# Comparing Negation in Logic Programming and in Prolog

Krzysztof R. Apt and Frank Teusink

*CWI*

*P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*

**Abstract**

We compare here two uses of negation – in logic programming and in Prolog. As in Prolog negation is defined by means of meta-programming facilities and the cut operator, this requires a careful reexamination of the assumptions about the underlying syntax and a precise definition of the computational processes involved.

After taking care of these matters we establish a formal result showing an equivalence in appropriate sense between these two uses of negation. This result allows us to argue about correctness of various known Prolog programs which use negation by reasoning about the corresponding general logic programs.

*AMS Subject Classification (1991):* 68Q40, 68T15.
*CR Subject Classification (1991):* F.3.2., F.4.1, H.3.3, I.2.3.
*Keywords & Phrases:* Meta-variables, negation, logic programming, Prolog.
*Note:* This research was partly supported by the ESPRIT Basic Research Action 6810 (Compulog 2). This paper will appear in K.R. Apt and F. Turini (editors), Meta-programming in Logic Programming, The MIT Press.

## 1. Introduction

During the last 15 years, a lot of attention was devoted to the study of negation in logic programming. No less than seven survey articles on this subject were published. Just to mention two most recent ones: Dix [Dix93] and Apt and Bol [AB94].

The main reason for this interest is that in the logic programming setting negative literals can be used to model non-monotonic reasoning. The computation process of logic programming provides then a readily available computational interpretation. This is not the case with other approaches to non-monotonic reasoning. This computation process is called SLDNF-resolution and was proposed by Clark [Cla78]. Negation is interpreted in it using the "negation as finite failure" rule. Intuitively, this rule works as follows: for a ground atom $A$,

$$\neg A \text{ succeeds iff } A \text{ finitely fails,}$$

$$\neg A \text{ finitely fails iff } A \text{ succeeds,}$$

where "finitely fails" means that the corresponding evaluation tree is finite and all its leaves are marked as failed.

However, SLDNF-resolution is not a practical way of computing and usually one resorts to Prolog when seeking for a computational interpretation. But in Prolog negation is implemented in a different way, namely by the predicate (or synonymously relation symbol) `neg` defined internally by the following two clauses:

$$\texttt{neg(X)} \leftarrow \;\; \texttt{X}, !, \texttt{fail}. \tag{1}$$

$$\texttt{neg(X)} \leftarrow . \tag{2}$$

where "!" is the cut operator and `fail` is a Prolog built-in with the empty definition.

The intuition behind this definition is perhaps best revealed by first introducing the `if_then_else` predicate defined as follows:

```
if_then_else(P, Q, R) ← P,!,Q.
if_then_else(P, Q, R) ← R.
```

`if_then_else` is intended to model within Prolog the customary **if** $P$ **then** $Q$ **else** $R$ construct of imperative programming languages. Then `neg` can be equivalently defined by

```
neg(X) ← if_then_else(X, fail, □).
```

where □ is the empty query which immediately succeeds. So intuitively, `neg(X)` can be interpreted as "if `X` succeeds then fail else succeed".

It is usually tacitly assumed that logic programming and Prolog ways of dealing with negation are "equivalent", in the sense that SLDNF-resolution combined with the leftmost selection rule (henceforth called LDNF-resolution) properly reflects Prolog's way of handling negation. Upon closer scrutiny this assumption is far from being obvious. The above definition of the `neg` predicate and its use in programs calls upon a number of features which are present in Prolog, but absent in logic programming, and for which a formal treatment is lacking. These are:

- the use of meta-variables, that is variables which occur in an atom position, like `X` in the first clause,
- the use of meta-programming facilities that arise when applying this definition of `neg`, so in constructs of the form `neg(`$A$`)` where $A$ is an atom, or a query in general.

Additionally, two better understood, though not necessarily simpler to handle, features of Prolog need to be taken care of, namely:

- the ordering of the program clauses,
- the use of the cut operator "!".

The aim of this paper is to relate precisely these two uses of negation: in logic programming and in Prolog. To do this we appropriately tune the definition of the SLDNF-resolution given in Apt and Doets [AD94] to our present needs and formally define "Prolog trees" in the presence of the cut operator. Then we prove a result that shows an appropriate equivalence between these two definitions of negation.

The outcome of this study is that we can now interpret various results about correctness of general logic programs executed by means of the LDNF-resolution (see e.g. Apt [Apt94]) as correctness results about the corresponding Prolog programs that use negation.

## 2. Syntactic Matters

### *2.1 General Logic Programs*

To relate general logic programs to Prolog programs we have to be precise about the syntax. Fix a first-order language $\mathcal{L}$. To make this comparison possible we assume that

- a general program is a *sequence* and not a *set* of general clauses,
- the predicates !, `neg` and `fail` are not present in the language $\mathcal{L}$.

A *general clause* is defined in the usual way (see e.g. Lloyd [Llo87]), so as a construct of the form $A \leftarrow L_1, \ldots, L_n$, where $A$ is an atom and $L_1, \ldots, L_n$ are literals, i.e. atoms or their negations, all in the language $\mathcal{L}$. And a *query* is a finite sequence of literals. In the context of logic programming the negation connective is written as "$\neg$".

*2.2 Prolog Programs*
Prolog programs here considered are intended to be the programs that allow us to model the negation by means of the predicate `neg` defined by the clauses (1) and (2). However, the syntax of clause (1) creates a number of problems, even if we ignore the cut operator "!".

First of all, the use of the meta-variable `X` in clause (1) violates the syntax of the first-order logic. This use of `X` in the resolution process leads to further complications. Take an $n$-ary function symbol `p` in the language $\mathcal{L}$ and let $s_1, \ldots, s_n$ be some terms. Consider now the query $\texttt{neg(p(}s_1, \ldots, s_n\texttt{))}$. During Prolog computation process it resolves using the clause (1) to the query $\texttt{p(}s_1, \ldots, s_n\texttt{),!,fail}$. Now in the first query `p` occurs in a position of a function symbol, whereas in the second one `p` occurs in a position of a relation symbol. So every function symbol needs also to be accepted as a relation symbol.

Also conversely: take an $n$-ary relation symbol $p$ with some terms $s_1, \ldots, s_n$, and consider the general clause $p(s_1, \ldots, s_n) \leftarrow \neg p(s_1, \ldots, s_n)$. Its desired translation into a Prolog clause is $\texttt{p(}s_1, \ldots, s_n\texttt{)} \leftarrow \texttt{neg(p(}s_1, \ldots, s_n\texttt{))}$. In the head of the latter clause `p` occurs in a position of a relation symbol, whereas in its body in the position of a function symbol.

As in both cases `p` was arbitrarily chosen, we conclude that to render the resolution process meaningful we need to accept that the classes of function symbols and of relation symbols in the underlying language coincide.

This is clearly in violation with the (usually tacit) assumption that in the first-order language, say $\mathcal{L}$, fixed above, the classes $F_m$ and $R_n$ of, respectively, its function symbols of arity $m$ and its relation symbols of arity $n$ are pairwise disjoint for $m, n \geq 0$. In short, the use of the clause (1) cannot be properly accounted for by just referring to the first-order logic.

A simple solution to the above mentioned two problems is to modify the syntax of the language $\mathcal{L}$ by allowing

- *meta-variables*, so variables that can occur in atoms positions, both in the queries and in the clause bodies,
- *ambivalent syntax*, so – in this case – by assuming that the classes of function and relation symbols coincide.

The latter can be achieved by extending $\mathcal{L}$ to a language in which for each $m \geq 0$ $F_m \cup R_m$ are the classes of both its function symbols and relation symbols. Thus in this language terms and atoms coincide.

Additionally, we assume that

- the predicates !, `neg` and `fail` are present in the underlying language,
- ! is a built-in 0-ary predicate (with a meaning to be explained later), and no clause uses it in its head,
- `neg` is a built-in predicate defined by the clauses (1) and (2), so no other clause uses it in its head,
- `fail` is a built-in 0-ary predicate with the empty definition, so no clause uses it in its head.

The last two assumptions ensure that `neg` and `fail` are indeed defined internally in the desired way. For the purposes of syntax the cut operator "!" is viewed here as a 0-ary predicate with the empty definition. This might suggest that its meaning coincides with that of `fail`. However, this is not the case. Its real, operational, "meaning" will be defined in Section 4 by means external to the resolution process.

So in the resulting language, apart of the customary atoms, also !, `fail` and meta-variables are admitted as atoms (henceforth called *special atoms*).

Now, a *Prolog program* is defined as a sequence of Prolog clauses preceded by the clauses (1) and (2). In turn a *Prolog clause* is a construct of the form $A \leftarrow B_1, \ldots, B_n$, where $A, B_1, \ldots, B_n$ are atoms in the language $\mathcal{L}$, and $A$ is not a special atom. And a *Prolog query* is a finite sequence of atoms. For brevity, in the examples of Prolog programs, we drop the listing of the clauses (1) and (2). Finally, we denote sequences of atoms or literals by bold capital letters.

Note that at this stage we use two notions of an atom – one within the language $\mathcal{L}$ and another in its ambivalent extension just defined. From the context it will be always clear to which of these two languages we refer.

### *2.3 Restricted Prolog Programs*

The translation of a general program to a Prolog program is now straightforward and as expected: we just replace everywhere a logic programming literal $\neg A$ by Prolog's atom neg($A$) and prefix the resulting program with the clauses (1) and (2). In short, the logic programming negation connective "$\neg$" is traded for the built-in predicate `neg`. Similarly, a general query is translated to a Prolog query by replacing everywhere $\neg A$ by neg($A$).

This translation process maps every general program (resp. general query) onto a Prolog program. However, not every Prolog program (resp. Prolog query) is the result of translating a general program (resp. general query). Indeed, in general the cut operator "!" can be used in any Prolog clause, not only (1).

Let us now characterize the Prolog programs (resp. Prolog queries) which are the result of the above translation of general programs (resp. general queries). We call them *restricted Prolog programs* (resp. *restricted Prolog queries*). To this we translate "back" every Prolog program (resp. Prolog query) onto a general program (resp. general query) by replacing everywhere neg($A$) by $\neg A$, and omitting the clauses (1) and (2) that define the `neg` predicate. Then a Prolog program (resp. Prolog query) is restricted if the outcome of this reverse translation is a syntactically legal general program (resp. general query). For example the Prolog query `neg(q),q` is restricted because its reverse translation is $\neg q, q$, whereas neither `neg(q(neg(a)))` nor `p(q),q` is restricted because their respective reverse translations violate the syntactic assumptions concerning general programs.

Of course, it is possible to define the class of restricted Prolog programs and queries directly, though the resulting definition is rather tedious.

We now define a *resolvent* of a Prolog query as follows.

**Definition 2.1** Consider a non-empty Prolog query $A, \mathbf{M}$ and a Prolog clause $c$. Let $H \leftarrow \mathbf{L}$ be a variant of $c$ variable disjoint with $A, \mathbf{M}$ and let $\theta$ be an mgu of $A$ and $H$. Then $(\mathbf{L}, \mathbf{M})\theta$ is called a *resolvent* of $A, \mathbf{M}$ and $c$ *with an mgu* $\theta$. □

The only unusual feature in the present setting is, that now the mgu's also bind the meta-variables. Also, note that the selected literal is always the leftmost literal.

It is worthwhile to mention that a resolvent of a restricted Prolog query w.r.t. a restricted Prolog program is not necessarily a restricted Prolog query. This is due to the use of clause (1), which introduces a cut atom. Thus, the Prolog queries generated in a computation of a restricted Prolog query are not necessarily restricted Prolog queries. However, the Prolog queries so generated do have one important property: they do not contain meta-variables. To prove this fact we need a stronger property.

**Definition 2.2**

- An atom $A$ is called *unsafe* if one of the following holds:
  - $A$ is a meta-variable,

  - $A$ is `neg(X)` where `X` is a variable,
  - $A$ is `neg(neg(s))` where `s` is a term.

- A Prolog query is called *meta-safe* if none of its atoms is unsafe. □

For example, the Prolog query `X, p(X)` is not meta-safe because its leftmost atom is a meta-variable, `neg(X)` is not meta-safe because the argument of `neg` is a meta-variable, and `neg(neg(p(X)))` is not meta-safe because the argument of the outermost `neg` predicate is itself a `neg` predicate.

Note that restricted Prolog queries and bodies of the restricted Prolog clauses are meta-safe.

**Lemma 2.3** *Let $Q$ be a meta-safe Prolog query and $P$ a restricted Prolog program. Then all resolvents of $Q$ are meta-safe.*

**Proof:** Let $Q$ be of the form $A, \mathbf{L}$, and let $(\mathbf{M}, \mathbf{L})\theta$ be a resolvent of $Q$, with an input clause $c$ and mgu $\theta$. As $Q$ is meta-safe, we know that $\mathbf{L}\theta$ is meta-safe. We prove that $\mathbf{M}\theta$ is meta-safe as well. Three cases arise.

**Case 1** : $c$ is clause (1).

Then $\mathbf{M}\theta$ is of the form $B$`,!,fail`, where $A$ is of the form `neg(`$B$`)`. But $Q$ is meta-safe, so $B$ is neither a meta-variable nor of the form `neg(`$B'$`)`. So $\mathbf{M}\theta$ is meta-safe.

**Case 2** : $c$ is clause (2).

Then $\mathbf{M}\theta$ is the empty query, so obviously meta-safe.

**Case 3** : $c$ is different from clauses (1) and (2).

Then the body of $c$ is meta-safe, and consequently so is $\mathbf{M}\theta$.

This proves that $(\mathbf{M}, \mathbf{L})\theta$ is meta-safe. □

**Corollary 2.4** *All Prolog queries generated in a computation of a restricted Prolog query and a restricted Prolog program are meta-safe.* □

In Prolog, if the selected atom is a meta-variable, an *error* arises. The above result thus shows that no errors arise in Prolog computations for queries and programs that are obtained by a translation of a general query and a general program.

## 3. Computing with General Logic Programs: LDNF-resolution

As the next step we define the LDNF-resolution that allows us to compute with general logic programs. The definition of LDNF-resolution given here is derived in a straightforward way from that of the SLDNF-resolution given in Apt and Doets [AD94]. Apart of the fact that we view in this paper a general program as a finite sequence and not as a finite set of general clauses, the differences are that:

- the leftmost selection rule is used,
- *floundering*, so –in this context– an abnormal termination due to selection of a non-ground literal is ignored.

In this way we bring the procedural interpretation of general programs closer to that of the corresponding Prolog programs and make the subsequent comparison possible. Recall from Clark [Cla78] and Lloyd [Llo87] that floundering is a problem that arises only when dealing with the semantic aspects of the SLDNF-resolution, which are irrelevant here.

Before giving the definition of LDNF-resolution, we recall the definitions of *resolvent* and *pseudo-derivation*.

**Definition 3.1** Consider a non-empty general query $L, \mathbf{M}$ and a general clause $c$.

- Suppose $L$ is a positive literal.

  Let $H \leftarrow \mathbf{L}$ be a variant of $c$ variable disjoint with $L, \mathbf{M}$ and let $\theta$ be an mgu of $L$ and $H$. Then $(\mathbf{L}, \mathbf{M})\theta$ is called a *resolvent* of $L, \mathbf{M}$ and $c$ *w.r.t.* $L$, *with an mgu* $\theta$.

  We write then $L, \mathbf{M} \overset{\theta}{\underset{c}{\Longrightarrow}} (\mathbf{L}, \mathbf{M})\theta$, and call it a *positive derivation step*. We call $H \leftarrow \mathbf{L}$ the *input clause* of the derivation step.

- Suppose $L$ is a negative literal. Then $\mathbf{M}$ is called a *resolvent* of $L, \mathbf{M}$ with the identity substitution $\epsilon$ *w.r.t.* $L$.

  We write then $L, \mathbf{M} \overset{\epsilon}{\underset{\emptyset}{\Longrightarrow}} \mathbf{M}$, and call it a *negative derivation step*.

- A general clause $c$ is called *applicable* to an atom if it has a variant the head of which unifies with the atom. □

Fix, until the end of this section, a general program $P$.

**Definition 3.2** A (finite or infinite) sequence $Q_0 \overset{\theta_1}{\underset{c_1}{\Longrightarrow}} Q_1 \cdots Q_n \overset{\theta_{n+1}}{\underset{c_{n+1}}{\Longrightarrow}} Q_{n+1} \cdots$ of derivation steps is called a *pseudo derivation of* $P \cup \{Q_0\}$ if

- $Q_0, \ldots, Q_n, \ldots$ are general queries,
- $\theta_1, \ldots, \theta_n, \ldots$ are substitutions,
- $c_1, \ldots, c_n, \ldots$ are general clauses of $P$, or $\emptyset$,

and for every step involving selection of a positive literal the following condition holds:
**Standardization apart**: the input clause employed is variable disjoint from the initial general query $Q_0$ and from the substitutions and input clauses used at earlier steps. □

Intuitively, an LDNF-derivation is a pseudo derivation in which the deletion of every negative literal is justified by means of a subsidiary (finitely failed LDNF-) tree. This brings us to consider special types of trees, called *forests*.

**Definition 3.3** A *forest* is a system $\mathcal{F} = (\mathcal{F}, T, subs)$ where

- $\mathcal{F}$ is a set of trees,
- $T$ is an element of $\mathcal{F}$ called the *main tree*, and
- *subs* is a function assigning to some nodes of trees in $\mathcal{F}$ a ("subsidiary") tree from $\mathcal{F}$.

By a *path* in $\mathcal{F}$ we mean a sequence of nodes $N_0, \ldots, N_i, \ldots$ such that for all $i$, $N_{i+1}$ is either an immediate descendant of $N_i$ in some tree in $\mathcal{F}$, or the root of the tree $subs(N_i)$. The *depth* of $\mathcal{F}$ is the length of the longest path in $\mathcal{F}$. □

Thus a forest is a special directed graph with two types of edges – the "usual" ones stemming from the tree structures, and the ones connecting a node with the root of a subsidiary tree. An LDNF-tree is a special type of forest, built as a limit of certain finite forests: *pre-LDNF trees*.

**Definition 3.4** A *pre-LDNF-tree* (relative to $P$) is a forest whose nodes are queries. Leaves can be unmarked, or can be marked as either *success* or *failure*. The class of pre-LDNF-trees is defined inductively:

- For every general query $Q$, the forest consisting of the main tree which has the single unmarked node $Q$ is a pre-LDNF-tree (an *initial* pre-LDNF-tree),

- If $\mathcal{T}$ is a pre-LDNF-tree, then any *extension* of $\mathcal{T}$ is a pre-LDNF-tree.

Before defining the notion of an *extension* of a pre-LDNF-tree, we need to define the notion of *successful* and *finitely failed* trees: for $T \in \mathcal{T}$,

- $T$ is called *successful*, if one of its leaves is marked as *success*, and
- $T$ is called *finitely failed*, if it is finite and all its leaves are marked as *failure*.

Now, an *extension* of a pre-LDNF-tree $\mathcal{T}$ is defined by performing the following actions for every non-empty general query $Q$ (with leftmost literal $L$) which is an unmarked leaf in some tree $T \in \mathcal{T}$:

- Suppose that $L$ is a positive literal.
  - If $Q$ has no resolvents w.r.t. $L$ and a clause from $P$:
    Mark $Q$ as *failure*.
  - If $Q$ has such resolvents:
    For every clause $c$ from $P$ which is applicable to $L$, choose one resolvent $Q'$ of $Q$ w.r.t. $L$ and $c$, with an mgu $\theta$, and add this as an immediate descendant of $Q$ in $T$. Choose the input clauses in such a way that all branches of $T$ remain pseudo derivations.
- Suppose that $L$ is a negative literal, say $\neg A$.
  - If $subs(Q)$ is undefined:
    Add a new tree $T'$, consisting of the single node $A$, to $\mathcal{T}$, and let $subs(Q) = T'$.
  - If $subs(Q)$ is defined and successful:
    Mark $Q$ as *failure*.
  - If $subs(Q)$ is defined and finitely failed:
    Add the resolvent $Q - \{L\}$ of $Q$ as the only immediate descendant of $Q$ in $T$.

Additionally, all empty queries are marked as *success*. □

Note that, if no tree in $\mathcal{T}$ has unmarked leaves, then trivially $\mathcal{T}$ is an extension of itself, and the extension process becomes stationary.

Next, we define LDNF-trees as the limit of sequences of pre-LDNF-trees. Every pre-LDNF-tree is a tree with two types of edges between possibly marked nodes, so the concepts of *inclusion* between such trees and of *limit* of a growing sequence of such trees have a clear meaning.

**Definition 3.5**

- An *LDNF-tree* is a limit of a sequence $\mathcal{T}_0, \ldots, \mathcal{T}_\alpha, \ldots$ such that $\mathcal{T}_0$ is an initial pre-LDNF-tree, and for all $i$ $\mathcal{T}_{i+1}$ is an extension of $\mathcal{T}_i$.
- An *LDNF-tree for* $Q$ is an LDNF-tree in which $Q$ is the root of the main tree.
- A (pre-)LDNF-tree is called *successful* (resp. *finitely failed*) if the main tree is successful (resp. finitely failed).
- An LDNF-tree is called *finite* if no infinite path exists in it (cf. Definition 3.3). □

In Figure 1, we show how the notions of initial pre-LDNF-trees and extensions of pre-LDNF-trees are used to construct a P-tree.

Finally, we recall the notion of a computed answer substitution.

**Definition 3.6** Consider a branch in the main tree of a (pre-)LDNF-tree for $Q$ which ends with the empty query. Let $\alpha_1, \ldots, \alpha_n$ be the consecutive substitutions along this branch.

Then the restriction $(\alpha_1 \cdots \alpha_n)|Q$ of the composition $\alpha_1 \cdots \alpha_n$ to the variables of $Q$ is called a *computed answer substitution* (*c.a.s.* for short) of $Q$. □

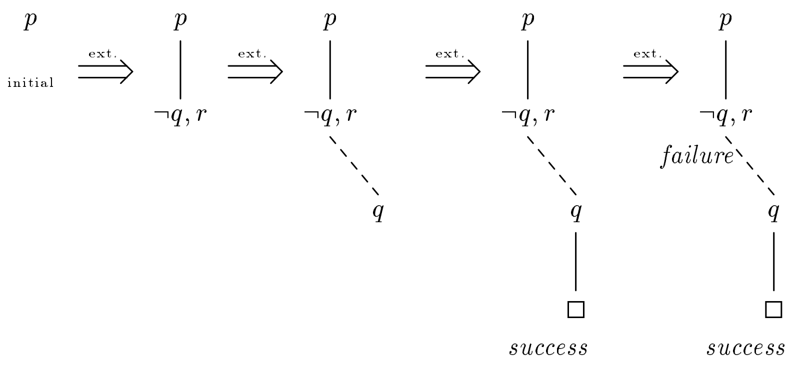



Figure 1: Step-by-step construction of an LDNF-tree for the query $p$ w.r.t. the general program $p \leftarrow \neg q, r \quad q \leftarrow$.

## 4. Computing with Prolog programs: P-resolution

In this section, we define the computation process used in Prolog to find answers to queries, which we call *P-resolution.* To this end we proceed in two steps.

First, we restrict the LDNF-resolution to logic programs, so general logic programs without negation, by simply disregarding the selection of a negative literal. We call the resulting computation process *LD-resolution.*

Then, we extend the LD-resolution to Prolog programs by allowing the choice of a meta-variable or of a cut atom as a selected atom. In the first case an error is reported, and in the second case the computation tree constructed so far is appropriately pruned.

To better understand the issues involved in defining the effect of the cut operator, let us consider the definition of a predicate p:

$$\begin{array}{l} \mathtt{p(s_1)} \leftarrow \mathbf{L}_1\,\mathtt{.} \\ \ldots \\ \mathtt{p(s_i)} \leftarrow \mathbf{M}\mathtt{,!,}\mathbf{N}\mathtt{.} \\ \ldots \\ \mathtt{p(s_k)} \leftarrow \mathbf{L}_k\,\mathtt{.} \end{array}$$

Here, the $i$-th clause contains a cut atom (there could be others, either in the same clause, or in other clauses). Now, suppose that during the execution of a query, some atom $\mathtt{p}(\mathbf{t})$ is resolved using (a variant of) the $i$-th clause, and that later on, the cut atom thus introduced becomes the leftmost atom. Then, according to the customary definition of the cut operator "!", once the indicated occurrence of ! is selected:

1. all other ways of resolving $\mathbf{M}$ are discarded, and
2. all derivations using (variants of) the $i+1$-th to $k$-th clause for p are discarded.

Note that this operational definition of the behaviour of the cut operator depends on the leftmost selection rule, and on viewing a program as a sequence of clauses, instead of a set of clauses.

To model this operational behaviour of the cut operator in P-resolution, we have to define it in terms of a pruning operator on LD-trees, but first, let us give an example of the behaviour of the cut operator. Consider the following Prolog program:

```
p ← q,!,t.   q ← r,!,t.   r ← s.   s ← .
p ← .        q ← .        r ← .
```

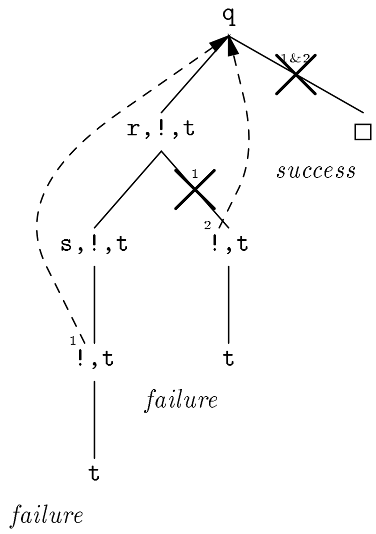



Figure 2: A computation tree for the query q

In Figure 2, an LD-tree for the query q is shown. In this tree, there are two nodes with a cut atom as the leftmost atom. Both of these cut atoms are introduced by resolving q in the root node of the tree. We say that their *origin* is the root node. In the figure, we use dashed arrows to to point from a selected cut atom to its origin. The two cut atoms that appear as leftmost atoms are marked as 1 and 2 respectively. Now, consider the cut atom marked as 1. Execution of this cut atom results in pruning: the middle branch has to be pruned according to rule 1, and the rightmost branch has to be pruned following rule 2. Execution of the cut atom marked as 2 also leads to a pruning of the rightmost branch (using rule 1 for the cut operator). In the figure, the pruned branches are marked using a cross. The label on the cross refers to the cut atoms that where responsible for the pruning of that branch.

Now, we can restate the behaviour of the cut operator as a pruning operator on LD-trees. Consider an LD-tree $T$. Let $Q$ be a node in $T$ with a cut atom as the selected atom and let $Q'$ be the origin of this cut atom (i.e. the node that introduced this cut atom). Then, execution of this cut atom results in pruning all branches that are to the right of $Q$, contain $Q'$, and do not contain $Q$.

In the tree of Figure 2, the order in which selected cut atoms where processed, was not important. However, in general, the order *is* important. Consider the LD-tree for $p$ in Figure 3. Here, there are three nodes with a cut atom as leftmost atom, marked as 1, 2 and 3, respectively. Suppose we would process them from right to left. First, the cut atom marked as 3 would prune the rightmost branch. Then, the cut atoms marked as 2 and 1 would prune the third and the second branch from the left, respectively. The resulting tree would consist of the leftmost branch only. On the other hand, when processed from left to right, the cut atom marked as 1 would prune the middle two branches. As a result, the cut atoms marked as 2 and 3 would disappear, which would prevent the rightmost branch from being pruned. Thus, the resulting tree would consists of the leftmost branch *and* the rightmost branch.

In Prolog, answers are computed using a left to right depth-first strategy. In particular, Prolog processes the cut atoms in the tree from left to right. On the other hand, LD-resolution is defined in a breadth-first manner: the process of extending a pre-tree consists of extending all unmarked leaves of that tree simultaneously. To solve this problem, we have to refine LD-resolution so that the depth-first strategy is used instead of the breadth-first strategy. At first sight it seems that to this end we have to implement the backtracking mechanism used by Prolog. Fortunately, it is not so. A

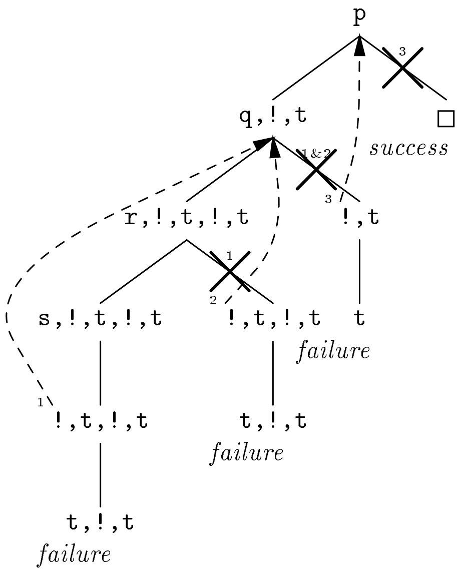



Figure 3: A computation tree for the query `p`

simpler alternative is to generate at each stage all direct successors of the *leftmost* unmarked leaf only. In this way the backtracking process is taken care of automatically.

Having discussed the modifications of the LD-resolution we now model the computation process of Prolog, by providing a formal definition of P-resolution. The central notion in this definition is that of a *P-tree*. We define them as the limit of a sequence of *pre-P-trees*, which in turn are a subclass of a class of ordered trees called *semi-P-trees*.

**Definition 4.1** A *semi-P-tree* (relative to $P$) is an ordered tree whose nodes contain queries, possibly marked with *success*, *failure*, or *error*. □

In an ordered tree, by definition for every node there is a strict total order on its children. To define the behaviour of the cut operator, we use these total orders to define a partial order on the nodes of an ordered tree.

**Definition 4.2** Let $m, n$ be two nodes in an ordered tree. We say that $n$ is *to the right of* $m$ if for some predecessors $m'$ and $n'$ of $m$ and $n$, respectively,

- $m'$ and $n'$ are siblings,
- $m'$ is strictly smaller than $n'$ in the total order on the children of a node. □

The first step in defining pre-P-trees is to define the effect of the cut operator.

**Definition 4.3** Let $\mathcal{B}$ be a branch in a semi-P-tree, and let $Q$ be a node in this branch with a cut atom as the leftmost atom. Then, the *origin* of this cut atom is the first predecessor of $Q$ in $\mathcal{B}$ that contains less cut atoms than $Q$. □

To see that this definition properly captures the informal meaning of the origin note that, when following a branch from top to bottom, the cut atoms are introduced and removed in a First-In Last-Out manner.

**Definition 4.4** Let $\mathcal{T}$ be a semi-P-tree, $Q$ a query in $\mathcal{T}$ which has a cut atom as the leftmost atom, and $Q'$ be the origin of this cut atom. Then, the operator $cut(\mathcal{T}, Q)$ removes from $\mathcal{T}$ all the nodes that are descendants of $Q'$ and lie to the right of $Q$. □

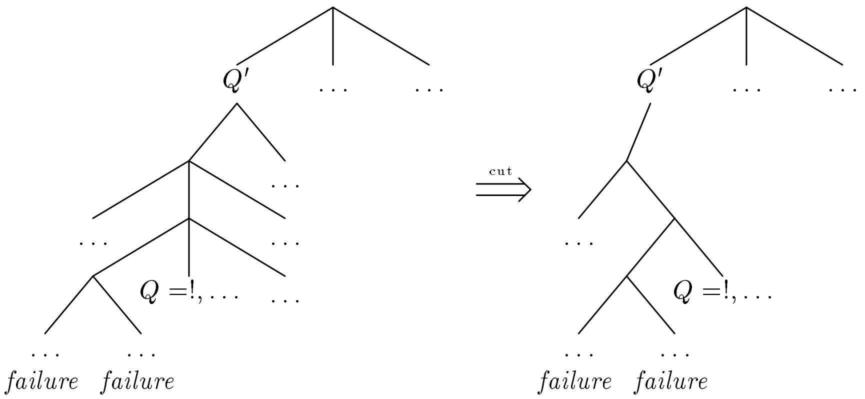



Figure 4: The effect of the operator $cut(\mathcal{T}, Q)$

In Figure 4, we illustrate the effect of $cut(\mathcal{T}, Q)$.

**Definition 4.5** The class of *pre-P-trees* is defined as follows:

- For every query $Q$, the tree consisting of the single unmarked node $Q$ is a pre-P-tree (an *initial* pre-P-tree).
- If $\mathcal{T}$ is a pre-P-tree, then any *extension* of $\mathcal{T}$ is a pre-P-tree.

An *extension* of a pre-P-tree $\mathcal{T}$ is defined as follows:

Let $Q$ be the leftmost unmarked leaf in $\mathcal{T}$. If $Q$ is the empty query, mark $Q$ as *successful*. Otherwise, let $Q$ be of the form $A, \mathbf{M}$.

- Suppose $A$ is an ordinary atom (i.e. not a special atom).
  - If $Q$ has no resolvents w.r.t. a clause from $P$:
    Mark $Q$ as *failure*.
  - If $Q$ has such resolvents:
    For every clause $c$ from $P$ which are applicable to $A$, choose one resolvent $Q'$ of $Q$ w.r.t. $c$ and add this as a child of $Q$ in $T$. Choose the input clauses in such a way that all branches of $T$ remain pseudo derivations. Order these children according to the the order in which their input-clauses appear in $P$.
- Suppose $A$ is a cut atom.
  Apply the operation $cut(\mathcal{T}, Q)$.
  Provide $Q$ with a single child $\mathbf{M}$.
- Suppose $A$ is a meta-variable.
  Mark $Q$ as *error*. □

We now define P-trees as the limit of sequences of pre-P-trees. In Figure 5, we show how the notions of initial pre-P-trees and extensions of pre-P-trees can be used to construct a P-tree (the program used in the figure is the translation of the program used in Figure 1). Note that in this Figure, the result of the 'cut step' (that is, the fifth tree) is not itself part of the sequence of extensions; it was added to clarify the use of the cut operator in the construction of P-trees.

To be able to define the limit of a sequence of pre-P-trees, we have to define a notion of an *inclusion* between pre-P-trees, and of the *limit* of a growing sequence of pre-P-trees. For pre-LD-trees and pre-LDNF-trees, these notions were obvious. In the case of pre-P-trees, the pruning that takes place when extending a pre-P-tree, complicates the matters a bit.

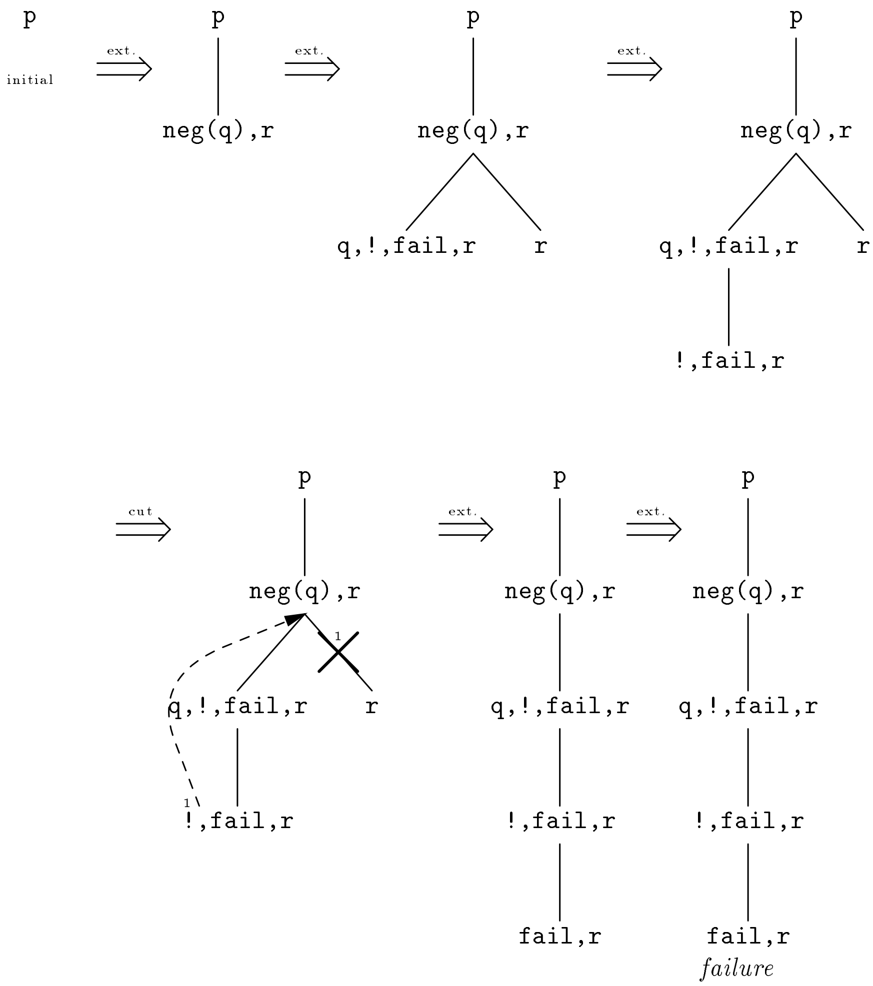



Figure 5: Step-by-step construction of a P-tree for the Prolog query `p` w.r.t. the Prolog program `p` $\leftarrow$ `neg(q), r.` `q` $\leftarrow$`. .`

**Definition 4.6** Let $\mathcal{T}$ and $\mathcal{T}'$ be pre-P-trees. $\mathcal{T}$ is said to be *included* in $\mathcal{T}'$ if $\mathcal{T}'$ can be constructed from $\mathcal{T}$ by means of one of the following two operations:

1. adding some children to a leaf of $\mathcal{T}$.
2. removing a single subtree from $\mathcal{T}$, provided its root is not a single child in $\mathcal{T}$.

We say that $\mathcal{T}$ is *properly included* in $\mathcal{T}'$, if $\mathcal{T}$ is included in $\mathcal{T}'$ and $\mathcal{T}'$ is not included in $\mathcal{T}$. We use $\subset$ to denote the transitive closure of the relation "$\mathcal{T}$ is properly included in $\mathcal{T}'$" and define $\mathcal{T} \subseteq \mathcal{T}'$ as $(\mathcal{T} \subset \mathcal{T}') \vee (\mathcal{T} = \mathcal{T}')$. □

Note that operation (2) never turns an internal node into a leaf.

**Lemma 4.7** *The relation $\subset$ is a strict partial order on pre-P-trees.*

**Proof:** We have to prove that the conditions for a strict partial order hold.

1. $\mathcal{T} \not\subset \mathcal{T}$

   Suppose by contradiction that $\mathcal{T} \subset \mathcal{T}$. Then, there exists a $\mathcal{T}'$ such that $\mathcal{T}$ is properly included in $\mathcal{T}'$, and $\mathcal{T}' \subseteq \mathcal{T}$. There are two cases:

   - $\mathcal{T}'$ is constructed by adding children to a leaf of $\mathcal{T}$.
     But then, some node $Q$ that is a leaf in $\mathcal{T}$, is an internal node in $\mathcal{T}'$. By definition of inclusion, and the fact that $\mathcal{T}' \subseteq \mathcal{T}$, $Q$ is an internal node in $\mathcal{T}$. This is in contradiction with the fact that $Q$ is a leaf $\mathcal{T}$.
   - $\mathcal{T}'$ is constructed by pruning a single subtree from $\mathcal{T}$.
     By definition of inclusion, the parent of the pruned subtree has at least two children in $\mathcal{T}$, and therefore, it has at least one child in $\mathcal{T}'$. Moreover, new nodes can only "grow" from leaves. Thus subtrees pruned from $\mathcal{T}$ can never be "regenerated", to reconstruct $\mathcal{T}$ out of $\mathcal{T}'$. Therefore, $\mathcal{T}' \not\subseteq \mathcal{T}$, which leads to a contradiction.

2. $\mathcal{T} \subset \mathcal{T}'$ and $\mathcal{T}' \subset \mathcal{T}''$ imply $\mathcal{T} \subset \mathcal{T}''$.

   Straightforward by the definition of $\subset$. □

**Corollary 4.8** *The relation $\subseteq$ is a partial order on pre-P-trees.* □

Clearly, with this notion of inclusion, we have that if $\mathcal{T}$ extends $\mathcal{T}'$ in the sense of Definition 4.5, then $\mathcal{T}' \subseteq \mathcal{T}$, so we can use this notion of extension to construct monotonously growing chains of pre-P-trees.

**Definition 4.9**

- A *P-tree* is a limit of a sequence $\mathcal{T}_0, \ldots, \mathcal{T}_i, \ldots$ such that $\mathcal{T}_0$ is an initial pre-P-tree, and for all $i$, $\mathcal{T}_{i+1}$ is an extension of $\mathcal{T}_i$.
- *A P-tree for $Q$* is a P-tree whose root is the query $Q$.
- An P-tree is called *finite* if no infinite branch exists in it. □

Formally, this definition is justified by the fact that every countable partial order with the least element (here the relation $\subseteq$ on pre-P-trees with the initial pre-P-tree as least element) can be canonically extended to a countable cpo (see e.g. Gierz [GHK$^+$80]).

Next, we define the concepts of *successful* and *finitely failed* P-trees.

**Definition 4.10**

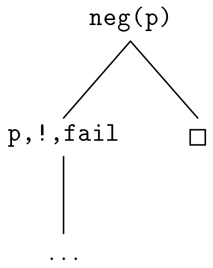



Figure 6: A P-tree for the query `neg(p)` w.r.t. $\mathtt{p} \leftarrow \mathtt{p}$.

- A P-tree is called *successful* if one of its leaves is marked as *success*.
- A (pre-)P-tree is called *finitely failed*, if it is finite, and all its leaves are marked as *failure*. □

Note that in P-trees, in contrast to LDNF-trees, some leaves can be unmarked. Whenever this is the case, the P-tree will contain exactly one infinite branch to the left of all these unmarked leaves. Such unmarked leaves represent the resolvents the Prolog computation process did not reach, because it got "trapped" in an infinite derivation (the infinite branch). For example, take the program $\mathtt{p} \leftarrow \mathtt{p}$., and the query `neg(p)`. Its P-tree is shown in Figure 6. This tree contains a branch ending with a leaf containing the empty query. However, this leaf is never reached by the Prolog computation process (and therefore never marked) because there is an infinite branch to the left of it.

Finally, it is clear how to define the notion of a computed answer substitution.

**Definition 4.11** Consider a successful derivation in a pre-P-tree for $Q$. Let $\alpha_1, \ldots, \alpha_n$ be the consecutive substitutions along this branch.

Then the restriction $(\alpha_1 \cdots \alpha_n)|Q$ of the composition $\alpha_1 \cdots \alpha_n$ to the variables of $Q$ is called a *computed answer substitution* (*c.a.s.* for short) of $Q$. □

## 5. Correspondence between LDNF-trees and P-trees

In this section, we prove that there is a close correspondence between (computed answers of) LDNF-trees and P-trees. More precisely, we prove that termination results on general programs w.r.t. LDNF-resolution translate directly into termination of their translated Prolog programs w.r.t. Prolog computation. For this purpose, we start by examining finite LDNF-trees, and their corresponding P-trees.

**Theorem 5.1** *Let $\mathcal{T}_L$ be a finite LDNF-tree for a general query $Q$. Then, there exists a finite P-tree $\mathcal{T}_P$ for $Q$ such that $\mathcal{T}_L$ and $\mathcal{T}_P$ have the same set of computed answers.*

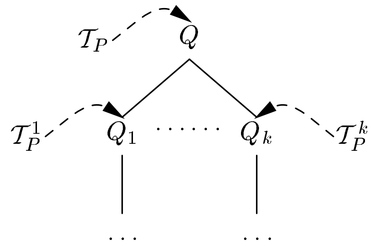
**Proof:** We prove the claim by induction on the depth of LDNF-trees (cf. Definition 3.3). Assume that the claim holds for all LDNF-trees of depth less than $r$. We have to prove the claim for LDNF-trees of depth $r$.

Let $\mathcal{T}_L$ be an LDNF-tree for $Q$ of some finite depth $r$. In the remainder of this proof, we identify a general query with its translation into a Prolog query. From the context it will always be clear whether we refer to a general query, or a Prolog query. Two cases arise.

- Suppose that $Q$ is of the form $A, \mathbf{L}$.

  Let $Q_1, \ldots, Q_k$ $(k \geq 0)$ be the children of $Q$ in $\mathcal{T}_L$. Let, for $i \in [1..k]$, $\mathcal{T}_L^i$ denote the subtree of $\mathcal{T}_L$ starting at $Q_i$.

  As, for $i \in [1..k]$, $\mathcal{T}_L^i$ is finite and of depth less than $r$, by induction hypothesis there exists a P-tree $\mathcal{T}_P^i$ for $Q_i$ such that $\mathcal{T}_P^i$ contains the same computed answers as $\mathcal{T}_L^i$. Now consider the semi-P-tree $\mathcal{T}_P$ with root $Q$, children $Q_1, \ldots, Q_k$ (ordered according to the order of their input clauses in $P$) and, for $i \in [1..k]$, $\mathcal{T}_P^i$ as the subtree starting at $Q_i$, as depicted by the following diagram:




To prove that $\mathcal{T}_P$ is a P-tree for $Q$, it is sufficient to show that all pruning caused by selection of cut atoms is guaranteed to be local to the respective subtrees $\mathcal{T}_P^i$ (for $i \in [1..k]$). Neither $Q$, nor its children $Q_1, \ldots, Q_k$ in $\mathcal{T}_P$, contain a cut atom, so no atom in $\mathcal{T}_P$ has $Q$ as its origin. It follows from the definition of the cut operator that all pruning is indeed local to the respective subtrees $\mathcal{T}_P^i$. Thus $\mathcal{T}_P$ is a P-tree for $Q$. From its construction, it follows that it contains the same computed answers as $\mathcal{T}_L$. Moreover, it is finite.

- Suppose that $Q$ is of the form $\neg A, \mathbf{L}$.

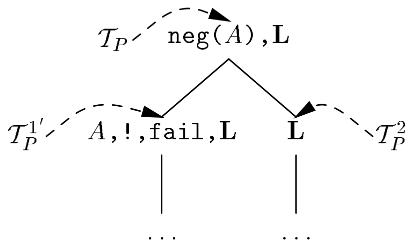
  Let $\mathcal{T}_L^1$ be the subtree of $\mathcal{T}_L$ starting at the root of $subs(Q)$. As the LDNF-tree $\mathcal{T}_L^1$ for $A$ is finite and of depth less than $r$, by induction hypothesis there exists a finite P-tree $\mathcal{T}_P^1$ for $A$ that has the same computed answers as $\mathcal{T}_L^1$. There are two sub-cases.

  - Suppose that $Q$ has a child in $\mathcal{T}_L$.

    Then, $\mathcal{T}_L^1$ is finitely failed, and therefore $\mathcal{T}_P^1$ is finitely failed as well. But then, we can construct a finitely failed P-tree $\mathcal{T}_P^{1'}$ for $A$`,!,fail,`$\mathbf{L}$. In this P-tree, the cut atom introduced at the root will never be reached.

    Let $\mathcal{T}_L^2$ be the subtree of $\mathcal{T}_L$ starting at the single child $\mathbf{L}$ of $Q$. As the LDNF-tree $\mathcal{T}_L^2$ for $\mathbf{L}$ is finite and of depth less than $r$, by induction hypothesis there exists a finite P-tree $\mathcal{T}_P^2$ for $\mathbf{L}$ that has the same computed answers as $\mathcal{T}_L^2$.

    Using $\mathcal{T}_P^{1'}$ and $\mathcal{T}_P^2$ we can construct a finite P-tree $\mathcal{T}_P$ for $Q$ that has the same computed answers as $\mathcal{T}_L$. This tree has the following form:




  - Suppose that $Q$ has no children in $\mathcal{T}_L$.

    Then, $\mathcal{T}_L^1$ is successful, and therefore $\mathcal{T}_P^1$ is successful as well. But then we can construct a finitely failed P-tree $\mathcal{T}_P^{1'}$ for $A$`,!,fail,`$\mathbf{L}$, in which the cut atom present in its root is selected at some point.

    Let $\mathcal{T}_P$ be the semi-P-tree such that its root is $Q$, and the subtree starting at the single child $A$`,!,fail,`$\mathbf{L}$ of $Q$ is $\mathcal{T}_P^{1'}$. In this tree, the origin of the cut atom that appears in the single child of $Q$, is $Q$. This cut atom is the selected atom in some node within $\mathcal{T}_P^{1'}$. Thus $\mathcal{T}_P$ is a P-tree for $Q$, because the potential second child of $Q$, that would contain the query $\mathbf{L}$ has been pruned at some stage. Thus $\mathcal{T}_P$ is finitely failed, just as $\mathcal{T}_L$ is. □

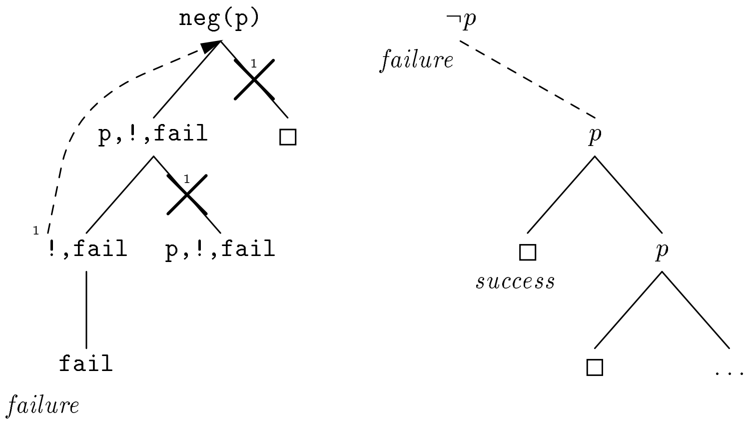



Figure 7: A P-tree and an LDNF-tree for `neg(p)`

Thus if we have a general query $Q$ that terminates w.r.t. a general program $P$, we know that Prolog computation on that query and that program will terminate, and give the same computed answers as LDNF-resolution.

Now what if we have a finite P-tree for a restricted Prolog query $Q$ and a restricted Prolog program $P$? Consider the following restricted Prolog program

$$p \leftarrow$$
$$p \leftarrow p$$

and the restricted Prolog query `neg(p)`. The P-tree and LDNF-tree for this query and this program are shown in Figure 7 (note that the pruned branches are not really part of the P-tree for `neg(p)`, but existed at some point during the construction of this P-tree). In this example, the P-tree is finite, because the potentially infinite branch caused by the clause $p \leftarrow p$ is pruned. However, in the LDNF-tree, this branch has been constructed in full, and therefore this LDNF-tree is infinite.

## 6. Applications

Due to the presence of cut in the definition of the predicate `neg` it is difficult to reason in a declarative way about Prolog programs that use negation. In other words, it is not clear how to prove correctness of such programs using their declarative interpretation.

We now show how this is possible using the results of this paper. The key observation is that Theorem 5.1 provides a crucial relationship between the computational behaviour of Prolog programs and their translations into general logic programs.

In the subsequent discussion we assume that the variables in the input clauses and the mgu's are chosen in a fixed way. We can then assume that for every Prolog program $P$ and Prolog query $Q$ there exists exactly one P-tree, and similarly for general logic programs, general queries and LDNF-trees.

So consider a restricted Prolog program $P$ with a restricted query $Q$ and their translation $P_L$ and $Q_L$ onto a general logic program and a general logic query, respectively. To reason about correctness of $P$ with $Q$ it is sufficient to reason about $P_L$ and $Q_L$. Indeed, suppose that we proved already that all LDNF-derivations of $P$ and $Q$ are finite. Then by Theorem 5.1 the P-tree for $P_L$ and $Q_L$ is finite, and $P_L$ with $Q_L$ and $P$ with $Q$ have the same set of computed answers.

As an example consider the following well-known Prolog program `TRANS` about which one claims that it computes the transitive closure a binary relation `e`:

```
trans(X, Y, E, Avoids) ← member([X, Y], E).
trans(X, Z, E, Avoids) ←
```

```
    member([X, Y], E),
    neg(member(Y, Avoids)),
    trans(Y, Z, E, [Y | Avoids]).

member(X, [X | Xs]) ← .
member(X, [Y | Xs]) ← member(X, Xs).
```

In Apt [Apt94] the following facts about its translation $\mathtt{TRANS}_L$ to a general logic program and a binary relation `e` were established:

- all LDNF-derivations of `trans(X, Y, e, [ ])` are finite,
- the computed answer substitutions of `trans(X, Y, e, [ ])` determine all pairs of elements which form the transitive closure of `e`.

Now, by Theorem 5.1 the same conclusions can be drawn about the original program `TRANS`.

The fact that above approach to correctness is limited to restricted Prolog programs is in our opinion not serious. In fact, we noticed that practically all "natural" Prolog programs that use negation are restricted.

## 7. Related Work

We conclude by briefly discussing related work.

There is an enormous literature on the subject of negation in logic programming, see, e.g., the references in the surveys cited in the introduction. However, to our knowledge, no work has been done on negation in Prolog.

The use of the ambivalent syntax was first advocated in mathematical logic by Richards [Ric74], in the logic programming setting by Kalsbeek [Kal93] and Jiang [Jia94], and in the programming languages area by Chen, Kifer and Warren[CKW89] in their language proposal HiLog. In each of these references different versions of ambivalence are assumed. For example, in Kalsbeek [Kal93] atoms can appear as terms and in Jiang [Jia94] formulas can appear as terms. Here we use an alternative version of ambivalence, which amounts to identification of atoms and terms, though we also allow meta-variables.

The definition of the LDNF-resolution given in Section 3 is derived from the definition of the SLDNF-resolution provided in Apt and Doets [AD94]. An alternative definition of SLDNF-resolution was given earlier by Martelli and Tricomi [MT92]. Both definitions overcome problems encountered in the original definition of Clark [Cla78].

In our operational semantics of Prolog programs, given in Section 4, we also provided a meaning to the cut operator. The problem of formalizing the meaning of cut has been studied in a number of publications during the last 10 years. Jones and Mycroft [JM84] defined various semantics for Prolog with cut. This work was pursued by Arbab and Berry [AB87], Debray and Mishra [DM88], and more recently by Lilly and Bryant [LB92].

In the literature several alternatives to the cut operator have been proposed – see e.g. Moss [Mos86] and more recent Hill, Lloyd and Shepherdson [HLS90].

## References

[AB87] B. Arbab and D.M. Berry. Operational and denotational semantics of Prolog. *Journal of Logic Programming*, 4(4):309–329, 1987.

[AB94] K.R. Apt and R. Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 19-20:9–71, 1994.

[AD94] K.R. Apt and K. Doets. A new definition of SLDNF-resolution. *Journal of Logic Programming*, 18(2):177–190, 1994.

[Apt94] K. R. Apt. Program verification and Prolog. In E. Börger, editor, *Specification and Validation methods for Programming languages and systems.* Oxford University Press, 1994. To appear.

[CKW89] W. Chen, M. Kifer, and D.S. Warren. Hilog: A first-order semantics for higher-order logic programming constructs. In *Proceedings of the North-American Conference on Logic Programming*, Cleveland, Ohio, October 1989.

[Cla78] K.L. Clark. Negation as failure. In H. Gallaire and G. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[Dix93] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In Andre Fuhrmann and Hans Rott, editors, *Logic, Action and Information. Proceedings of the Konstanz Colloquium in Logic and Information (LogIn '92)*. DeGruyter, 1993.

[DM88] S.K. Debray and P. Mishra. Denotational and operational semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91, 1988.

[GHK+80] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M.W. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, 1980.

[HLS90] P.M. Hill, J.W. Lloyd, and J.C. Shepherdson. Properties of a pruning operator. *Journal of Logic and Computation*, 1(1):99–143, 1990.

[Jia94] Y. Jiang. Ambivalent logic as the semantic basis fo metalogic programming: I. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming*, pages 387–401. MIT Press, June 1994.

[JM84] N.D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *International Symposium on Logic Programming*, pages 281–288, 1984.

[Kal93] M. Kalsbeek. The vanilla meta-interpreter for definite logic programs and ambivalent syntax. Technical Report CT-93-01, Department of Mathematics and Computer Science, University of Amsterdam, The Netherlands, 1993.

[LB92] A. Lilly and B.R. Bryant. A prescribed cut for Prolog that ensures soundness. *Journal of Logic Programming*, 14(4):287–339, 1992.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation – Artificial Intelligence. Springer-Verlag, 1987. Second, extended edition.

[Mos86] C. Moss. Cut & Paste – defining the impure primitives of Prolog. In E. Shapiro, editor, *Proceedings of the International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 686–694. Springer Verlag, 1986.

[MT92] M. Martelli and C. Tricomi. A new SLDNF-tree. *Information Processing Letters*, 43(2):57–62, 1992.

[Ric74] B. Richards. A point of reference. *Synthese*, 28:431–445, 1974.