SLWV - A Logic Programing Theorem Prover

Luis Moniz Pereira[†] , Luis Caires[¥] and José Alferes[*]

AI Centre, Uninova
2825 Monte da Caparica, Portugal

Abstract: The purpose of this work is to define a theorem prover that retains the procedural aspects of logic programing. The proof system we propose (SLWV[1] resolution) is defined for a set of clauses in the implicational form (keeping to the form of logic programs), not requiring contrapositives, and has an execution method that respects the execution order of literals in a clause, preserving the procedural flavor of logic programming.
SLWV resolution 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.
Our work can be seen as an extension to logic programs that goes 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 of that form).
As this approach does not require alternative clause contrapositives, it provides for better control over the search space. 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 can be useful in debugging.

Keywords: Logic Programming, Negation, Resolution, Theorem Proving

Introduction

The purpose of this work is to define a theorem prover that retains the procedural aspects of logic programing. In order to keep to the clausal form of logic programs, the proof system we define (SLWV resolution) applies, without loss of generality, to sets of clauses in the implicational form, though not requiring any contrapositive variants. Thus the form of clauses we consider is:

$H <- L_1, \ldots, L_n.$

where $n \geq 0$ and $H, L_1, \ldots, L_n$ are literals. As usual a literal is an atom A or its (classical) negation $\neg A$.

Clauses with negative literals in the head are transformed into a single contrapositive clause with $\perp$ there, where the new symbol denotes falsehood. Accordingly, refuting a top goal G is equivalent to adding the clause $\neg G/\perp$ and finding a derivation for $\perp$.

In order to preserve the procedural aspect of the language, it is our aim to maintain the execution order of literals in a clause. So we can't use contrapositives. The proof system we propose can be seen as a combination of SL-resolution [Chan73] and

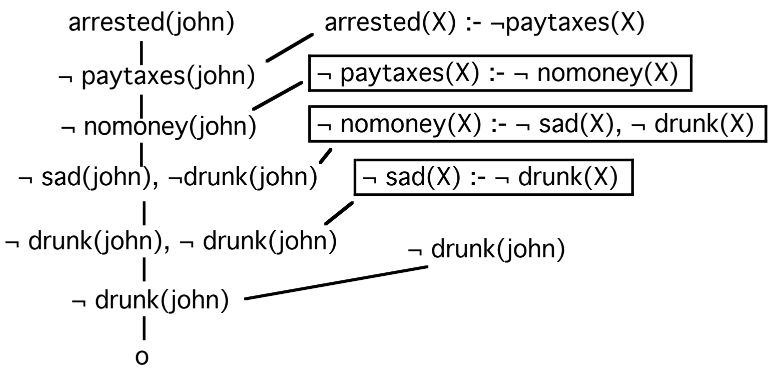
† Fax: +351-1-2955641 Phone: +351-1-2953156 E-mail: lmp@fct.unl.pt
¥ Fax: +351-1-2955641 Phone: +351-1-2953156 E-mail: lxc@fct.unl.pt
* Fax: +351-1-2955641 Phone: +351-1-2953156 E-mail: jja@fct.unl.pt

1 **S**elected **L**inear **W**ithout contrapositive clause **V**ariants.

case-analysis that admits a form of linear derivation. As an example let us consider a procedural-like representation of a SL-resolution for program:

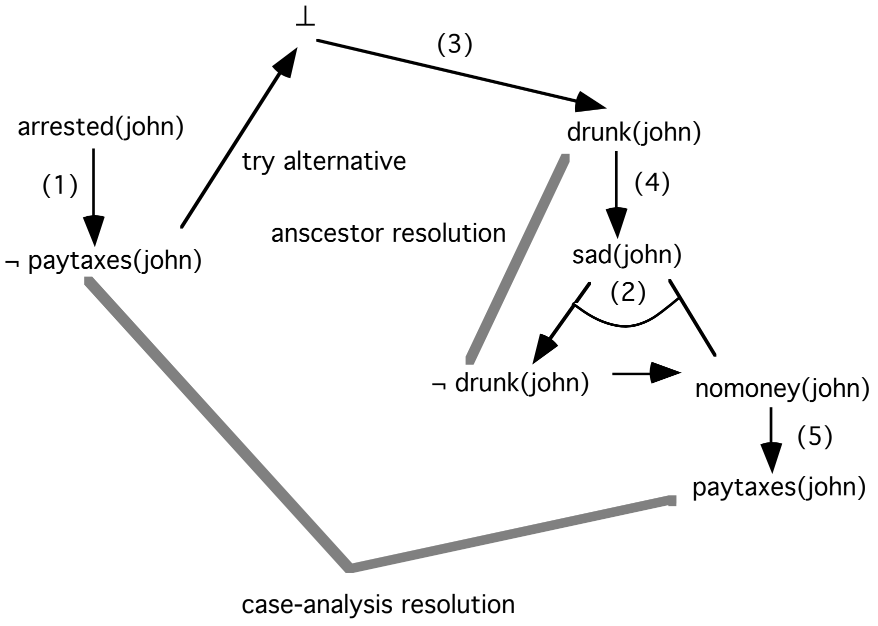
```
(1) arrested(X) :- ¬ paytaxes(X).
(2) sad(X):- ¬ drunk(X), nomoney(X).
(3) ¬ drunk( john ).                (4) drunk(X) :- sad(X).
(5) nomoney(X) :- paytaxes(X).
```

and query `?- arrested(john).`



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

Clauses inside boxes show where a contrapositive must be used to resolve a literal. In such cases, our method, instead of relying on contrapositives, tries alternative branches of the ancestors of the literal, in order to find a complementary literal, and then resolving both, using case-analysis. The graph below illustrates in a simple way how our method proceeds, in this example (arrows show a possible execution trace).



Here we can see that the case-analysis resolution corresponds to the use of a contrapositive[2].

2 The equivalence between our method and SL-resolution, specially in what concerns the use of contrapositives, is discussed in section 5 herein.

The organization of this paper is as follows: in sections 1 to 3 we define the proof system used for normal programs. In section 4 we examine the connections with resolution proofs, prove its equivalence to SL-resolution (thus proving it is sound and complete) and give some examples. Section 5 explains how we extend the method for clauses with negative heads. In Section 6 we define a new kind of answers (disjunctive answers) and explain how a theorem prover that keeps the procedural aspect of logic programs can handle this problem easily. Section 7 describes a simple executor for the method. Finally, in section 8, we draw some conclusions.

1. Language

The class of programs we consider initiallyare ordered finite sets of clauses of the implicative form $H♦B_1, B_2,\ldots B_n$ (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. Δ is the empty L-Clause. The intended meaning of a L-Clause PN#F is the disjunction $F \vee A_1 \vee A_2 \vee\ldots\vee A_n$, for A_i ε 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. Δ-Rules

Error! [R1] **Error!** [R2]

Error! if Aσ is complement of Aiσ for some Ai ε PN and σ [R3][3]

2.2. ♦Rule

Error! for some A ε PN+<g>[4], if H ♦ B ε Π and Hσ = Aσ [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.

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

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

2.3. &-Rule

Error! [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 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 (or F-PB) 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 $C_0 < C_1 < .. \; < .. < C_n < C_k$. Note that if $C_i < C_j$ for $i \neq k$ in S then $C_i < C_j$ in S', for $j \neq k$, and $C_k = \max(\Pi)$ wrt S'.
Additionaly, the i-promoting of a label PN = $<A_1,A_2,\ldots,A_i,\ldots,A_n>$ is defined as the label PN* = $<A_i,A_1,\ldots,A_{i-1},A_{i+1}\ldots,A_n>$. (Shuffles and promotions will be used solely 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 Δ, then F_n is the last L-Clause of B.
PB2'. If F_n is PN # ∂ then F_{n+1} is Δ.
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 Δ. Otherwise, let **CL** = CL(A) U (**U** 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.

Example: Let the program **Π** = { 1:p ♦ a,b ; 2:p ♦ ¬a ;3:b }. The proof branches for goal p in that program are:

	S_n	
{} # p	[1,2,3]	by PB1
{p} # (a,b)	[2,3,1]	by PB4.1.2

now, splitting of branches occurs (by PB3). For the left one, we have

{p} # a	[2,3,1]	
{p,a} # ¬a	[3,2,1]	by PB4.1.2
Δ	[3,2,1]	by PB4.1

by PB2 this is the last L-clause of this branch. The right branch will produce.

{p} # b	[2,3,1]	
{p,b} # ∂	[2,1,3]	by PB4.1.2
Δ	[2,1,3]	by PB2'

again by PB2 this is the last L-clause of this branch.❏

In [Pere90] we provided a soundness and completeness result for the above system. The proof is inspired in [Shut77] and proceeds as follows: After exhibiting some properties of F-PBs, we prove that if every F-PB contains Δ, then F is deducible; finally, we build a model that satisfies S = ΠU{¬F} from the hypothesis that there is a F-PB that does not contain Δ: so if S is unsatisfiable, all F-PBs must contain Δ, and F is deducible.

Completeness Theorem [Pere90]: If S is unsatisfiable then every F-PB contains Δ and F is deducible.❏

Soundness Theorem [Pere90]: If F is deducible, then **M** |=F in any model **M** that satisfies Π.❏

4. SLWV Connection with Resolution Proofs

In this section we begin by defining a SLWV linear derivation. Then we show how this derivation can be mapped into SL ones and vice versa.

4.1. SLWV Linear Derivation

Definition: A refutation for G from Π is a proof that every G-PB contains Δ. 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 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 Δ, 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 $A\sigma$ is complement of $AL_i\sigma$ 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: Let us consider now an example similar to the one given in the introduction. For brevity we use short predicate names.
Π = {arr(X)♦ ¬pay(X) ; sad(X)♦ nom(X) ; arr(X)♦ dr(j); dr(X)♦ sad(X); nom(X) ♦ pay(X)}.

The SLWV refutation of arr(X) is:

{}#arr(X) by D1
{arr(X)}# ¬pay(X) by D4.1.2

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.❏

{arr(j), ¬pay(j)}# dr(j)	by D4.1.2
{arr(j), ¬pay(j), dr(j)}# sad(j)	by D4.1.2
{arr(j),¬pay(j),dr(j),sad(j)}# (¬dr(j), nom(j))	by D4.1.2
{arr(j),¬pay(j),dr(j),sad(j)}# ¬dr(j) {arr(j),¬pay(j),dr(j),sad(j)}# nom(j)	by D3
{arr(j), ¬pay(j), dr(j), sad(j), nom(j)}# pay(j)	by D4.1
Δ	by D4.1

Example: Let Π = { 1: p(0) ♦ ;2:r(x) ♦ p(x), 3:r(y) ♦ ¬p(f(x)),p(x) }. Note that the third clause has a kind of recursion in its body. In fact this clause is equivalente to p(f(x)) ♦ ¬r(Y),p(X). The first version of the clause turns things more unreadble. Nevertheless one could write it like that.

The SLWV refutation for r(f(f(0))) is:

	S_n	
{} # r(f(f(0)))	[1,2,3]	by D1
{r(f(f(0)))} # p(f(f(0)))	[1,3,2]	by D4.1.2
{r(f(f(0))),p(f(f(0)))} # ¬p(f(X)) {r(f(f(0))),p(f(f(0)))} #p(X)	[1,2,3]	by D4.1.2 and D3
{r(f(f(0))),p(f(f(0)))} # p(f(0))	[1,2,3]	by D4.1

[here some steps are omitted for brevity]

{r(f(f(0))), p(f(f(0))), p(f(0))} # p(0)	[1,3,2]	
{r(f(f(0))), p(f(f(0))), p(f(0)), p(0)} # ∂	[1,3,2]	by D4.1.2
Δ	[1,3,2]	by D2'

4.2. 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.

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

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** $\underline{\varnothing^*}$ ¬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 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.❑

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).

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

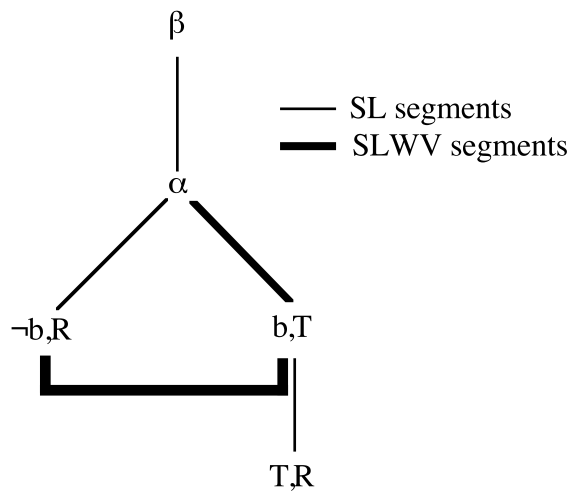


fig 1

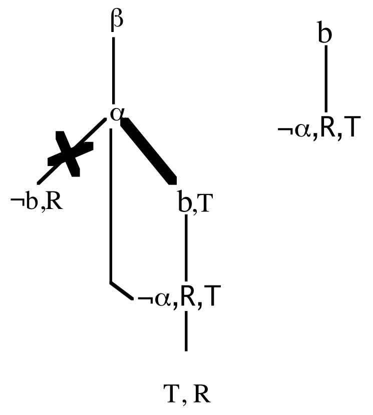


fig 2

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 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.

8 As the actual implementation does.

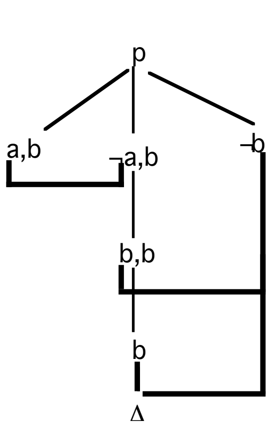


fig 3

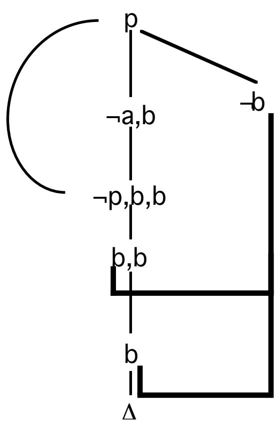


fig 4

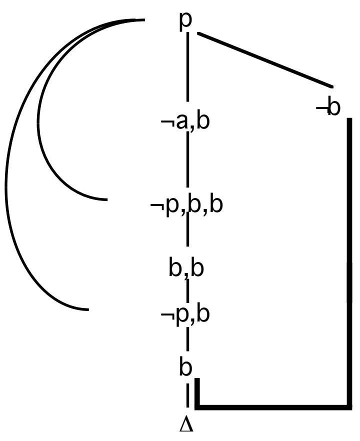


fig 5

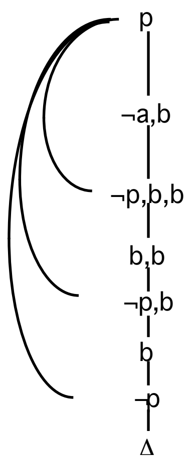


fig 6

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.

4.3. Mapping SL derivations to SLWV ones

5. 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 U {⊥♦ g}∅⊥, so Π |- g iff Π U {⊥♦ 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 $\neg p \blacklozenge \mathbf{B}$ is written, it will be not used for reducing $\neg p$ until the search space related to all ancestors below $\perp$ is exhausted.

Nevertheless, consider the following L-Resolvent $\{ \perp, a_1,a_2,\ldots a_k \} \# \neg p$, and suppose the original clause set contained some clauses of the form $\neg p \blacklozenge B_i$. 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 $\{ \perp, a_1,a_2,\ldots a_k ,\neg p\} \# \mathbf{B}$. Note that this procedure is actually a shortcut for the (permissible) derivation :

$(\perp, a_1,a_2,\ldots a_k)\neg p$
$(\perp, a_1,a_2,\ldots a_k,\neg p) (p,\mathbf{B})$ (reducing $\perp$)
$(\perp, a1,a2,\ldots ak,\neg p)p(\perp, a1,a2,\ldots ak,\neg p)\mathbf{B}$ (&-Rule)
$(\perp, a_1,a_2,\ldots a_k,\neg p)\ \mathbf{B}$ (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 $\{\perp, a_1,a_2,\ldots a_k ,\neg p\} \# \mathbf{B}$ 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].

6. Disjunctive answers

The intuitive desired meaning of a disjunctive answer is readily shown with an example. Suppose we have the program:
Π = { nomoney(john) ♦ ¬nomoney(mary), nopay(father); nopay(father) },

meaning that john or mary (or both) have no money if their father is not paid, and their father is not paid. The query ♦nomoney(X), intends to ask about who has no money. As in the program there is no X such that nomoney(X) can be derived, the answer to that query is no. But we know that one of john or mary have no money; more generally, there may be a set of terms such that at least one of them has some property or obeys some relation. A disjunctive answer to the (disjunctive) query ⁄♦ nomoney(X) in Π should be X=john or X=mary, denoting that nomoney(john)⁄nomoney(mary) is a logical consequence of Π.

Our purpose in this section is first to present a definition of answers and of disjunctive answers. We will then argue that SLWV, due to its logic programing like strategy, is suitable to capture the concept of disjunctive answer by informally describing how it can provide such answers.

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

Definition: An answer to a query ♦P(**X**) in a program Π is a substitution σ (for the variables of P(**X**)) such that Π ≈ ¬P(**X**)σ is inconsistent.

This can be seen as the usual definition of an answer to a query. We now extend this definition to disjunctive answers.

Definition: A disjunctive answer to a query ⁄♦P(**X**) is a finite set of substitutions Σ such that Π ≈**Error!**
In other words, Σ is a disjunctive answer to a query ⁄♦P(**X**) such that
Error!) P(**X**)σ is a logical consequence of Π.

To solve the problem of how to find such answers we prove a proposition that states when there exists a disjunctive answer to a query.

Proposition: There exists a disjunctive answer Σ to a query ⁄♦P(**X**) in Π iff *tg* (top goal) is a logical consequence of Π ≈ {*tg* ♦ P(**X**)} (*tg* being a new predicate symbol, not occuring elsewhere in Π).

Proof: According to Herbrand's theorem [Chang73], Π_1 = Π ≈ *tg* ♦ P(**X**) derives *tg* iff there exists a finite subset of Π_2 = Π ≈ *tg* ♦P(**X**)σ_1≈…≈*tg* ♦P(**X**)σ_n≈… (σ_1, …, σ_n, … being all the possible Herbrand Universe substitutions for P(**X**)) deriving *tg* (regarding that the top goal *tg* does not introduce new symbols on Π_1). Let S be the finite subset S of { P(**X**)σ_1, …, P(**X**)σ_n, … } made out from the bodies of clauses with head *tg* from the finite subset of Π_2 that derives it. As *tg* does not occur in Π, for it to be derived from Π_2, according to the resolution principle, the disjunction of the elements of S, must logically follows from Π_2. Thus the set of substitutions applied to P(**X**) on S is, by definition, a disjunctive answer Σ to a query⁄♦P(**X**) in Π. ❏

Based on this proposition, for resolving a query⁄♦P(**X**) in SLWV we implicitly add to the program the clause *tg* ♦P(**X**) and consider the new query ♦*tg*, keeping track of the substitutions for P(**X**) everytime the system *uses* this special clause. Given the procedural aspects of SLWV execution, memory of such substitutions can be easily kept.

Example: Reconsider the program given at the beginning of this section and the query ⁄♦nomoney(X). A derivation for this query, based on SLWV plus an additional (rightmost) set to keep track of the substitutions and the special clause tg ♦ nomoney(X) is[10]:

```
1.    {}#tg                                   ## {}
2.    {tg}# nm(X)                             ## {nm(X)}
```

[10] For the sake brevity, in this derivation nm stands for nomoney, np for nopay, j for john, m for mary and f for father.

3. {tg, nm(j)}# ¬nm(m) {tg, nm(j)}# np(f) ## {nm(j)}
4. {tg, nm(j), ¬nm(m)}# nm(X) {tg, nm(j)}# np(f) ## {nm(j),nm(X)}
5. {tg, nm(j), ¬nm(m)}# nm(m) {tg, nm(j)}# np(f) ## {nm(j),nm(m)}
6. {tg, nm(j),np(f)} # δ ## {nm(j),nm(m)}
7. Δ ## {nm(j),nm(m)}

As expected, a disjunctive answer for the query is {X/john,X/mary}.

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 kernel predicates written in Prolog[11]. For simplicity we won't consider the kind of answers described in last section.

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 *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 subsumes normal programs.

7.2. Preprocessing

In order to have a single positive literal in the head each clause ¬ L :- Body is transformed into **false** :- L, Body. Introducing a topgoal :- G is equivalent to introducing the clause G/**false**, as seen in section 5 above.

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.

[11] The complete code is available on request.

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[12] 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 4.2 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].

7.4. 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

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

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]
```

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).

7.5. 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. Conclusions

We've defined a new theorem proving method that allows for procedural logic programing with classical negation. In particular, Horn clause programs under this system, have a trace that can be mapped quite directly to SLDNF ones. Implementationwise, these programs don't make calls to kernel predicates, so their runtime under our executor is comparable to that under normal Prolog.
We showed how for disjunctive answers our theorem prover is suitable.
For its simplicity and Prolog-like strategy, our method turns out to be attractive for programing in clausal logic, being well suited to extend Prolog programming with classical negation. Furthermore, procedural information regarding clause and goal orders is adhered to, which is still a desireble feature of logic programing.

Comparing this method with the ones presented in [Love87,Stic85,Stic86], we find that it has the advantage of not needing contrapositives variants.
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 significant 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), though, as it is based on model elimination, the user doesn't have control over the execution order.

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

[Pere90] Pereira L., L. Caires and J. Alferes: Classical Negation in Logic Programs. In **Proceedings of the Seventh "Seminário Brasileiro de Inteligência Artificial"** , Campina Grande PB, Brazil, Nov. 90.

[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.