VII SBIA, nov/1990, Campina Grande, PB

Classical Negation in Logic Programs

Luis Moniz Pereira, Luis Caires and José Alferes

AI Center, Uninova

jja@fctunl.rccn.pt

2825 Monte da Caparica, Portugal

Abstract: The purpose of this work is to extend logic programming beyond normal programs, as defined in [Lloy87], and thus beyond(positive) definite clause programming, by allowing also definite negative heads. Thus we admit program clauses with both positive and (classically) negated atoms conjoined in the body, and at most one literal as its head (clauses with disjunctions of literals in the head are transformed into a single clause in that form).
The proof system we propose (SLWV) can be seen as a combination of SL-resolution [Chan73] and case-analysis, that admits a form of linear derivation. We prove its soundness and completeness, give it an operational semantics by defining a standard derivation and produce an implementation.
As this approach does not require alternative clause contrapositives, it provides for better control over the search space, and maintains the execution order of literals in a clause, preserving the procedural flavor of logic programming. We provide a method of execution keeping to the implicational clausal form of program statements typical of Prolog (without the use of clause contrapositives), adding an increased expressiveness, but at a tolerable computational cost for regular Prolog programs. The implementation relies on the source program being preprocessed into directly executable Prolog. Since preprocessing only involves the addition of three additional variables to each predicate definition while keeping the overall program structure untouched, a directly recognizable execution pattern that mimics Prolog is obtained: this is useful in debugging.

Keywords: Logic Programming, Negation, Theorem Proving, Resolution

Introduction

The purpose of this work is to extend logic programming beyond normal programs, defined in [Lloy87], and thus beyond definite clause programming. We increase expressiveness, by allowing both positive and (classically) negated atoms conjoined in the body, e.g. `arrested(X) :(¬paytaxes(X),` where ¬ stands for the classical negation, clauses with disjunction of literals in the head, e.g. `sad(X); drunk(X) :- nomoney(X),` and negative heads, e.g. `¬ drunk( john ).`

Clauses with a disjunction of literals in the head are transformed into a single equivalent clause with just one (arbitrary) literal there. Thus the second clause above is transformed into, say, `sad(X) :- ¬ drunk(X), nomoney(X).`

Clauses with negative literals in the head are transformed into a contrapositive with ⊥ there (where the new symbol denotes falsehood. Accordingly, resolving a topgoal G is equivalent to resolving G/⊥). The last clause presented is transformed into, `⊥ :( drunk(john).`

In order to keep the procedural aspect of the language, it is also our aim to maintain the execution order of literals in a clause. So we can't make direct use of contrapositives. The proof system we propose can be seen as a combination of SL-resolution [Chan73] and case-analysis that admits a form of linear derivation. As an example let us consider an SL-resolution of the program:

```
(1) arrested(X) :- ¬ paytaxes(X).
(2) sad(X); drunk(X) :- nomoney(X).
(3) ¬ drunk( john ).                    (4) drunk(X) :- sad(X).
(5) nomoney(X) :- paytaxes(X).
```

and the query `?- arrested(john).`

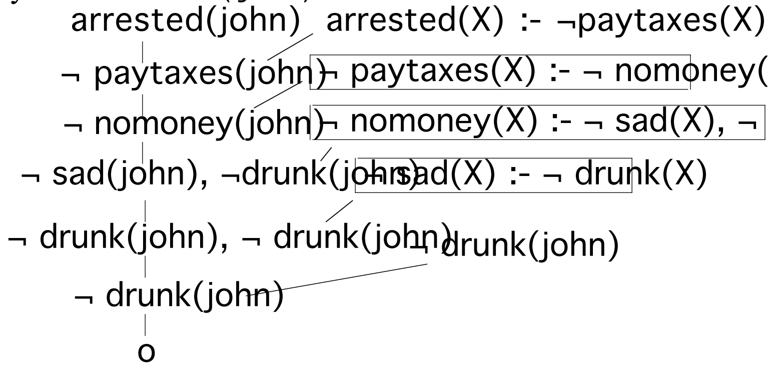
arrested(john)
arrested(X) :- ¬paytaxes(X)
¬ paytaxes(john)
¬ paytaxes(X) :- ¬ nomoney(
¬ nomoney(john)
¬ nomoney(X) :- ¬ sad(X), ¬
¬ sad(john), ¬drunk(john)
¬ sad(X) :- ¬ drunk(X)
¬ drunk(john), ¬ drunk(john)
¬ drunk(john)
¬ drunk(john)
o

fig. 1

Clauses inside boxes show where a contrapositive is used. In such cases, our method, instead of considering the contrapositives, tries alternative branches of the ancestors, in order to find the complementary literal and then resolve upon it, using a form of case-analysis. The graph bellow tries to illustrate in a simple way how our method proceeds, for the same example (arrows shows a possible execution trace).

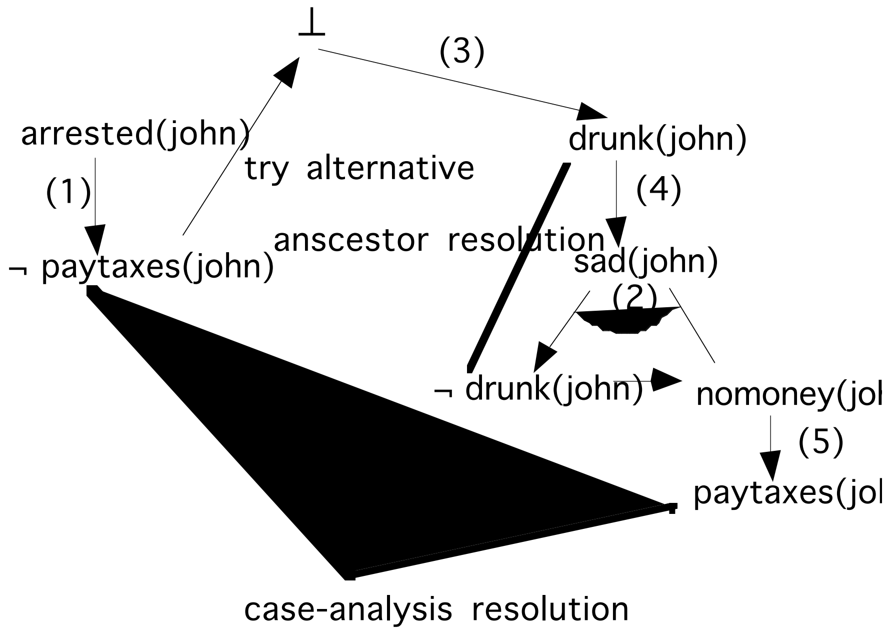

fig. 2

Here we can see that the case-analysis resolution corresponds to the use of a contrapositive[1].

The organization of this paper is as follows: in sections 1 to 4 we define the proof system used and prove its completeness and soundness, for normal programs. In section 5 we explore some connections with resolution proofs and give some examples. Section 6 explains how we extend the method for clauses with negative heads. In section 7 we

[1] The equivalence between our method and SL-resolution, specially in what concerns of using contrapositives, is discussed in section 5.1 herein.

describe a simple executor for the method. Finally in section 8, we draw comparisons with related work.

1. Language

The class of programs we consider are ordered finite sets of clauses of the implicative form $H \blacklozenge B_1, B_2, \_ B_n$ (or H ♦ **B** for short) where H is an atom and the B_j for all j are literals. n may be 0, in which case we write H ♦ _, where _ stands for an atom satisfied by all models. For a clause C H ♦ **B**, we write BodyC for **B**.

1.1 Definitions

Labels are finite sequences of literals. &-clauses are conjunctions of literals. L-Clauses are pairs L#G, where L is a label and G a &-clause. L-Resolvents are finite sequences of L-Clauses. _ is the empty L-Clause. The intended meaning of a L-Clause PN#F is the disjunction $F \vee A_1 \vee A_2 \vee \_ \vee A_n$, for $A_i \varepsilon$ PN. Thus F logically follows from ø#F.

Now, let Π be a program, PN a label, σ,τ substitutions.

2. Rules

These rules are the basic inferences of the system, and state the deducibility relation of formulae (which are L-Clauses).

2.1. _-Rules

$$\frac{}{\_}\ [R1] \qquad\qquad \frac{}{PN\#\_}\ [R2]$$

$$\frac{}{PN\#A} \quad \text{if } A\sigma \text{ is complement of } Ai\sigma \text{ for some } Ai \ \varepsilon \ PN \text{ and } \sigma \qquad [R3]^2$$

2.2. ♦ Rule

$$\frac{PN+<g> \# B\sigma}{PN \# g} \quad \text{for some } A \ \varepsilon \ PN+<g>^3, \text{ if } H \blacklozenge B \ \varepsilon \ \Pi \text{ and } H\sigma = A\sigma \qquad [R4]$$

Any such A verifying the above stated condition is called a **reducible** literal. The actual A used is said to be **reduced** by H ♦ B upon application of R4.

2.3. &-Rule

$$\frac{PN \# g PN \# G}{PN \# (g,G)} \qquad [R5]$$

3. Derivations and Proof Branches

As usual, we say that a literal F is deducible if there is a proof of ø#F using the above rules in sequence. Since there is a rule (R5) involving more that one premise, the proofs in this system are tree-shaped. However, we can devise a linear refutation scheme, by allowing

2 This rule can be seen as an instance of a Gentzen cut.

3 A+B is the concatenation of the labels A and B.

formulae to be sequences of L-Clauses (L-Resolvents). So, we start off with ø#F, and apply the rules backwards. Application of R4 then always introduces a non singular L-Resolvent. When applying any rule to the latest L-Resolvent in the sequence, we select a fixed L-Clause in it and proceed. In order to prove completeness, a notion of proof branch for F will be introduced as a (recursively construed) branch of a proof tree for F.

3.1 Definitions

Let S be an imposed total ordering over Π, and C a clause of Π.

|C| is the ordinal index of C in Π wrt S.

Let k be a natural number.

We define the k-shuffle S' of S to be the ordering over Π such that if S establishes $C_0 < C_1 < .. < C_k < .. < C_n$, then S' establishes $C_0 < C_1 < .. \quad < .. < C_n < C_k$. Note that if $C_i < C_j$ for i_k in S then $C_i < C_j$ in S', for j _ k, and $C_k = \max(\Pi)$ wrt S'.

Additionaly, the i-promoting of a label PN = $<A_1,A_2,\_,A_i,\_,A_n>$ is defined as the label PN* = $<A_i,A_1,\_,A_{i-1},A_{i+1}\_,A_n>$. (Shuffles and promotions will be used to insure fairness in reductions.)

Let CL(A) = { H ♦ B ε Π | Hσ=Aσ for some σ} be the clause set for a literal A. Note that if A is a negative literal, CL(A) is empty.

Let F_i be L-Clauses and S_i a ordering over Π.

A proof branch B for G wrt Π (or a G-PB) is a sequence F_1-S_1, F_2-S_2, .. F_n-S_n defined as follows (unless expressed otherwise, S_{n+1} is S_n):

PB1. F_1 is ø # G.

PB2. If F_n is _, then F_n is the last L-Clause of B.

PB2'. If F_n is PN # _ then F_{n+1} is _.

PB3. If F_n is PN # (g, G), then F_{n+1} is either PN # g or PN # G . (branching)

PB4. If F_n is PN # A , with A a literal, then

PB4.1 If Aσ is complement of AL_iσ for some AL_i ε PN and substitution σ,then F_{n+1} is _. Otherwise, let **CL** = CL(A) ∪ (**∪** CL(A_i) for all A_i ε PN).

PB4.1.1 **CL** is empty. Then F_n is last L-Clause of B.

PB4.1.2 **CL** is not empty. Let C be min(**CL**) wrt S_n, and AR the atom that introduced C in **CL**. Now, we consider two cases: If ARσ = Aσ, we set F_{n+1} as PN+<A> # BodyCσ. Otherwise ARσ = **A**σ, for some **A** ε PN=$<A_1,A_2,\_,A_k,\_,A_n>$. If there are several $\mathbf{A}_j$ in this condition, we let **A** be the rightmost such $\mathbf{A}_j$. Finally, let F_{n+1} be PN* # BodyCσ. In both cases, let S_{n+1} be the |C|-shuffle of S_n. So shuffles (and promotions) are only used in this subcase.

Notes:

(1) Proof branches are formed by applying the basic inference rules backwards. PB2, PB2' and PB4.1 relate respectively to [R1], [R2] and [R3]. PB3 relates to [R5] and PB4.1.2 relates to [R4].

(2) The notions of reducible and reduced literal stated in 2.2 also apply in the context of the PB4.1.2 step. Furthermore, we assume that substitutions are applied throughout the whole branch.

4. Completeness and Soundness

We now provide a soundness and completeness result for the above system.

The proof is inspired in [Shut77] and will proceed as follows: First we exhibit some properties of F-PBs. We prove afterwards that if every F-PB contains _, then F is deducible. Finally we build a model that satisfies $S = \Pi \cup \{\neg F\}$ from the hypothesis that there is a F-PB that does not contain the _: so if S is unsatisfiable, all F-PBs must contain _, and F is deducible.

Lemma 4.1: If every F-PB is finite, then there are finitely many F-PBs[4].

Proof: By König's lemma, a tree with a finite branching factor and finite length branches has a finite number of branches.o

Lemma 4.2: If every F-PB contains _, then F is deducible.

Proof: If every F-PB contains _, then every F-PB is finite. Let B be such a F-PB. Now, since by 4.1 there is only a finite number of F-PBs, let's make the following induction (on the length of PBs) : if F_n is _, then F_n is deducible by R1. Otherwise, by the definition of PB, F_n has the shape of a conclusion of some inference rule. Since the premises of this rule occur in B after F_n in F-PBs, the PBs for those premises have smaller lengths than any PB for F_n, so by induction hypothesis, the premises are deducible. This makes F_n also deducible. Since F is F_1, F is deducible.o

4.3 Definitions

A refutation for G from Π is a proof that every G-PB contains _. If there is a refutation of G from Π we say that G is Π-refutable, or simply refutable if Π is understood.

We now define negative and positive parts of L-clauses F_n.

Let F_n be PN # G. A literal **A** is a **positive** part of F_n if **A** ε $PN \cup \{G\}$. An atom is a **negative** part of F_n if $\neg A$ ε $PN \cup \{G\}$. Note that in any model **M**, the following holds:

(1) If A is a positive part of F_n, then **M** |= A implies **M** |= F_n.

(2) If A is a negative part of F_n, then **M** |_ A implies **M** |= F_n.

Note also that a positive literal may be both a negative and a positive part; not so for a negative literal, since ¬ does not occur embedded.

Let W be a set of formulae. HB(W) is the Herbrand base over W.

We now arrive to the main lemma:

Lemma 4.4: If there is a F-PB that does not contain _, then there is a model **M** over $HB(\Pi \cup F)$ such that **M** |= Π and **M** |_ F (eg. $\Pi \cup \{\neg F\}$ is satisfiable).

It's proof will follow from several sublemmas, stated and proved below.

Let B be such a F-PB (in the conditions of Lemma 4.4).

Lemma 4.4.1: If A is reducible in F_n by C, then B contains a reduction of A by C.

Proof: If A is reduced in F_n , then we have the result. Otherwise, A is also reducible in F_{n+1}, by definition of PB. Since B does not contain _, every step in B must be a reduction. After each reduction, an i-shuffle occurs, so the number of clauses "lesser" than C wrt the current S_m decreases. Moreover, each reduced literal is promoted, so B must contain a step m in which C = min(Π) wrt S_m, with A the rightmost C-reducible literal. So B contains a reduction of A by C.o

4 Let Π = { p ♦ q } and G = p. Then every G-PB has length ω : ø#p_{p}#q_{p}#q_{p}#q_

We now define positive and negative parts of B: A is a positive (negative) part of B if A is a positive (negative) part of some Fn of B.

Lets now consider the proof of Lemma 4.4. Let **M** be a model construed as follows. Let P = $p(x_1,x_2,\_,x_n)$ be a predicate letter. Then **M**(P) is the set of n-tuples $(t_1,t_2,\_t_n)$ from HB(ΠUF) such that $p(t_1,t_2,.\_t_n)$ is an instance of some negative part of B or does not occur at all either as a positive or a negative part.

Proposition 4.4.2: If a literal C is a positive part of B , then **M** |_ C.

Proof: Since B does not contain _, C is not both positive and negative, for otherwise a cancelation would occur by PB4.1. So, the claim follows by the definition of the model.o

Proposition 4.4.3: If C is a clause H ♦ **B** , then **M** |= C.

Proof : if C is ever applied in B, **B** _ _ for otherwise B would contain _ by PB2'. So **B** is $\{B_j\}$. But then both H and some B_j must occur as positive parts of B. Since A and B_j are literals, **M** |_ H and **M** |_ B_j so, **M** |= C. If C is never applicable, by Lemma 4.4.1 no instance of H occurs as a (reducible) positive part of B. Then **M** |= H, so **M** |= C.o

Since F occurs as a positive part of B, **M** |_ F. By Proposition 4.4.3 **M** |= Π. So Lemma 4.4 is proved.o

Completeness Theorem

If S is unsatisfiable then every F-PB contains _ by Lemma 4.4. By Lemma 4.2, F is deducible.o

Soundness Theorem

If F is deducible, then **M** |=F in any model **M** that satisfies Π.

Proof: Let **M** be a model such that **M** |= Π, P_i be premises for some inference rule and C its conclusion. We must show that if **M** |= P_i, then **M** |= C.

For R1 and R2, the result is immediate. For R3, note that PN # A is a tautology. For R4, suppose that **M** |= PNU{g}#Bσ. This means that **M** |= Bσ v g v ((**V**A_i) for A_i ε PN). Let A be as in 2.2. Since **M** |= Hσ ♦ Bσ, **M** |= Bσ implies **M** |= Aσ, so **M** |= g v ((**V**A_i) for A_i ε PN). If **M** |_ Bσ, then also **M** |= g v ((**V**Ai) for A_i ε PN). So **M** |= PN**#**g anyway.

Finally, for R5, suppose M |= PN#g and M |= PN#G. Then clearly **M** |= PN#(g,G), and we have the result.o

5. SLWV and Connections with Resolution Proofs

We now consider the linear refutation method over L-Resolvents suggested in 3 (assuming that the selected literal is the leftmost one). Let R be a L-Resolvent. Then, the following defines a (SLWV) linear derivation G for a literal G.

D1. F_1 is ø # G.

D2. If F_n is _, then F_n is the last L-Resolvent of G.

D2'. If F_n is PN # _ R then F_{n+1} is R.

D3. If F_n is PN # (g, G) R, then F_{n+1} is PN # g PN # G R.

D4. If F_n is PN # A R, with A a literal, then

D4.1 If Aσ is complement of AL_iσ for some AL_i ε PN and substitution σ,then F_{n+1} is R.

D4.1.1 **CL** is empty. Then F_n is last L-Resolvent of B.

D4.1.2 **CL** is not empty. This case is everywhere identical to PB4.1.2, with R included throughout, so it is omitted for brevity[5].

Example: For notational convenience, in the sequel we sometimes write (*x,y,z,_*)g for {*x,y,z,_*}#g. Let Π = { p ♦ a,b ; p ♦ ¬a,b ; p ♦ ¬b }. Then the following constitutes a **SLWV** refutation of p:

1.	()p		2.	(p)a(p)b	by D4.1.2
3.	(p,a)¬a(p,a)b(p)b	by D4.1.2	4.	(p,a)b(p)b	by D4.1
5.	(p,a,b)¬b(p)b	by D4.1.2	6.	(p)b	by D4.1
7.	(p,b)¬a(p,b)b	by D4.1.2	8.	(p,b,¬a)¬b(p,b)b	by D4.1.2
9.	(p,b)¬b	by D4.1	10.	Δ	by D4.1

5.1 Mapping SLWV derivations to SL ones

Let α ∅ β denote a valid SL-resolution step between consecutive resolvents α and β and ∅* its transitive closure.

Proposition: Let a ∅* b, **R** be a SL-derivation[6] **D** of b,**R** from a. Then there is a SL-derivation **D'** of ¬a,**R** from ¬b (named its inversion).

Proof: (by induction on the number of steps of the derivation **D**) Induction base: suppose that **D** is a ∅ b,**R**. Then a was resolved upon by a program clause a ♦ b,**R**. Consider the variant ¬b ♦ ¬a,**R**. Forthwith, **D'** is obtained. Otherwise, suppose that a ∅* b,**R** in more than one step. Look for the step of **D** in which b was introduced, say, by a clause *C* of the general form c ♦ **H**,b,**B** (c is then the immediate ancestor of b). So **D** has the form a ∅* c, **F** ∅ **H**,b,**B**,**F** ∅* b,**R** (where **R** = **B**,**F**). We have two cases: either **H** is void or not.

In the first case, *C* is c ♦ b,**B**. By induction hypothesis, the inversion **D*** ¬c ∅* ¬a, **F** of a ∅* c, **F** exists. Consider the clause variant ¬b ♦¬c,**B** (clause inversion step). Then **D'** is ¬b ∅ ¬c,**B** ∅* ¬a,**F**,**B**.

If **H** is non void, notice that **H**,b,**B**,**F** ∅* b,**R** contains a refutation **H** ∅* Δ of **H**. Now, he have two subcases: either this refutation contains ancestor cancelation steps or not so. In the latter case, **D'** is ¬b ∅ **H**,¬c,**B** $\underline{∅*}$ ¬c,**B**∅* ¬a,**F**,**B**, where the underlined steps are the refutation of **H** in **D**. Otherwise, suppose the subderivation **H** ∅* Δ uses ancestor cancelations (in this case, due to the derivation chain inversion, the ancestors used will no longer be available). However, the particular ancestor used in such a step must be some literal resolved with a program clause in the segment a ∅* c,**F** : this literal will appear complemented in some resolvent of the inversion ¬c ∅* ¬a,**F**, at some clause inversion step. So, when including the subderivation **H** ∅* Δ in **D'** we must, so to speak, defer those

5 **Remark:** It is interesting to argue soundness of the inference system by justifying its rules in terms of valid resolution steps. For application of rules R1,R2 and R5 soundness is easily recognized.For R4, if g is the selected literal, we have an instance of a resolution step with a side clause. Otherwise, suppose some A_i ε PN is selected for resolving upon. This A_i is clearly an ancestor for g. So, application of R4 results in suspending the current resolvent (by introducing g in PN) , followed by the choice of another candidate side clause for A_i. For R3, let **G** be the A_i mentioned in 2.1 above. Such **G** is either (i) an ancestor of A or (ii) not so. (i) if **G** is an ancestor of A, R3 merely rewords the ancestor cancelation step available in SL-Resolution. (ii) if **G** is not an ancestor of A, let H♦{R_i}, **G**, {R_j} be the clause that introduced **G** into PN. Now, as suggested above, we can envisage **G** as a place holder for the resolvent **C** = **G**, {R_j}, and the R3 step as the resolution on A of Fn=PN#A R with **C**, since {R_j} was left in R by the R4 step that introduced **G** in PN.o

6 We assume that a leftmost selection function is used. However, the argument could be easily extended.

ancestor cancelation steps by reordering the resolvent, so as to later eliminate them by a simple merge operation immediately after the step where the (now complemented) ancestor is introduced.o

Example

Let Π = { p ♦ a,b ; a ♦ c,d ; c ♦ ¬a }.

Let **D** be p ∅* d,b be p ∅ a,b ∅ c,d,b ∅ ¬a,d,b $\underline{\varnothing}$ d,b. Note the underlined step; ¬a canceled with its ancestor a. Now **D'** is ¬d ∅ c,¬a ∅ ¬a,¬a $\underline{\varnothing}$ ¬p,b. The underlined step contains an implicit merge operation; in this example no deferring was needed, because the matching ancestor was already present.

A SL derivation can be obtained for every (left) SLWV derivation by removing every cancelation performed between existing disjunctive branches[7]. Such cancelations always occur (cf. fig. 1).

The basic translation step

For every cancelation in the stated conditions, consider the proof obtained by inserting the inverted derivation below the step where the cancelation actually occurs (cf. fig. 2).

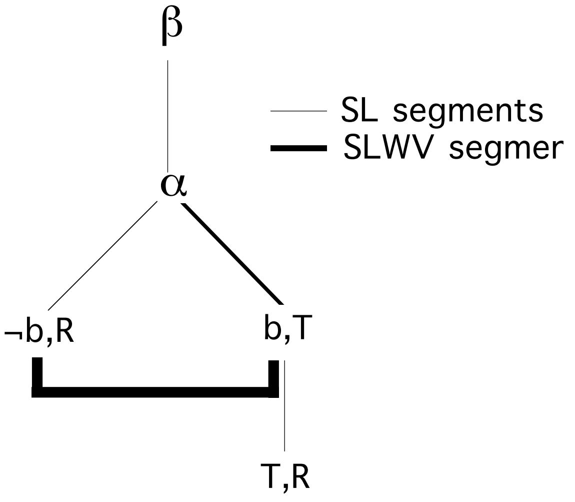

fig 3

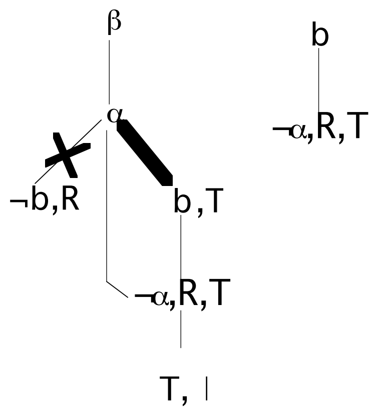

fig 4

Using a left selection function[8] , β ∅* α ∅* ¬b,R is an SL derivation. By the proposition above, the inverse derivation D* b ∅* ¬α, R exists, which in fig 2 has been used to reduce b in the right branch of the disjunct. Note that D* can still use cancelation with ancestors in β ∅* α freely, since this segment, while possibly taking part in α ∅* ¬b,R, is not affected by the inversion. The resulting proof will then become β ∅* α ∅* b,T ∅* ¬α,R,T $\underline{\varnothing}$ T,R, where the underlined step is an ancestor cancelation upon α. If there are no more cancelations involving the inverted branch α ∅* ¬b,R (crossed over in the figure), it can be removed (cf. footnote, previous page).

To translate some SLWV proof to a SL one, we iteratively apply the simple translation step described above. After every simple translation step, the original proof

7 Branches neither ending in Δ nor taking part in some cancellations are irrelevant to a particular proof.

8 As the actual implementation does.

contains one less "cross" cancelation between disjunctive branches so, when the process terminates, we have a pure SL derivation.

Example

Let Π = { p ♦ a,b ; p ♦ ¬a,b ; p ♦ ¬b }.
We have the following SLWV derivation tree for p (with three cancelations) in fig. 3.

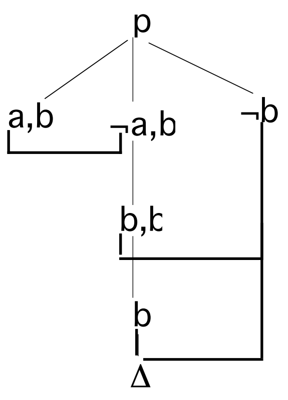

fig 5

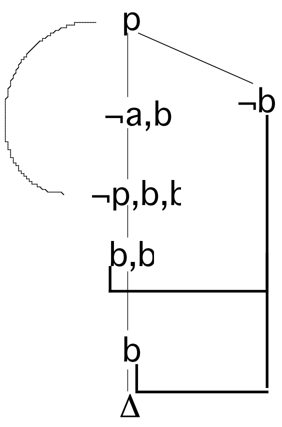

fig 6

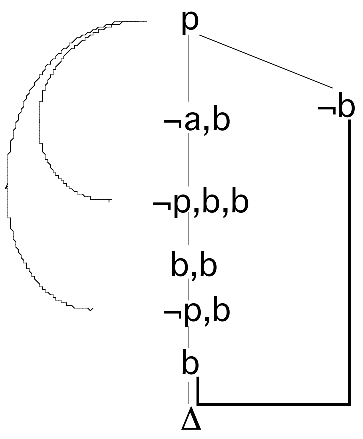

fig 7

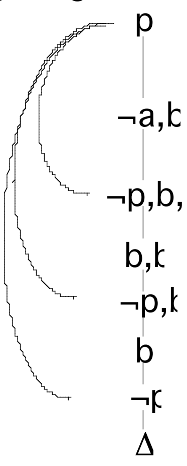

fig 8

Note that p ∅ a,b is pure SL and takes part in a cancelation (on a). After a simple translation operation (with inversion ¬a ∅¬p,b) we get the proof of fig 4.

Since there are no more cancelations involving this branch, we remove it from the proof. Next, we consider first the innermost cancelation on b (note that the branch up to the outermost cancelation is not pure SL). Using the inversion b ♦¬p, we arrive at the proof displayed in fig 5. Using the same procedure, the proof of fig 6 is obtained. This final derivation is a pure SL proof, as claimed above.

6 Defining negative definite program clauses

Up to now we have considered only program clauses with positive head literals. However, a clause ¬H♦ **B** can be soundly rewritten as ⊥♦H, **B** where ⊥ is a new logical constant standing for 'false'. This suggests the translation of every negative definite clause in a corresponding clause for ⊥. Now Π |- g iff |- Π -> g iff |- (Π & ¬ g)∅ ⊥ .

Thus Π |- g iff |- Π ∪ {⊥♦ g}∅⊥, so Π |- g iff Π ∪ {⊥♦ g} |- ⊥. Now, a proof of ⊥ in this setting has as first two lines

1. ⊥
2. {⊥} g

Thus, whenever programs with negative definite clauses are defined, in establishing a derivation we can start as usual from the relevant literal to be proved (such literal can now freely be either positive or negative), but considering ⊥ as a "root" ancestor.

Note however that a tradeoff exists when this approach is applied for the leftmost/ancestor-order. For when a clause ¬p ♦ **B** is written, it will be not used for reducing ¬p until the search space related to all ancestors below ⊥ is exhausted.

Nevertheless, consider the following L-Resolvent { ⊥, $a_1,a_2,\_a_k$ } # ¬p, and suppose the original clause set contained some clauses of the form ¬p ♦B_i. Those clauses have been rewritten as $\perp\blacklozenge p, B_i$. However, since we are reducing the outer literal of the L-Resolvent, we can safely apply the original variant $\neg p \blacklozenge \mathbf{B}$ to obtain $\{ \perp, a_1,a_2,\_a_k ,\neg p\} \# \mathbf{B}$. Note that this procedure is actually a shortcut for the (permissible) derivation :

$(\perp, a_1,a_2,\_a_k)\neg p$

$(\perp, a_1,a_2,\_a_k,\neg p) (p,\mathbf{B})$ (reducing $\perp$)

$(\perp, a1,a2,\_ak,\neg p)p(\perp, a1,a2,\_ak,\neg p)\mathbf{B}$ (&-Rule)

$(\perp, a_1,a_2,\_a_k,\neg p)\ \mathbf{B}$ (cancelation)

Additionally, since we are now considering reductions with clauses for $\neg p$ (a certain subset of the clauses for $\perp$), the $\neg p$ introduced in $\{\perp, a_1,a_2,\_a_k ,\neg p\} \# \mathbf{B}$ can be envisaged as a $\perp_{\neg p}$ (e.g, as if clauses for $\perp$ of the form $\perp\blacklozenge p ,B_i$ were "marked" $\perp_{\neg p}\blacklozenge p ,B_i$) in such a way that alternative clauses for $\neg p$ obtained by the ancestor-order strategy will be used before any other clauses for $\perp$[9].

7. On the implementation of SLWV

Next we present a simple executor for the method above, relying on a preprocessing of clauses and on the addition of a few specific predicates written in Prolog[10]. We begin with a description of the external syntax of programs. Afterwards we describe the preprocessor and internal syntax. Then we present two new mechanisms to execute preprocessed programs, cancel and climb, their implementation in Prolog, and give some ideas on how the executor can be thought in terms of a low level implementation, comparing the two new mechanisms with existing ones in Prolog. Finally we explain how to avoid duplicate solutions.

7.1 External syntax

A program is a set of clauses of the form *head :- body* where *head* is a literal or a disjunction of literals separated by ';' and *body* is a literal or a conjunction of literals. *head :- true* can also be represented as *head* .

This syntax extends Horn clauses to allow negative literals anywhere and disjunction in the head, and subsumes normal programs.

7.2 Preprocessing

In a preliminary phase, the preprocessor transforms program clauses into the language described in section 1. A clause with more than one literal in the head is transformed into a single equivalent clause with just one (arbitrary) literal there. (No contrapositives are generated or used.)

$L_1 ;L_2; ...; L_n$:- Body is transformed into, say, L_1 :- $\neg L_2, ..., \neg L_n$, Body

After this, and in order to have a single positive literal in the head each clause $\neg$ L :- Body is transformed into **false** :- L, Body. Introducing a topgoal :- G is equivalent to introducing the clause G / **false**, as seen in section 6 above.

9 Eg. $\{ \perp, a_1,a_2,\_a_k \} \# \neg p \quad \varnothing \{ \perp_p, a_1,a_2,\_a_k,\perp_{\neg p} \} \# \mathbf{B}.$

10 The complete code is available on request.

According to the method, calls for a goal must be associated with a label set and a clause ordering. As no clauses have a negative head, for greater efficiency in applying clauses by rule **D**4.1.2, and also for faster application of **D**4.1, we represent the label set PN by two lists, P and N: P with the positive and N with the negative literals. In each goal, additionally to P and N, we introduce a further parameter C, specifying the clauses it can use. When C is unbound, every clause for it is to be considered.

Consequently, each clause is augmented by the preprocessor with the three argument variables P, N and C. Hence, each clause with p(**X**) as head

p(**X**) :- b_p(**Y**), ¬ b_n(**Z**), _ is transformed into

```
p(X,P,N,index):- bp(Y,[x-p(X)|P],N,_), $neg( bn(Z,[x-p(X)|P],N,_)),_
```

where `index` is the number of this clause, starting with 0, in the sequence of clauses for p(**X**).

In all goals, `x-p(X)` is fronted to `p`: the tag x is used to state that literal `p` has been reduced. Suspended literals[11] contain a free variable parameter instead; this parameter is used to check for reduction by cancelation of goals with complementary ones either in P or N (cf. 7.3 below), by binding it to x when cancelation occurs.

`$neg` is a specific predicate enveloping the atom of each negated goal, to be explained later.

In order to enable access to the underlying Prolog in our programs, literals of the form `call(Goal)` are preprocessed simply to `Goal`.

The splitting of proof branches (**D**3) is guaranteed by Prolog's execution of the conjunction.

7.3 The cancel mechanism

To insure application of rule **D**4.1, when trying to solve a goal one has, first of all, to check for a cancelation in PN. This mechanism can be seen as a match of a suspended literal with the head of the clause contrapositive with the negation of the goal at the head (cf. section 5.1 above).

So for each predicate p(**Y**), the preprocessor adds, as a first clause for p:

```
p(Y,_,N,_) :- member(x-p(Y),N).                        [CR]
```

where the symbol `x` indicates, by binding within the list element, that a cancelation has occurred.

For negative goals we add, as first clause for **$neg**, a clause similar to [CR].

7.3 The climb mechanism

When solving a goal p(**Y**), after trying all clauses with this literal at the head, to insure the complete application of rule **D**4.1.2, the executor must try alternative clauses for the ancestors, in order to find an alternative disjunctive branch containing a literal ¬p(**Z**). We call this the climb mechanism. To invoke this mechanism, the preprocessor adds, as the last clause for every p(**X**):

```
p(X,P,N,_) :- climb(P,N,p(X)).                    [CC]
```

[11] By a suspended literal we mean a non-ancestor literal that was introduced in the label set, according to **D**4.1.2.

where **climb** is a specific predicate, that in turn chooses an element of the P label set and reconsiders it (as we don't have preprocessed negated heads) on its alternative clauses.

```
climb(P,N,G) :-
     choose_one(P,AnsPred,ClauseNumber,NewP),
     translate(AnsPred,[X-G|NewP],N,ClauseNumber,AnsGoal),
     AnsGoal,
     X == x.
```

To insure that in the present derivation the original goal actually cancels, as explained before in 7.2, a check for the binding of `X` with `x` is performed.

translate is a fact predicate with an instance introduced for each program predicate by the preprocessor. It transforms elements of P or N into the form of preprocessed predicate calls (i.e. p(**X**,P,N,C)) given P,N, and C, and vice-versa. p(**X**) introduces:

```
translate(p(X), P, N, C, p(X,P,N,C)) :- !.
```

As this problem is the same for negative literals, we have as last clause for **$neg**, similar to [CC].

The predicate `choose_one(P,AnsPred,ClauseNumber,NewP)` chooses from `P` the ancestor alternative clause with `AnsPred` at the head and number `ClauseNumber`, and builds for it the new `P` list `NewP`. Here various search strategies can apply. The simplest one seams to choose the next clause for the closest ancestor, and then the others by backtracking in this order. A possible alternative is to choose first clauses that lead directly to the complementary literal. This alternative is much more efficient, but has to keep information about the call graph, which may be worth considering if a low level implementation is produced.

In fact, we can think about this mechanism as an elaboration of the standard Prolog backtracking that keeps some information about the failed branch. This information is mainly about variable bindings and failed goals (corresponding here to suspended literals). This fact, supported with the similarity between the cancel mechanism and the matching of goals with clause heads, suggests a low level implementation (eventually based on some modification in a virtual Prolog machine).

7.4 Avoiding duplicate solutions

The climb mechanism, conjoint with backtracking, introduces the problem of undesirable repeated solutions. For example, consider the following program:

```
(1) p :- q.                              (2) p :- ¬q.
```

with the top goal `?- p`. In the first solution p calls q using (1), q invokes the climb mechanism and succeeds by cancelling with ¬q of (2). By backtracking p calls ¬q using (2), q invokes the climb mechanism and succeeds by cancelling with q of (1), reaching this way the second solution. In fact this solution is exactly the one reached at the beginning.

Another way to see the problems, is to say that it happens because both, negative and positive literals, can invoke the climb mechanism. In fact if only one type of literals could invoke the climb, no such problem could arise. But if we only allow climbing on, say, positive literals there is no chance for these literals to cancel, because there are never negative suspended literals. So the type of literals that can invoke the climb mechanism has no need for the cancel mechanism. We can be more specific by saying that for each predicate name either there exists the climb mechanism for positive literals and the cancel mechanism for negative ones or vice-versa. For each one, according to a static analysis of the problem, we can choose the alternative that seams to be the most efficient.

With a low level implementation this problem disappears, because there is only one mechanism that subsumes climb and backtracking.

8. Comparison with related work

Horn clause programs under this executor, don't make calls to specific predicates, so their run time is comparable to that under normal Prolog.

Comparing this method with the ones presented in [Love87,Stic85,Stic86], we find that it has the advantage of not needing syntactic variants, namely contrapositives.

Another method for Logic Programming with negation that doesn't make use of contrapositives is that of [Plai88]. This method builds for each problem a deduction system where the rules and axioms depend on the clause set. This approach seems to be more complex than ours and has an execution trace very different from Prolog. It presents a significative overhead in the execution of definite clause programs.

The method described in [Mant88], has the advantage of being both very simple and efficient (at least for a certain class of programs). Although, as it is based on model elimination, the user doesn't have control over the execution order, being, this way very difficult to use it as a logic programming language.

For its simplicity and Prolog-like strategy, our method turns out to be more attractive than the above ones, given also that it is amenable to a simple preprocessing, not incurring in the overhead of an interpreter. Thus it is well suited to extend Prolog programming with negation.

Acknowledgements

We thank ESPRIT Basic Research Project COMPULOG (no. 3012), Instituto Nacional de Investigação Científica, and Gabinete de Filosofia do Conhecimento for their support.

References

[Chan73] Chang C. and R. Lee.: **Symbolic Logic and Mechanical** *Theorem Proving* ,Academic Press, New York, 1973.

[Eshg89] Eshghi, K. and R. Kowalski.: Abduction Compared with Negation as Failure, **Logic Programming: Proceedings of the Sixth International Conference,** (Levi and Martelli eds.), MIT Press, 1989.

[Lloy87] Lloyd, J.: **Foundations of Logic Programming** , second edition, Springer-Verlag, 1987.

[Love87] Loveland, D.W.: Near Horn Prolog. In J. Lassez, editor,**Logic Programming: Proceedings of the Fourth International Conference,** pages 456-469, The MIT Press, 1987.

[Mant88] Manthey, R. and Bry: F. SATCHMO: a theorem prover implemented in Prolog. In **Proceedings of CADE 88 (9th Conference on Automated Deduction),** Argonne, Illinois, 23-26 May 88, LNCS, Springer Verlag.

[Plai88] Plaisted, D. A.: Non-horn clause logic Programing without Contrapositives. **Journal of Automated Reasoning** 4 (1988) pages 287-325.

[Shut77] Shutte, K.: **Proof Theory**. Springer-Verlag, 1977.

[Stic85] Stickel, M.E. et al: An Analysis of Consecutively Bounded Depth-First Search with Applications in Automated Deduction, **Proceedings of the Ninth International Joint Conference of Artificial Intelligence**, Los Angeles California, August 85.

[Stic86] Stickel, M.E.: A Prolog Technology Theorem Prover: Implementation by an extended Prolog Compiler, **Proceedings of the Eigth International Conference in Automated Deduction**, Oxford, England, July 86.