Printing Requisition / Graphic Services

26249

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number,** to **extension 3451.**

TITLE OR DESCRIPTION: CS-88-01 Rev.

DATE REQUISITIONED: ~~Dec 12/88~~ Jan. 9/89 | DATE REQUIRED: ASAP | ACCOUNT NO.: 1 1 2 1 6 1 6 1 4 1 4 1

REQUISITIONER - PRINT: L | PHONE: 2112 | SIGNING AUTHORITY

MAILING INFO - NAME: S. DeIngelis | DEPT.: C.S. | BLDG. & ROOM NO.: DC 2314 | ☑ DELIVER ☐ PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 34 | NUMBER OF COPIES: ~~15~~ 20

TYPE OF PAPER STOCK: ☑ BOND ☐ NCR ___ PT. ☑ COVER ☐ BRISTOL ☑ SUPPLIED ☐ ___

PAPER SIZE: ☑ 8½ x 11 ☐ 8½ x 14 ☐ 11 x 17 ☐ ___

PAPER COLOUR: ☑ WHITE ☐ ___ | INK: ☑ BLACK ☐ ___

PRINTING: ☐ 1 SIDE ___ PGS. ☑ 2 SIDES ___ PGS. | NUMBERING: FROM ___ TO ___

BINDING/FINISHING: 3 down left side ☑ COLLATING ☑ STAPLING ☐ ___ HOLE PUNCHED ☐ PLASTIC RING

FOLDING/PADDING | CUTTING SIZE

Special Instructions

With mats + Disks Enclosed.

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

D 0 1
D 0 1
D 0 1

TYPESETTING — QUANTITY

PAP 0 0 0 0 0 0 — T 0 1
PAP 0 0 0 0 0 0 — T 0 1
PAP 0 0 0 0 0 0 — T 0 1

PROOF

PRF
PRF
PRF

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

FLM — C 0 1
FLM — C 0 1
FLM — C 0 1
FLM — C 0 1
FLM — C 0 1

PMT

PMT — C 0 1
PMT — C 0 1
PMT — C 0 1

PLATES

PLT — P 0 1
PLT — P 0 1
PLT — P 0 1

STOCK

0 0 1
0 0 1
0 0 1
0 0 1

BINDERY

RNG — B 0 1
RNG — B 0 1
RNG — B 0 1
MIS 0 0 0 0 0 0 — B 0 1

OUTSIDE SERVICES

$ ___ COST

TAXES - PROVINCIAL ☐ FEDERAL ☐

GRAPHIC SERV. OCT. 85 482-2

Compiling A Default Reasoning System into Prolog

David L. Poole
Department of Computer Science

Research Report CS-88-01
January 1988

Compiling A Default Reasoning System into Prolog

David Poole
Logic Programming and Artificial Intelligence Group,
Department of Computer Science,
University of Waterloo,
Waterloo, Ontario, Canada, N2L3G1
(519) 888-4443
dlpoole@dragon.waterloo.edu

January 4, 1988

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 the 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 [Reiter80,McCarthy86,Cox87,Poole88].

The systems implemented so far (eg., [Brewka86,Lifschitz85,Ginsburg87,PGA87]) 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 [Poole88,PGA87] by compiling its input into Horn clauses with negation as failure, thereby allowing direct use the advances in logic programming implementation technology. Both the compiler and the compiled code can take advantage of these improvements.

We have been running this implementation on standard Prolog compilers (in particular Quintus Prolog) and it outperforms all other default reasoning systems that the author is aware of. It is, however, not restricted to the control structure of Prolog. There is nothing in the compiled code which forces it to use Prolog's search strategy. Logic programming and other researchers are working on alternate control structures which seem very appropriate for default and abductive reasoning. Advances in parallel inference (e.g., [Moto-Oka84]), constraint satisfaction [Dincbas87,Van Hentenryck87] and dependency directed backtracking [de Kleer86,Doyle79,Cox82] should be able to be directly 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 [Kowalski79]. We can let the control people concentrate on efficiency of Horn clause systems, and these will then 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 that are requited for default and abductive reasoning.

This is a step towards having representation and reasoning systems which are designed for correctness being able to use the most efficient of control strategies, so we can have the best of expressibility and efficiency.

2 Theorist Framework

Theorist [Poole88,PGA87] is a logical reasoning system for default reasoning and diagnosis. It is based on the idea of theory formation from a fixed set of possible hypotheses.

This implementation is of the version of Theorist described in [Poole88]. The user provides three sets of first order formulae

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

Δ is a set of formulae which act as *possible hypotheses*, any ground instance of which can be used in an explanation if consistent.

C is a set of closed formulae taken as constraints. The constraints restrict what can be hypothesised.

We assume that $\mathcal{F} \cup C$ is consistent.

Definition 1 a **scenario** of $\mathcal{F}, \Delta$ is a set $D \cup \mathcal{F}$ where D is a set of ground instances of elements of Δ such that $D \cup \mathcal{F} \cup C$ is consistent.

Definition 2 If g is a closed formula then an **explanation** of g from $\mathcal{F}, \Delta$ is a scenario of $\mathcal{F}, \Delta$ which implies g.

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

$\mathcal{F} \cup D \models g$ and
$\mathcal{F} \cup D \cup C$ is consistent

$\mathcal{F} \cup D$ is an explanation of g.

In other papers we have described how this can be the basis of default and abductive reasoning systems [PGA87,Poole88,Poole87b,Poole86]. If we are using this for prediction then possible hypotheses can be seen as defaults. [Poole88] describes how this formalism can account for default reasoning. 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 [PGA87]. We will refer to possible hypotheses as defaults.

One restriction that can be made with no loss of expressive power is to restrict possible hypotheses to just atomic forms with no structure [Poole88]. 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 wffs; 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 [Poole84].

A *wff* is a well formed formula made up of arbitrary combination of implication ("=>", "< −"), disjunction ("*or*", ";"), conjunction ("*and*", "&", ",") and negation ("*not*", "~") of atomic symbols. Variables follow the Prolog convention of being in upper case. There is no explicit quantification.

A *name* is an 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 $(\forall w) \in \mathcal{F}$; i.e., the universal closure of w (all variables universally quantified) is a fact.

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

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 $(\forall d \Rightarrow w) \in \mathcal{F}$.

constraint w.
: where w is a wff means $\forall w \in \mathcal{C}$.

prolog p.
: where p is an atomic symbol means any Theorist call to p should be

proven in Prolog. This allows us to use built-in predicates of pure Prolog. One should not expect Prolog's control predicates to work.

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

3 Overview of Implementation

In this section we assume that we have a 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 can be proven. I.e., 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 predicates which were used in the proof.

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

Algorithm 1 to explain g

1. try to prove g from $\mathcal{F} \cup \Delta$. If no proof exists, then g is not explainable. 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 $\mathcal{F} \cup C$ by failing to prove it is inconsistent.

3.1 Consistency Checking

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

Lemma 1 *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 $D = \{d_1, ..., d_n\}$*

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

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). □

This lemma says that we can show that $\mathcal{F} \cup C \cup \{d_1, ..., d_n\}$ is consistent if we can show that for all i, $1 \leq i \leq n$, $\mathcal{F} \cup C \cup \{d_1, ..., d_{i-1}\} \nvdash \neg d_i$. 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 2 *If $\mathcal{F} \cup C$ is consistent, g is explainable 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 $\mathcal{F} \wedge C \wedge \{d_1, ..., d_{i-1}\} \nvdash \neg d_i$ for all $i, 1 \leq i \leq n$.*

Proof: If g is explainable 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 $\mathcal{F} \cup D \cup C$ is consistent. So there is a ground proof of g from $\mathcal{F} \cup D$. By the preceding lemma $\mathcal{F} \cup D \cup C$ is consistent so there can be no sound proof of inconsistency. That is, we cannot prove $\mathcal{F} \wedge C \wedge \{d_1, ..., d_{i-1}\} \vdash \neg d_i$ for any i. □

This leads us to the refinement of algorithm 1:

Algorithm 2 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 $\mathcal{F} \wedge C \wedge \{d_1, ..., d_{i-1}\}$. 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

Notice that theorem 2 says that g is explainable if there is a ground proof. There is a problem that arises to translate the preceding algorithm (which assumes ground proofs) into an algorithm 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 generated. Consider the following example:

Example 1 Let $\Delta = \{p(X)\}$. That is, any instance of p can be used if it is 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 is explainable 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 we want g to be explainable.

Theorem 3 *It is not adequate to only consider interpretations in the Herbrand universe of* $\mathcal{F} \cup \Delta \cup C$ *to determine explainability.*

Proof: consider the example above; the Herbrand universe is just the set $\{a\}$. Within this domain there is no consistent explanation to explain g. □

This shows that Herbrand's theorem is not applicable to the whole system. It is, however, applicable to each of the deduction steps [Chang73].

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

Algorithm 3 To determine if g is explainable 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;
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 explainable.

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

Example 2 Consider $\mathcal{F}$ and Δ as in example 1 above. 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 must show that all of the instances are inconsistent. The above algorithm says we replace Y with a constant β. $p(\beta)$ is consistent with the facts. Thus we can show g is explainable.

3.3 Incremental Consistency Checking

Algorithm 3 assumed that we only check consistency at the end. We cannot replace free variables by a unique constant until the end of the computation. This algorithm can be further refined by considering cases where we can check consistency at the time the hypothesis is generated.

Theorem 1 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 Consider the two alternate set of facts:

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

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 seems to be two obvious solutions to this problem, the first is to allow the consistency check to return constraints on the values (eg., [Edmonson87]). The alternate (and simpler) solution is to delay the evaluation of the consistency check until all of the variables are bound (as in [Naish86]), 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 simplest 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.

4 The Deduction System

This implementation is based on linear resolution [Chang73,Loveland78]. This is complete in the sense that if g logically follows from some *consistent* set of clauses A, then there is a linear resolution proof of g from A.

SLD resolution of Prolog [Llord87] can be seen as linear resolution with the contrapositive and ancestor search removed.

To implement linear resolution in Prolog, 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

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

   as rules. Each of these can then 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 (i.e., we treat $\neg p(\overline{X})$ as an atom $notp(\overline{X})$).

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 $\neg L$ and then when we have proven L we can discharge the assumption.

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 [Poole86], but this implementation uses the very simple form given above. Another tempting possibility is to use the near-Horn resolution of [Loveland87]. More work needs to be done on this area.

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 up conditions on the proof of alternate answers that we are assuming are not true in order to have the disjunctive answer.

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

4.2 Conversion to Clausal Form

It is desirable that we can convert an arbitrary well formed formula into clausal (or rule) form without mapping out subterms. Instead of distributing, we do this 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 [Chang73] 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.

The alternate [Poole84] 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 can be embedded into the compiler by using Prolog "or" instead of actually building the p. (Alternatively we can define "or" by defining

the clause $(p;q) \leftarrow p$ and $(p;q) \leftarrow q$.) 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 [Poole84].

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 search strategy in the compiled code. Each Theorist wff gets locally translated into a set of Prolog clauses.

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 the constraints.

prove_not_r meaning $\neg r$ can be proven from the facts and the constraints.

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 will contain all of the information that we need 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 constraints 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 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 source 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,

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

means

$$\mathcal{F} \cup Ths \cup \neg P \cup N \models r(-args-)$$

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 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 that T_1 is the consistent hypotheses before we try to explain r, and and T_2 is the list of consistent hypotheses which includes T_1 and those hypotheses assumed to explain r. Similarly, D_1 is the list of deferred hypotheses before we consider the goal and D_2 is the list of resulting deferred hypotheses used in explaining r.

Anc contains the ancestors of the goal. As in the previous case, 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 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

$$\mathcal{F} \cup T_2 \cup D_2 \cup \neg P \cup N \cup A_2 \models r(-args-)$$

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

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

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[1]:

```
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.

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 rules is either a literal or of the form $H \leftarrow Body$ (written "`if(H,Body)`") where H is a literal and *Body* is a conjunction and disjunction of literals.

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

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 4 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 are the assumptions made in explaining gr.

Example 5 the fact

$$father(randy, jodi)$$

gets translated into

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

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

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

Example 6 the rule

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

gets translated into

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

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

Example 7 the fact

$$father(randy, jodi)$$

gets translated into

$$prove\_father(randy, jodi, \_, \_).$$

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 8 The rule

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

gets translated into

$$\begin{array}{l} prove\_p(A, B, anc(C, D)) : - \\ \quad prove\_q(A, B, anc([p(A)|C], D)), \\ \quad (\quad prove\_r(A, B, anc([p(A)|C], D)), \\ \qquad prove\_s(A, B, anc([p(A)|C], D)) \\ \quad ; \quad prove\_t(A, B, anc([p(A)|C], D))), \\ \quad prove\_m(A, B, anc([p(A)|C], D)). \end{array}$$

$$\begin{array}{l} ex\_p(A, ths(B, C, D, E), anc(F, G), ans(H, I)) : - \\ \quad ex\_q(A, ths(B, J, D, K), anc([p(A)|F], G), ans(H, L)), \\ \quad (\quad ex\_r(A, ths(J, M, K, N), anc([p(A)|F], G), ans(L, O)), \\ \qquad ex\_s(A, ths(M, P, N, Q), anc([p(A)|F], G), ans(O, R)) \\ \quad ; \quad ex\_t(A, ths(J, P, K, Q), anc([p(A)|F], G), ans(L, R))), \\ \quad ex\_m(A, ths(P, C, Q, E), anc([p(A)|F], G), ans(R, I)) \end{array}$$

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 theory for the proving, 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 $prove\_b_i$ (and $prove\_not\_b_i$), 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 $rule(F, R)$ declares R to be a fact or constraint rule (depending on the value of F). Constraints can only be used for proving; facts can be used for explaining as well as proving. 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 5.7, and can be ignored on first reading.

```
rule(F,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)),
   ( F=fact,
     prolog_cl((ExH:-ExB))
   ; F=constraint).

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

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 both facts and constraints 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 are more efficient than others (see for example [Smith86] for a discussion of such issues).

The declarations are as follows:

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

declare_constraint(C) :-
```

```
    nnf(C,even,N),
    rulify(constraint,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(H, N) where H is either "*fact*" or "*constraint*" and N is the negation of a fact or constraint 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 declared as such.

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

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

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

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

contrapos(H, D, T) where H is either "*fact*" or "*constraint*", and (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.

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

contrapos(H,D,(L;R)) :- !,
```

```
    contrapos(H,D,L),
    contrapos(H,D,R).

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

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

Example 9 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*

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

5.4 Possible Hypotheses

The other class of things we have to worry about is the class of possible hypotheses. As described in [Poole88] 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 only prove a hypothesis if we have already assumed it:

   $$prove\_d(-args-, Ths, Anc) :-$$
   $$\quad member(d(-args-), Ths).$$

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

   $$ex\_d(-args-, ths(T, T, D, D), Anc, ans(A, A)) :-$$
   $$\quad member(d(-args-), T).$$

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:

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

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):

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

The following compiles directly 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 10 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 Relations defined in Prolog

We can define some relations to be executed in Prolog. This means that we can prove the *prove* and *ex* forms by calling the appropriate Prolog definition.

```
declare_prolog(G) :-
   new_lit("ex_",G, [ths(T,T,D,D), _, ans(A,A)], ExG),
   prolog_cl((ExG :- G)),
   new_lit("prove_",G,[_,_],PrG),
   prolog_cl((PrG :- G)).
```

5.6 Explaining Observations

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

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

check_consis([],T,T).
check_consis([H|D],T1,T) :-
```

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

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 $newans(G) \leftarrow G$, and if we ever try to prove the negation of a top level goal, we add that instance to the list of alternate answers. In this implementation we also check that G is not identical to a higher level goal. This removes most cases where we have a redundant assumption (i.e., when we are not gaining a new answer, but only adding redundant information).

```
ex_not_newans(G,ths(T,T,D,D),anc(Pos,Neg),ans(A,[G|A])) :-
   \+ id_anc(G,anc(Pos,Neg)).

id_anc(n(G),anc(_,N)) :- !, id_member(G,N).
id_anc(G,anc(P,_)) :- id_member(G,P).
```

5.7 Ancestor Search

Our linear resolution 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. 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(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 11 *if we do a 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.

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 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,constraint).
:- op(1150,fx,default).
:- op(1150,fx,prolog).
:- 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 of a formula is an 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 mapping out subterms. 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.

```
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 12 the wff

(a or not b) and c ⇒ d and (not e or f)

with parity *odd* gets translated into

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

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. These indicate how this can be done. 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),
   !.

constraint C :-
   declare_constraint(C),
   !.
```

7 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 actually compile the resulting code from this translater as we could any other Prolog code; this make 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 half the speed of the corresponding interpreted or compiled code of the underlying Prolog system. This seems

reasonable as all we are doing is one membership of an empty list (which should immediately fail) for each procedure call. Thus for each procedure call, we do one extra Prolog call which fails. The contrapositive of the clauses are never used.

8 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.

As far as this compiler is concerned there are a few issues which arise:

- Is there a more efficient way to determine that a goal can succeed because it unifies with an ancestor [Poole86,Loveland87]?
- Can we incorporate a cycle check that has a low overhead? A simple, but expensive, version is implemented in some versions of our compiler which checks for identical ancestors.
- Are there optimal ordering which we can put the compiled clauses in so that we get answer most quickly [Smith86]? At the moment the compiler just puts the elements of the bodies in an arbitrary ordering. The optimal ordering depends, of course, on the underlying control structure.
- Are there better ways to do the consistency checking when there are variables in the hypotheses?

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).
constraint 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

constraint 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])).
prove_not_emu(A,B,anc(C,D)) :-
   prove_birdsfly(A,B,anc(C,[emu(A)|D])).
```

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 Brenda Parsons and Amar Shan for valuable comments on a previous version of this paper. This research was supported under NSERC grant A6260.

References

[Brewka86] G. Brewka, "Tweety – Still Flying: Some Remarks on Abnormal Birds, Applicable Rules and a Default Prover", *Proc. AAAI-86*, pp. 8-12.

[Chang73] C-L. Chang and R. C-T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[Cox82] P. T. Cox, *Dependency-directed backtracking for Prolog Programs.*

[Cox87] 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.

[Dincbas87] M. Dincbas, H. Simonis and P. Van Hentenryck, *Solving Large Combinatorial Problems in Logic Programming*, ECRC Technical Report, TR-LP-21, June 1987.

[Doyle79] J. Doyle, "A Truth Maintenance System", *Artificial Intelligence*, Vol. 12, pp 231-273.

[de Kleer86] J. de Kleer, "An Assumption-based TMS", *Artificial Intelligence, Vol. 28, No. 2*, pp. 127-162.

[Edmonson87] R. Edmonson, ????

[Enderton72] H. B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, Orlando.

[Genesereth87] M. Genesereth and N. Nilsson, *Logical Foundations of Artificial Intelligence*, Morgan-Kaufmann, Los Altos, California.

[Ginsburg87] M. L. Ginsburg, *Computing Circumscription*, Stanford Logic Group Report Logic-87-8, June 1987.

[Goebel87] 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.

[Kowalski79] R. Kowalski, "Algorithm = Logic + Control", *Comm. A.C.M.* Vol 22, No 7, pp. 424-436.

[Lifschitz85] V. Lifschitz, "Computing Circumscription", *Proc. IJCAI85*, pp. 121-127.

[Llord87] J. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 2nd Edition.

[Loveland78] D. W. Loveland, *Automated Theorem Proving: a logical basis*, North-Holland, Amsterdam.

[Loveland87] D. W. Loveland, "Near-Horn Logic Programming", *Proc. 6th International Logic Programming Conference.*

[McCarthy86] J. McCarthy, "Applications of Circumscription to Formalising Common Sense Knowledge", *Artificial Intelligence*, Vol. 28, No. 1, pp. 89-116.

[Moto-Oka84] 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.

[Naish86] L. Naish, "Negation and Quantifiers in NU-PROLOG", *Proc. 3rd Int. Conf. on Logic Programming*, Springer-Verlag, pp. 624-634.

[Neufeld87] 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.

[Poole84] D. L. Poole, "Making Clausal theorem provers Non-clausal", *Proc. CSCSI-84*. pp. 124-125.

[Poole86] D. L. Poole, "Gracefully adding Negation to Prolog", *Proc. Fifth International Logic Programming Conference*, London, pp. 635-641.

[Poole86] D. L. Poole, "Default Reasoning and Diagnosis as Theory Formation", Technical Report, CS-86-08, Department of Computer Science, University of Waterloo, March 1986.

[Poole87a] D. L. Poole, "Variables in Hypotheses", *Proc. IJCAI-87*, pp. 905-908.

[Poole87b] D. L. Poole, *Defaults and Conjectures: Hypothetical Reasoning for Explanation and Prediction*, Research Report CS-87-54, Department of Computer Science, University of Waterloo, October 1987, 49 pages.

[Poole88] D. L. Poole, *A Logical Framework for Default Reasoning*, to appear *Artificial Intelligence*, Spring 1987.

[PGA87] 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.

[Reiter80] R. Reiter, "A Logic for Default Reasoning", *Artificial Intelligence*, Vol. 13, pp 81-132.

[Smith86] D. Smith and M. Genesereth, "Ordering Conjunctive Queries", *Artificial Intelligence.*

[Van Hentenryck87] P. Van Hentenryck, "A Framework for consistency techniques in Logic Programming" IJCAI-87, Milan, Italy.

X-24601 REV. 3/86

VENDOR CODE: 390020 REMITTANCE ADVICE

CHECK NUMBER 0756632

VENDOR NAME: UNIVERSITY OF WATERLOO

CHECK DATE: 12/19/88

INV NUMBER	INV DT	PO #	CD	AMT BILLED	DISCOUNT	DEBIT AMT	NET AMT
CS8801	110388	B 003048	V	4.00	.00	.00	4.00

received Jan. 6/89

ATTACH. CODE: N=NONE B=BSS V=VENDOR X=BOTH

2704607

A DIVISION OF THE BOEING COMPANY ** P.O. BOX 3707 ** SEATTLE, WA 98124

BOEING SUPPORT SERVICES

CHECK AMOUNT → *******4.00

University of Waterloo
Department of Computer Science
Waterloo, Ontario N2L 3G1

November [redacted] 1988

INVOICE

The Boeing Company
Library Acquisitions
P.O. Box 3707 M/S 74-60
Seattle, WA. 98124-2207
U.S.A.

Purchase Order L9939

REPORT(S) ORDERED

CS-88-01

TOTAL COST $4.00 (This price includes postage)

Would you please make your cheque or international bank draft payable to the **Computer Science Department, University of Waterloo** and forward to my attention.

Thanking you in advance.

Yours truly,

Susan DeAngelis

Susan DeAngelis (Mrs.)
Research Report Secretary
Computer Science Dept.

/sd

Encl.

NOV - 2 1988

BOEING TECHNICAL LIBRARY

OFFICIAL PURCHASE ORDER

PURCHASE ORDER NUMBER	PLEASE REFERENCE THIS PURCHASE ORDER NO. ON ALL INQUIRIES, INVOICES BILLS OF LADING, ETC.	P.O. DATED YR. MO. DAY
L9939		88/10/18

SHIP TO
THE BOEING COMPANY
LIBRARY ACQUISITIONS
P.O. BOX 3707 M/S 74-60
SEATTLE, WA 98124-2207
U.S.A.

BILL TO
THE BOEING COMPANY
LIBRARY ACQUISITIONS
P.O. BOX 3707 M/S 74-60
SEATTLE, WA 98124-2207
U.S.A.

QUANTITY	ISBN	TITLE DESCRIPTION	EST. UNIT PRICE
1		POOLE, D L. COMPILING A DEFAULT REASONING SYSTEM INTO PROLOG. NOTE: 1988. NOTE: CS-88-01. NOTE: 109552. NOTE: HARD COPY. EDITION: LATEST.	30.00

G. O. Pierce
AUTHORIZED SIGNATURE

ACCOUNTS
OCT 27 1988
PAYABLE

U.P.S. SHIPPING ADDRESS:

BOEING CENTRAL RECEIVING
LIBRARY, 237-1563, MS 74-60
BLDG 4-63, DR 27
LOGAN N & N 6TH ST
RENTON, WA 98055

UNIVERSITY OF WATERLOO
WATERLOO, ONT
N2L 3G1
CANADA

DIRECT ORDER INQUIRIES TO:

THE BOEING COMPANY
LIBRARY ACQUISITIONS
P.O. BOX 3707, M/S 74-60
SEATTLE, WA 98124-2207
TEL:(206) 237-1563

DEC - 9 1988

National Centre for Software Technology

Gulmohar Cross Road No. 9, Juhu, Bombay 400 049

Telephone: 62 96 06/62 95 74
Telex: 011-78260 NCST IN

Dy. Librarian

24.11.88

Ref:NCST/LIB/ 6188

Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
U.S.A.

Dear Sir,

We hereby place our firm order for the titles (Technical Report '1988) listed below/in the enclosure.

Please send your proforma invoice in quadruplicte for advance payment. Also indicate the handling/shipping charges and discount applicable to R & D Centres/educational institutions.

Awaiting an early receipt of invoice.

Yours faithfully,

Shah
(Dy. Librarian)

1. D. Poole, "Compiling a default reasoning system into Prolog." CS-88-01

sent with compliments
Dec. 12/88

COMMANDE DE LIVRE / *BOOK ORDER* → *TO QUOTE* | N° 0856878/00 /LIB | DATE 26 10 88

QTÉ/*COP.*	TITRE/*TITLE*	AUTEUR/*AUTHOR* - ÉDITEUR/*PUBLISHER*...
001	COMPILING A DEFAULT REASONING SYSTEM INTO PROLOG CS 88 01 RESEARCH REPORTS 1988	POOLE D. AUT./*AUTH.* VOL. ANNÉE/*YEAR* VERS. ED./*PUBL.*
MESSAGE		
EXP./*SHIP*	SURFACE	

Sent with compliments Nov. 29/88

dawson france
SERVICE LIBRAIRIE-*BOOKS DEPT.*
AGRÉMENT DOUANE I 20
B.P. 40
91121 PALAISEAU CEDEX (FRANCE)
TÉL. : (1) 69.09.01.22 - TÉLEX 600 394 F
Télécopieur (1) 64.54.83.26

WATERLOO (UNIVERSITY OF)
SUSAN DE ANGELIS-FAC.MATHEMATICS
DPT COMPUTER SCIENCE
WATERLOO ONTARIO N2L 3G1
CANADA

872311 billon CERB

OUR ACCT:*X

EXPÉDITION PAQUET POSTE
SHIPMENT BY BOOKPOST
FACT. 3 EX. / *INVOICE IN TRIPLICATE*

REMISE LIBRAIRE SVP / *BOOKSELLER'S DISCOUNT PLEASE*

IMPORTANT : *NOTICE TO PUBLISHERS* / RECOMMANDATION AUX ÉDITEURS

- ***TO COMPLY WITH UNIVERSAL POSTAL UNION REGULATIONS***
  - *A - PLEASE STICK THIS LABEL ON THE PARCEL,*
  - *B - AFFIX ON THE PARCEL THE GREEN CUSTOMS LABEL "C1" COMPLETED WITH THE NET VALUE (AS APPEARING ON YOUR INVOICE) AND THE DESCRIPTION OF CONTENT (EDUCATIONAL BOOK, ETC...) OR,*
  - *AFFIX OR ANNEX EITHER A COPY OF YOUR INVOICE OR A CUSTOMS DECLARATION "C2/CP3".*
  - *FAILURE TO COMPLY WITH THE ABOVE COULD RESULT IN THE CUSTOMS WITHHOLDING CONSIGNMENT.*

- **AFIN DE SATISFAIRE AUX EXIGENCES POSTALES ET DOUANIÈRES**
  - A - VEUILLEZ COLLER CETTE ÉTIQUETTE SUR LE COLIS,
  - B - VEUILLEZ FIXER SUR LE COLIS L'ÉTIQUETTE VERTE DE DOUANE "C1" AVEC LE PRIX NET (TEL QU'IL APPARAIT SUR VOTRE FACTURE) ET LA DESCRIPTION DU CONTENU (LIVRE D'ENSEIGNEMENT, ETC...) OU JOINDRE VOTRE FACTURE OU UNE DÉCLARATION C2/CP3.
  - LE DÉFAUT DE CETTE PROCÉDURE POURRAIT PROVOQUER UN BLOCAGE EN DOUANE.

SIMON FRASER UNIVERSITY

PF30022

BURNABY, B.C. V5A 1S6

VENDOR NO. 41650 PAGE

BATCH	INVOICE NO.	P/O NO.	INVOICE AMOUNT	BATCH	INVOICE NO.	P/O NO.	INVOICE AMOUNT
	REPORT		2.00				

received + sent report

SEP 28 1988

38832

REMITTANCE STATEMENT PLEASE DETACH

TOTAL 2.00

SIMON FRASER UNIVERSITY

SCHOOL OF COMPUTING SCIENCE
FACULTY OF APPLIED SCIENCES

BURNABY, BRITISH COLUMBIA V5A 1S6
Telephone: (604) 291-4277

AUGUST 25, 1988

UNIVERSITY OF WATERLOO
DEPT. OF COMPUTER SCIENCE
WATERLOO, ONTARIO
N2L 3G1

DEAR SIRS:

PLEASE FIND ENCLOSED MY CHEQUE ~~IN THE AMOUNT~~ OF $2.00.
PLEASE SEND ME TECH REPORT ~~#CS-88-01~~ COMPILING A DEFAULT
REASONING SYSTEM INTO ~~PROLOG BY~~ D. POOLE.

THANK YOU,

J. Delgrande /dlw

JIM DELGRANDE
ASSISTANT PROFESSOR

Printing Requisition / Graphic Services

15071

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number,** to **extension 3451.**

TITLE OR DESCRIPTION: CS-88-01

DATE REQUISITIONED: Aug. 16

DATE REQUIRED: ASAP

ACCOUNT NO.: 1 1 2 6 6 1 4 4 1

REQUISITIONER – PRINT

PHONE: 2192

SIGNING AUTHORITY: Sue DeAngelis

MAILING INFO – NAME: S. DEANGELIS

DEPT.: C. S.

BLDG. & ROOM NO.: DC 2314

☑ DELIVER ☐ PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 38

NUMBER OF COPIES: 20

TYPE OF PAPER STOCK: ☑ BOND ☐ NCR ____ PT. ☑ COVER ☐ BRISTOL ☑ SUPPLIED ☐ ____

PAPER SIZE: ☑ 8½ x 11 ☐ 8½ x 14 ☐ 11 x 17 ☐ ____

PAPER COLOUR: ☑ WHITE ☐ ____

INK: ☑ BLACK ☐ ____

PRINTING: ☐ 1 SIDE ____ PGS. ☑ 2 SIDES ____ PGS.

NUMBERING: FROM ____ TO ____

BINDING/FINISHING: ☑ COLLATING ☑ STAPLING 3 down left side ☐ ____ HOLE PUNCHED ☐ PLASTIC RING

FOLDING/PADDING

CUTTING SIZE

Special Instructions

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE: D 0 1

TYPESETTING — QUANTITY | LABOUR CODE: PAP 0 0 0 0 0 0 — T 0 1

PROOF — PRF — ____

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE: FLM — C 0 1

PMT — PMT — C 0 1

PLATES — PLT — P 0 1

STOCK — ____ — 0 0 1

BINDERY — RNG — B 0 1; MIS 0 0 0 0 0 0 — B 0 1

OUTSIDE SERVICES — $ ____ COST

TAXES – PROVINCIAL ☐ FEDERAL ☐

GRAPHIC SERV. OCT. 85 482-2

Printing Requisition / Graphic Services

1207

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number,** to **extension 3451.**

TITLE OR DESCRIPTION: CS-88-01

DATE REQUISITIONED: Jan. 6/88

DATE REQUIRED: ASAP.

ACCOUNT NO.: 1 2 6 2 6 0 1 4 1

REQUISITIONER - PRINT: S. DeAngelis

PHONE: 2192.

SIGNING AUTHORITY: Kym Hungerford

MAILING INFO – NAME: S. DeAngelis

DEPT.: CS

BLDG. & ROOM NO.: MC 6081 E

☑ DELIVER ☐ PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 38

NUMBER OF COPIES: 50

TYPE OF PAPER STOCK: ☑ BOND ☐ NCR ____ PT. ☑ COVER ☐ BRISTOL ☑ SUPPLIED ☐ ____

PAPER SIZE: ☑ 8½ x 11 ☐ 8½ x 14 ☐ 11 x 17 ☐ ____

PAPER COLOUR: ☑ WHITE ☐ ____ INK: ☑ BLACK ☐ ____

PRINTING: ☐ 1 SIDE ____ PGS. ☑ 2 SIDES ____ PGS.

NUMBERING: FROM ____ TO ____

BINDING/FINISHING: ☑ COLLATING ☑ STAPLING ☐ ____ HOLE PUNCHED ☐ PLASTIC RING

3 down left side

FOLDING/PADDING

CUTTING SIZE

Special Instructions

Math front + backs enclosed

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

D 0 1
D 0 1
D 0 1

TYPESETTING — QUANTITY

PAP 0 0 0 0 0 0 — T 0 1
PAP 0 0 0 0 0 0 — T 0 1
PAP 0 0 0 0 0 0 — T 0 1

PROOF

PRF
PRF
PRF

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

FLM — C 0 1
FLM — C 0 1
FLM — C 0 1
FLM — C 0 1
FLM — C 0 1

PMT

PMT — C 0 1
PMT — C 0 1
PMT — C 0 1

PLATES

PLT — P 0 1
PLT — P 0 1
PLT — P 0 1

STOCK

0 0 1
0 0 1
0 0 1
0 0 1

BINDERY

RNG — B 0 1
RNG — B 0 1
RNG — B 0 1
MIS 0 0 0 0 0 0 — B 0 1

OUTSIDE SERVICES

$ ____ COST

TAXES – PROVINCIAL ☐ FEDERAL ☐

GRAPHIC SERV. OCT. 85 482-2