Compiling A Default Reasoning System into Prolog

David Poole
Department of Computer Science,
University of British Columbia,
Vancouver, B.C., Canada, V6T 1W5 (604) 228-6254
poole@cs.ubc.ca

1 July 1990

Abstract

Artificial intelligence researchers have been designing representation systems for default and abductive reasoning. Logic Programming researchers have been working on techniques to improve the efficiency of Horn Clause deduction systems. This paper describes how one such default and abductive reasoning system (namely *Theorist*) can be translated into Horn clauses (with negation as failure), so that we can use the clarity of abductive reasoning systems and the efficiency of Horn clause deduction systems. We thus show how advances in expressive power that artificial intelligence workers are working on can directly utilise advances in efficiency that logic programming researchers are working on. Actual code from a running system is given.

1 Introduction

Many people in Artificial Intelligence have been working on default reasoning and abductive diagnosis systems [35, 20, 4, 29]. The systems implemented so far (eg., [1, 16, 12, 34, 32]) are only prototypes or have been developed in ways that cannot take full advantage in the advances of logic programming implementation technology.

Many people are working on making logic programming systems more efficient. These systems, however, usually assume that the input is in the form of Horn clauses with negation as failure. This paper shows how to implement the default reasoning system Theorist [32, 29, 30] by compiling its input into Horn clauses with negation as failure, thereby allowing direct use advances in logic programming implementation technology. Both the compiler and the compiled code can take advantage of these improvements.

This work should be seen as an instance of Stickel's [39] proposal for a Prolog technology theorem prover. Rather than redesigning and extending a Prolog compiler [40], this is done by compiling to Prolog[1]. Rather than concentrating on the needs for theorem proving, this work has concentrated on the needs for representing common sense knowledge, in particular for default and abductive reasoning.

We have been running this implementation on standard Prolog compilers, and it outperforms all other default reasoning systems that the author is aware of. It is, arguably, not restricted to the control structure of Prolog. There is nothing in the compiled code which forces it to use Prolog's depth-first search strategy; all it requires is the implementation of Horn clauses with negation as failure. Logic programmers and other researchers are working on alternate control structures which seem very appropriate for default and abductive reasoning. Advances in parallel inference (e.g., [22]), constraint satisfaction [5, 42] and dependency directed backtracking [7, 6, 3] should be applicable to the code produced by this compiler.

We are thus effecting a clear distinction between the control and logic of our default reasoning systems [15]. We can let the control people concentrate on improving the efficiency of Horn clause systems, which will be directly applicable to those of us building richer representation systems. The Theorist system has been designed to allow maximum flexibility in control strategies while still giving us the power of assumption-based reasoning required for default and abductive reasoning.

This is a step towards having representation and reasoning systems which

[1]Independently and subsequently Stickel [41] has also developed a compiler from a Theorem prover to Prolog. He has, however, concentrated on the needs for theorem proving applications. These are not emphasised in this paper (see section 7).

are designed for correctness being able to use the most efficient control strategies. We want the best of expressibility and efficiency.

2 Theorist Framework

Theorist [32, 29, 30] is designed to be a very simple logical reasoning system for default and abductive reasoning. It is based on the idea of theory formation from a fixed set of possible hypotheses.

We assume a first order language [9] with a countable set of constant symbols (see section 2.1 for the syntax accepted by this implementation). A ground instance of a formula is obtained by substituting variable free terms of the language for variables in the formula.

The user provides:

$\mathcal{F}$ is a set of closed formulae called the *facts*. These are intended to be true in the world being modelled. $\mathcal{F}$ is assumed to be consistent.

Δ is a set of (possibly open) formulae which act as *possible hypotheses.*

Definition 2.1 A **scenario** of $(\mathcal{F}, \Delta)$ is a set D of ground instances of elements of Δ such that $D \cup \mathcal{F}$ is consistent.

Definition 2.2 If g is a closed formula, an **explanation** of g from $(\mathcal{F}, \Delta)$ is a scenario of $(\mathcal{F}, \Delta)$ which, together with $\mathcal{F}$, implies g.

That is, g can be explained from $(\mathcal{F}, \Delta)$ if there is a set D of ground instances of elements of Δ such that

$$\begin{array}{l} \mathcal{F} \cup D \models g \text{ and} \\ \mathcal{F} \cup D \text{ is consistent} \end{array}$$

D is an explanation of g.

Definition 2.3 An **extension** of $(\mathcal{F}, \Delta)$ is the set of logical consequences of the $\mathcal{F}$ together with a maximal (with respect to set inclusion) scenario of $(\mathcal{F}, \Delta)$.

In other papers we have described how the Theorist framework can be the basis of default and abductive reasoning systems [32, 29, 30]. If we are using Theorist for prediction then possible hypotheses can be seen as defaults [29, 30]. This is also a framework for abductive reasoning where the possible hypotheses are the base causes we are prepared to accept as to why some observation was made [30]. In this paper we refer to possible hypotheses as defaults, but the implementation can be used for either.

One restriction that can be made with no loss of expressive power is to restrict possible hypotheses to just atomic forms with no structure [29]. This is done by naming the defaults.

2.1 Syntax

The syntax of Theorist is designed to be of maximum flexibility. Virtually any syntax is appropriate for formulae; the formulae are translated into Prolog clauses without mapping out subterms. The theorem prover implemented in the Compiler can be seen as a non-clausal theorem prover [26].

Variables, constants, function symbols, predicate symbols, terms and atomic symbols (atoms) are defined as in Prolog [17]. A well formed formula (a *wff*) is made up of arbitrary combinations of implication ("`=>`", "`<-`"), disjunction ("`or`", `;`), conjunction ("`and`", "`&`", "`,`") and negation ("`not`", "`~`") of atomic symbols. As in Prolog, There is no explicit quantification; all facts are assumed to be universally quantified and all queries existentially quantified.

names are atomic symbol with only free variables as arguments.

The following gives the syntax of the Theorist code:

fact w.

where w is a wff, means that $(\forall w) \in \mathcal{F}$;[2] i.e., the universal closure of w is a fact.

default d.

where d is a name, means that $d \in \Delta$; i.e., d is a possible hypothesis.

[2] $\forall w$ is the universal closure of w. That is, all variables in w are universally quantified. If $\overline{V}$ are the free variables in w, $(\forall w)$ is $\forall \overline{V}\ w$. Similarly $\exists w$ is the existential closure of w: all variables are existentially quantified.

default $d : w$.

where d is a name and w is a wff means w, with name d can be used in a scenario if it is consistent. Formally it means $d \in \Delta$ and $(\forall(d \Rightarrow w)) \in \mathcal{F}$.

explain w.

where w is an arbitrary wff, gives all explanations of $\exists w$.

predict w.

where w is a arbitrary ground wff, returns "yes" if w is in every extension of $(\mathcal{F}, \Delta)$ and "no" otherwise. If it returns "yes", a set of explanations is returned, if it returns "no" then a scenario from which g cannot be explained is returned (this follows the framework of [30]).

3 Overview of Implementation

In this section we assume that we have a first order predicate calculus deduction system (denoted $\vdash$) which has the following properties (such a deduction system will be defined in the next section):

1. It is sound (i.e., if $A \vdash g$ then $A \models g$).
2. It is complete in the sense that if g follows from a consistent set of formulae, then g (or some formula more general than g) can be proven. That is, if A is consistent and $A \models g$ then $A \vdash g$.
3. If $A \vdash g$ then $A \cup B \vdash g$; i.e., adding in extra facts will not prevent the system from finding a proof which previously existed.
4. It can return instances of certain predicates used in the proof.

The basic idea of the implementation follows the definition on explainability:

Procedure 3.1 [3] to explain g from $(\mathcal{F}, \Delta)$

[3]This is called a procedure as, in general, it is not decidable whether an atom can be explained.

1. try to prove g from $\mathcal{F} \cup \Delta$. If no proof exists, then g cannot be explained. If there is a proof, let D be the set of instances of elements of Δ used in the proof. We then know
$$\mathcal{F} \cup D \models g$$
by the soundness of our proof procedure.

2. show D is consistent with $\mathcal{F}$ by failing to prove it is inconsistent. As $\mathcal{F}$ is consistent, the completeness of our proof procedure will ensure that if $\mathcal{F} \cup D$ is inconsistent, a proof for $\neg D$ from $\mathcal{F}$ will be found.

3.1 Consistency Checking

The following two theorems are important for implementing the consistency check:

Lemma 3.2 If A is a consistent set of formulae and D is a finite set of ground instances of possible hypotheses, then if we impose arbitrary ordering on the elements of $D = \{d_1, ..., d_n\}$

$$\begin{gathered} A \cup D \text{ is inconsistent} \\ \text{if and only if} \\ \text{there is some } i,\ 1 \leq i \leq n \text{ such that } A \cup \{d_1, ..., d_{i-1}\} \text{ is consistent and} \\ A \cup \{d_1, ..., d_{i-1}\} \models \neg d_i. \end{gathered}$$

Proof: If $A \cup D$ is inconsistent there is some least i such that $A \cup \{d_1, ..., d_i\}$ is inconsistent. Then we must have $A \cup \{d_1, ..., d_{i-1}\}$ is consistent (as i is minimal) and $A \cup \{d_1, ..., d_{i-1}\} \models \neg d_i$ (by inconsistency). $\square$

This lemma says that we can show that $\mathcal{F} \cup \{d_1, ..., d_n\}$ is consistent if we can show that for all i, $1 \leq i \leq n$, $\mathcal{F} \cup \{d_1, ..., d_{i-1}\} \not\vdash \neg d_i$. If our theorem prover can show there is no proof of all of the $\neg d_i$, then we have consistency.

This lemma indicates that we can implement Theorist by incrementally failing to prove inconsistency. We need to try to prove the negation of the default in the context of all previously assumed defaults. Note that this ordering is arbitrary.

The following theorem expands on how explainability can be computed:

Theorem 3.3 If $\mathcal{F}$ is consistent, g cannot be explained from $(\mathcal{F}, \Delta)$ if and only if there is a ground proof of g from $\mathcal{F} \cup D$ where $D = \{d_1, ..., d_n\}$ is a set of ground instances of elements of Δ such that $\mathcal{F} \wedge \{d_1, ..., d_{i-1}\} \not\vdash \neg d_i$ for all $i, 1 \leq i \leq n$.

> **Proof:** If g can be explained from $\mathcal{F}, \Delta$, there is a set D of ground instances of elements of Δ such that $\mathcal{F} \cup D \models g$ and $\mathcal{F} \cup D$ is consistent. So there is a ground proof of g from $\mathcal{F} \cup D$. By the preceding lemma $\mathcal{F} \cup D$ is consistent so there can be no sound proof of inconsistency. That is, we cannot prove $\mathcal{F} \cup \{d_1, ..., d_{i-1}\} \vdash \neg d_i$ for any i. $\square$

This leads us to the refinement of procedure 3.1:

Procedure 3.4 to explain g from $(\mathcal{F}, \Delta)$

1. Build a ground proof of g from $(\mathcal{F} \cup \Delta)$. Make D the set of instances of elements of Δ used in the proof.

2. For each i, try to prove $\neg d_i$ from $\mathcal{F} \cup \{d_1, ..., d_{i-1}\}$. If all such proofs fail, D is an explanation for g.

Note that the ordering imposed on the D is arbitrary. A sensible one is the order in which the elements of D were generated. Thus when a new hypothesis is used in the proof, we try to prove its negation from the facts and the previously used hypotheses. These proofs are independent of the original proof and can be done as they are generated as in negation as failure (see section 3.3), or can be done concurrently.

3.2 Variables

Theorem 3.3 says that g can be explained if there is a ground proof. A problem arises in translating the preceding procedure (which assumes ground proofs) into an procedure which does not build ground proofs (eg., a standard resolution theorem prover), as we may have variables in the forms we are trying to prove the negation of.

A problem arises when there are variables in the d_i generated. Consider the following example:

Example 3.5 Let $\Delta = \{p(X)\}$. That is, any instance of $p(X)$ can be used if consistent. Let $\mathcal{F} = \{\forall Y(p(Y) \Rightarrow g), \neg p(a)\}$. That is, g is true if there is a Y such that $p(Y)$.

According to our semantics, g can be explained with the explanation $\{p(b)\}$, which is consistent with $\mathcal{F}$ (consider the interpretation $I = \{\neg p(a), p(b)\}$ on the domain $\{a, b\}$), and implies g.

However, if we try to prove g, we generate $D = \{p(Y)\}$ where Y is free (implicitly a universally quantified variable). The existence of the fact $\neg p(a)$ should not make it inconsistent, as g can be explained.

So we need to generate a ground proof of g. This leads us to:

Procedure 3.6 To determine if g can be explained from $(\mathcal{F}, \Delta)$

1. generate a proof of g using elements of $\mathcal{F}$ and Δ as axioms. Make D_0 the set of instances of Δ used in the proof;
2. form D_1 by replacing free variables in D_0 with unique constants[4];
3. add D_1 to $\mathcal{F}$ and try to prove an inconsistency (as in the previous case). If a complete search for a proof fails, g is explained.

This procedure can now be directly implemented by a resolution theorem prover.

Example 3.7 Consider example 3.5. If we try to prove g, we use the hypothesis instance $p(Y)$. This means that g is provable from any instance of $p(Y)$. To show g cannot be explained, we replace Y with a constant β. $p(\beta)$ is consistent with the facts. Thus g is explained.

3.3 Incremental Consistency Checking

Procedure 3.6 assumed that we only check consistency at the end. We cannot replace free variables by a unique constant until the end of the computation.

[4]This is justified here as the desire to build a ground proof. It can also be justified by noticing that free variables in defaults are existentially quantified (we only need to assume that some individual exists). To show this is inconsistent we Skolemise the existentially quantified variable.

This procedure can be further refined by considering cases where we can check consistency at the time the hypothesis is generated.

Lemma 3.2 shows that we can check consistency incrementally in whatever order we like. The problem is to determine whether we have generated the final version of a set of hypotheses. If there are no variables in our set of hypotheses, then we can check consistency as soon as they are generated. If there are variables in a hypothesis, then we cannot guarantee that the form generated will be the final form of the hypothesis.

Example 3.8 Consider the two alternate sets of facts:

$$\begin{array}{lll}
\Delta & = \{ & p(X) \; \} \\
\mathcal{F}_1 & = \{ & \forall X \; p(X) \wedge q(X) \Rightarrow g, \\
 & & \neg p(a), \\
 & & q(b) \; \} \\
\mathcal{F}_2 & = \{ & \forall X \; p(X) \wedge q(X) \Rightarrow g, \\
 & & \neg p(a), \\
 & & q(a) \; \}
\end{array}$$

Suppose we try to explain g by first explaining p and then explaining q. Once we have generated the hypothesis $p(X)$, we have not enough information to determine whether the consistency check should succeed or fail. The consistency check for $\mathcal{F}_1$ should succeed (i.e, we should conclude with a consistent instance, namely $X = b$), but the consistency check for $\mathcal{F}_2$ should fail (there is no consistent value for the X which satisfies p and q). We can only determine the consistency after we have proven q.

There are two obvious solutions to this problem; the first is to allow the consistency check to return constraints on the values. An alternate solution is to delay the evaluation of the consistency check until all of the variables are bound (as in [23]), or until we know that the variables cannot be bound any more. In particular we know that a variable cannot be bound any more at the end of the computation.

The implementation described in this paper does the simpler form of incremental consistency checking, namely it computes consistency immediately for those hypotheses with no variables and delays consistency checking until the end for hypotheses containing variables at the time they are generated.

Even though it may be desirable to detect when all of the variables in a term become instantiated, it was decided that this would impose too great an overhead in current Prolog systems,

3.4 Comparison with other implementations

This specification can be seen a a refinement of Reiter's [35] top-down proof procedure for normal defaults without prerequisites, but with a further refinement of how to handle variables.

This procedure is also related to the algorithms of Pople [33] and Cox and Pietrzykowski [4]. Neither of these have a predefined set of possible hypotheses, but rather they allow the assumption of hypotheses on syntactic grounds (e.g., there is no way to derive them).

4 The Deduction System

In this section we describe the deduction system that we assumed in the previous section. This implementation is based on model elimination, in particular the MESON proof procedure [18, 39]. This is complete in the sense that if g logically follows from some *consistent* set of clauses A, then there is a MESON proof of g from A.

SLD resolution of Prolog [17] can be seen as MESON with the contrapositive and ancestor search removed.

To implement MESON in Prolog [39], we add two things

1. we use the contrapositive of our clauses. If we have the clause

   $$L_1 \vee L_2 \vee ... \vee L_n$$

   then we create the n rules

   $$\begin{array}{l} L_1 \leftarrow \neg L_2 \wedge ... \wedge \neg L_n \\ L_2 \leftarrow \neg L_1 \wedge \neg L_3 \wedge ... \wedge \neg L_n \\ ... \\ L_n \leftarrow \neg L_1 \wedge ... \wedge \neg L_{n-1} \end{array}$$

   Each of these can be used to prove the left hand literal if we know the other literals are false. Here we are treating the negation of an atom as a different Prolog atom.

2. the ancestor cancellation rule. While trying to prove L we can assume $\neg L$. We have a subgoal proven if it unifies with the negation of an ancestor in the proof tree. This is an instance of proof by contradiction. We can see this as assuming $\neg L$; when we have proven L we discharge the assumption. We only make the assumption when we know it can be discharged.

One property of the deduction system that we want is the ability to implement definite clauses with a constant factor overhead over using Prolog. One way to do this is to keep two lists of ancestors of any node: P the positive (non negated atoms) ancestors and N the negated ancestors. Thus for a positive subgoal we only need to search for membership in N and for a negated subgoal we only need to search P. When we have definite clauses, there are no negated subgoals, and so N is always empty. Thus the ancestor search always consists of checking for membership in an empty list. The alternate contrapositive form of the clauses are never used.

Alternate, more complicated ways to do ancestor search have been investigated [28], but this implementation uses the very simple form given above.

4.1 Disjunctive Answers

For the compiler to work properly we need to be able to return disjunctive answers. We need disjunctive answers for the case that we can prove only a disjunctive form of the query.

For example, if we can prove $p(a) \vee p(b)$ for the query $?p(X)$, then we want the answer $X = a$ or b. This can be seen as "if the answer is not a then the answer is b".

To have the answer $a_1 \vee ... \vee a_n$, we need to have a proof of "If the answer is not a_1 and not a_2 and ... and not a_{n-1} then the answer is a_n". We collect instances of the top level goal that we are assuming are not true in order to prove an instance of the top level goal.

This idea is implemented by being able to assume an instance of the negation of the top level goal as long as we add it to the set of answers. To do this we carry the alternate answers that we are assuming in proving the top level goal.

4.2 Conversion to Clausal Form

It is desirable to convert an arbitrary well formed formula into clausal (or rule) form without mapping out subterms. Instead of distributing, this can be done by creating a new term to refer to the disjunct.

Once a formula is in negation normal form, then the normal way to convert to clausal form [2] is to convert something of the form

$$\alpha \vee (\beta \wedge \gamma)$$

by distribution into

$$(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

and so mapping out subterms.

An alternate [26] is to create a new relation p parameterised with the variables in common with α and $\beta \wedge \gamma$. We can then replace $\beta \wedge \gamma$ by p and then add

$$(\neg p \vee \beta) \wedge (\neg p \vee \gamma)$$

to the set of formulae.

This is essentially the same idea that most Prolog implementations use to implement a disjunction in the body of a rule. It is interesting to note that it does work in general [26].

This is implemented using Prolog "or"[5] instead of actually building the p. We build up the clauses so that the computation runs without any multiplying out of subterms. This is an instance of the general procedure of making clausal theorem provers into non-clausal theorem provers [26].

5 Details of the Compiler

In this section we give actual code which converts Theorist code into Prolog code. The compiler is described here in a bottom up fashion; from the construction of the atoms to compilation of general formulae.

The compiler is written in Prolog and the target code for the compiler is Prolog code (in particular Horn clauses with negation as failure). There are no "cuts" or other non-logical "features" of Prolog which depend on Prolog's

[5]If the underlying Prolog does not have disjunction, it can be easily added by defining the clause $(p; q) \leftarrow p$ and $(p; q) \leftarrow q$.

Figure 1: Steps of the compiler

search strategy in the compiled code (although there is in the compiler). Each Theorist wff gets locally translated into a set of Prolog clauses.

Figure 1 shows the forms that the formulae take, and the procedures that effect the change, between the user giving the facts and defaults and the resulting code being put in the Prolog database or written to a file.

The term "rule" is used as a technical term for an intermediate form. One formula produces many rules. Note that when we are using rules, we have already formed contrapsotives.

5.1 Target Atoms

For each Theorist predicate symbol r there are 4 target predicate symbols, with the following informal meanings:

prove_r meaning r can be proven from the facts, and a given set of hypotheses.

prove_not_r meaning $\neg r$ can be proven from the facts and a given set of hypotheses.

ex_r meaning r can be explained from $(\mathcal{F}, \Delta)$.

ex_not_r meaning $\neg r$ can be explained from $(\mathcal{F}, \Delta)$.

The arguments to these built predicate symbols contain all of the information needed to prove or explain instances of the source predicates.

5.1.1 Proving

For relation $r(-args-)$ in the source code we want to produce object code which says that $r(-args-)$ (or its negation) can be proven from the facts and the current set of assumed hypotheses.

For the source relation

$$r(-args-)$$

(which is to mean that r is a relation with $-args-$ being the sequence of its arguments), we have the corresponding target relations

$$prove\_r(-args-, Ths, Anc)$$

$$prove\_not\_r(-args-, Ths, Anc)$$

which are to mean that r (or $\neg r$) can be proven from the facts and ground hypotheses Ths with ancestor structure Anc. These extra arguments are:

Ths is a list of ground instances of defaults. These are the defaults we have already assumed and so define the context in which to prove $r(-args-)$.

Anc is a structure of the form $anc(P, N)$ where P and N are lists of atoms. Interpreted procedurally, P is the list of positive (not negated) ancestors of the goal in a proof and N is the list of negated ancestors in a proof. As described in section 4 we conclude some goal if it unifies with its negated ancestors.

Declaratively,

$$prove\_r(-args-, Ths, anc(P, N))$$

is true when

$$\mathcal{F} \models \left(\bigwedge_{h \in Ths} h\right) \wedge \left(\bigwedge_{p \in P} \neg p\right) \wedge \left(\bigwedge_{n \in N} n\right) \Rightarrow r(-args-)$$

The definition of $prove\_not\_r$ is the same except that $\neg r(-args-)$ appears in the place of $r(-args-)$.

5.1.2 Explaining

There are two target relations for explaining associated with each source relation r. These are ex_r and ex_not_r.

For the source relation:

$$r(-args-)$$

we have two target new relations for explaining r:

$$ex\_r(-args-, Ths, Anc, Ans)$$

$$ex\_not\_r(-args-, Ths, Anc, Ans)$$

These mean that $r(-args-)$ (or $\neg r(-args-)$) can be explained, with

Ths is the structure of the incrementally built hypotheses used in explaining r. There are two statuses of hypotheses we use; one the defaults that are ground and so can be proven consistent at the time of generation; the other the hypotheses with free variables at the time they are needed in the proof, for which we defer consistency checking (in case the free variables get instantiated later in the proof). Ths is essentially two difference lists, one of the ground instances of defaults already proven consistent and one of the deferred defaults. Ths is of the form

$$ths(T_1, T_2, D_1, D_2)$$

which is to mean, procedurally, that T_1 is the list of consistent ground hypotheses before we try to explain r, and T_2 is T_1 together with the consistent ground hypotheses assumed to explain r. Similarly, D_1 is the list of deferred hypotheses before we consider the goal and D_2 is D_1 and the deferred hypotheses used in proving r.

Anc contains the ancestors of the goal. As for proving, this is a pair of the form $anc(P, N)$ where P is the list of positive ancestors of the goal, and N is the list of negated ancestors of the goal.

Ans contains the answers we are considering in difference list form $ans(A_1, A_2)$, where A_1 is a disjunct of the answers before proving the goal, and A_2 is the answers after proving the goal.

The semantics of

$$ex\_r(-args-, ths(T_1, T_2, D_1, D_2), anc(P, N), ans(A_1, A_2))$$

is defined by

$$\mathcal{F} \models \left(\bigwedge_{g \in T_2} g\right) \wedge \left(\bigwedge_{h \in D_2} h\right) \wedge \left(\bigwedge_{p \in P} \neg p\right) \wedge \left(\bigwedge_{n \in N} n\right) \Rightarrow r(-args-) \vee A_2$$

where $T_1 \subseteq T_2$, $D_1 \subseteq D_2$ and $A_1 \subseteq A_2$, and such that

$$\mathcal{F} \cup T_2 \text{ is consistent}$$

The definition of *ex_not_r* is the same except that $\neg r(-args-)$ appears in the place of $r(-args-)$.

5.1.3 Building Atoms

The procedure *new_lit*(*Prefix, Reln, Newargs, Newreln*) constructs a new atom, *Newreln*, with predicate symbol made up of *Prefix* prepended to the predicate symbol of *Reln*, and taking as arguments the arguments of *Reln* together with *Newargs*. For example,

?– new_lit("ex_",reln(a,b,c),[T,A,B],N).

yields

N = ex_reln(a,b,c,T,A,B)

The procedure is defined as follows[6]:

```
new_lit(Prefix, Reln, NewArgs, NewReln) :-
   Reln =.. [Pred | Args],
   name(Pred,PredName),
   append(Prefix, PredName, NewPredName),
   name(NewPred,NewPredName),
   append(Args, NewArgs, AllArgs),
   NewReln =.. [NewPred | AllArgs].
```

5.2 Compiling Rules

The next simplest compilation form we consider is the intermediate form called a "rule". Rules are statements of how to conclude the value of some relation. Each Theorist fact corresponds to a number of rules (one for each literal in the fact). Each rule gets translated into Prolog rules to explain and prove the head of the rule.

[6]The verbatim code is actual implementation code given in standard Edinburgh notation. I assume that the reader is familiar with such notation.

Rules use the intermediate form called a "literal". A literal is either an atomic symbol or of the form $n(A)$ where A is an atomic symbol. A rule is either a literal or of the form $H \leftarrow Body$ (written "`if(H,Body)`") where H is a literal and $Body$ is formed from conjunctions and disjunctions of literals.

We translate rules of the form

$$h(-x-) \leftarrow b_1(-x_1-), b_2(-x_2-), ..., b_n(-x_n-).$$

into the internal form (assuming that h is not negated)

$$\begin{aligned}
&ex\_h(-x-, ths(T_0, T_n, D_0, D_n), anc(P, N), ans(A_0, A_n)) :- \\
&\quad ex\_b_1(-x_1-, ths(T_0, T_1, D_0, D_1), anc([h(-x-)|P], N), ans(A_0, A_1)), \\
&\quad ex\_b_2(-x_2-, ths(T_1, T_2, D_1, D_2), anc([h(-x-)|P], N), ans(A_1, A_2)), \\
&\quad ..., \\
&\quad ex\_b_n(-x_n-, ths(T_{n-1}, T_n, D_{n-1}, D_n), anc([h(-x-)|P], N), ans(A_{n-1}, A_n)).
\end{aligned}$$

That is, we explain h if we explain each of the b_i, accumulating the explanations and the answers. Note that if h is negated, then the head of the clause will be of the form ex_not_h, and the ancestor form will be $anc(P, [h(-x-)|N])$. If any of the b_i are negated, then the corresponding predicate will be $ex\_not\_b_i$.

Example 5.1 the rule

$$gr(X, Y) \leftarrow f(X, Z), p(Z, Y)$$

gets translated into

$$\begin{aligned}
&ex\_gr(X, Y, ths(D, E, F, G), anc(H, I), ans(J, K)) :- \\
&\quad ex\_f(X, Z, ths(D, M, F, N), anc([gr(X, Y)|H], I), ans(J, O)), \\
&\quad ex\_p(Z, Y, ths(M, E, N, G), anc([gr(X, Y)|H], I), ans(O, K)).
\end{aligned}$$

To explain gr we explain both f and p. The initial assumptions for f should be the initial assumptions for gr, and the initial assumptions for p should be the initial assumptions plus those made to explain f. The resulting assumptions after proving p are the assumptions made in explaining gr.

Example 5.2 the fact

$$father(randy, jodi)$$

gets translated into

$$ex\_father(randy, jodi, ths(T,T,D,D), B, ans(A,A)).$$

We can explain $father(randy, jodi)$ independently of the ancestors; we need no extra assumptions, and we create no extra answers.

Similarly we translate rules of the form

$$h(-x-) \leftarrow b_1(-x_1-), b_2(-x_2-), ..., b_N(-x_n-).$$

into

$prove\_h(-x-, T, anc(P,N)) :-$
$\quad prove\_b_1(-x_1-, T, anc([h(-x-)|P], N)),$
$\quad ...,$
$\quad prove\_b_n(-x_n-, T, anc([h(-x-)|P], N)).$

Example 5.3 the rule

$$gr(X,Y) \leftarrow f(X,Z), p(Z,Y)$$

gets translated into

$prove\_gr(X, Y, D, anc(H, I)) :-$
$\quad prove\_f(X, Z, D, anc([gr(X,Y)|H], I)),$
$\quad prove\_p(Z, Y, D, anc([gr(X,Y)|H], I)).$

That is, we can prove gr if we can prove f and p. Having $gr(X,Y)$ in the ancestors means we can prove $gr(X,Y)$ by assuming $\neg gr(X,Y)$ and then proving $gr(X,Y)$.

Example 5.4 the fact

$$father(randy, jodi)$$

gets translated into

$$prove\_father(randy, jodi, A, B).$$

Thus we can prove $father(randy, jodi)$ for any explanation and for any ancestors.

Disjuncts in the source body (;) get mapped into Prolog's disjunction. The answers and assumed hypotheses should be accumulated from whichever branch was taken. This is then executed without mapping out subterms.

Example 5.5 The rule

$$p(A) \leftarrow q(A), (r(A), s(A); t(A)), m(A).$$

gets translated into

```
prove_p(A, B, anc(C, D)) :-
    prove_q(A, B, anc([p(A)|C], D)),
    (   prove_r(A, B, anc([p(A)|C], D)),
        prove_s(A, B, anc([p(A)|C], D))
    ;   prove_t(A, B, anc([p(A)|C], D))),
    prove_m(A, B, anc([p(A)|C], D)).

ex_p(A, ths(B, C, D, E), anc(F, G), ans(H, I)) :-
    ex_q(A, ths(B, J, D, K), anc([p(A)|F], G), ans(H, L)),
    (   ex_r(A, ths(J, M, K, N), anc([p(A)|F], G), ans(L, O)),
        ex_s(A, ths(M, P, N, Q), anc([p(A)|F], G), ans(O, R))
    ;   ex_t(A, ths(J, P, K, Q), anc([p(A)|F], G), ans(L, R))),
    ex_m(A, ths(P, C, Q, E), anc([p(A)|F], G), ans(R, I))
```

Note that P is the resulting explanation from either executing r and s or executing t from the explanation J.

5.2.1 The Code to Compile Rules

The following relation builds the desired structure for the bodies:

$$make\_bodies(B, T, [Ths, Anc, Ans], ProveB, ExB)$$

where B is a disjunct/conjunct of literals (the body of the rule), T is a scenario used in proving B, Ths is a theory structure for explaining, Anc is an ancestor structure (of form $anc(P, N)$), Ans is an answer structure (of form $ans(A0, A1)$). This procedure makes $ProveB$ the body of forms $prove\_b_i$ (or $prove\_not\_b_i$), and ExB a body of the forms ex_b_i.

```
make_bodies((H,B), T, [ths(T1,T3,D1,D3), Anc, ans(A1,A3)],
                    (ProveH,ProveB), (ExH,ExB)) :-
   !,
   make_bodies(H,T,[ths(T1,T2,D1,D2),Anc,ans(A1,A2)],ProveH,ExH),
   make_bodies(B,T,[ths(T2,T3,D2,D3),Anc,ans(A2,A3)],ProveB,ExB).

make_bodies((H;B),T,Ths,(ProveH;ProveB),(ExH;ExB)) :-
   !,
   make_bodies(H,T,Ths,ProveH,ExH),
   make_bodies(B,T,Ths,ProveB,ExB).

make_bodies(n(A), T, [Ths,Anc,Ans], ProveA, ExA) :-
   !,
   new_lit("prove_not_", A, [T,Anc], ProveA),
   new_lit("ex_not_", A, [Ths,Anc,Ans], ExA).

make_bodies(A, T, [Ths,Anc,Ans], ProveA, ExA) :-
   new_lit("prove_", A, [T,Anc], ProveA),
   new_lit("ex_", A, [Ths,Anc,Ans], ExA).
```

The procedure $rule(R)$ declares R to be a fact rule. R is either a literal or of the form $if(H,B)$ where H is a literal and B is a body.

$prolog\_cl(C)$ is a way of asserting to Prolog the clause C. This can either be asserted or written to a file to be consulted or compiled. The simplest form is to just assert C.

$make\_anc(H)$ is a procedure which ensures that the ancestor search is set up properly for H. It is described in section 5.5, and can be ignored on first reading.

```
rule(if(H,B)) :-
   !,
   make_anc(H),
   make_bodies(H,T,[Ths,Anc,Ans],ProveH,ExH),
   form_anc(H,Anc,Newanc),
   make_bodies(B,T,[Ths,Newanc,Ans],ProveB,ExB),
   prolog_cl((ProveH:-ProveB)),
   prolog_cl((ExH:-ExB)).
```

```
rule(H) :-
   make_anc(H),
   make_bodies(H,T,[ths(T,T,D,D),_,ans(A,A)],ProveH,ExH),
   prolog_cl(ProveH),
   prolog_cl(ExH).
```

form_anc$(L, A1, A2)$ means that $A2$ is the ancestor form for subgoal L with previous ancestor form $A1$.

```
form_anc(n(G), anc(P,N), anc(P,[G|N])) :- !.
form_anc(G, anc(P,N), anc([G|P],N)).
```

5.3 Forming Contrapositives

For facts we convert the user syntax into negation normal form (section 6.2), form the contrapositives, and declare these as rules.

Note that here we choose an arbitrary ordering for the clauses in the bodies of the contrapositive forms of the facts. No attempt has been made to optimise this, although it is noted that some orderings may be more efficient than others (see for example [37] for a discussion of such issues).

The declarations are as follows:

```
declare_fact(F) :-
   nnf(F,even,N),
   rulify(N).
```

nnf(*Wff,Parity,Nnf*) (section 6.2) means that *Nnf* is the negation normal form of *Wff* if *Parity=even* and of $\neg$*Wff* if *Parity=odd*. Note that we *rulify* the normal form of the negation of the formula.

rulify(N) where N is the negation of a fact in negation normal form (see section 6.2), means that all rules which can be formed from N (by allowing each atom in N being the head of some rule) should be passed on to *rule* so they can be compiled further. This procedure is defined by induction on the structure of the first argument.

```
rulify((A,B)) :- !,
   contrapos(B,A),
   contrapos(A,B).

rulify((A;B)) :- !,
   rulify(A),
   rulify(B).

rulify(n(A)) :- !,
   rule(A).

rulify(A) :-
   rule(n(A)).
```

contrapos(D, T) where (D, T) is (the negation of) a formula in negation normal form means that all rules which can be formed from (D, T) with head of the rule coming from T should be formed. Think of D as the literals for which the rules with them as heads have been formed, and T as those which remain to be as the head of some rule. This procedure considers the cases that T could be in.

```
contrapos(D, (L,R)) :- !,
   contrapos((R,D),L),
   contrapos((L,D),R).

contrapos(D,(L;R)) :- !,
   contrapos(D,L),
   contrapos(D,R).

contrapos(D,n(A)) :- !,
   rule(if(A,D)).

contrapos(D,A) :-
   rule(if(n(A),D)).
```

Example 5.6 if we are to *rulify* the negation normal form

$$n(p(A)), q(A), (r(A), s(A); t(A)), m(A)$$

we generate the following rule forms, which can then be given to *rule*

$$\begin{array}{l}
p(A) \leftarrow q(A), (r(A), s(A); t(A)), m(A) \\
n(q(A)) \leftarrow (r(A), s(A); t(A)), m(A), n(p(A)) \\
n(r(A)) \leftarrow s(A), m(A), q(A), n(p(A)) \\
n(s(A)) \leftarrow r(A), m(A), q(A), n(p(A)) \\
n(t(A)) \leftarrow m(A), q(A), n(p(A)) \\
n(m(A)) \leftarrow (r(A), s(A); t(A)), q(A), n(p(A))
\end{array}$$

Note that we always create as many rules as there are literals in the original formula (given that we do not allow equivalences or exclusive-or).

5.4 Possible Hypotheses

The other class of things we have to worry about is the class of possible hypotheses. As described in [29] and outlined in section 2, we only need worry about atomic possible hypotheses.

If $d(-args-)$ is a possible hypothesis (default), then we want to form the target code as follows:

1. We can prove a hypothesis if we have already assumed it:

$$\begin{array}{l}
prove\_d(-args-, Ths, Anc) :- \\
\quad member(d(-args-), Ths).
\end{array}$$

2. We can explain a default if we have already assumed it:

$$\begin{array}{l}
ex\_d(-args-, ths(T, T, D, D), Anc, ans(A, A)) :- \\
\quad member(d(-args-), T).
\end{array}$$

3. We can explain a hypothesis by assuming it, if it has no free variables, we have not already assumed it and it is consistent with everything assumed before:

$$\begin{array}{l}
ex\_d(-args-, ths(T, [d(-args-)|T], D, D), Anc, ans(A, A)) :- \\
\quad \text{variable\_free}(d(-args-)), \\
\quad \backslash + member(d(-args-), T, \\
\quad \backslash + prove\_not\_d(-args-, [d(-args-)|T], anc([], [])).
\end{array}$$

4. If a hypothesis has free variables, it can be explained by adding it to the deferred defaults list (making no assumptions about its consistency; this will be checked at the end of the explanation phase):

   $ex\_d(-args-, ths(T, T, D, [d(-args-)|D], Anc, ans(A, A)) : -$
   $\quad \backslash+\text{variable\_free}(d(-args-)).$

The following compiles defaults into such code:

```
declare_default(D) :-
   make_anc(D),
   new_lit("prove_",D,[T,_],Pr_D),
   prolog_cl((Pr_D :- member(D,T))),
   new_lit("ex_",D, [ths(T,T,Defer,Defer), _, ans(A,A)], ExD),
   prolog_cl((ExD :- member(D, T))),
   new_lit("ex_",D, [ths(T,[D|T],Defer,Defer), _, ans(A,A)], ExDass),
   new_lit("prove_not_",D, [[D|T],anc([],[])],Pr_not_D),
   prolog_cl((ExDass :- variable_free(D), \+member(D,T),
                 \+Pr_not_D)),
   new_lit("ex_",D, [ths(T,T,Defer,[D|Defer]), _, ans(A,A)], ExDefer),
   prolog_cl((ExDefer :- \+ variable_free(D))).
```

Example 5.7 The default

$birdsfly(A)$

gets translated into

$prove\_birdsfly(A, B, C) : -$
$\quad member(birdsfly(A), B)$
$ex\_birdsfly(A, ths(B, B, C, C), D, ans(E, E)) : -$
$\quad member(birdsfly(A), B)$
$ex\_birdsfly(A, ths(B, [birdsfly(A)|B], C, C), D, ans(E, E)) : -$
$\quad variable\_free(birdsfly(A)),$
$\quad \backslash+member(birdsfly(A), B),$
$\quad \backslash+prove\_not\_birdsfly(A, [birdsfly(A)|B], anc([], []))$
$ex\_birdsfly(A, ths(B, B, C, [birdsfly(A)|C]), D, ans(E, E)) : -$
$\quad \backslash+variable\_free(birdsfly(A))$

5.5 Ancestor Search

Our model elimination theorem prover must recognise that a goal has been proven if it unifies with an ancestor in the search tree. To do this, it keeps two lists of ancestors, one containing the positive (non negated) ancestors and the other the negated ancestors. At run-time it searches one of these lists for an ancestor that unifies with the current goal.

At compile time we add such rules for each predicate symbol. When the ancestor search rules for predicate p are defined, we assert *ancestor_recorded(p)*, so that we do not attempt to redefine the ancestor search rules.

make_anc(L) tells the system that it should make the ancestor search rules for the literal L. This is called for each predicate that is used in a rule (section 5.2) or default (section 5.4).

```
make_anc(Name) :-
   ancestor_recorded(Name),
   !.
make_anc(n(Goal)) :-
   !,
   make_anc(Goal).
make_anc(Goal) :-
   Goal =.. [Pred|Args],
   same_length(Args,Nargs),
   NG =.. [Pred|Nargs],
   make_bodies(NG,_,[ths(T,T,D,D),anc(P,N),ans(A,A)],ProveG,ExG),
   make_bodies(n(NG),_,[ths(T,T,D,D),anc(P,N),ans(A,A)],ProvenG,ExnG),
   prolog_cl((ProveG :- member(NG,N))),
   prolog_cl((ProvenG :- member(NG,P))),
   prolog_cl((ExG :- member(NG,N))),
   prolog_cl((ExnG :- member(NG,P))),
   assert(ancestor_recorded(NG)).
```

Example 5.8 If we call

$$make\_anc(gr(A, B))$$

we create the Prolog clauses

$prove\_gr(A, B, C, anc(D, E)) :-$
    $member(gr(A, B), E).$
$prove\_not\_gr(A, B, C, anc(D, E)) :-$
    $member(gr(A, B), D).$
$ex\_gr(A, B, ths(C, C, D, D), anc(E, F), ans(G, G)) :-$
    $member(gr(A, B), F).$
$ex\_not\_gr(A, B, ths(C, C, D, D), anc(E, F), ans(G, G)) :-$
    $member(gr(A, B), E).$

This is only done once for the *gr* relation.

5.6 Explaining Observations

$expl(G, T0, T1, Ans)$ means that $T1$ is an explanation of Ans (Ans being the disjunct of alternate answers) from the facts given $T0$ is already assumed. The query G is an arbitrary wff.

```
expl(G,T0,T1,Ans) :-
   ground(N),
   declare_fact('<-'(newans(N,G) , G)),
   ex_newans(N,G,ths(T0,T,[],D),anc([],[]),ans(G,Ans)),
   ground(D),
   check_consis(D,T,T1).
```

$check\_consis(D, T0, T1)$, where D is a set of hypotheses and $T0$ is a set of hypotheses consistent with the facts, is true if $T1$ is $T0$ together with D and is consistent with the facts.

```
check_consis([],T,T).
check_consis([H|D],T1,T) :-
   new_lit("prove_not_", H,[T1,anc([],[])], Pr_n_H),
   \+ Pr_n_H,
   check_consis(D,[H|T1],T).
```

$ground(D)$ means that D is made ground by substituting unique constants for the variables.

To obtain disjunctive answers we have to know if the negation of the top level goal is called. This is done by declaring the fact $newans(N, G) \leftarrow G$, and if we ever try to prove the negation of a top level goal, we add that instance to the list of alternate answers. The index N is used to ensure that different proofs do not interfere with each other; we only want to use the new possible answer for the correct goal.

```
:- dynamic ex_not_newans/5.
ex_not_newans(N,G,ths(T,T,D,D),anc(Pos,Neg),ans(A,(G;A))) :-
   member(newans(N,_),Pos).
```

Although it does not make a different to correctness, it is important for efficiency to remove old instances of rules for *newans*. We currently do this by making sure that we always find all proofs and then removing the asserted clauses. This can be done simply by writing *prolog_cl* such that it retracts the clause when it is backtracked over.

6 Interface

In this section a minimal interface is given. We try to give enough so that we can understand the conversion of the wff form into negation normal form [2] and the parsing of facts and defaults. There is, of course, much more in any usable interface than described here.

6.1 Syntax Declarations

All of the declarations we use will be defined as operators. This will allow us to use infix forms of our wffs, for extra readability. Here we use the standard Edinburgh operator declarations which are given in the spirit of being enough to make the rest of the description self-contained.

```
:- op(1150,fx,fact).
:- op(1150,fx,default).
:- op(1150,fx,predict).
:- op(1150,fx,explain).
:- op(1130,xfx,:).
:- op(1110,xfx,<-).
:- op(1110,xfx,=>).
:- op(1100,xfy,or).
:- op(1000,xfy,and).
:- op(1000,xfy,&).
:- op(950,fy,~).
:- op(950,fy,not).
```

6.2 Converting to Negation Normal Form

We want to convert an arbitrarily complex formula into a standard form called *negation normal form*. Negation normal form [2] of a formula is a logically equivalent formula consisting of conjunctions and disjunctions of literals (either an atom or of the form $n(A)$ where A is an atom). The relation defined here puts formulae into negation normal form without distributing. Usually we want to find the negation normal form of the negation of the formula, as this is the form suitable for use in the body of a rule.

The predicate used is of the form

$$nnf(Fla, Parity, Body)$$

where

Fla is a formula with input syntax

$Parity$ is either *odd* or *even* and denotes whether Fla is in the context of an odd or even number of negations.

$Body$ is a tuple which represents the negation normal form of the negation of Fla if $Parity$ is even and the negation normal form of Fla if $Parity$ is odd.

This is defined by covering all of the forms that the formula could take.

```
nnf((X => Y), P,B) :- !,
   nnf((Y or not X),P,B).
nnf((Y <- X), P,B) :- !,
   nnf((Y or not X),P,B).
nnf((X & Y), P,B) :- !,
   nnf((X and Y),P,B).
nnf((X , Y), P,B) :- !,
   nnf((X and Y),P,B).
nnf((X ; Y), P,B) :- !,
   nnf((X or Y),P,B).
nnf((X and Y),P,B) :- !,
   opposite_parity(P,OP),
   nnf((not X or not Y),OP,B).
nnf((X or Y),even,(XB,YB)) :- !,
   nnf(X,even,XB),
   nnf(Y,even,YB).
nnf((X or Y),odd,(XB;YB)) :- !,
   nnf(X,odd,XB),
   nnf(Y,odd,YB).
nnf((~ X),P,B) :- !,
   nnf((not X),P,B).
nnf((not X),P,B) :- !,
   opposite_parity(P,OP),
   nnf(X,OP,B).
nnf(F,odd,F).
nnf(n(F),even,F) :- !.
nnf(F,even,n(F)).

opposite_parity(even,odd).
opposite_parity(odd,even).
```

Example 6.1 the wff

$$(a\ or\ not\ b)\ and\ c \Rightarrow d\ and\ (not\ e\ or\ f)$$

with parity *odd* gets translated into

$$d, (n(e); f); n(a), b; n(c)$$

the same wff with parity *even* (i.e., the negation of the wff) has negation normal form:

$$(n(d); e, n(f)), (a; n(b)), c$$

6.3 Theorist Declarations

The following define a subset of the Theorist declarations. Essentially these operators just call the appropriate compiler instruction.

```
fact F :- declare_fact(F),!.

default N : H :-
   !,
   declare_default(N),
   declare_fact((H <-N)),
   !.
default N :-
   declare_default(N),
   !.
```

The *explain* command writes out all explanations for instances of the query:

```
explain G :-
   expl(G,[],D,A),
   write([D,' is an explanation for ',A]),
   fail.
```

6.4 Prediction

In [29] we present a sceptical view of prediction, arguing that one should predict what is in every extension. This, propositionally at least, corresponds to circumscription [10]. The following theorem proved in [29, theorem 2.6] (a similar theorem has been given by Ginsberg [12] for circumscription under the unique names and domain closure assumptions) gives us a hint as to how prediction can be implemented.

Theorem 6.2 g is in every extension of $(\mathcal{F}, \Delta)$ iff there exists a set Σ of explanations of g such that there is no scenario of $(\mathcal{F}, \Delta)$ inconsistent with every member of Σ.[7]

We can use theorem 6.2 to consider another way to view membership in every extension. Consider two antagonistic agents Y and N trying to determine whether g should be predicted or not. Y comes up with explanations of g, and N tries to find where these explanations fall down (i.e., tries to find

[7]In other words there is no explanation of the negation of the disjunct of the elements of Σ.

a scenario S which is inconsistent with all of Y's explanations). Y then tries to find an explanation of g given S. If N cannot defeat Y's explanations then g is in every extension (one of Y's explanations is in every extension), and if Y cannot find an explanation from some S constructed by N then g is not in every extension (in particular g is not in any extension of S).

A direct implementation of such a procedure would have the protocol [30]:

- First Y finds an explanation of g, then N tries to find an explanation of the negation of that explanation.
- When N succeeds in finding a counter argument Y retries to find another explanation.
- When Y succeeds in finding another explanation, N is restarted to find an explanation for the negation of this explanation (as well as the negation of the other explanations).

Unfortunately the protocol of having two processes where one is retried when the other succeeds cannot be implemented in sequential Prolog[8].

There seems to be two ways to implement the procedure in sequential Prolog:

1. We can let the Y process run its full course and then let N find counter explanations to all of the explanations.
2. We can let N be the master process that calls Y to produce explanations as needed.

The following sections give implementations of both of these approaches. Note that both of these implementations only work for testing whether a ground atom is in all extensions. They do not do answer extraction [14].

6.5 Generate Explanations First

The following code implements the idea of generating all explanations of the query, and then finding counter arguments. What we really want is for the

[8]The problem is that we need two stacks for the processes, as one is expanding and shrinking when the other is waiting for a retry.

first "bagof" to generate the explanations in a demand-driven fashion, and then just print the explanations needed.

predict(G) is true for ground G is G is in all extensions of the facts and possible hypotheses

```
predict G :-
  bagof(E,expl(G,[],E,G),Es),
  predct(G,Es).
```

predct(G, Es) where Es is the set of explanations of G determines whether there is a counter to all of the explanations Es.

```
predct(G,Es) :-
   find_counter(Es,[],S),!,
   write(['No, ',G,' cannot be explained from ',S]).
predct(G,Es) :-
   write(['Yes, ',G,' is in all extensions. Explanations are:',Es]).
```

find_counter$(Es, S0, S1)$ where Es is a list of explanations, and $S0$ is a scenario, is true if $S1$ is a scenario that is a superset of $S0$ that is inconsistent with every element of Es.

```
find_counter([],S,S).
find_counter([E|R],S0,S2) :-
   member(D,E),
   expl(n(D),S0,S1,n(D)),
   find_counter(R,S1,S2).
```

6.6 Generate Explanations on Demand

The alternate definition is where N is the master process and can call Y to generate new explanations. The inefficiency here is due to Y having to restart from scratch each time. The other thing we need to notice is that N should not consider every permutation of explanations if it fails to find an explanation. This is why there is a cut after generating Y's explanations.

pred2(G, S) is true if G is in every extension of $(\mathcal{F} \cup S, \Delta)$.

```
pred2(G,S) :-
   expl(G,[],E,G),
  check_consis(E,S,_),
  !,
  \+ (( member(D,E),
        expl(n(D),S,S1,n(D)),
        \+ pred2(G,S1))).
```

This prccedure is similar to the previous definition except when we are trying to find an alternate explanation given a counter argument. Instead of considering the next element in the list of all explanations, we generate a new explanation from scratch.

This procedure is much more efficient that the first one when there are many explanations but only one or a few are needed in order to establish that the goal is in all extensions. However, it is much less efficient than the first procedure if many of the explanations are needed in order to establish that the goal is in all extensions.

Note that both of these procedures are concrete implementations of the abstract dialectical procedure for implementing membership in all extensions. This abstract procedure could be more accurately implemented by having multiple processes each computing explanations.

6.7 Comparison with other implementations

Independently of the work described in this paper, others have developed very similar procedures to the one presented here, as implementations of circumscription [34, 12]. These, however have not at all concentrated on efficiency issues as we have here. The algorithms of both Ginsberg [12] and Przymusinski [34] can be seen as variants of the first way (generating all explanations first) to implement membership in all extensions.

To aid the reader we give a vocabulary transform to understand the difference between Przymusinski's algorithm and ours.

First note that fixed predicates can be treated in Theorist by having both the predicate and its negation as a possible hypothesis [8].

Second, where this treat the problem of explanation as assuming hypotheses to imply a goal:

$$\mathcal{F} \models h_1 \wedge h_2 \wedge \ldots \wedge h_n \Rightarrow g$$

it can also be viewed as finding the consequence of the negated goal

$$\mathcal{F} \models \neg g \Rightarrow \neg h_1 \vee \neg h_2 \vee \ldots \vee \neg h_n$$

Our sets of explanations are implicitly disjoined, whereas in the consequence form they are conjoined.

Our implementation of explanation can be seen as a MESON version of Przymusinski's MILO resolution which is based on ordered linear resolution. The MILO leaves correspond to explanations (but negated as above). $Deriv(T, C)$ is the negation of the disjunct of explanations.

One main difference is that we try to check consistency as soon as possible; if an explanation is inconsistent, we want to know so that the search space can be reduced.

Przymusinski's algorithm [34] and Ginsberg's algorithm [12] correspond to the first implementation of membership in all extensions.

7 Refinements

In this paper we have given the bare bones of our compiler. The details should be enough to build an implementation. There are a number of refinements that we have implemented that are not presented here. Most of these are compiler switches, so that these refinements can be turned on and off for experimental purposes.

7.1 Sound Unification

The theorem prover we have given here is not sound because of the lack of the "occurs-check" in Prolog unification. We have not concentrated on it here as it is well covered by Stickel [39, 40]. Like Stickel, the Theorist implementation uses the idea of Plaisted [25]. When compiling rules, if there are two occurrences of a variable in the head of a clause, they are made into different variables and are unified at run time, with an occurs check.

For the common-sense reasoning examples we have been using, the occurs check has not been found to be a problem. Unlike theorem proving applications, we tend not to find "tricky" deductions (although I am sure we will be able to find them).

7.2 Depth-bound and Iterative Deepening

The second aspect that Stickel [39, 40] concentrated on was in the incompleteness of Prolog in that it can search infinite search branches even if there is a solution. We have implemented a depth-bound search by modifying *make_anc* to add a rule to cut the search if a depth-bound is exceeded. Iterative deepening can be implemented using this depth-bound by increasing it iteratively. The only problem is that we have to know whether a failure was due to natural causes (in which case it should be regarded as a finite failure), or whether it was due to the depth-bound being reached (in which case the depth-bound should be increased).

Again the profile of problems we have been considering has been different from theorem proving problems (e.g., in Stickel's tests [40]). Rather than all of our problems having trees with a large branching factor and some proofs at low depths, often the system is used more like a more powerful logic programming language, where the users ensure that the search trees are sparse (which is what we encourage them to do). As first pointed out by Meltzer [21], depth-bounds seem to be almost essential for any hypothetico-deductive account of reasoning. We are trying to empirically verify this, by trying to determine when a depth-bound is appropriate and needed.

7.3 Loop Check

Once we have an ancestor search, it is an easy matter to add a "loop check". This is implemented by having *make_anc* add a rule that cuts the search space if there is an ancestor in the tree that is identical[9] to the current subgoal [27]. This is accomplished by searching the opposite ancestor list to that searched by the ancestor search.

7.4 Combining Explanation and Prediction

One useful architecture for default and abductive reasoning is where explanation and prediction are combined into one system [30, 31]. Given an observation, we abduce to causes and then predict what follows from these hypothesises causes. To implement this, for each Theorist predicate p we create Prolog predicates *prove_p* (p can be proven from the facts), *ex_p* (p can

[9]Using the "==" of Edinburgh-syntax Prologs.

be explained using defaults and conjectures), *predict_p* (p can be explained using defaults only), and their negative forms. The *ex_p* and *predict_p* predicates are like the *ex_p* predicates used in this paper except they use different sets of possible hypotheses.

7.5 Incremental Explanations and Observations

When explaining observations in abductive reasoning, we would like to be able to incrementally add observations to be explained. One of the ideas we are experimenting with is to maintain the set of minimal and least presumptive [30] explanations so that new observations can be explained incrementally from the previous explanations. Whether such an idea can work in general, or whether one always runs out of space is something we are trying to investigate.

7.6 Simplifying and Pruning Explanations

The implementation finds one explanation for each proof. This does not guarantee that the explanations found will be either minimal or least presumptive [30]. We make sure that we do not have repeated instances of assumptions in the same explanation; another proof, however, may have needed a subset of the explanations.

Removing redundant explanations can be done at the end of the proof, but it would undoubtedly be better to not generate explanations that would be pruned. We are currently investigating two different strategies to handle this. The first is to use the idea of the previous section to find all explanations of part of the observations, prune the explanations that are not minimal and then try and explain further observations. The second idea is to build partial explanations, and to prune as soon as possible [24].

7.7 Explaining Answers

One feature of an expert system that makes it useful is the ability to explain answers. It is easy to add an extra argument to the Prolog predicates produces to return the proof tree, so that a Shapiro-like [36] debugging algorithm can be used. So far we have only built a simple debugger that can allow the user to traverse a proof tree.

7.8 Answer Extraction

One of the features of Theorist is that it always returns an explanation. What one knows is that there is a logical implication from the explanation given to the goal presented. Thus instead of the system making defeasible inferences, the explicit assumptions are returned so that the user can decide whether or not to accept them. This is most important for the membership in all extensions, where we either return the explanations that support the goal (and whose negation cannot be explained), or the scenario from which the goal cannot be explained.

One of the problems with the algorithm presented is that it returns a set of all explanations for the goal, rather than a minimal set of explanations of the goal. This also manifests itself in the inability to return disjunctive answers from membership in all extensions [14].

7.9 Refined Ancestor Search

One of the big problems that has not been addressed in this paper is the problem of removing redundant proofs. This has been found in practice, to be one of the biggest problems of using the MESON proof procedure. The problem is that whenever there is an ancestor resolution, there is always another proof tree that does the same ancestor resolution, but with the contrapositive of the intermediate rules. This problem also manifests itself in having non minimal disjunctive answers and being able to prove different permutations of disjunctive answers. While we could have solved this problem by hiding it in the simplification of explanations and answers, we preferred to have the problem in the open so that it could be solved in a clean way. One suggestion for solving this problem has been given by Spencer [38].

8 Runtime Considerations

What is given here is the core part of our current implementation of Theorist. This code has been used with Waterloo Unix Prolog, Quintus Prolog, C-prolog and Mac-Prolog. For those Prologs with compilers we can compile the resulting code from this translator as we could any other Prolog code; this makes it very fast indeed.

The resulting code when the Theorist code is of the form of definite clauses (the only case where a comparison makes sense, as it is what the two systems have in common), runs at about one quarter the speed of the corresponding interpreted or compiled code of the underlying Prolog system. About half of this extra cost is for the extra arguments to unify. The other factor is for one membership of an empty list for each procedure call: for each procedure call, we do one extra Prolog call to determine if the subgoal is a member of the negative ancestor list, which immediately fails. For definite clauses, contrapositives of the clauses are never used.

It may seem as though Theorist is always slower than Prolog. At one level of description this is obvious; as Theorist compiles to Prolog, it cannot be faster. It sometimes occurs that the natural Theorist description of a problem compiles into such efficient code, that it takes a very convoluted Prolog program to produce the same efficiency. One example of this is for the n-queens problem[10].

Example 8.1 Consider the Theorist representation of the 8 queens problem. $r(N)$ is true if there is a queen in row N. $q(M,N)$ is true if there is a queen in column M and row N. The facts specify the constraints on queens, in a standard way of stating how constraints on the numerical values of rows and columns. As the described implementation of Theorist does not use built-in functions, we axiomatise addition of positive integers *pplus*.

```
fact r(1) and r(2) and r(3) and r(4) and r(5) and r(6) and r(7)
          and r(8)  => goal.
fact q(X,Y) => r(Y).
default q(1,X).
default q(2,X).
default q(3,X).
default q(4,X).
default q(5,X).
default q(6,X).
default q(7,X).
default q(8,X).
fact not q(X,Y) <- q(X,Z), lt(Z,Y).
```

[10]Of course one would not really want to use depth-first search for this problem, but that is not the point here.

```
fact not q(X,Y) <- q(W,Z) ,
   ( pplus(X,D,W),(pplus(Y,D,Z); pplus(Z,D,Y))
   ; pplus(W,D,X),pplus(Y,D,Z) ).
fact lt(X,Y) <- pplus(X,Z,Y).
fact pplus(8,8,16).
fact pplus(8,7,15).
% other relations for pplus
fact pplus(1,2,3).
fact pplus(1,1,2).
```

The reason that this is efficient for a depth-first search is that the use of incremental consistency checking means that bad partial solutions are pruned as quickly as possible. To do this in Prolog requires a much less intuitive definition than the Theorist one presented here.

9 Conclusion

This paper has described in detail how we can translate Theorist code into Prolog so that we can use the advances in Prolog implementation technology.

We are currently working on many applications of default and abductive reasoning. Hopefully with compilers based on the ideas presented in this paper we will be able to take full advantage of advances in Prolog implementation technology while still allowing flexibility in specification of the problems to be solved.

Appendix: A Detailed Example

Consider the Theorist code:

> **fact** emu(A) => bird(A).
> **default** birdsfly(A): bird(A) => flies(A).
> **fact** not (birdsfly(A) and emu(A)).
> **fact** emu(tweety).
> **fact** bird(polly).

This appendix will show the exact code that this Theorist fragment gets compiled to according to the code in this paper.

fact emu(A) => bird(A).

gets translated into the forms for computing ancestor search for birds:

```
prove_bird(A,B,anc(C,D)) :-
   member(bird(A),D).
prove_not_bird(A,B,anc(C,D)) :-
   member(bird(A),C).
ex_bird(A,ths(B,B,C,C),anc(D,E),ans(F,F)) :-
   member(bird(A),E).
ex_not_bird(A,ths(B,B,C,C),anc(D,E),ans(F,F)) :-
   member(bird(A),D).
```

the rules for explaining and proving something is a bird:

```
prove_bird(A,B,anc(C,D)) :-
   prove_emu(A,B,anc([bird(A)|C],D)).
ex_bird(A,B,anc(C,D),E) :-
   ex_emu(A,B,anc([bird(A)|C],D),E).
```

the ancestor search for emus:

```
prove_emu(A,B,anc(C,D)) :-
   member(emu(A),D).
prove_not_emu(A,B,anc(C,D)) :-
   member(emu(A),C).
ex_emu(A,ths(B,B,C,C),anc(D,E),ans(F,F)) :-
   member(emu(A),E).
ex_not_emu(A,ths(B,B,C,C),anc(D,E),ans(F,F)) :-
   member(emu(A),D).
```

and the rules for explaining and proving something is a not an emu:

```
prove_not_emu(A,B,anc(C,D)) :-
   prove_not_bird(A,B,anc(C,[emu(A)|D])).
ex_not_emu(A,B,anc(C,D),E) :-
   ex_not_bird(A,B,anc(C,[emu(A)|D]),E).
```

The input:

default birdsfly(A): bird(A) => flies(A).

gets translated into

> **default** birdsfly(A).

which compiles into:

```
prove_birdsfly(A,B,anc(C,D)) :-
   member(birdsfly(A),D).
prove_not_birdsfly(A,B,anc(C,D)) :-
   member(birdsfly(A),C).
ex_birdsfly(A,ths(B,B,C,C),anc(D,E),ans(F,F)) :-
   member(birdsfly(A),E).
ex_not_birdsfly(A,ths(B,B,C,C),anc(D,E),ans(F,F)) :-
   member(birdsfly(A),D).
prove_birdsfly(A,B,C) :-
   member(birdsfly(A),B).
ex_birdsfly(A,ths(B,B,C,C),D,ans(E,E)) :-
   member(birdsfly(A),B).
ex_birdsfly(A,ths(B,[birdsfly(A)|B],C,C),D,ans(E,E)) :-
   variable_free(birdsfly(A)),
   \+member(birdsfly(A),B),
   \+prove_not_birdsfly(A,[birdsfly(A)|B],anc([],[])).
ex_birdsfly(A,ths(B,B,C,[birdsfly(A)|C]),D,ans(E,E)) :-
   \+variable_free(birdsfly(A)).
```

and the fact rule

$$flies(A) \leftarrow bird(A), birdsfly(A).$$

which is compiled into

```
prove_flies(A,B,anc(C,D)) :-
   member(flies(A),D).
prove_not_flies(A,B,anc(C,D)) :-
   member(flies(A),C).
ex_flies(A,ths(B,B,C,C),anc(D,E),ans(F,F)) :-
   member(flies(A),E).
ex_not_flies(A,ths(B,B,C,C),anc(D,E),ans(F,F)) :-
   member(flies(A),D).
```

```
prove_flies(A,B,anc(C,D)) :-
   prove_bird(A,B,anc([flies(A)|C],D)),
   prove_birdsfly(A,B,anc([flies(A)|C],D)).
ex_flies(A,ths(B,C,D,E),anc(F,G),ans(H,I)) :-
   ex_bird(A,ths(B,J,D,K),anc([flies(A)|F],G),ans(H,L)),
   ex_birdsfly(A,ths(J,C,K,E),anc([flies(A)|F],G),ans(L,I)).
```

and the fact rule

$$\neg bird(A) \leftarrow \neg flies(A), birdsfly(A).$$

which is compiled into

```
prove_not_bird(A,B,anc(C,D)) :-
   prove_not_flies(A,B,anc(C,[bird(A)|D])),
   prove_birdsfly(A,B,anc(C,[bird(A)|D])).
ex_not_bird(A,ths(B,C,D,E),anc(F,G),ans(H,I)) :-
   ex_not_flies(A,ths(B,J,D,K),anc(F,[bird(A)|G]),ans(H,L)),
   ex_birdsfly(A,ths(J,C,K,E),anc(F,[bird(A)|G]),ans(L,I)).
```

and the fact rule

$$\neg birdsfly(A) \leftarrow \neg flies(A), bird(A).$$

which is compiled into

```
prove_not_birdsfly(A,B,anc(C,D)) :-
   prove_not_flies(A,B,anc(C,[birdsfly(A)|D])),
   prove_bird(A,B,anc(C,[birdsfly(A)|D])).
ex_not_birdsfly(A,ths(B,C,D,E),anc(F,G),ans(H,I)) :-
   ex_not_flies(A,ths(B,J,D,K),anc(F,[birdsfly(A)|G]),ans(H,L)),
   ex_bird(A,ths(J,C,K,E),anc(F,[birdsfly(A)|G]),ans(L,I)).
```

The next declaration is

fact not (birdsfly(A) and emu(A)).

This gets translated into

```
prove_not_birdsfly(A,B,anc(C,D)) :-
   prove_emu(A,B,anc(C,[birdsfly(A)|D])).
ex_not_birdsfly(A,ths(B,C,D,E),anc(F,G),ans(H,I)) :-
   ex_emu(A,ths(B,C,D,E),anc(F,[birdsfly(A)|G]),ans(H,I)).
prove_not_emu(A,B,anc(C,D)) :-
   prove_birdsfly(A,B,anc(C,[emu(A)|D])).
ex_not_emu(A,ths(B,C,D,E),anc(F,G),ans(H,I)) :-
   ex_birdsfly(A,ths(B,C,D,E),anc(F,[emu(A)|G]),ans(H,I)).
```

fact emu(tweety).

gets translated into

```
prove_emu(tweety,A,B).
ex_emu(tweety,ths(A,A,B,B),C,ans(D,D)).
```

fact bird(polly).

gets translated into

```
prove_bird(polly,A,B).
ex_bird(polly,ths(A,A,B,B),C,ans(D,D)).
```

Acknowledgements

This work could not have been done without the ideas, criticism and feedback from Randy Goebel, Eric Neufeld, Paul Van Arragon, Scott Goodwin and Denis Gagné. Thanks to Andrew Csinger, Brenda Parsons and Amar Shan for valuable comments on previous versions of this paper. Thanks to anonymous reviewers for constructive reviews. This research was supported under NSERC grant OGPOO44121.

References

[1] G. Brewka, "Tweety – Still Flying: Some Remarks on Abnormal Birds, Applicable Rules and a Default Prover", *Proc. AAAI-86*, pp. 8-12.

[2] C-L. Chang and R. C-T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[3] P. T. Cox, *Dependency-directed backtracking for Prolog Programs.*

[4] P. T. Cox and T. Pietrzykowski, *General Diagnosis by Abductive Inference*, Technical report CS8701, School of Computer Science, Technical University of Nova Scotia, April 1987.

[5] M. Dincbas, H. Simonis and P. Van Hentenryck, *Solving Large Combinatorial Problems in Logic Programming*, ECRC Technical Report, TR-LP-21, June 1987.

[6] J. Doyle, "A Truth Maintenance System", *Artificial Intelligence*, Vol. 12, pp 231-273.

[7] J. de Kleer, "An Assumption-based TMS", *Artificial Intelligence, Vol. 28, No. 2*, pp. 127-162, 1986.

[8] J. de Kleer and K. Konolige, "Eliminating the fixed predicates from a circumscription", *Artificial Intelligence*, 39 (3), 391-398, 1989.

[9] H. B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, Orlando.

[10] D. W. Etherington, *Reasoning with Incomplete Information*, Morgan Kaufmann, 1988.

[11] M. Genesereth and N. Nilsson, *Logical Foundations of Artificial Intelligence*, Morgan-Kaufmann, Los Altos, California.

[12] M. L. Ginsberg, "A circumscriptive theorem prover", *Artificial Intelligence*, 39(2), 209-230, 1989.

[13] R. G. Goebel and S. D. Goodwin, "Applying theory formation to the planning problem" in F. M. Brown (Ed.), *Proceedings of the 1987 Workshop on The Frame Problem in Artificial Intelligence*, Morgan Kaufmann, pp. 207-232.

[14] N. Helft, K. Inoue and D. Poole, "Extracting answers in circumscription", *ICOT Technical Memorandum TM-855*, ICOT, 1989.

[15] R. Kowalski, "Algorithm = Logic + Control", *Comm. A.C.M.* Vol 22, No 7, pp. 424-436.

[16] V. Lifschitz, "Computing Circumscription", *Proc. IJCAI85*, pp. 121-127.

[17] J. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 2nd Edition, 1987.

[18] D. W. Loveland, *Automated Theorem Proving: a logical basis*, North-Holland, Amsterdam, 1978.

[19] D. W. Loveland, "Near-Horn Logic Programming", *Proc. 6th International Logic Programming Conference*, 1987.

[20] J. McCarthy, "Applications of Circumscription to Formalising Common Sense Knowledge", *Artificial Intelligence*, Vol. 28, No. 1, pp. 89-116, 1986.

[21] B. Meltzer, "The semantics of induction and the possibility of complete systems of inductive reasoning", *Artificial Intelligence*, 1 (1970), 189-192.

[22] T. Moto-Oka, H. Tanaka, H. Aida, k. Hirata and T. Maruyama, "The Architecture of a Parallel Inference Engine — PIE", *Proc. Int. Conf. on Fifth Generation Computing Systems*, pp. 479-488, 1984.

[23] L. Naish, "Negation and Quantifiers in NU-PROLOG", *Proc. 3rd Int. Conf. on Logic Programming*, Springer-Verlag, pp. 624-634, 1986.

[24] E. M. Neufeld and D. Poole, "Towards solving the multiple extension problem: combining defaults and probabilities", *Proc. Third AAAI Workshop on Reasoning with Uncertainty*, Seattle, pp. 305-312, 1987.

[25] D. A. Plaisted, "The occur-check problem in Prolog", *New Generation Computing*, 2 (1984), 309-322.

[26] D. L. Poole, "Making Clausal theorem provers Non-clausal", *Proc. CSCSI-84*. pp. 124-125, 1984.

[27] D. Poole and R. G. Goebel, "On Eliminating loops in Prolog", *SIGPLAN Notices, Vol 20, No 8*, August 1985, pp. 38-40.

[28] D. Poole and R. Goebel, "Gracefully adding Negation to Prolog", *Proc. Fifth International Logic Programming Conference*, London, pp. 635-641, 1986.

[29] D. Poole, "A Logical Framework for Default Reasoning", *Artificial Intelligence*, 36(1), 27-47, 1987.

[30] D. Poole, "Explanation and prediction: an architecture for default and abductive reasoning, *Computational Intelligence* 5(2), 1989, 97-110.

[31] D. Poole, "A methodology for using a default and abductive reasoning system", to appear *International Journal of Intelligent Systems*, 1990.

[32] D. L. Poole, R. G. Goebel and R. Aleliunas, "Theorist: A Logical Reasoning System for Defaults and Diagnosis", in N. Cercone and G. McCalla (Eds.) *The Knowledge Frontier: Essays in the Representation of Knowledge*, Springer Varlag, New York, 1987, pp. 331-352.

[33] H. E. Pople, "On the mechanization of abductive logic", *Proc. IJCAI-73* 147-152, 1973.

[34] T. C. Przymusinski, "An algorithm for computing circumscription", *Artificial Intelligence*, 38 (1), 49-74, 1989.

[35] R. Reiter, "A Logic for Default Reasoning", *Artificial Intelligence*, Vol. 13, pp 81-132.

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

[37] D. Smith and M. Genesereth, "Ordering Conjunctive Queries", *Artificial Intelligence* 1886.

[38] E. B. Spencer, "Avoiding Redundant Explanations" *Proc. NACLP-90* Austin, 1990.

[39] M. E. Stickel, "A Prolog Technology Theorem Prover", *New Generation Computing*, 2 (1984) 371-383.

[40] M. E. Stickel, "A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler", *Journal of Automated Reasoning*, 4 (1988) 353-380.

[41] M. E. Stickel, "A Prolog technology theorem prover: a new exposition and implementation in Prolog", Technical Note 464, SRI International, June 1989.

[42] P. Van Hentenryck, "A Framework for consistency techniques in Logic Programming" IJCAI-87, Milan, Italy.