COMP9417 Project
Shapiro's Model Inference System

Marc Ahrens, marca@cse.unsw.edu.au
Angela Finlayson, angf@cse.unsw.edu.au
Peter Gammie, peteg@cs.mu.oz.au
`http://www.cs.mu.oz.au/~peteg/CompSci/mis/`

March 23, 2002

Thanks to Mark Reid (mreid@cse.unsw.edu.au) for help with Section 3.

Contents

1 Introduction

This section is a brief introduction and overview of Machine Learning, Logic Programming, Inductive Logic Programming, and Shapiro's Model Inference System.

Source code for the applets used in this document can be found at:
http://www.cs.mu.oz.au/˜peteg/thesis/src/prolog/prolog.html"
(I've merged it with my thesis work).

1.1 Logic Programming

Logic Programming is the area that broadly includes languages based in a fairly direct way on a logic, which is usually taken to be classical first-order Horn-clause logic. These semantical foundations are usually augmented with *negation-as-finite-failure*, meta-programming facilities (e.g. `call/1`, `asserta/1`, `retract/1`) and type, mode, and module systems. Prolog is the original such language, dating from the early 1970's, and lacks all such niceties except the first two.

A tutorial introduction to Prolog can be found in [CM81], and [Llo87b] provides a thorough treatment of the semantics of a purified version of the language.

In the following, we assume familiarity with logic programming. The specialised terminology of this field is described in Appendix A.

1.2 Proof by Refutation

Prolog uses a form of *resolution* [Rob92] to execute logic programs. Restricted to Horn clauses, this proof technique uses the following rule of inference (a reworded version of [Sha81]):

> Let $C_1 = \leftarrow r_1, q_1, ..., q_n$ (a *definite query*), $C_2 = r_2 \leftarrow s_1, ..., s_n$ (a *definite clause*) be sentences in L. Provided that r_1 and r_2 have a *most general unifier* θ (i.e. $P = r_1\theta = r_2\theta$), then the *resolvent* of these two clauses is $(\leftarrow q_1, ..., q_n, s_1, ..., s_n)\theta$, and P is termed *the atom resolved upon*. C_1 is called the *left component*, and C_2 the *right component*.

The aim is to derive the empty clause, written $\square$, which is interpreted as "contradiction". This means that the query is inconsistent with the theory, from which we conclude that the negation of the query is in fact consistent with it. What's more, substitutions for the variables in the query that satisfy the latter are computed during the search for a refutation.

Horn clauses are particularly useful, as they have a very natural algorithmic interpretation; they have the flavour of the *procedures* found in imperative langauges. The output of such an executation is a sentence that satisfies the constraints asserted by the program, or failure if no such solution is found. In $r \leftarrow q_1, ..., q_n$, r is called the *head* of the clause and $q_1, ..., q_n$ the *body*. To execute the head, each conjunct of the body must be executed.

Refutation proof trees[1] provide a way of visualising the logical aspect of executing a Prolog program, in contrast to the procedural (or operational) view that a debugger for an imperative language (such as C) would provide. The completed refutation tree should be read as "because all the children are true, the parent is true in the context of the theory T".

1.3 Inductive Logic Programming

The goal of Inductive Logic Programming is to induce a compact, generalised, understandable-by-humans description of a set of facts in a very expressive formal language. The term was coined by Muggleton in 1991 [Sam93], but the field had been under active investigation for a long time before that. An overview of rule-learning (of which ILP is an instance) can be found in [Mit97, Chapter 10], and a substantial amount of ILP theory is presented in [NCdW97].

[1] As distinct from *resolution proof trees* where the emphasis is on what objects participate in each step of the proof. We elide such details here as the emphasis is on searching through the intermediate (ground) results of a proof to identify faulty clauses in the theory (Section 2).

Given the gamut of programming languages, we might ask what it is about logic programming that makes it a good target for induction. We defer answering this question until we have presented the Model Inference System (Section 5) so that we have a point of reference.

Alternatively, it may be asked why a more expressive formal language such as full first-order logic is not employed. The main reason is efficiency: a theory induced by an ILP system is effectively and efficiently *executable* via a Prolog-like theorem prover, and does not require a massively non-deterministic search to determine whether a particular statement follows.

There is a plethora of ILP systems, e.g. FOIL [QCJ93], MARVIN [SB86], Progol [Mug95], and many variations on some core ideas. The two fundamental approaches taken to the induction problem are to exploit the properties of the description language (e.g. MIS) and to use a statistical model (e.g. FOIL). Any effective ILP system needs to combine them in some way: the former can greatly increase efficiency by better structuring the search space, and the latter is mandated by noisy (real-world) data. Again, we discuss these after presenting MIS.

In order to provide some intuition about ILP, the following points contrast it broadly with decision tree inducers [Mit97, Chapter 3 and 10]:

- Horn-clause logic is much more expressive than the attribute/value framework used by decision trees (which is effectively propositional, or zeroth-order, logic). This means that the task of inducing a theory is both much harder and much less efficient, and so ILP should only be used when this extra expressiveness is necessary.

  More concretely, decision trees can only describe relationships between particular attributes and constants, whereas ILP can describe relationships amongst attributes.

- A theory induced by an ILP system can hold on an infinite domain; we might say the system can generalise to the infinite case. For example, MIS can learn an addition predicate that is defined on *all* triples of natural numbers.

  For this to be the case, it must be possible to express recursive definitions in the hypothesis language[2].

  Indeed, as all finite relations can be encoded in a propositional language, this is at the core of ILP's power. A survey that focusses on precisely this topic is [FY99], where several techniques for providing structured recursion operators (such as divide-and-conquer) are discussed.

- ILP systems can (usually) make use of *background knowledge*, although it is a matter of current research as to how to do this efficiently.

1.4 Philosophical Concerns

Abstractly, we might pose the question: just what is a valid inductive generalisation? From a classical philosophical viewpoint[3], a satisfactory answer to this question is needed if we are ever to escape the "I think, therefore I am" solipsism of Descartes and be assured that it is possible to make valid assertions about the inter-subjective realm.

Hume [Hum85] argued quite forcefully in the eighteenth century that there can be no logically sound reason to generalise from experience to univerally true statements. The essence of his argument is that we must assume that nature is uniform if we were to grant validity to such statements, and that such an assumption cannot be verified indepedently of experience.

So, are we completely stuck? Surely not. In the context of machine learning we're simply looking for good descriptions that fit the data we have available – there is no inherent claim for universal generality here. Indeed, completeness is not always in itself desirable – a topic we take up in Section 3.

[2]Once we allow such objects in the hypothesis space, we have all sorts of computability issues to deal with. For example, it is no longer the case that the process of checking that the theory entails the observations necessarily terminates.

[3]Prior to Kant, that is.

```
P <- the program containing only the empty clause

while(true) {
    read a new example.

    while(P is not consistent with the available examples) {
        if(P entails e and e is a negative example) {
            weaken / specialise P.
        }

        if(P does not entail e and e is a positive example) {
            strengthen / generalise P.
        }
    }
}
```

Figure 1: The skeleton of a generalise/specialise learner.

In contrast to empirical scepticism, we can simply assume that generalisations are necessary (perhaps as an aid to comprehending large amounts of data) and move along to the question of what a *good* way of discovering them is. In concert with this, we need to know what the *requirements* for effective learning are; can we, for example, learn simply by being exposed to positive examples of a concept?

While the former question is a matter of research, the latter one and others similar to it have been studied in some detail by computational learning theorists – see [JORS99] for an in-depth coverage of such.

Shapiro adopts the *learning-in-the-limit* model [Gol67] in order to prove various properties of his algorithms. Informally, the idea is that, given a *presentation* of facts in an observation language L_o, the learner outputs a series of conjectures drawn from a hypothesis language L_h that eventually converges to a finite description of them[4]. Note that there is no introspection requirement: the learner never realises that it has happened upon the correct description. Thus, the crux of Hume's argument – that we cannot establish uniformity – is side-stepped.

In the context of this model, both positive and negative examples are required; without negatives we could trivially output the everywhere-true program, and similarly for the no-positives case.

1.5 An Overview of the Model Inference System

The Model Inference System, by Ehud Y. Shapiro [Sha81, Sha83] is an instance of the class of incremental generalise/specialise algorithms, which have the skeleton shown in Figure 1.

Shapiro formalises the *Model Inference Problem* as follows [Sha81]:

> Suppose we are given a first order language L and two subsets of it: an *obervational language* L_o and a *hypothesis language* L_h with $\Box \in L_o \subset L_h \subset L$. In addition, assume that we are given an oracle for some unknown model M of L. The Model Inference Problem is to find a finite L_o-complete axiomatization of M.

Unpacking this definition, we have the following:

- MIS takes as input a sequence of observations, where every statement α in L_o (eventually) appears and is flagged as either *true* or *false*. Such pairs are called *facts*.

[4] Obviously there are plenty of technical caveats if this is to go through: for example, the observations must completely characterise the target set, which is required to be describable in some formalism (in other words, it must be at least recursively enumerable).

- It is assumed that there is an *oracle* that can determine the truth of any ground statement in L relative to the model M[5]. This kind of learning is termed *supervised* as it relies on the existence of an entity capable of answering questions about the domain, which is not always a reasonable assumption (Section 5).

  The oracle can be formalised as a set facts $< \alpha, Boolean >$, each of which reflect the truth or falsity of α in the target model M.

- The output of this algorithm is a *conjecture*, which is a set of Horn clauses that entail all the positive observations, and none of the negatives, that have been presented to it so far.

  The original MIS [Sha81] can actually learn theories in full first order logic, but we restrict it to Horn Clauses (i.e. Prolog rules) so that the learnt descriptions are executable. More technically, there are some restrictions on the languages involved; Shapiro proves a theorem that $L_o = \{$ ground atoms of $L\}$, $L_h = \{$ Horn sentences of $L\}$ is *admissible*, i.e. a model defined over L is learnable in the limit.

MIS itself consists of two parts: roughly, a systematic way of finding bugs in programs (discussed in Section 2), and a search for fixes for these bugs (Section 3).

Both of these exploit the clean semantics of the pure Horn Clause core of Prolog, and hence run into difficulties with the non-logical parts of the language (e.g. the cut - `!/0`, unsound unification and the incompleteness of Prolog's proof strategy). Indeed, learning with the cut has since been shown to be difficult [BGT93].

Some further *a priori* observations:

- MIS assumes a fixed language; that is, it doesn't attempt to deal with the situation where a change in representation is required. In machine learning terms, it never tries to restrict the hypothesis language to the propositional sub-language of L, even when the observations can be described in such a framework.

  This is hardly surprising, however: it is quite difficult for even experts to invent effective transformations from "real world" domains to artificial languages.

- MIS is *incremental*, which means that a theory can always be later revised (although the state of the learner needs to be retained). This has implications for efficiency and completeness, which we discuss in more detail in Section 5.

- Shapiro proves a completeness result relative to Gold's learning in the limit framework with a *most general refinement operator*. Again, this is not of great practical interest for reasons detailed in Section 3.

- MIS can exploit background information, provided the system is made aware of it at the beginning of the induction process. See Section 5.1 for more discussion on background knowledge invention and use.

[5]In less logical language, MIS assumes that the user has an intuitive idea of or can empirically verify the ground facts in L. Loosely, the collection of these facts is termed a *model*, and the point of this exercise is to find a set of clauses that describes the model.

2 The Contradiction Backtracing Algorithm

The contradiction backtracing algorithm is the most interesting part of Shapiro's algorithmic error diagnosis framework [Sha83], as it has some philosophical implications for the refinement of scientific theories [Sha81]. This algorithm exploits the simple logical semantics of pure Prolog, and so an understanding of refutation theorem proving in general and the resolution calculus specifically is required (Section 1.2) to understand this section.

The following inputs are required:

- A theory T that proves a negative observation O^-. We say that T is *overly-general*; it is *unsound* relative to the concept the oracle has in mind.
- An oracle capable of answering queries about the truth of ground atoms in the model M.

The core of the algorithm is the following: as pure Prolog has only one mechanism for getting work done (a SLDNF-resolution step: unification of a body goal with the head of a clause), there are only a very few possible sources of bugs. In particular, if T can prove O^-, then it must be the case that there is at least one clause whose head is asserted false by the oracle, and whose body is deemed to be true in M (there is a proof of this assertion in [Sha83, p38]).

Given this characteristic of the faulty clause, diagnosing incorrect solutions boils down to searching T for one that has it. [Sha83, Section 3.2] presents several querying strategies with varying *query complexity*, which is defined to be the asymptotic number of times the oracle is consulted in terms of the size of the computation (proof) tree. In effect, this metric measures the data-hungriness of this diagnosis algorithm.

The search implemented by our applet is *top down*, shown in Figure 2 and described in [SS94, p337] like so:

> ...At each node it tries to find a false [child]. The algorithm recurses with any false [child] found. If there is no false [child], then the current node constitutes a counterexample [and indicates the faulty clause], as the goal at the node is false and all of its [children] are true.

It should be apparent that this algorithm queries the oracle a number of times that is a linear function of the depth of the proof tree.

Shapiro proves that his *divide-and-query* strategy (not presented here) is query-optimal, that is, it makes the most efficient use of the oracle for this task. The idea is an application of the classic divide-and-conquer paradigm, and bears a striking similarity to binary search: imagine the proof tree as a linear structure; query the node that lies nearest the middle, then recurse in the half where the bug must lie. This algorithm is logarithmic in the number of nodes in the proof tree.

Once such a bug has been discovered, the Model Inference System retracts the clause that has been identified as the culprit of the missing solution, and *marks* it as being unsatisfactory. A later pass will *refine* such marked clauses into more specific ones (Section 3).

2.1 Other Diagnosis Algorithms

While the backtracing algorithm can determine which clause is responsible for the generation of a false solution, logic programs can exhibit other unwanted behaviour:

- Finitely failing to prove something that it should be able to. Shapiro developed a *missing solution* algorithm that localises the cause of such bugs.
- Exhausting the stack during a proof attempt. There is a *stack overflow* algorithm that determines the cause of this mis-behaviour.

Shapiro has presented (some of) these algorithms in at least three places [Sha81, Sha83, SS94], with varying degrees of sophistication. These algorithms are useful for declarative debugging [Llo87a], independently of their usefulness in machine learning.

```
%
% Art Of Prolog: Program 17.15: Top-down diagnosis of a false solution
% false_solution(A, Clause) :-
%       If A is a provable false instance, then Clause
%       is a false clause in the program. Top down algorithm.
%
false_solution(A, Clause) :-
        fp_solve(A, Proof),
        false_goal(Proof, Clause).

fp_solve(true, true) :- !.
fp_solve((A,B), (ProofA, ProofB)) :- !,
        fp_solve(A, ProofA), fp_solve(B, ProofB).
fp_solve(A,    (A :- builtin)) :- builtin(A), !, call(A).
fp_solve(A,    (A :- Proof)) :- clause(A, B), fp_solve(B, Proof).

false_goal((A :- B),Clause) :-
        false_conjunction(B,Clause), !.
false_goal((A :- B),(A :- B1)) :-
        extract_body(B,B1).

false_conjunction(((A :- B),Bs),Clause) :-
        query_goal(A,false), !,
        false_goal((A :- B),Clause).
false_conjunction((A :- B),Clause) :-
        query_goal(A,false), !,
        false_goal((A :- B),Clause).
false_conjunction((A,As),Clause) :-
        false_conjunction(As,Clause).

extract_body(true,true).
extract_body((A :- B),A).
extract_body(((A :- B),Bs),(A,As)) :-
        extract_body(Bs,As).

query_goal(A,true) :- builtin(A).
query_goal(Goal,Answer) :-
        not(builtin(Goal)),
        writeln(['Is the Goal', Goal, 'true?']),
        read(Answer).
```

Figure 2: A Prolog program for backtracing contradictions.

3 Refinement

We saw in Section 2 how MIS goes about specialising a theory that is too general; the counterpart problem of generalising a theory that is too specific is carried out with the use of a *refinement operator*.

The following inputs are required:

- A theory T that fails to prove a positive observation O^+. We say that T is *too-specific*; it is *incomplete* relative to the concept the oracle has in mind.
- An oracle capable of answering queries about the truth of ground atoms in the model M.
- A set of *marked clauses* that the backtracing algorithm has previously deemed too general to describe M.
- A *refinement operator*, that, when applied to a marked clause, yields a logically weaker (i.e. more specific) clause.

  It should be noted that, even though the new candidate clause is less general than the previously-considered one, adding it does not decrease the number of facts entailed by the current theory. (It may not actually increase it, though.) So, the new conjecture should be considered a generalisation of the previous one.

Shapiro's generalisation algorithm is parameterised by both the refinement operator and the strategy that makes use of it. We deal with each in the following sections.

3.1 Refinement Operators

A *refinement operator* ρ suggests a logically weaker plausible replacement to a hypothesis refuted by the diagnosis algorithm outlined in Section 2. Formally, q is a refinement of p if it satisfies the following two criteria:

1. $p \vdash q$, i.e. everything provable from q is provable from p, or equivalently, p is at least as general as q.
2. $size(p) < size(q)$ under some size metric. In logic programming terms, Shapiro uses the following:

   > The *size* of a sentence p, $size(p)$, is the number of symbol occurrences in p (excluding punctuation symbols) minus the number of distinct variables occurring in p.

   Intuitively, building a monotonic size increase into the notion of refinement means that all refinements of p are more specific/weaker than p, as it requires more things to be provable (in the case of adding a body goal) or more structure to be present in the query (if a function symbol is introduced, or two or more variables are unified). This exploits the intimate relationship between the syntax and semantics of logic programs.

A refinement operator ρ is then defined to be a set-valued function that maps a statement p to a subset of its refinements, with some technical side-conditions.

A function satisfying this definition induces a partial order (where "larger" is interpreted to mean more-specific) over the statements of the language L, and can be visualised as a directed acyclic graph (generally termed a *Hasse diagram* in the context of partial orders). If all statements in L appear in the graph at some point, and are transitively reachable from the most-general clause $\square$, then the refinement operator is termed *complete* for the language L.

Shapiro proves the existence of a refinement operator complete for any first order language in [Sha81]. This property in itself is not particularly desirable, however, as it leads to inefficient induction of theories due to the exponential explosion of the size of the refinement graph. It is

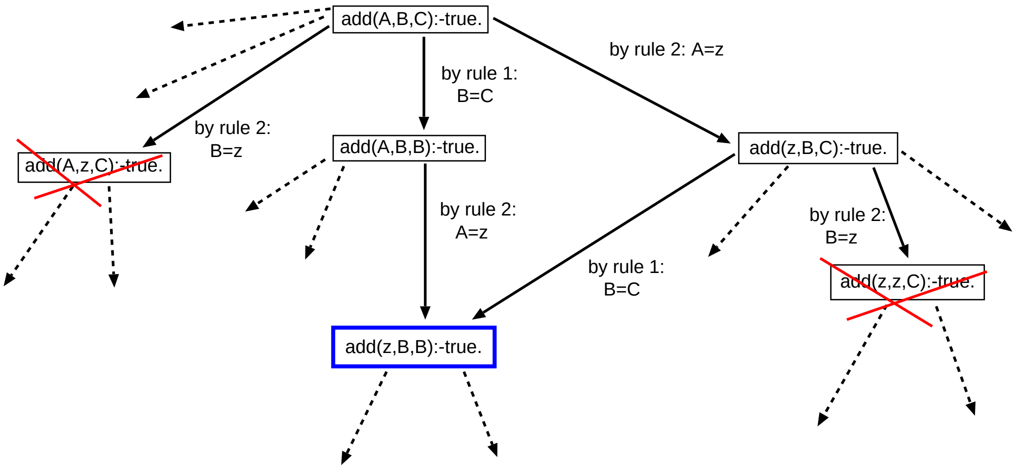

Figure 3: The success path through the refinement DAG for the base clause of `add/3`.

more interesting to define refinement operators that work on subsets of L that are more likely to contain satisfactory theories, as this is the only way the search is going to be tractable.

One class of interesting logic programs that can be enumerated via a refinement operator in practice is the that of the *definite clause grammars*. Using such an operator, it is possible to learn the syntax of simple programming languages.

The refinement operator used in Section 4 is defined as follows: Let $PSyms$ be the set of predicate symbols in L_h, $FSyms$ the function symbols; then $q \in \rho(p)$ iff q is a modification of p in precisely one of the following ways:

1. Let U, V be distinct variables in p; unify them in q (i.e. $q = p\{U/V\}$, read as "substitute V for U in p").

2. Let $f/n \in FSyms$ and U be a variable in p; let $q = p\{U/f(A_1, ..., A_n)\}$ where the A_i are new variables.

3. Let $g/n \in PSyms$, A_i distinct variables in p and $p = h : -b_1, ..., b_m$; let $q = h : -b_1, ..., b_m, g(A_1, ..., A_n)$.

See Section 4.1 for a discussion of some of the pragmatic issues of defining a useful refinement operator.

Figures 3 and 4 show how the two clauses for the `add/3` relation can be derived using these rules. Note that in practice there will probably be several other clauses included in the theory, such as the symmetric base case `add(B, z, B)`.

3.2 Using Refinement Operators

In essence, the bug-correction algorithm employed by MIS is little more than a search through an enumeration of all clauses in L_h. Given the intractability of this search in general, the given refinement operator ρ is used to bias the enumeration towards clauses that are more likely to be correct.

The search itself is *general-to-specific*, starting with the empty theory $\Box$. The advantage of this strategy over a *specific-to-general* one is that there is a simple starting point; for the latter the initial theory would have to contain – notionally, at least – all ground statements in L_h. This set will be infinite if there is at least one function symbol with an arity of one or more in the language (which will be the case in all interesting domains), and so is difficult to represent on a physical machine.

As ρ induces a partial (and not total) order on L_h, three sets need to be maintained:

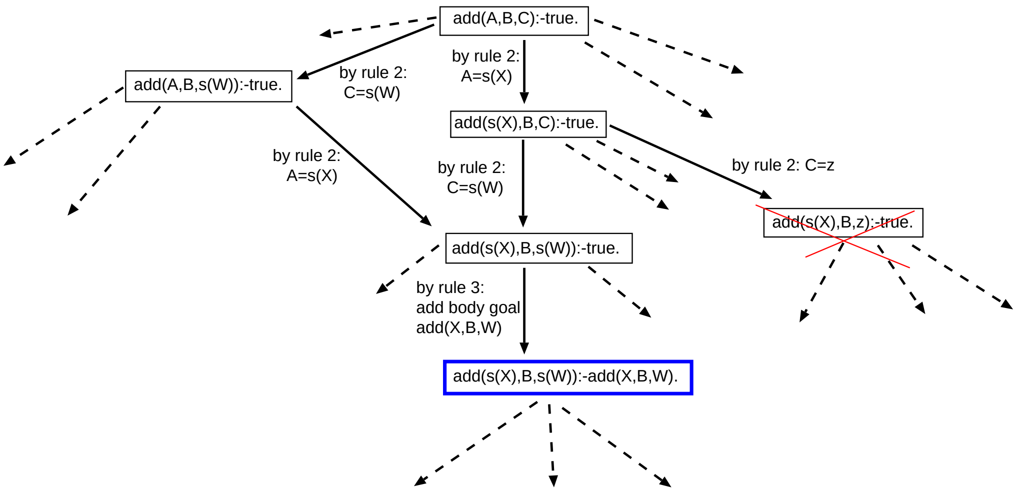

Figure 4: The success path through the refinement DAG for the recursive clause of `add/3`.

- The *theory* set contains all clauses that have been examined and found to be consistent with the facts.
- The *marked* set contains all clauses that have been flagged by the contradiction backtracing algorithm as the culprits for the proof of a negative observation, and have yet to be refined.
- The *dead* set contains all marked clauses that have been refined.

The search itself is straightfoward: select the next marked clause, add it's refinements to the theory and move it to the dead set. Note that the ordering of the marked set must be *fair*, for otherwise the algorithm may get stuck examining an infinite path that does not contain useful clauses. This is analogous to a Prolog proof attempt failing to terminate even though the query logically follows from the program, due to it's incomplete depth-first-search strategy.

In the context of Figure 1, searching the refinement DAG is purely a generalisation step; the consistency of the resulting theory with the facts is established by the framework itself.

4 Examples of MIS Learning Concepts

The applet on this page is a straightfoward implementation of the MIS algorithm, with very few refinements. Usage instructions:

1. Select either a canned learning scenario from the choice widget, or select the function symbols and predicate symbols in L_o. (Note that `./2` is the list constructor function symbol in Prolog by convention.)

2. Enter zero or more facts in the "bulk data entry" text box. These facts have the form:

   ```
   predicate_to_be_learned(a1, ..., an), true.
   predicate_to_be_learned(b1, ..., bn), false.
   ```

   Note that the applet allows facts to be added incrementally as well.

3. Click "done".

4. At this point MIS will attempt to find a theory that is consistent with the entered facts (if there were any). This is non-interactive, except when the oracle is consulted for contradiction backtracing (Section 2).

5. The applet will then enter an interactive loop, where facts are read and the theory is refined.

4.1 Implementation Notes and Empirical Obervations

This section is a collection of brief observations that stem from actually implementing MIS.

- The starting theory contains all predicate symbols in L_o with distinct variables for arguments. This is, in effect, the empty program $\Box$ specialised to L_o – they both entail all facts in L_o.

- MIS learns quite well from sparse data – it learnt the `less_than/2` relation from 15 facts (some of which are redundant).

- If the entered facts are inconsistent (the same fact is deemed both true and false), then the algorithm goes into an infinite loop. This is a manifestation of MIS's lack of noise handling.

- The theories induced tend to contain a lot of irrelevant clauses and be non-standard. The former can be dealt with via better refinement pruning and vetting strategies and perhaps a post-process of the theory.

- The algorithm as it stands cannot learn theories that use background knowledge, as most predicates do not behave well if their arguments do not have particular instantion modes. An example of this is `add/3`: the call `add(X, Y, Z)` fails to terminate. This is a manifestation of Prolog's incomplete search rule.

  The options for solving this are:

  – Use a complete proof technique. This is computationally intractable, but then again, so is MIS' search through the refinement graph.

  – Add meta-information (mode and perhaps type information) for the background predicates, and throw out all clauses that violate these prescriptions.

- Due to the previous point, the refinement operator in the applet is restricted to construct clauses with only a single body goal, which must not be a variant of the head. Again, mode information or a complete search strategy is necessary if more than one is allowed.

5 Conclusion

Having presented Shapiro's Model Inference System, we can now address some of the questions posed in the introduction:

- Why use a logic-based hypothesis language, rather than ones with other, perhaps simpler, semantic foundations?

  An immediate, if partial, answer to this is that logic has a very simple and regular semantics which is intimately reflected in it's syntax. This is exemplified in Section 3, where "refinement" is characterised by a simple syntactic criterion.

  In terms of the error diagnosis tools, pure Prolog has the advantage that it only has one way of getting work done: viz, unifying a goal with the head of a clause. Combined with the lack of side-effects, this implies that we can think about predicates purely in terms of their input/output behaviour. In other words, we can (mostly) treat predicates as black boxes and only unfold those that are known to be buggy. This is in contrast to C, where the presence of side-effects allow the source of a bug to be syntactically quite distant from where it manifests.

  Finally, for the purposes of the Model Inference System itself, adding or removing definite clauses have a predictable and incremental effect on a theory. Again, this follows from the simple proof-theoretic semantics of pure Prolog.

- How onerous is the requirement for an oracle for the learning domain?

  The most obvious atypicality of MIS is its reliance on an oracle for the concept it is trying to induce in addition to the training dataset. While this requirement is perfectly reasonable for abstract symbolic domains (such as inducting grammars of programming languages or axiomatising mathematical theories), it fails dismally when the domain has an *a priori* unknown structure. Concretely, this implies that MIS and indeed any supervised learning technique would be completely useless for, e.g., inducting a grammar for whale songs [Ric00].

- Can MIS be extended to handle noisy data?

  The presence of noise in the training data implies that there will be some overlap between the sets of positive and negative data. As first-order logic does not pretend to deal with contradictions in a very sophisticated way[6], it is not at all obvious how to retain the logical properties so effectively exploited by MIS in their presence.

- Can MIS scale to interesting learning tasks?

  MIS must make efficient use of the oracle if it is to induce larger-than-toy programs. As our implementation is quite naive and resource-hungry on even small examples, we have no empirical data on this front.

  The other issue that needs to be addressed is limiting the exponential growth in the refinement graph so that interesting programs can be induced in a reasonable amount of time. One way of doing this is to introduce meta-information such as type and mode annotations for the predicate and function symbols in L_h. *Biasing* the search almost always leads to incompleteness, which can be problematic.

These last three issues indicate that we must go beyond pure classical first-order logic to find a learning algorithm usable on practical, non-trivial domains. The most obvious recourse is the same as that for the physical sciences: apply statistics. See [MR99] for some recent work on this.

[6] Indeed, if a theory contains a contradiction then it contains every statement in L.

5.1 Multiple Predicate Learning

Some concepts cannot be learnt with a single set of clauses that all have the same head symbol (that is, define a single predicate). An example of this is anywhere where two or more levels of recursion (analogous to nested or sequenced loops) are required to express the relation, such as in successor-arithmetic multiplication, and a list-of-lists to lists flattening transformation.

Hence, it is necessary that auxilliary predicates can be introduced. If the learning algorithm can do this automatically, it is said to do *predicate invention*.

MIS will not create such *theoretical terms* on it's own initiative, although it can do *multi-predicate learning* – which is to learn several predicates as part of the one learning task. Additionally, MIS can either learn these predicates concurrently, or learn one and then use it as background knowledge for a later one. In either case, the learning task must be structured, as the oracle must be able to answer queries on every predicate in the target theory.

5.2 MIS Against FOIL

FOIL [QCJ93] is a well known ILP system that has quite distinct properties from MIS.

- FOIL uses an incomplete search rule based on a greedy search using the *information gain* heuristic (in a similar way to ID3, C4.5) to build Horn Clause descriptions of the data.
- MIS is *incremental*, i.e. a theory is never considered "finished" – it can be revised at any time (although the state of the learner needs to be retained). Conversely, FOIL is a bulk, or one-shot learner – give it all the data and it gives you what it considers to be the best description of it.
- MIS uses an *intensional* check against the facts; it explicitly attempts to prove them using the current conjecture. FOIL uses an *extensional* covers check while searching for clauses to build a theory from, which is much faster but (even more) incomplete.
- FOIL does *unsupervised* learning, i.e. there is no need for an idea about what the learnt predicate(s) should look like or what it should do. Conversely, MIS requires that the model being inferred comes equipped with an oracle (something that can evaluate the truth of a predicate at any point in the domain).
- FOIL can handle noisy data, whereas pure MIS cannot.

References

[BGT93] F. Bergadano, D. Gunetti, and U. Trinchero. The Difficulties of Learning Logic Programs with Cut. *Journal of AI Research*, 1:91–107, 1993.

[CM81] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[FY99] Pierre Flener and Serap Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *Journal of Logic Programming*, 41(2-3):141–195, 1999.

[Gol67] E. M. Gold. Language identication in the limit. *Information and Control*, 10(5):447–474, 1967.

[Hum85] David Hume. *A Treatise on Human Nature*. Penguin Classics, 1985.

[JORS99] S. Jain, D. Osherson, J. Royer, and A. Sharma. *Systems That Learn: An Introduction to Learning Theory*. MIT Press, second edition, 1999.

[Llo87a] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5:133–154, 1987.

[Llo87b] J.W. Lloyd. *Foundations of logic programming*. Symbolic Computation – *Artificial Intelligence*. Springer-Verlag, 1987.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[MR99] Eric McCreath and Mark Reid. A noise resistant model inference system. In *Discovery Science*, pages 252–263, 1999.

[Mug95] S. Muggleton. Inverse entailment and Progol. *New Generation Computing Journal*, 13:245–286, 1995.

[NCdW97] S. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes In Artificial Intelligence*. Springer-Verlag, 1997.

[QCJ93] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, volume 667, pages 3–20. Springer-Verlag, 1993.

[Ric00] Peter Rickwood. Automatic characterisation of humpback whale song. School of Computer Science and Engineering, UNSW, June 2000.

[Rob92] J. A. Robinson. Logic and logic programming. *Communications of the ACM*, 35(3):40–65, March 1992.

[Sam93] Claude Sammut. The Origins of Inductive Logic Programming: A Prehistoric Tale. In S. Muggleton, editor, *Third International Workshop on Inductive Logic Programming*, 1993.

[SB86] C. A. Sammut and R. B Banerji. *Machine Learning: An Artificial Intelligence Approach*, volume 2, chapter Learning Concepts by Asking Questions., pages 167–192. Morgan Kaufmann, 1986.

[Sha81] E.Y. Shapiro. Inductive inference of theories from facts. Technical report, Department of Computer Science, Yale University, 1981.

[Sha83] E.Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog: advanced programming techniques*. Logic programming. MIT Press, 2nd edition, 1994.

A Logic and Logic Programming Definitions

Most of these definitions follow Lloyd [Llo87b].

Unconstrained resolution, while an improvement on its predecessors, is still a massively non-deterministic search through a very large space of possible proofs. Interestingly, there is a class of clauses – the *definite* clauses – over which a variant of resolution can be made quite efficient.

Assume or give a brief overview of most-general-unifiers. A good overview of resolution theorem proving and its relationship to logic programming is [Rob92].

Atom: If p is a predicate symbol and t_i are terms, then $p(t_1, ..., t_n)$ is an *atom*. Note this is at odds with the builtin meta-predicates `atom` and `atomic`.

Chronological backtracking: In Prolog, a failing computation causes a re-examination of the most-recent choice of the clause to execute, in a context where there were several alternatives. The computation proceeds with some other selection.

Clause: A *clause* is a logical formula in *conjunctive normal form*, which has the schema: $p_1 \wedge ... \wedge p_m \rightarrow q_1 \vee ... \vee q_n$ or, equivalently, $\neg p_1 \vee ... \vee \neg p_n \vee q_1 \vee ... \vee q_n$ where p_i and q_i are atoms.

Clocksin and Mellish [CM81, Appendix B] give a Prolog program to convert arbitrary first-order formulae into clausal form.

Functor: A function symbol/arity pair. These have the same role as records in other languages (`structs` in C) which is to aggregate data. These are the *data constructors* in the context of type systems.

Ground: A term or atom is *ground* if it contains no variables.

Horn Clause: A clause containing at most one positive literal, written (with the commas interpreted as $\wedge$): $a \leftarrow b_1, ..., b_n$ (a *definite clause*) or $\leftarrow b_1, ..., b_n$ (a *definite goal*). Note that introducing negated body goals into the language violates this definition.

Instance of: One term is an instance of another if it is *more instantiated* than it, in the sense that it is the result of applying some substitution to the other.

Formally: τ is an instance of ρ if there exists a substitution θ such that $\rho\theta = \tau$, written $\tau \preceq \rho$.

Mode: instantiation states – i.e. whether a variable is bound or free, or in terms of predicates, whether an argument position is an *input* or an *output* (or both). Note that the instantiation state of a term can be arbitrarily complex.

Module System: Some mechanism for dividing a program into separate compilation units, for the purposes of abstraction.

Negation-as-finite-failure: Prolog uses the *closed-world-assumption*: if it cannot prove a goal using the clauses in it's database, and the proof attempt fails after a finite number of steps, then the goal is deemed to be false. This rule means that Prolog implements a *non-monotonic* logic: it is not necessarily the case that adding more clauses to the theory will lead to more things being provable. This is in contrast to classical first-order logic, which is monotonic.

Predicate Symbol: A *predicate symbol* is a string/arity pair that names a set of clauses (all of which have the same predicate symbol as heads).

Pure Prolog: Restriction of the language to Horn clause logic – i.e. no higher-order constructs (`call/1`, `findall/3`, ...), meta-predicates (`functor/3`, `arg/3`, ...) or built in arithmetic (`is/2`, ...).

Resolution: A sound and complete proof technique for first-order clausal logic, invented by Robinson in the early 1960's [Rob92]. Prolog uses a variant of full resolution, where constraints are imposed on clauses and the inputs to the resolution step, called SLD-NF resolution (linear resolution over definite clauses with a selection function, combined with the non-monotonic negation-as-failure rule [Llo87b]).

Substitution: A *substitution* is a function that maps a set of variables to a set of terms, and is usually denoted θ. We write $\rho\theta = \tau$ to indicate that the term τ results when we apply substitution θ to term τ.

Term: A *term* is one of: a variable, a constant symbol (e.g. a numeral), or a compound structure $f(a_1, ..., a_n)$ where a_i are terms. In the latter case, f is called a *function symbol*, n is the arity of the term, and f/n is the *functor*.

Type: A type is a collection of objects deemed to have some common property, such as the set of natural numbers.

Unification: Unification can be viewed as a generalisation of pattern matching (such as that found in Haskell) in that variables can be bound in both the pattern *and* the object matched against, rather than just the pattern.

Unification is used in Prolog for assignment, memory allocation, pattern matching, argument binding, etc. As such it can be heavily optimised provided we can discover (essentially) mode information.

More formally, we assume the existence of a sound method of finding the *most general unifier* (mgu) of two terms, written $\theta = \text{unify}(t_1, t_2)$. See [Llo87b] for more details.

Variants: Two terms are *variants* if they are identical modulo a variable renaming, which is analogous to α-convertibility in the lambda calculus.

If τ and ρ are variants, we write $\tau \cong \rho$. Obviously $\tau \preceq \rho \wedge \rho \preceq \tau \Rightarrow \tau \cong \rho$.