeeds when the card of rank `X`, suit `Y` followed by the card of rank `Z`, suit `W` is a legal combination.

Example 6. Two concepts learned by Clint

Concept 1:

`legal_tuple(X, Y, Z, W)` ← `face(X)` ∧ `suit(Y)` ∧ `not_face(Z)` ∧ `suit(W)`

`legal_tuple(X, Y, Z, W)` ← `not_face(X)` ∧ `suit(Y)` ∧ `face(Z)` ∧ `suit(W)`

Concept 2:

`legal_tuple(X, Y, Z, W)` ← `precedes_rank(X, Z)` ∧ `not_lower_suit(Y, W)`

`legal_tuple(X, Y, Z, W)` ← `precedes_rank(X, Z)` ∧ `lower_suit(Y, W)`

□

An important aspect about these experiments is the number of needed examples to learn a correct concept-description. When comparing Clint to Foil and taking into account the facts that Clint generates its own examples and Foil learns a 6-place predicate, the number of examples needed by Clint seems large. (For Foil the number of supplied examples was in both experiments 30.) Let us explain this seemingly large number of examples. Foil is

Table 1. Experimental results.

	Concept 1	Concept 2
Number of clauses	2	2
Number of user supplied examples	2	2
Number of generated examples	17	16
Number of positive examples	15	14
Number of negative examples	4	4
Number of literals in starting clauses	15–17	18–20

a general-to-specific system that learns by adding conditions to the body of a clause. Given this strategy the considered concepts are very simple because the bodies of the clauses contain only two literals.[2] For Clint, which searches specific-to-general the learned concepts are much harder. Indeed, Clint has to drop between 11 and 18 literals from the body of the clause. Furthermore, the number of examples needed by Clint to learn a clause in this domain depends mainly on the number of literals in the starting clause and not on the complexity or the number of literals in the final clauses (see (De Raedt, 1991) for a discussion of the complexity of Clint). For Foil the most important factor seems to be the complexity of the target clauses. Therefore we expect that when the clauses to be learned would contain more literals, Foil would need much more examples than Clint in order to learn a good approximation of the concepts.

Analogies prove nothing, that is quite true,
but they can make one feel more at home
—Sigmund Freud—

5. Constructive induction by analogy

In this section we present the companion system of Clint: Cia. It attempts to overcome one of the main limitations of classical concept-learning systems by suggesting interesting new concepts to the user.

A second issue that is addressed by the Cia techique is:

Can the learning process itself be learned and if so, how can this be done? or
Can we apply machine learning techniques to machine learning?

Although there has not been much work to answer these questions, there are some interesting initial studies such as Stepp, Whitehall and Lawrence (1988), who discuss general issues in learning learning machines, Rendell, Seshu and Tcheng (1987), who try to learn conditions under which to apply specific machine learning techniques and Vrain and Lu (1988), who apply analogy to do incremental concept-learning.

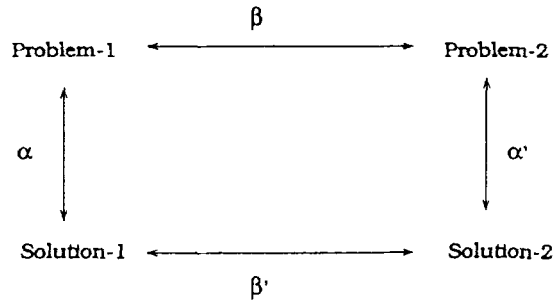

Figure 1. Analogy.

5.1. Applying analogy on concept-learning

We first consider the classical analogy paradigm as proposed by Chouraqui (1985), Kodratoff (1988), and Kodratoff (1990). It is shown in figure 1.

Definition 18. *A target* `Problem-2` *is solved by analogy to a source* `Problem-1` *by considering the similarity relations* β *and* β' *and the mappings* α *and* α'. `Solution-2` *is called an analogous solution to a target* `Problem-2` *iff the following relations hold:*

1. α(`Problem-1, Solution-1`).
2. α'(`Problem-2, Solution-2`).
3. β(`Problem-1, Problem-2`).
4. β'(`Solution-1, Solution-2`).

These relations depend on the particular application of analogy. In our setting we have that `Problem-1` and `Problem-2` are concept-learning problems, more specifically, sets of examples to be generalized. We also have that `Solution-1` and `Solution-2` are concept-descriptions which are complete and consistent for the examples of `Problem-1` and `Problem-2` respectively. The problem addressed by Cia is the following one:

- **Given:**
  - a target concept-learning problem `Problem-2`
  - a set of source concept-learning problems:
    {(`Problem-i, Solution-i`) | $\forall i$: α (`Problem-i, Solution-i`) holds}
  - the relations α', β and β'
- **Find:** an analogous solution to `Problem-2`

We will assume that α is accomplished by Clint although the technique could in principle also be applied in the context of other concept-learners. So, in order to describe our Cia technique in detail, we only have to define α', β and β'.

5.2. The Constructive Induction by Analogy technique

The main assumption behind the Cia technique is the following one:

Assumption: *Concept-descriptions are often very similar in the sense that they are instantiations of the same second order schema.*

The instantiations are obtained by applying second order substitutions. An illustration of this assumption can be found in example 7.

Example 7. The assumption of Cia

Consider the concept-description:

$$\texttt{lighter (X, Y)} \leftarrow \texttt{weight(X, XW)} \wedge \texttt{weight(Y, YW)} \wedge \texttt{less(XW, YW)}$$

It is an instantiation of the following second-order schema with substitution

$$\Theta = \{p = \texttt{lighter}, q = \texttt{weight}, r = \texttt{less}\}$$
$$\exists p, q, r\colon p(\texttt{X}, \texttt{Y}) \leftarrow q(\texttt{X}, \texttt{XW}) \wedge q(\texttt{Y}, \texttt{YW}) \wedge r(\texttt{XW}, \texttt{YW})$$

Other instantiations of this schema together with their substitution are for instance

$$\texttt{same\_color(X, Y)} \leftarrow \texttt{color(X, XC)} \wedge \texttt{color(Y, YC)} \wedge \texttt{eq(XC, YC)}$$
$$\Theta = \{p = \texttt{same\_color}, q = \texttt{color}, r = \texttt{eq}\}$$

and

$$\texttt{brothers(X, Y)} \leftarrow \texttt{son(X, XP)} \wedge \texttt{son(Y, YP)} \wedge \texttt{eq(XP, YP)}$$
$$\Theta = \{p = \texttt{brothers}, q = \texttt{son}, r = \texttt{eq}\}$$

and

$$\texttt{near(X, Y)} \leftarrow \texttt{part\_of(X, XP)} \wedge \texttt{part\_of(Y, YP)} \wedge \texttt{next\_to(XP, YP)}$$
$$\Theta = \{p = \texttt{near}, q = \texttt{part\_of}, r = \texttt{next\_to}\}$$

□

It is important to note that the Cia assumption (or a variant of the assumption) has been made and exploited by several other researchers:

- The Blip-system (Morik, 1989; Wrobel, 1988; Emde, Habel & Rollinger, 1983; Thieme, 1989) uses second-order schemata in a model-driven approach to learning.
- Yokomori (1986) shows how (a variant of) second-order schemata can be used in a logic programming setting to decrease the memory needs of the knowledge base and to perform analogical reasoning.
- Dieter Poetschke (1989) showed that many programs such as `append` and `plus` are analogous. He was also investigating ways to exploit these analogies.

The Cia technique in Clint employs the observed similarity by deriving from each learned concept-description a second-order schema (see algorithm 2). This schema specifies the relations among predicates in the concept-description. The derivation of a schema from a clause is straightforward. One only has to replace all predicate names in the concept-description by predicate variables. Once such a schema is derived, it is stored to be matched with starting-clauses when new concepts are learned.

Algorithm 2 shows when Clint calls Cia. This occurs at two different places:

- when Clint has found a consistent starting clause covering a certain positive example, Clint will call the first part of Cia; this part tries to guess a correct clause for the concept being learned by asking clause questions
- after learning and asserting one clause for a concept, Clint may call the second part of Cia; this part attempts to invent or discover new concepts by asking new term questions

Let us now present these two parts of Cia (see algorithm 4).

Algorithm 4. Constructive Induction by Analogy

```
procedure cia_part_1(c: consistent clause): returns generalization of c
    d := void
    for all second-order schemata T do
        if match(T, c) with substitution Θ
        then if TΘ does not cover negative examples and d = void
            then ask the clause question TΘ to the user
                if TΘ is correct
                then if the user does not want to continue
                    then return TΘ
                    else d := TΘ
                    endif
                else find the substitution κ ⊂ Θ such that pmatch(T, c) for κ
                    ask the new term question Tκ to the user
                    if the user names the predicate variable in Tκ
                       by a substitution Λ
                    then assert TκΛ in the knowledge base
                    endif
                endif
            else find the substitution κ ⊂ Θ such that pmatch(T, c) for κ
                ask the new term question Tκ to the user
                if the user names the predicate variable in Tκ
                   by a substitution Λ
                then assert TκΛ in the knowledge base
                endif
            endif
        endif
    endfor
    return d
endproc
```

```
procedure cia_part_2(c: consistent clause)
    for all second-order schemata T do
        if pmatch(T, c) with substitution Θ and not match(T, c)
        then ask the new term question TΘ to the user
            if the user names TΘ by a substitution Λ
            then assert TΘΛ in the knowledge base
            endif
        endif
    endfor
endproc
```

cia_part_1 tries to make a guess at the correct concept-description by completely matching a consistent starting clause derived by Clint with one of its schemata. If the starting clause matches a schema, Cia proposes the instantiated second-order schema to the user and asks whether the resulting clause is correct for the concept being learned. If it is not, it may be that the proposed clause is actually a clause for an unknown but useful concept. Then the user may name the concept, such that the system can assert it in its knowledge-base. This corresponds to asking a new term question for a partial match with the schema. On the other hand, if the user confirms that Cia has discovered the target concept-description, Cia may halt or it can continue to propose concept-descriptions by instantiating other schemata in order to discover new concepts. If the target concept is not found, Clint has to continue in order to learn the concept. Once the concept is learned by Clint, a second order schema is derived from the learned clause. In that case, it may occur that the schemata and the starting clause do not match completely, but partially (only the bodies match). Then, more new predicates may be proposed. This is done in cia_part_2. When desired, Cia can generate some instances (examples) of proposed concept-descriptions in order to facilitate the recognition of the predicate.

Thus far, the basic algorithm of the Cia technique has been presented. Several improvements and variants are possible. The main objection against the use of the basic algorithm as it stands is that its pure syntactic nature gives rise to uninteresting as well as redundant questions (cf. also example 8). In order to avoid such questions we augmented the algorithm with the following tests:

- Before asking a question, it is verified that it is not redundant. This is done by testing whether $T\Theta$, the concept to be proposed, is not equivalent to a previously proposed one.

  If we want to test whether question q_1 is redundant by comparing it to a previously asked question q_2, we can use two different tests:
  — A naive test which merely verifies: $\models (q_1 \leftrightarrow q_2)$.[3] This can, e.g., be done using simple subsumption tests in two directions (see, e.g., (Loveland, 1978))
  — A more complicated test which verifies $KB \models (q_1 \leftrightarrow q_2)$.

  For the second kind of test one can use a depth-bounded version of Buntine's subsumption technique (Buntine, 1988) (applied in two directions) or an extensional test to check equivalence. Buntine's test is quite fast and sound but unfortunately not complete; this means that all subsumptions found by it are legitimate but existing subsumptions will not always be found. The extensional test is complete (for nonrecursive predicates without

functors) but rather slow since it derives all ground instances that are covered by both descriptions. Of course, both tests are applied on the current knowledge base; in case there is still an error or an incompleteness in the knowledge base, they may give rise to wrong conclusions.

- Before proposing a new predicate, it should be sure that it does not exist yet (it might have been in the original knowledge base or it might already have been learned by Clint). This is verified in a similar way.
- Proposed predicate-definitions should be in a most simple form. A definition is in a most simple form if it is impossible to properly generalize the clause without changing the semantics. As an example: the concept grandfather(GF, GC) ← father(GF, C) ∧ parent(C, GC) ∧ male(GF) is not in a most simple form since male(GF) can be deleted without changing the semantics of the clause. In order to know whether a clause is in a most simple form we delete each condition once, and verify (using our subsumption test(s)) whether the resulting clauses are equivalent to the original one.

Apart from asking relevant questions, some further improvements are possible:

- Cia can be extended further by allowing transformations on learned schemata. The extension is motivated by the observation that when $\exists p, q, r$: $p(X, Y) \leftarrow q(X) \wedge r(X, Y)$ is a useful schema then it is quite likely that symmetric versions of the schema such as $\exists p, q, r$: $p(X, Y) \leftarrow q(Y) \wedge r(X, Y)$ are also useful. The second schema can be obtained by applying a transformation on the first one (replace the variables of unary predicates that also occur in the head by other variables occurring in the head). Several such transformations can be used in Cia and when Cia learns a new schema, all possible transformations are applied on it. If the result of such a transformation is a new schema that is not a renaming of an old one, it is stored in the memory. Because the use of transformations is less justifiable from the analogy point of view, its use is optional.
- The schemata encode information about source problems. In the context of Clint-Cia, Clint may sometimes learn an incorrect clause from which Cia will derive a schema. However, when it is later detected that the clause, learned by Clint, is incorrect and the clause is retracted, we should also retract the schemata that are derived from the incorrect clause. The reason is that it can no longer be considered as a source problem. There is of course one exception to this rule, which occurs when the considered schemata can also be derived from other learned clauses that still remain in the knowledge base. This enhancement has been found useful in practice. It does not require much computation as one can index clauses under schemata.
- One quite technical optimization concerns the matching procedure with respect to the equality predicate eq. In order to avoid new term questions like $\exists p$: $p(X, Y) \leftarrow$ male(X) ∧ eq(X, Y) we allow the equality predicate only to match those predicate-variables of schemata whose source-instantiations were equality predicates. Although the equality predicate may only match such predicate-variables, it is allowed that predicate-variables with as source instantiation the equality predicate match nonequality predicates.

An example session with the integrated Clint-Cia system is shown in example 8.

Example 8. An example session with Clint-Cia

Suppose that we start with the following knowledge:

```
parent(alice, rose) ←        parent(leon, rose) ←
parent(rose, luc) ←          parent(rose, ann) ←
parent(etienne, luc) ←       parent(etienne, ann) ←
parent(laura, etienne) ←

female(alice) ←              male(luc) ←
female(rose) ←               male(etienne) ←
female(ann) ←                male(leon) ←
female(laura) ←
```

In the next session, user input is in **bold** and system output typewritten. Furthermore, the Cia-part is surrounded by brackets ⟨Cia Cia⟩.

Suppose Clint first learns the concept father.

?-learn.
Next example ? @ **father(etienne, luc).**
Classification ? @ **true.**
Starting clause for father(etienne, luc) in L_0:

father(F, C) ← male(F) ∧ male(C) ∧ parent(F, C)

The starting clause is derived from the knowledge base; the corresponding justification is { male(luc), male(etienne), parent(etienne, luc) }. Observe that the starting clause contains only relations that hold for the example. When the option of Clint to query the user about relevance, would have been on, Clint would have asked whether male(luc), male(etienne), parent(etienne, luc) are relevant for father(etienne, luc). The first language definition requires that all variables in the body of a clause also occur in the head of the starting clause.

Is father(rose, luc) true? { male(F) } deleted @ **no.**
Is father(leon, rose) true ? { male(C) } deleted @ **yes.**
Generalized clause:

father(F, C) ← male(F) ∧ parent(F, C)

No more generalizations. Asserted clause:

father(F, C) ← male(F) ∧ parent(F, C)

```
⟨Cia:
Derived Schemata:
∃: father, male, parent: father(F, C) ← male(F) ∧ parent(F, C)
Schema number 1
and after applying transformation (1):
∃: father, male, parent: father(F, C) ← male(C) ∧ parent(F, C)
Schema number 2
Cia⟩
```

The following transformation was used: if the head of the schema is of arity >1 then derive a schema such that the variables of the unary predicates in the body are replaced by other ones; these variables also have to occur in the head.

Suppose the user now decides to learn a second predicate.

?-learn
Next example ? @ **grandmother(alice, luc).**
Classification ? @ **true.**
Starting clause for grandmother(alice, luc) in L_0:

```
grandmother(G, C) ← male(C) ∧ female (G)
```

No more valid generalizations. Asserted clause:

```
grandmother(G, C) ← male(C) ∧ female(G)
```

There are no more generalizations in the language since deleting any of the two conditions would give rise to an unlinked variable.

```
⟨Cia
Derived Schemata:
∃ grandmother,male, female: grandmother(G, C) ← male(C) ∧ female(G)
Cia⟩
```

No transformations are applied since all transformations yield a symmetric variant.

?-learn.
Next example ? @ **grandmother(rose, luc).**
Classification ? @ **false.**

The user signals that there is an error in the knowledge base. To recover from the error (a covered negative example), the system queries the user about the proof tree. If female and male would have been basic predicates, these questions would not have been asked.

```
Is female(rose) true ? @ yes.
Is male(luc) true ? @ yes.
Retracting incorrect clause:

    grandmother(G, C) ← male(C) ∧ female(G)

⟨Cia
Retracting Derived Schemata:
∃ grandmother,male, female: grandmother(G, C) ← male(C) ∧ female(G)
Cia⟩
```

Since there are no more instantiations of the schema in the knowledge base, it is also retracted from the knowledge base.

```
True fact grandmother(alice, luc) is not covered
Starting clause for grandmother(alice, luc) in L0

    grandmother(G, C) ← male(C) ∧ female(G)

Starting clause covers negative example grandmother(rose, luc):
Shifting base to L1
```

L_1 allows for clauses for which each condition can have one variable not occurring in the head. No relations among these existentially quantified variables are allowed.

```
Starting clause for grandmother(alice, luc) in L1

    grandmother(G, C) ← male(C) ∧ female(G) ∧ father(F, C) ∧
        parent(M1, C) ∧ parent(G, M2)

Starting clause covers false fact grandmother(rose, luc):
Shifting bias to L2
```

L_2 also allows for relations among the existentially quantified variables introduced in L_1.

```
Starting clause for grandmother(alice, luc) in L2:

    grandmother(G, C) ← male(C) ∧ female(G) ∧ father(F, C) ∧
        parent(F, C) ∧ parent(M1, C) ∧ parent(G, M2) ∧
        eq(M1, M2) ∧ female(M1) ∧ male(F)
```

Although the system knows for instance that `father(F, C)` implies `parent(F, C)`, it keeps both literals in the body of the clause. This allows to facilitate the generalization process. This knowledge is, however, used by the generalization operator in order to simplify final clauses and to optimize the search.

```
Is grandmother(leon, luc) true ? { female(G) } deleted @ no.
Is grandmother(alice, ann) true ? { male(C) } deleted @ yes.
Generalized clause:

    grandmother(G, C) ← female(G) ∧ father(F, C) ∧ parent(F, C) ∧
        parent(M1, C) ∧ parent(G, M2) ∧ eq(M1, M2) ∧
        female(M1) ∧ male(F)

Is grandmother(laura, luc) true ? { female(M1) } deleted @ yes.

    grandmother(G, C) ← female(G) ∧ father(F, C) ∧ parent(F, C) ∧
        parent(M1, C) ∧ parent(G, M2) ∧ eq(M1, M2) ∧ male(F)

No more valid generalizations. Asserted clause:

    grandmother(G, C) ← female(G) ∧ parent(M1, C) ∧
        parent(G, M2) ∧ eq(M1, M2)
```

The final clause is simplified. Given the knowledge base, the final clause is semantically equivalent to the original one.

```
⟨Cia
Proposing new predicates by partial matching ...
The starting clause for grandmother partially matches schema 1:
∃ p: p(F, C) ← male(F) ∧ father(F, C)
This question is not posed as it is not in a most simple form
The starting clause for grandmother partially matches schema 1:
∃ p: p(F, C) ← male(F) ∧ parent(F, C)
This question is not posed as it is redundant with father
The starting clause for grandmother partially matches schema 1:
∃ p: p(F, C) ← female(F) ∧ parent(F, C)
Name p ? @ mother.
Asserting:

    mother(F, C) ← female(F) ∧ parent(F, C)

The starting clause for grandmother partially matches schema 2:
∃ p: p(F, C) ← male(C) ∧ father(F, C)
Name p ? @ ignore.
```

In order not to have too many predicates in the knowledge base, the user ignores this predicate although the proposed predicate is a plausible one.

```
The starting clause for grandmother partially matches schema 2:
∃ p: p(F, C) ← female(C) ∧ parent(F, C)
Name p ? @ daughter.
Asserting:

    daughter(F, C) ← female(C) ∧ parent(F, C)

The starting clause for grandmother partially matches schema 2:
∃ p: p(F, C) ← male(C) ∧ parent(F, C)
Name p ? @ son.
Asserting:

    son(F, C) ← male(C) ∧ parent(F, C)

End of partial matching
Derived Schemata:
∃ grandmother, female, parent, eq:
grandmother(G, C) ← female(G) ∧ parent(M1, C) ∧ parent(G, M2) ∧ eq(M1, M2)
Schema 3
Cia⟩
```

The system derives some more schemata by applying transformations and finally halts. □

6. Experiments with Cia

Although the above example is quite simple, it gives a good idea of our Clint-Cia algorithm and its potential use. In order to better demonstrate Clint-Cia's utility we tested it on two extended learning tasks:

- An experiment on the family domain using the integrated Clint-Cia system with both parts of Cia.
- An experiment on the domain of simple chess rules. In this experiment, Clint was enhanced with Cia's first part.

6.1. The family domain

6.1.1. A description of the experiment

In this experiment, we started from a knowledge base containing the predicates `parent`, `male`, `female` and `married`. The knowledge base consisted of data from 4 generations; in total there were about 140 facts. The aim was to learn the following 29 predicates: `father`, `mother`, `son`, `daughter`, `grandparent`, `grandson`, `granddaughter`, `grandfather`, `grandmother`, `grandparent_in_law`, `sibling`, `brother`, `sister`,

sibling_in_law, sister_in_law, brother_in_law, parent_in_law, mother_in_law, father_in_law, son_in_law, daughter_in_law, uncle_or_aunt, uncle, aunt, niece, nephew, cousin, female_cousin and male_cousin. In total the definitions for these predicates consisted of 31 clauses. The only disjunctive predicate was sibling_in_law (3 clauses). We used the integrated Clint-Cia system for learning this knowledge base.

Both steps of Cia were applied and once a predicate was learned by Clint, all symmetric variants of the associated schema were asserted. Also, when an erroneous definition was learned by Clint and later retracted, the associated schema (and its variants) were retracted. Furthermore, the option in Clint to reduce the search space by querying the user for the relevant arguments (see example 4) was used to remove some of the irrelevant literals from the body of the starting clause. Notice that although this option allowed us to discard many irrelevant relations from the justification, many other irrelevant relations remained in the justification. Suppose, e.g., that you learn grandfather(leon, luc) then the reduced justification would—depending on the known predicates—be based on mother(rose, luc), female(rose), daughter(leon, rose), parent(leon, rose), son(rose, luc), male(leon), male(rose).

A good heuristic to minimize the interaction with Clint-Cia is to learn the basic (and most difficult) predicates first. We learned these predicates in the following order: grandparent, sibling, parent_in_law, grandparent_in_law, sibling_in_law, cousin. Afterwards it was easy to learn the other ones. Only three clauses were learned using the pure Clint system. These three clauses resulted in the following schemata:

∃ grandparent, parent, eq:
grandparent(G, C) ← parent(G, M1) ∧ parent(M2, C) ∧ eq(M1, M2) Schema A

∃ parent_in_law, married, parent:
parent_in_law(G, C) ← parent(G, M) ∧ married(M, C) Schema B

∃ father,male, parent:
father(F, C) ← male(F) ∧ parent(F, C) Schema C

These schemata and their transformed variants were the only ones that were used by Clint-Cia. They proved sufficient to learn (or help learning) the other predicate. These other predicates were learned in basically two different ways. The most difficult ones (sibling, three clauses for sibling_in_law, parent_in_law, grandparent_in_law, and cousin) are only learnable in language L_i, $i > 0$ of Clint given the knowledge base and the order of learning the predicates. For these predicates, Clint first had to shift the bias and then Cia could learn the correct definitions using its first step. The three clauses for sibling_in_law were learned by matching with only one clause. The remaining predicates, were directly derived from the starting clause in the first language of Clint (14 predicates) or invented by matching with this starting clause (8 predicates). Learning those clauses was very similar to the session in example 8.

During the learning process Cia proposed only 3 new predicates which were ignored by the user. They corresponded roughly to the following concepts: mother_of_daughter,

brother_of_sister, sister_in_law_of_brother_in_law. Notice that they all represent meaningful concepts. Furthermore, 5 symmetric variants of desired predicate-definitions were proposed. With symmetric variants, we mean predicates that are equivalent but whose argument positions are interchanged. For example, if father is defined in the usual way and father' is defined by father(X, Y) ↔ father'(Y, X), then father and father' are symmetric variants. Clearly, these questions can be avoided by allowing the user to interchange the argument positions of proposed new terms. Also, Cia avoided 8 redundant questions: either the corresponding concepts already existed or the questions were already posed or the concepts were not in a most simple form.

To summarize: using Clint-Cia, only 3 clauses were learned by the pure mechanism of Clint, 9 clauses were learned by Cia after a shift of bias by Clint, 14 clauses were learned directly by Cia and 8 new predicates were invented by Cia. This is in contrast to learning the whole set of predicates with Clint which would have resulted in much more questions. With Clint, each clause is only obtained after several membership questions to the user.

6.1.2. Considering variants of the experiment

We believe that an important reason for the success of this experiment is the use of the option in Clint of the relevant arguments. When this option would not have been used (or less irrelevant literals would have been identified), then Cia would have posed more questions. On the one hand, this would result in the invention of more predicates but on the other hand more proposed concepts would have to be ignored.

A second important issue is the order of learning the predicates. What would have happened when the predicates were learned in a different order? Small variations on the order in which the predicates were learned, would only give rise to small variations on the results. However, if the order would have been radically different, the results would be significantly different. To illustrate this using an extreme case, consider what happens when the predicate cousin, one of the most difficult predicates, would be learned first. Given our initial knowledge base, the predicate cousin is only learnable in L_7 and two clauses are needed to correctly define it. Learning cousin would give rise to large starting clauses because the language allows existential quantifiers at level 3. Large starting clauses result in the generation of many examples. Furthermore, the schemata derived from the clauses for cousin would contain up to 8 literals and many existential quantified variables. This implies that these schemata match only few starting clauses and would almost never be used by Cia. Learning cousin first, is—certainly—an extreme case and it is normal that doing so is much more difficult than learning it when sibling and uncle_or_aunt are known. If sibling is known, the predicate is learnable in L_5 while if both sibling and uncle_or_aunt are known, it is learnable in L_3. It is important to see that these difficulties also apply to other concept-learners and even to humans; they seem inherent to concept-learning. Nevertheless, the problem is not as bad as it may seem. Our order for learning the most difficult predicates first may be modified in many ways while obtaining similar results. Indeed, if one takes care that sibling is learned before sibling_in_law and sibling_in_law or uncle_or_aunt before cousin the results would have been nearly the same.

6.2. Learning simple chess rules

We are well aware that the domain of family-relations is particularly well suited for our Cia technique because there are many (structurally) related predicates in it. Therefore, we present a learning task in the chess domain.

In this experiment,[4] it was our aim to learn the rules that govern simple chess movements. The simple chess rules are the first kind of rules that one teaches novice chess players. Roughly speaking, they state for each piece of chess which movement is allowed without taking into account the notion of chess and special rules such as *en passant* and *rocade*. For instance, for a rook, they state:

> a rook may move from tile t_1 to tile t_2 if t_1 and t_2 are on the same row or the same column and there is no piece standing in between t_1 and t_2 and either there is no piece on t_2 or a piece of the opponent.

For the other pieces there are similar rules. The rules for pawns and knights are particularly difficult.

To learn the knowledge base, consisting of definitions of simple chess movements, we started from the knowledge base specified in example 9. The aim was then to learn the following predicates: `pawn_move/5`, `bishop_move/5`, `knight_move/5`, `rook_move/5`, `king_move/5` and `queen_move/5`. The argument structure of these predicates is the same:

> `x_move(A1, A2, A3, A4, A5)` succeeds iff piece `A1` that is standing on tile `A2` in boardsituation `A5` may move to tile `A3` that is occupied by piece `A4` (possibly the empty piece) in boardsituation `A5`.

Example 9. A chess knowledge base to learn acceptable moves.

The knowledge base contains the following predicates:

- `rook/1`, `bishop/1`, `knight/1`, `queen/1`, `king/1` and `pawn/1`: these predicates succeed when their argument is a rook, bishop, knight, queen, king or pawn respectively. E.g., `rook(white(R))` is true.
- `opposite_color/2` succeeds when its arguments are pieces having a different color. E.g., `opposite_color(white(R), black(B))` is true.
- `board_no/1` succeeds when its argument is a number specifying a known chess position. In order to learn. Clint was given a small number (8) of specific chess boards, which were all numbered. It is for these specific chess boards that Clint is supposed to learn and generate examples.
- `occupies/3` succeeds when argument 2 is a location on the board, argument 1 is a piece and argument 3 is a board number such that the piece of argument 1 is located on position argument 2 of board argument 3. If there is no piece on the specified location, argument 1 yields the value `empty`.
- `empty/1` succeeds when its argument is the value `empty`.

- same_row/2 succeeds when its two arguments are board locations that are in the same row. E.g., same_row(g8, a8) is true. There are similar definition for same_diagonal/2 and same_column/2.
- nothing_in_between/3 succeeds when argument 1 is a board number, and arguments 2 and 3 are locations on the board such that there is no piece in between the locations on the specified board. nothing_in_between/3 only succeeds when the locations are on the same diagonal, row or column.
- startposition/2 succeeds when the first argument is a piece that is initially on the position specified by argument 2. E.g., startposition(white(Pa), e7) is true.
- forward/3 succeeds when the first argument is a piece and the third argument is a position that lies beyond the second argument from the player's point of view, e.g., forward(white(Pa), a2, a4) succeeds.
- adjacent/2 succeeds when its arguments specify adjacent locations on the chess board. E.g., adjacent(e5, f4) is true.
- vertical_distance_1/2, vertical_distance_2/2, horizontal_distance_1/2, horizontal_distance_1/2 succeeds when the number of columns (resp. rows) between the locations is 1 (resp. 2). E.g., vertical_distance_1(b4, e5) is true. □

The system was also given the following knowledge:

- when learning x_move, the condition x(A1) should appear in all clauses for x_move
- when learning x_move, no condition y(A4) should appear in a clause for x_move
- when learning x_move, the conditions occupies(A1, A2, A5) and occupies(A3, A4, A5) should appear in all clauses for x_move

This information was simply encoded in the language.[5] Although this knowledge is—in no way—needed by the system to learn the given concepts, it was used because:
- The same kind of knowledge is also communicated to novel players when learning the same kind of rules.
- It makes all membership questions to the user relevant from the application point of view.[6]

The target-knowledge base consists of 21 clauses, which can all be described in the language of completely bound Horn-clauses (L_0). Of these 21 clauses, 7 are structurally different, i.e., 7 schemata are needed to make abstraction of these 21 clauses. Two schemata are sufficient to make abstraction of all clauses for rook_move, bishop_move, queen_move, king_move. Two additional schemata are needed to make abstraction of knight_move and for each clause of pawn_move an additional schema is needed. A clause for each schema is shown in example 10.

Example 10. Clauses for schemata of the chess experiments

```
rook_move(A1, A2, A3, A4, A5) ← board_nr(A5) ∧ rook(A1) ∧
    same_row(A2, A3) ∧ occupies(A1, A2, A5) ∧ occupies(A4, A3, A5) ∧
    empty(A4) ∧ nothing_in_between(A5, A2, A3)
```

```
rook_move(A1, A2, A3, A4, A5) ← board_nr(A5) ∧ rook(A1) ∧
     same_row(A2, A3) ∧ occupies(A1, A2, A5) ∧ occupies(A4, A3, A5) ∧
     opposite_color(A1, A4) ∧ nothing_in_between(A5, A2, A3)

knight_move(A1, A2, A3, A4, A5) ← board_nr(A5) ∧ knight(A1) ∧
     occupies(A1, A2, A5) ∧ occupies(A4, A3, A5) ∧
     opposite_color(A1, A4) ∧ vertic_distance_1(A2, A3) ∧
     horiz_distance_2(A2, A3)

knight_move(A1, A2, A3, A4, A4) ← board_nr(A5) ∧ knight(A1) ∧
     occupies(A1, A2, A5) ∧ occupies(A4, A3, A5) ∧
     empty(A4) ∧ vertic_distance_1(A2, A3) ∧ horiz_distance_2(A2, A3)

pawn_move(A1, A2, A3, A4, A5) ← board_nr(A5) ∧ pawn(A1) ∧
     occupies(A1, A2, A5) ∧ occupies(A4, A3, A5) ∧ empty(A4) ∧
     startposition(A1, A2) ∧ same_column(A2, A3) ∧
     forward(A1, A2, A3) ∧ vertic_distance_2(A2, A3)

pawn_move(A1, A2, A3, A4, A5) ← board_nr(A5) ∧ pawn(A1) ∧
     occupies(A1, A2, A5) ∧ occupies(A4, A3, A5) ∧ empty(A4) ∧
     same_column(A2, A3) ∧ forward(A1, A2, A3) ∧
     vertic_distance_1(A2, A3)

pawn_move(A1, A2, A3, A4, A5) ← board_nr(A5) ∧ pawn(A1) ∧
     occupies(A1, A2, A5) ∧ occupies(A4, A3, A5) ∧
     opposite_color(A1, A4) ∧ same_diagonal(A2, A3) ∧
     forward(A1, A2, A3) ∧ adjacent(A2, A3)
```

In this second experiment, the aim was to test Cia's first part and to see how well its guessing of concept-descriptions performs.[7] The following conventions were employed in the experiments:

- the normal version of Clint was used together with Cia's part one; no relevance questions were posed.
- for each clause a schema was derived; this schema was retracted when the corresponding clause was retracted and there were no other learned clauses that justified the schema
- no transformations on schemata were applied
- as soon as an incorrect clause was learned, a counter-example to that clause was presented to the system; this resulted in the retraction of the incorrect clause (and possibly its associated schema)

Using these conventions, two different experiments were performed: one learning the predicates in a simple order for Cia, i.e., starting by `rook` and ending with `pawn` and the other one in the reverse order. The results are shown in tables 2 and 3. The tables show: the predicate being learned, the number of clauses for the predicate, the number of user-

Table 2. Experimental results.

Experiment 1	Ro	Bi	Qu	Ki	Kn	Pa	Total
clauses	4	2	6	2	4	3	21
user examples	4	2	6	2	5	3	22
Clint examples	12	—	—	—	10	10	32
Cia 1 good	2	2	6	2	1	1	14
Cia 1 bad without continuation	—	2	2	1	—	12	17
Cia 1 bad with continuation	—	—	—	3	—	8	11
schemata	2	—	—	—	3	2	7

Table 3. Experimental results.

Experiment 2	Pa	Kn	Ki	Qu	Bi	Ro	Total
clauses	3	4	2	6	2	4	21
user examples	3	4	2	6	2	4	21
Clint examples	15	8	7	—	—	—	30
Cia 1 good	—	2	—	6	2	4	14
Cia 1 bad without continuation	3	—	13	1	—	—	17
Cia 1 bad with continuation	—	—	—	4	—	—	4
schemata	3	2	2	—	—	—	7

supplied examples, the number of generated examples by Clint, the number of good guesses by Cia, the number of bad guesses by Cia (this is divided into two different numbers: the first one gives the number of bad guesses before a good clause was guessed, the second one gives the number of guesses after a good clause was guessed; this difference is related to the option in Cia to continue after guessing a good clause; see also the algorithms) and the number of derived schemata from clauses for the predicate. In our context, because Cia part 2 was turned off, the option without continuation is the most relevant one.

These results indicate that the order of learning the predicates is not always important. Indeed, one would have expected that the first experiment would have required less clause questions than the second one because on the average the system knows less schemata. This is not the case in our experiments because:

- the size of the schemata for `rook_move` is smaller than the one for `bishop_move` and the ones for `pawn_move`.
- one special event occurred in the first experiment: one clause for `knight_move` that was learned by Clint contained a redundant condition, i.e., given our knowledge base of chessboards, it was a syntactic variant of the target clause. The syntactic variant was retained and the corresponding schema was successfully applied when learning `pawn_move`. This event also explains why 3 schemata were derived from `knight_move` and only two from `pawn_move` in the first experiment.

The experiments are quite successful. Indeed, the number of generated questions is—on the average—quite acceptable to learn a rather complicated knowledge base. If we only count the bad questions of Cia without continuation, we have a rate of success of about

45 percent for clause questions. If they are counted with continuation we get 1/3 for the first experiment and 2/5 for the second one.

Although the number of questions generated by Cia is—from the global point of view—quite acceptable, this is not the case at certain points: the learning of pawn_move in the first experiment and the learning of king_move in the second experiment. At those points many questions are generated. In both cases this can be explained by the fact that the size of the starting clause is relatively large as compared to the size of the known schemata.

If we estimate the number of membership questions that would have been required by Clint by taking the average number of generated membership questions for the clauses learned by Clint, we arrive at an expected number of generated membership questions of about 90. If we compare 90 membership questions to 32 (30) membership questions and 31 clause questions, we see that one clause question corresponds to about two membership questions.

6.3. Evaluation of the experiments with Cia

6.3.1. A fundamental difficulty

Before evaluating our experiment with Clint-Cia, we want to point out a fundamental difficulty with evaluating systems as Cia. Although, it has been argued in the literature (Kibler & Langley, 1988) that machine learning is an experimental science and that machine learning techniques can be validated by carefully designed and controlled experiments, we believe that for certain techniques (of which Cia is one) this does not apply. The work on discovery systems like, e.g., AM and Eurisco (Lenat, 1983; Lenat & Brown, 1983) has received much critique (Ritchie & Hanna, 1983) because there is no straightforward experiment nor a sound theoretical basis which allows to evaluate them. However, it is not doubted that the programs work and that AM and Eurisco are excellent pieces of artificial intelligence research. We believe that the same problem with evaluation methods holds for Cia. The only weak theoretical basis on which it relies is that schemata are useful abstractions of predicate definitions. This usefulness of schemata was also noted and exploited by several other researchers (but never justified) such as Emde, Habel and Rollinger (1983), Morik (1989), Wrobel (1989), Thieme (1989), and Yokomori (1986). The only other justification we can provide for the Cia technique is the description of working experiments. Performing systematic experiments is not feasible because the experiments depend too much on the learning situation and application. There is—for instance—no known generator of relevant knowledge bases.

Despite the fundamental difficulties involved in evaluating Clint-Cia, we believe that our experiments show that

> *the* Constructive Induction by Analogy *technique is useful at least for domains with many (structurally) related predicates.*

Furthermore, the experiments in the chess domain show that there are nontrivial domains that have structurally related predicates.

6.3.2. Asking less questions

In other domains, with less structure, the only problem may be that the system asks too many questions to the user or proposes uninteresting predicates. As far as the number of questions is concerned it is very important that Cia learns in the context of Clint. The reason why the number of proposed concept-descriptions is constrained from the start, is that Cia matches schemata only with starting clauses generated by Clint. These clauses have—usually—only a small number of conditions in their body. Another important property of these starting clauses is that they cover the positive example supplied by the user, which means that the clause can be fully instantiated such that all conditions in it are true. This property assures that all concepts proposed by Cia are meaningful ones because there are known positive examples for these concepts.

This does not mean, however, that there may not be situations in which the system generates too many questions. There are many ways to ask less questions:

- First, one may want to use only the first part of Cia (this is an option in Cia). This will result in less questions since part 2 is the part in which most matches occur.
- Second, bad schemata should be excluded from the knowledge-base. E.g., if Clint learned `vehicle(X) ← car(X)`, it should not enter the derived schema in its knowledge-base, since this schema is too general to be useful. One way to avoid entering such schemata, is to ask the user whether the schema is interesting enough to be added into the knowledge-base. Another possibility is to measure the interestingness of the known schemata and retract the ones that fall below a certain threshold. Such a measure can take into account, e.g., the rate of successful applications of the schema and the regularity of the schema. In the first case, one could award and punish schemata for each application, like is done with rules in, e.g., Sage (Langley, 1985).
- Third, the schemata may be refined to also take into account some of the semantics of the original predicate. Two promising refinements are:
  — adding information about the types of the arguments in the original predicate and
  — use information about the properties of the predicate (this is related to, e.g., symmetry, number restrictions in KL-ONE (Brachman & Schmolze, 1985)).

  As an example of these extensions, consider the `father` definition. For `father` we have that
  — the types of `F` and `C` are related: the type of `F` is a subset of the type of `C` (all males are humans) and
  — there is exactly one father and two parents for each `C`, while for each `F` there can be many `C`'s.

  Refined schemata will match less clauses and hence they will give rise to less questions.
- Fourth, if there would be an explicit goal (like in Wrobel (1988)) for the learning system, then it would also be much easier to decide whether a question is useful or not. However, it is not clear how Clint-Cia can become goal- or demand-driven.

6.3.3. Analogy

If we reconsider the framework of analogy from section 5.1, then it is clear that the Cia technique can be viewed as if it were performing analogy. First, although problems are

specified by a positive example and a knowledge base, Cia acts on the starting clause computed from the positive example, the knowledge base and the language. So, problems are represented by starting clauses and solutions are clauses that are instantiations of particular schemata.

- α(Problem-1, Solution-1) is satisfied if Solution-1 is a clause learned by Clint for Problem-1;
- α'(Problem-2, Solution-2) is satisfied if Solution-2 is the instantiated schema $T\Theta$ obtained by (partially or completely) matching the schema with the starting-clause;
- β'(Solution-1, Solution-2) is satisfied if both clauses corresponding to Solution-1 and Solution-2 are instantiations of the schema T
- β(Problem-1, Problem-2) is satisfied if both clauses (partially or completely) match the same schema T.

Some people may argue that Cia only performs a very weak form of analogy since only syntactical information is mapped from source to target and there is no model of causality taken into account.[8] In terms of the framework of Kodratoff (1990), our notion of analogy would be termed trivial. However, there is no reason why trivial analogies cannot be useful. First, augmented with more semantic information about the source problem, more information could be mapped (see example 12). Secondly, some researchers as, e.g., G. Tecuci (personal communication) have argued that one could explain Cia in a causal context. However, we feel that the point whether Cia performs analogy or not, is not really important and more a matter of terminology. The main reason why we explained Cia as performing analogy is that is was inspired in this way and also that it is a very convenient way to explain what happens.[9] Thirdly, the experiments with Cia have shown that some questions generated by Cia certainly involve analogy. This point is worked out in some detail in example 11.

Example 11. Analogous questions generated by Cia

Let us consider the following clause generated by Clint:

```
rook_move(A1, A2, A3, A4, A5) ← board_nr(A5) ∧ rook(A1) ∧
    same_row(A2, A3) ∧ occupies(A1, A2, A5) ∧
    occupies(A4, A3, A5) ∧ opposite_color(A1, A4) ∧
    nothing_in_between(A5, A2, A3)
```

The corresponding schema was used by Cia to generate the following clause question:

```
bishop_move(A1, A2, A3, A4, A5) ← board_nr(A5) ∧ bishop(A1) ∧
    same_diagonal(A2, A3) ∧ occupies(A1, A2, A5) ∧
    occupies(A4, A3, A5) ∧ opposite_color(A1, A4) ∧
    nothing_in_between(A5, A2, A3)
```

Clearly those two clauses are very much related; the only difference being that same_row is replaced by same_diagonal and rook by bishop. Now, for everyone knowing the rules of chess it is known that a bishop is allowed to move along diagonals in a way that a rook is allowed to move along rows or columns. This is also the way in which a bishop is analogous to a rook. □

6.3.4. Towards an intelligent predicate editor

Example 11 does not only show that Cia sometimes realizes real analogy, but it also motivates the use of our Constructive Induction by Analogy technique in a different fashion. In the example, Cia might also have generated a different question, which corresponds to the proposed clause with `same_diagonal` replaced by `vertic_distance_2`. Such a question might be generated when there is also a literal for `vertic_distance_2` in the starting clause. Clause questions related to other instantiations of the same schema can also be imagined.

Therefore, it seems interesting to present the clause questions corresponding to instantiations of one schema not as individual clause questions but as one more complicated, but more compact question. This question would then allow the user to edit the clause he/she wants. This is illustrated in example 12.

Example 12. Editing clauses

Consider the previous example and the two clause questions:

```
bishop_move(A1, A2, A3, A4, A5) ← board_nr(A5) ∧ bishop(A1) ∧
     same_diagonal(A2, A3) ∧ occupies(A1, A2, A5) ∧
     occupies(A4, A3, A5) ∧ opposite_color(A1, A4) ∧
     nothing_in_between(A5, A2, A3)

bishop_move(A1, A2, A3, A4, A5) ← board_nr(A5) ∧ bishop(A1) ∧
     vertic_distance_2(A2, A3) ∧ occupies(A1, A2, A5) ∧
     occupies(A4, A3, A5) ∧ opposite_color(A1, A4) ∧
     nothing_in_between(A5, A2, A3)
```

These questions could be replaced by showing the user:

```
bishop_move(A1, A2, A3, A4, A5) ← board_nr(A5) ∧ bishop(A1) ∧
     (same_diagonal(A2, A3) or vertic_distance_2(A2, A3)) ∧
     occupies(A1, A2, A5) ∧ occupies(A4, A3, A5) ∧
     opposite_color(A1, A4) ∧ nothing_in_between(A5, A2, A3)
```

and asking him/her which instantiation(s) she/he wants. In order to point out the analogy with previously learned clauses or concepts the source clause should be shown next to these questions

```
rook_move(A1, A2, A3, A4, A5) ← board_nr(A5) ← rook(A1) ∧
     same_row(A2, A3) ∧
     occupies(A1, A2, A5) ∧ occupies(A4, A3, A5) ∧
     opposite_color(A1, A4) ∧ nothing_in_between(A5, A2, A3)
```

□

Observe that this method is also applicable when there are more than two instantiations of one schema. Furthermore, the proposed method generates at most one question for each schema and exploits the analogy in a better way. Therefore, it may be even more feasible in practice than the original Cia technique.

6.3.5. Learning how to learn

The integrated system Clint-Cia realizes constructive induction since it is able to invent new concepts. However, it also implements a kind of learning how to learn. Indeed, learning new concepts with Clint allows to derive (and use) new schemata and since the behavior of Clint-Cia depends very much on the schemata, changing the set of schemata results in changing the learning. Hence, Clint-Cia adapts its learning: it gradually evolves from a pure data-driven learner (Clint) to a mixed model- and data-driven algorithm. One expects, that as more useful schemata are known, that this learning behavior will improve (this was certainly the case in our experiments). One promising further aspect is that schemata (i.e., learning knowledge) acquired in one domain could also be used in other domains. If one compares Clint-Cia to the learning learning machines discussed by Stepp, Whitehall and Lawrence (1988), one can easily see that Clint-Cia has all characteristics of learning learning machines (De Raedt & Bruynooghe, 1989b), except that it does not chunk or transform its knowledge into a more operational form. None of the systems discussed by Stepp et al. possess so many of these characteristics.

7. Related work

7.1. For Clint

The work on Clint is related to the work on versionspaces (Mitchell, 1982), learning apprentices (Kodratoff & Tecuci, 1987; Tecuci & Kodratoff, 1990), explanation based learning (Mitchell, Keller & Kedar-Cabelli, 1986; De Jong & Mooney, 1986), experimentation (Subramanian & Feigenbaum, 1986; Sammut & Banerji, 1986; Krawchuk & Witten, 1988), indirect relevance (Buntine, 1987), identification in the limit and debugging (Shapiro, 1983), shift of bias (Utgoff & Mitchell, 1982; Utgoff, 1986), and inductive logic programming (Muggleton & Buntine, 1988; Muggleton & Feng, 1990; Rouveirol & Puget, 1989; Quinlan, 1990).

Clint's basic algorithm is very simlar to version-spaces, except that Clint stores the negative examples instead of the G-set. For rich description languages this is often more efficient (De Raedt & Bruynooghe, 1988). The search strategy of Clint within one language corresponds closely to the breadth-first algorithm proposed in Mitchell (1982).

As Clint builds justifications for examples, which can be considered as a kind of plausible explanation, in an inductive way it attempts to overcome the strong theory requirements imposed by explanation based learning. Therefore it also needs more than one example to find a new definition.

Clint asks membership questions to the user, just like Factoring (Subramanian & Feigenbaum, 1986), Marvin (Sammut & Banerji, 1986) and Alvin (Krawchuk & Witten, 1988).

However, none of these techniques copes with indirect relevance, because they all require that the relevant relations in an example are specified. Subramanian and Feigenbaum use some kind of propositional logic, while we use subsets of first order logic. Factoring is more powerful but less general than Clint because it requires that the space is factorable. For Marvin and Alvin it is less clear which concepts can be learned. Also, their generalization operator is incomplete. Alvin is less efficient and more powerful because it always tries to construct crucial objects, while Clint constructs significant objects. Also, Marvin and Alvin cannot shift their bias and only cope with a more limited form of indirect relevance (from some given relations it is possible to induce other ones). Coping with indirect relevance in Clint is very similar to the technique in the Plausible Generalization Algorithm (Buntine, 1987). Only, the Plausible Generalization Algorithm is unable to shift its bias or to generate examples.

Clint is related to the learning component of Disciple (Kodratoff & Tecuci, 1987; Tecuci & Kodratoff, 1990). However, it does not suffer from some of the problems with Disciple (Tecuci, 1988) such as the use of a restricted versionspace approach, limited explanations and no means for error-recovery.

The debugging method used in Clint is adapted from Shapiro (1983), and optimized not to ask questions about basic predicates. Previous work on shifting the bias concentrated on propositional logic (Utgoff, 1986). One of the main contributions of Clint is the introduction of a series of languages, which are subsets of first order logic and which can be used to shift the bias. These languages are computed dynamically from one example and the knowledge-base.

Clint is also related to other systems that induce logic programs from examples. This includes the approaches of Quinlan (1990), Muggleton and Buntine (1988), Muggleton and Feng (1990), and Rouveirol and Puget (1989). There are, however, some important differences between Clint and these other approaches:

- The approaches of Quinlan (1990), Muggleton and Buntine (1988), and Muggleton and Feng (1990) aim at inducing rules from large collections of examples. Therefore they focus on one concept and use heuristics to prune the search. Furthermore, they do not address issues such as convergence and learnability, example-generation, shift of bias and the interactions among the learning of different predicates.
- On the other hand, Clint is not applicable to learning rules from a given large collection of examples. Two features of Clint prevent this: Clint is unable to cope with uncertain and approximate rules and Clint always generates examples.

Despite these differences, we are convinced that an integration of both approaches could be realized. One interesting feature of Clint for the other systems is the shift of bias. Clint could also be adapted towards these other systems by generalizing from given examples. This could be realized by computing the starting clauses from the given examples and computing the least general generalization of them.

Interesting further extensions of Clint are described in De Raedt (1991), De Raedt, Feyaerts and Bruynooghe (1991), De Raedt, Bruynooghe and Martens (1991) and De Raedt and Bruynooghe (1990b). They are concerned with handling integrity constraints while learning, the acquisition of fact knowledge about objects and the use of negation and a three-valued logic for concept-learning.

7.2. For Cia

The Cia technique is related to model driven learning in Blip (Morik, 1989; Emde, Habel & Rollinger, 1983; Wrobel, 1989), applying analogy on concept-learning (Vrain & Lu, 1988), analogy (Yokomori, 1986), constructive induction (Rendell, 1989; Muggleton & Buntine, 1988; Muggleton, 1987) and discovery (Lenat, 1983).

The second order schemata and the model-driven way of using them is very similar to the Blip learner (Morik, 1989; Emde, Habel & Rollinger, 1983; Wrobel, 1989). However, Blip mainly uses the schemata to learn given concepts while Cia also uses them to invent new concepts. Also, the application conditions of a schema are completely different in Blip, where a heuristically restricted generate and test method is used, while in Cia a schema is matched against the starting-clause derived by Clint. The procedure to decide whether to accept or reject the instantiated schema is also different: Blip takes into account the number of positive and negative instances while Cia also relies on the user. In Clint-Cia it is not always possible to consider the positive and negative instances, since there may only be one example given. Blip can also invent new concepts in context (Wrobel, 1988). In Wrobel (1988) the invention is demand-driven while in Cia it occurs as opportunities for learning new concepts arise. This learning in context allows to reduce the search-space. An important difference with respect to learning how to learn is that the schemata are automatically derived by Clint-Cia but not by Blip (but see (Thieme, 1989)). Also, Blip is a passive learner, which does not want to ask the user questions while Clint-Cia is an active learner, which asks many questions to the user.

In the much more general setting of logic programming the analogy between different predicate definitions was also noticed and exploited by Yokomori (1986). He defines a notion of analogy of which ours seems to be a special case. He also indicates how it might be used to solve problems by analogy but he does not address issues of learning (and learning how to learn) and when to apply this analogical reasoning.

Cia uses the same paradigm as Vrain and Lu (1988) in which concept-learning is viewed as problem-solving and analogy is applied. However, Vrain and Lu apply analogy on a part of the concept-learning process (changing a concept-description for a certain example) whereas we apply it on whole concept-learning tasks.

Cia is also related to work on constructive induction (Muggleton & Buntine, 1988; Muggleton, 1987; Rendell, 1989; Mehra, Rendell & Wah, 1989). However, this work on constructive induction has concentrated on discovering an appropriate vocabulary for reexpressing one particular given concept learning problem. Our approach differs from these approaches since it attempts to suggest the user valuable concepts to include in the knowledge base. Also, the suggested new concepts are not used for the given task but rather for future learning tasks. Seen in this light, Cia is closer to system like AM and Eurisco (Lenat, 1983) which attempt to define interesting concepts. Nevertheless, the context, the heuristics and evaluation methods in the approach of Lenat and ours are completely different.

Acknowledgments

Since this work started in 1987, many people have made useful suggestions on Clint-Cia in various stages. We would like to thank G. Tecuci, G. Sablon, J.F. Puget, and D. De Schreye

for various discussions on Clint-Cia. Most of all we are grateful to Yves Kodratoff and Katharina Morik for many discussions, comments, suggestions and for inviting Luc at their institutes in Paris and Bonn during the summers of 1988 and 1989. It was during these visits that most of the work on Clint-Cia was inspired by the systems Disciple and Blip. They were supported by the Belgian National Fund for Scientific Research by means of an IBM-travel grant (Paris) and the German National Research Center for Computer Science (Bonn). Maurice and Luc are supported by the Belgian National Fund for Scientific Research.

We would also like to thank the referees for their valuable comments and suggestions.

Notes

1. `eq` is a system-defined predicate that is always considered by Clint. It is defined by `eq(X, X)` ←.
2. Our concept 1 was encoded by Quinlan (1990) without the predicates `suit` and `rank`; this explains why Foil needed only two literals to describe the concepts.
3. For clause questions, this form is retained. However, for new term questions we have to instantiate the head of the proposed schema to the same predicate by applying a second order substitution Θ such that $q_2\Theta = q_1\Theta$ and Θ introduces a predicate name that is not already present in the knowledge base.
4. These experiments were carried out with the help of Hans Wouters.
5. Alternatively, using a recent extension of Clint, this knowledge could have been stated by means of constraints, see De Raedt (1991) and De Raedt, Bruynooghe and Martens (1991).
6. This point is related to the used knowledge representation. Given the knowledge representation there is no other way of preventing pieces to move from one boardsituation to another one or to move a piece p from a tile on which p does not stand to another tile. Apart from generating additional membership questions, there seems to be no other way around this problem, even when a different knowledge representation would have been used.
7. In this application with the sketched knowledge base it is impossible to invent novel concepts; this is due to the additional knowledge that was integrated in Clint's language.
8. The problems with viewing Cia as performing analogy were pointed out to us by Yves Kodratoff, Katharina Morik, Gheorge Tecuci and Stefan Wrobel.
9. We follow Sigmund Freud in this respect.

References

Aben, M., & Van Someren, M. (1990). Heuristic refinement of logic programs. In L.C. Aiello (Ed.), *Proceedings of the Ninth European Conference on Artificial Intelligence* (pp. 7–12). Pitman.

Brachman, R.J., & Schmolze, J.G. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science, 9*, 171–216.

Buntine, W. (1987). Induction of horn-clauses: Methods and the plausible generalization algorithm. *International Journal of Man-Machine Studies, 26*, 499–520.

Buntine, W. (1988). Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence, 36*, 375–399.

Chouraqui, E. (1985). Construction of a model for reasoning by analogy. In L. Steels and J.A. Campbell (Eds.), *Progress in artificial intelligence*. Ellis Horwood.

De Jong, G., & Mooney, R. (1986). Explanation based learning: An alternative view. *Machine Learning, 1*, 145–176.

De Raedt, L. (1991). *Interactive concept-learning*. Ph.D. thesis, Department of Computer Science, Katholieke Universiteit Leuven.

De Raedt, L., & Bruynooghe, M. (1988). On interactive concept-learning and assimilation. In D. Sleeman (Ed.), *Proceedings of the Third European Working Session On Learning* (pp. 167–176). Pitman.

De Raedt, L., & Bruynooghe, M. (1989a). Constructive induction by analogy. In *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 476–477). Morgan Kaufmann.

De Raedt, L., & Bruynooghe, M. (1989b). Constructive induction by analogy: A method to learn how to learn? In *Proceedings of the Fourth European Working Session on Learning* (pp. 189-200). Pitman.

De Raedt, L., & Bruynooghe, M. (1989c). On explanation and bias in inductive concept-learning. In *Proceedings of the Third European Knowledge Acquisition for knowledge based systems Workshop* (pp. 338-353).

De Raedt, L., & Bruynooghe, M. (1989d). Towards friendly concept-learners. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 849-856). Morgan Kaufmann.

De Raedt, L., & Bruynooghe, M. (1990a). Indirect relevance and bias in inductive concept-learning. *Knowledge Acquisition*, 2, 365-370.

De Raedt, L., & Bruynooghe, M. (1990b). On negation and three-valued logic in interactive concept-learning. In L.C. Aiello (Ed.), *Proceedings of the Ninth European Conference on Artificial Intelligence* (pp. 207-212). Pitman.

De Raedt, L., Bruynooghe, M. & Martens, B. (1991). Integrity constraints and interactive concept learning. In *Proceedings of the Eighth Machine Learning Workshop* (pp. 394-398). Morgan Kaufmann.

De Raedt, L., Feyaerts, J., & Bruynooghe, M. (1991). Acquiring object-knowledge for learning systems. In Y. Kodratoff (Ed.), *Proceedings of the Fifth European Working Session on Learning* (pp. 245-264), volume 2 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Dietterich, T.G., & Michalski, R.S. (1985). Discovering patterns in sequences of events. *Artificial Intelligence, 25*, 257-294.

Emde, W. (1989). An inference engine for multiple theories. In K. Morik (Ed.), *Knowledge Representation and Organization in Machine Learning*, volume 347 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Emde, W., Habel, C.U., & Rollinger, C.R. (1983). The discovery of the equator or concept driven learning. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (pp. 455-458). Morgan Kaufmann.

Genesereth, M., & Nilsson, N.J. (1987). *Logical foundatons of artificial intelligence*. Morgan Kaufmann.

Kibler, D., & Langley, P. (1988). Machine learning as an experimental science. In *Proceedings of the Third European Working Session on Learning* (pp. 81-92). Pitman.

Kodratoff, Y. (1988). *Introduction to machine learning*. Pitman.

Kodratoff, Y. (1990). Combining similarity and causality in creative analogy. In L.C. Aiello (Ed.), *Proceedings of the Ninth European Conference on Artificial Intelligence* (pp. 398-403). Pitman.

Kodratoff, Y., & Tecuci, G. (1987). Disciple-1: Interactive system in weak theory fields. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 271-273). Morgan Kaufmann.

Krawchuk, B., & Witten, I. (1988). On asking the right questions. In *Proceedings of the Fifth Machine Learning Conference* (pp. 15-22). Morgan Kaufmann.

Langley, P. (1985). Strategy acquisition governed by experimentation. In L. Steels and J. Campbell (Eds.), *Progress in artificial intelligence*. Ellis Horwood.

Lenat, D. (1983). The role of heuristics in learning by discovery: Three case studies. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. Tioga Publishing Company.

Lenat, D., & Brown, J.S. (1983). Why AM and EURISCO appear to work. *Artificial Intelligence, 23*, 269-294.

Loveland, D.W. (1978). *Automated theorem proving: A logical basis*. North-Holland.

Mehra, P., Rendell, L.A., & Wah, B.W. (1989). Principled constructive induction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 651-657). Morgan Kaufmann.

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. Morgan Kaufmann.

Michalski, R.S., Carbonell, J.G., & Mitchell, T.M. (1983). *Machine Learning: An Artificial Intelligence Approach*, volume 1. Morgan Kaufmann.

Minton, S. (1988). Quantitative results concerning the utility of explanation based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 564-569). Morgan Kaufmann.

Mitchell, T.M. (1982). Generalization as search. *Artificial Intelligence, 18*, 203-226.

Mitchell, T.M., Keller, R.M., & Kedar-Cabelli,, S.T. (1986). Explanation based generalization: A unifying view. *Machine Learning, 1*, 47-80.

Mitchell, T.M., Mahadevan, S. & Steinberg, L.I. (1985). Leap: A learning apprentice for VLSI design. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 573-580). 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*. Springer-Verlag.

Muggleton, S. (1987). Duce, an oracle based approach to constructive induction. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 287–292). Morgan Kaufmann.

Muggleton, S., & Buntine, W. (1988). Machine invention of first order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning* (pp. 339–351). Morgan Kaufmann.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*. Oshima, Tokyo, Japan.

Poetschke, D. (1989). Analogical reasoning for second generation expert systems. In K. Jantke (Ed.), *Analogical and Inductive Inference* volume 397 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Quinlan, J.R. (1990). Learning logical definition from relations. *Machine Learning*, *5*, 239–266.

Rendell, L. (1989). Comparing systems and analyzing functions to improve constructive induction. In *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 461–464). Morgan Kaufmann.

Rendell, L., Seshu, R., & Tcheng, D. (1987). More robust concept-learning using dynamically variable bias. In *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 66–78). Morgan Kaufmann.

Ritchie, G.D., & Hanna, F.K. (1983). AM: A case study in AI methodology. *Artificial Intelligence*, *23*, 249–268.

Rouveirol, C., & Puget, J.-F. (1989). A simple solution for inverting resolution. In K. Morik (Ed.), *Proceedings of the Fourth European Working Session on Learning* (pp. 201–211). Pitman.

Sammut, C., & Banerji, R. (1986). Learning concepts by asking questions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach*, volume 2. Morgan Kaufmann.

Shapiro, E.Y. (1983). *Algorithmic program debugging*. The MIT Press.

Smith, R.G., Mitchell, T.M., Winston, H.A., & Buchanan, B.G. (1985). Representation and use of explicit justifications for knowledge base refinement (pp. 673–680). In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann.

Stepp, R.E., Whitehall, B.L., & Lawrence, B.H. (1988). Towards intelligent machine learning algorithms. In *Proceedings of the Eighth European Conference on Artificial Intelligence* (pp. 333–338). Pitman.

Subramanian, D., & Feigenbaum, J. (1986). Factorization in experiment generation. In *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 518–522). Morgan Kaufmann.

Tecuci, G. (1988). *DISCIPLE: A theory, methodology and system of expert knowledge acquisition*. Ph.D. thesis, Universite Paris-Sud, Orsay.

Tecuci, G., & Kodratoff, Y. (1990). Apprenticeship learning in nonhomogeneous domain theories. In Y. Kodratoff and R.S. Michalski (Eds.), *Machine learning: An artificial intelligence approach*, volume 3. Morgan Kaufmann.

Thieme, S. (1989). The acquisition of model-knowledge for a model-driven machine learning approach. In K. Morik (Ed.), *Knowledge Representation and Organization in Machine Learning* volume 347 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Utgoff, P.E. (1986). Shift of bias for inductive concept-learning. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, (Eds.), *Machine learning: An artificial intelligence approach*. Morgan Kaufmann.

Utgoff, P.E., & Mitchell, T.M. (1982). Acquisition of appropriate bias for concept learning. In *Proceedings of the Second National Conference on Artificial Intelligence* (pp. 414–418). Morgan Kaufmann.

Vrain, C., & Lu, C.R. (1988). An analogical method to do incremental learning of concepts. In *Proceedings of the Third European Working Session on Learning* (pp. 227–235). Pitman.

Wrobel, S. (1988). Automatic representation adjustment in an observational discovery system. In D. Sleeman (Ed.), *Proceedings of the Third European Working Session on Learning*. Pitman.

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* (pp. 253–262). Springer-Verlag.

Yokomori, T. (1986). Logic program forms. *New Generation Computing*, *4*, 305–320.