On the Practical Use of Negation in a Prolog Compiler

Juan José Moreno-Navarro, Susana Muñoz-Hernández

Abstract

The inclusion of negation among the logical facilities of LP has been a very active area of research, and several techniques have been proposed. However, the negation capabilities accepted by current Prolog compilers are very limited. In this paper, we discuss the possibility to incorporate some of these techniques in an efficient way in a Prolog compiler. Our idea is to mix some of the existing proposals guided by the information provided by a global analysis of the source code.

Keywords: Negation in Logic Programming, Constraint Logic Programming, Program Analysis, Implementations of Logic Programming.

1 Introduction

The desire to write logic programs that use negation is as old as logic programming. There are many reasons for this necessity: On one hand negation plays a very important role for knowledge representation and most of its uses cannot be simulated with positive programs. On the other hand, the possibility to use negation can contribute to include all the capabilities of logic in a programming language (what also includes equality, computable functions and higher order logic).

For these reasons, the research community on negation in LP has made a lot of efforts to propose different ways to understand and incorporate negation into programming languages. Most of the interesting proposals rely on semantics, and a considerable amount of papers in logic programming conferences are devoted to these subjects. Surprisingly, only a small subset of these ideas have arrived to the field on implementation and have produced modifications to Prolog compilers. In fact, the negation capabilities incorporated by current Prolog compilers are rather limited. To our knowledge, the only negation techniques that are present in a Prolog compiler are:

- The (unsound) negation as failure rule, that is present in most Prolog compilers (Sicstus, Bin Prolog, Quintus, etc.).
- The sound (but incomplete) delay technique of Gödel or Nu-Prolog that applies negation as failure when the variables of the negated goal are ground. It is well known that it has the risk of floundering.
- The constructive negation of Eclipse, that was announced in earlier versions but has been removed from recent releases.

LSIIS, F. Informática, U.P.M., Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain, E-mail: `jjmoreno@fi.upm.es`, URL: `http://lml.ls.fi.upm.es/~jjmoreno`. E-mail: `susana@lml.ls.fi.upm.es`. Work partially supported by the spanish project TIC96.1012-C02-02.

This paper is devoted to study the possibility to go a bit further than these experiences, and to design the steps needed to extend a Prolog compiler with a negation subsystem.

The paper does not try to propose any new method, but to combine existing techniques to make some negation techniques useful for practical application. The novelty appears in the techniques used for the combination and the combination strategy.

At the moment, we are interested in techniques with a single and simple semantics. For this reasons we adopt the most simple possibility: Closed Word Assumption (CWA) [10] by program completion and Kunen's 3-valued semantics [14]. With respect to the techniques used, they need to share these semantics. Another important issue is that they must be "constructive", i.e. the program execution needs to search for the values that make a negated goal false. One can argue that Chan's constructive negation fulfil both points, but it is quite difficult to implement and expensive in terms of execution resources. So, our idea is to try to use the simplest technique as possible in any particular case. To help on this distinction, we need to use some tests to characterize the situation. To avoid the execution of these tests during the execution of the program, the results of a global analysis of the source code are used. The program analyses includes groundness detection, elimination of delays, and the determination of the finiteness of the number of solutions. All these analyses are incorporated in the CIAO compiler [6], an extension of Sicstus Prolog, and it will be used as the test bench for our proposals.

The rest of the paper is organized as follows. Section 2 presents some preliminaries: we discuss the negation techniques to be used, and briefly enumerates the characteristics of the program analyses. In order to present how the techniques can be introduced in a Prolog compiler, we start with the management of disequality constraints (Section 3), then we discuss the implementation of (a part of) constructive negation (Section 4). Intensional negation as well as the computation of universal quantified goals are studied in Section 5. Section 6 explains how to combine all the techniques. Finally, we conclude.

2 Preliminaries

In this section we introduce some previous work on negation and program analysis that will be used along the paper.

2.1 Treatment of Negation

Among the techniques that have been proposed to implement the computation of negated goals of the form $\neg Q$ in a program P based on the CWA, the most promising are the following:

- The negation as finite failure rule of Clark [10], which states that $\neg Q$ is a consequence of a program P if a finitely failed SLD tree for the query Q with respect to P exists, (in short, if Q finitely fails). The implementation provided by Prolog compilers is the following:

  ```
  naf (Q) :- Q, !, fail.
  naf (Q).
  ```

  that is unsound except if it is used for ground goals (i.e. Q has no free variables).
- There are many works related to the program completion of Clark [10], (see [16], [1]), some of them ([3, 4]) oriented to obtain a program that is a transformation of a original program P which introduce also the "only if" part of the predicate definitions (i.e., interpreting implications as equivalences).
- The constructive negation proposed by Chan [8, 9], and formalized in the context of CLP by Stuckey [21].

2.2 Information obtained by global analysis

In order to provide some heuristics to guide the computation of the negation process we will use some techniques of global analysis. Along the paper we assume that a part of the Prolog compiler produces to same degree of accuracy, the following information. At least this is true for the CIAO system:

- Groundness: The groundness analysis tries to identify those variables that are bound to a ground term in a certain point of the program. There are several papers and implementation of groundness analysis (see, for instance, [17]).
- Elimination of delays: In the presence of delays (or waits) the analysis tries to identify which of them are useless (so, removing them) or if there is a reordering of the goals that does not need the delay directive. See [12] for a reference.
- Finiteness of the number of solutions: The analysis is based on complexity and execution cost to determine if a goal has a finite number of solutions (even zero) or there are a potential infinite number of answers. The interested reader can consult [15, 2].

3 Management of disequality constraints

The first step in our management of negation is to handle disequalities between terms $t_1 \neq t_2$. Most Prolog implementations can work with disequalities if both terms are ground (built-in predicate `/==`). However, they cannot work in the presence of free variables. The "constructive" behaviour must allow the "binding" of a variable with a disequality: the solution to the goal `X /== t` is the constraint $X \neq t$. In fact, what we need is a implementation of CLP ($\mathcal{H}$) (constraints over the Herbrand Universe with equality and disequality). This capability is present in several CLP Prolog extensions (Prolog III for instance), but is not available in usual Prolog compilers. As we are going to prove, the inclusion of disequalities and constrained answers has a very low cost.

First of all, we need a representation for constraint answers. The disequation $c(X, a) \neq c(b, Y)$ introduces a disjunction $X \neq b \vee Y \neq a$. For this reason, we use conjunctions of disjunctions of disequations as normal forms. On the other hand, we will produce disequations by means of the negation of a equation $X = t(\overline{Y})$. This fact produces the universal quantification of the free variables in the equation, unless a more external quantification affects them. The negation of such equation is $\forall\ \overline{Y}\ X \neq t(\overline{Y})$. Also, universally quantified disequations are allowed in the constraints. More precisely, the normal form of constraints is:

$$\underbrace{\bigwedge(X_i = t_i)}_{\text{positive information}} \quad \underbrace{\wedge \bigvee_j \forall\ \overline{Z}_j^1\ (Y_j^1 \neq s_j^1) \wedge \ldots \wedge \bigvee_l \forall\ \overline{Z_j^n}\ (Y_l^n \neq s_l^n)}_{\text{negative information}}$$

where each X_i appears only in $X_i = t_i$, none s_k^r is Y_k^r and the universal quantification could be empty (leaving a simple disequality).

It is easy to redefine the unification algorithm to manage constrained variables. This very compact way to represent a normal form was firstly presented in [18] and differs from Chan's representation where only disjunctions are used[1].

Therefore, in order to include disequalities into a Prolog compiler we need to reprogram unification. It is possible if the Prolog version allows attributed variables ([7] for

[1] Chan treats the disjunctions by means of backtracking. The main advantage of this normal form is that the search space is drastically reduced.

instance, in Sicstus Prolog or in Eclipse, where they are called meta-structures). Such these variables let us keep associated information with each variable during the unification and that is what can be used to dynamically control the constraints.

Attributed variables are variables with an associated attribute, which is a term. We will associate to each variable a data structure containing a normal form constraint. They behave like ordinary variables, except that the programmer can supply code for unification, printing facilities and memory management. In our case, the printing facility is used to show constrained answers. The main task is to provide a new unification code.

Once the unification of a variable X with a term t is triggered, there are three possible cases (up to commutativity):

1. if X is a free variable and t is not a variable with a negative constraint, X is just bind to t,
2. if X is a free variable or bound to a term t' and t is a variable Y with a negative constraint, we need to check if X (or, equivalently, t') satisfy the constraint associated with Y. A predicate `satisfy` is used for this purpose.
3. if X is bound to a term t' and t is a term (or a variable bound to a term), the classical unification algorithm can be used.

A predicate `=/=`, to check disequalities, is defined in a similar way than unification. The main difference is that it incorporates negative constraints instead of bindings and the decomposition step (not studied before) can produce disjunctions.

As an example, let us show the constraints produced in certain situations. The attribute/constraint of a variable is represented as a list of list of pairs (variable, term) using a constructor `/`, i.e. the disequality $X \neq 1$ is represented as `X / 1`. When an universal quantification is used in a disequality (e.g. $\forall Y\ X \neq c(Y)$) the new constructor `fA`/2 is used (the previous constraint is represented as `fA (Y, X / c (Y))`. The first list is used to represent disjunctions while the list inside represents the conjunction of disequalities. We focus on the variable X.

SUBGOAL	ATTRIBUTE	CONSTRAINT
¬ `member (X,[1,2,3])`	$[[X/1, X/2, X/3]]$	$X \neq 1 \wedge X \neq 2 \wedge X \neq 3$
¬ `member (X,[1,2,3]), X =/= 2`	$[[X/1, X/3]]$	$X \neq 1 \wedge X \neq 3$
`member (X,[1]), X =/= 1`	`fail`	$false$
`X =/= 4`	$[[X/4]]$	$X \neq 4$
`X =/= 4; (X =/= 6, X =/= Y`	$[[X/4], [X/6, X/Y]]$	$X \neq 4 \vee (X \neq 6 \wedge X \neq Y)$
`member (X,[0,s(0),s(s(0))]),` `fA (Y, X =/= s (Y))`	0	$X = 0$

4 Constructive negation

The second technique we are going to implement is constructive negation. Constructive negation was proposed by Chan [8, 9] and it is widely accepted as the "most promising" method to handle negation with Kunen's 3-valued semantics (up to some extensions and modifications proposed by other authors). Although the first Chan's paper is credited as the presentation of the idea, but a "mistake" in the development of the technique (solved in the second one), it has still some interesting results from the implementation point of view. The main idea of constructive negation is easy to describe: in order to obtain the solutions of $\neg Q$ we proceed as follows:

1. Firstly, the solutions of Q are obtained getting a disjunction:
   $Q \equiv S_1 \vee S_2 \vee ... \vee S_n$

   Each of the component S_i can be understood as a conjunction of equalities:
   $S_i \equiv S_i^1 \wedge S_i^2 \wedge \ldots \wedge S_i^{m_i}$
2. Then the formula is negated and rearranged to obtain a normal form constraint:

$$\begin{array}{lll} \neg Q \equiv & \neg(S_1 \vee S_2 \vee \ldots \vee S_n) & \equiv \\ & \neg S_1 \wedge \neg S_2 \wedge \ldots \wedge \neg S_n & \equiv \\ & \neg(S_1^1 \wedge \ldots \wedge S_1^{m_1}) \wedge \ldots \wedge \neg(S_n^1 \wedge \ldots \wedge S_n^{m_n}) & \equiv \\ & (\neg S_1^1 \vee \ldots \vee \neg S_1^{m_1}) \wedge \ldots \wedge (\neg S_n^1 \vee \ldots \vee \neg S_n^{m_n}) & \end{array}$$

   The formula can be obtained in a different way depending on how we negate a solution. It can also be arranged in a disjunction of conjunctions according with the variables in each S_i^j.

Of course, the solution is not valid in general, because a goal can have an infinite number of solutions. [8] offers a technique to negate a solution and to normalize the previous formula. [18], working on a CLP framework as proposed by [21], adapted the idea (in a different but equivalent context) using our notion of constraint normal form. Given a constraint

$$\bigwedge_{i=1}^{m}(X_i = t_i) \wedge \bigwedge_{j=1}^{n} \bigvee_{k=1}^{n_j} \forall\, \overline{Z}_k^j\ (Y_k^j \neq s_k^j)$$

the negation will produce the following constraints:

- $\bigvee_{1=1}^{m} \forall\, \overline{Z}_i\ (X_i \neq t_i)$, where $\overline{Z}_i$ are the variables of t_i that are not quantified outside.
- $\bigwedge_{i=1}^{m}(X_i = t_i) \wedge \bigwedge_{k=1}^{n_l}(Y_l^k = s_l^k) \quad \wedge \quad \bigwedge_{j=1}^{l-1} \bigvee_{k=1}^{n_j} \forall\, \overline{Z}_k^j\ (Y_k^j \neq s_k^j)$, for all $l < n$.

Notice again that we are using a much more compact representation for the negated constraints than that proposed in [8].

Once we have explained how to negate a constraint, the rest is easy: for each solution, each possibility of the negation is combined with one of the others. All these different solutions are obtained by backtracking.

The implementation of $\neg Q$ (in Prolog `cneg (Q)`) works as follows:

1. First of all, all variables V of the goal Q are obtained.
2. Secondly, all the solutions of Q for variables in V are computed using `setof`/3. Each solution is a constraint in normal form.
3. The negation of each solution is computed and combined to obtain an answer to $\neg Q$ one by one.

Of course, this implementation is only valid when it is detected that the goal has a finite number of solutions. The full code ([19]) is available from the authors on request.

Some examples were this technique is useful are the following. They are extracted from a running session.

```
| ?- cneg (X =/= Y).
     X = Y ?;
     no
| ?- cneg (X=/=Y, member(Y,[1,2])).
     Y /= 1, Y /= 2 ?;
     X = 1, Y = 1 ?;
     X = 2, Y = 2 ?;
     no
| ?- cneg(member(Y,[X]),member(Y,[2])).
     X /= 2; Y /= 2 ?;
     no
```

```
| ?- cneg (X =/= X).
     true ?  ;
     no
| ?- cneg (cneg (X =/= X)).
     no
| ?- cneg(member(X,[1,2,3])).
     X /= 1, X /= 2, X /= 3 ?  ;
     no
| ?- cneg ([1,X] =/= [Y,2]).
     Y = 1, X = 2 ?;
     no
```

5 Intensional negation and universal quantification

Intensional negation [3, 4] uses a different approach to handle negation. A program transformation technique is used to add new predicates to the program in order to express the negative information. Informally, the *complement* of head terms of the positive clauses are computed and they are used later as the head of the negated predicate.

For example, if we have the program:

```
even (0).
even (s(s(X))) :- even (X).
```

the transformation produces a new predicate `not__even` that succeeds when `even` fails.

```
not__even (s(0)).
not__even (s(s(X))) :- not__even (X).
```

There are two problems with this technique. The first one is that in the presence on logical variables in the rhs of a clause, the new program needs to handle some kind of universal quantification construct. The second trouble is that, while the new program is semantically equivalent to the completed program, the operational behaviour can differs. In the presence of logical variables, the new predicate can generate all the possible values one by one, even when a more general answer can be given. The predicate:

```
p (X, X).
```

is negated by:

```
not__p (X, Y) :-                not__eq (0, s(Y)).
          not__eq (X, Y).       not__eq (s(X), 0).
                                not__eq (s(X), s(Y)) :- not__eq (X, Y).
```

if the program only contains natural numbers with `0` and `succ`. The query `not__p (X, Y)` will generate infinitely many answers, instead of the more general $X \neq Y$. An answer of the form $X \neq Y$ can only be replaced by an infinite number of equalities.

Our approach to manage this problem is to use constraints instead of concrete terms. All what we need is to have disequality constraints, what are yet included. So, the negated predicates of the previous examples, with our transformation, are the following:

```
not__even (X) :- X =/= 0, fA (Y, X =/= s(s(Y))).
not__even (s(s(X))) :- not__even (X).

not__p (X, Y) :- X =/= Y.
```

Notice that if the program only contains natural numbers, the first clause is equivalent to the one obtained above.

A bit more complicate is the first problem. For this purpose we have implemented a predicate `for_all`/2 that tries to detect if a goal Q is valid for all possible values of a list of variables $[X_1, \ldots, X_n]$ with a call `for_all ([X1, ..., Xn], Q)`.

Roughly speaking, the implementation, explained in more detail later, instantiates incrementally the variables $X_1, \ldots, X_n$ and tries to make Q true without further instantiation. The idea was sketched in [4] but without a concrete implementation.

5.1 Program transformation

We are going to present the transformation technique in a different way that [4]. They apply the transformation to a restricted class of programs that are powerful enough to cover all the cases. We prefer to apply the transformation to all kind of programs, what is closer to the behaviour of the compiler.

In order to formally define the negated predicate `not__p` for `p` we proceed step by step. First of all we need the definition of the complement of a term t. Without using constraints, the only way to represent the complement of a term is a set of terms. However, this set can be expressed by means of a constraint on a variable X that does not appear in the term (i.e. the constrained values for X are exactly the terms that are not t.

Definition: *Complement of a term*
The complement of a term t (not using the variable X) on the variable X (in symbols $Comp(t)$) is a constraint value for X, defined inductively as follows:

- $Comp(Y) = fail$
- $Comp(c) = (X \neq c)$, with c constant.
- $Comp(c(t_1, \ldots, t_n)) = \forall\ \overline{Z}(X \neq c(t_1, \ldots, t_n))$, with c a constructor and $\overline{Z}$ the variables of $t_1, \ldots, t_n$.

Without loss of generality we can consider that all the predicates has one argument, taking the tuple construction as a constructor. Given a set of clauses for a predicate p:

$$\begin{array}{l} \mathrm{C}_1\text{: } p(t^1) \ :- G_1. \\ \ldots \\ \mathrm{C}_m\text{: } p(t^m) :- G_m. \end{array}$$

we say that the **complement clause** of the program is

$$not\_\_p(X) \ :- Comp(t^1), \ldots, Comp(t^m).$$

assuming, by adequate renamings, that the terms do not share variables (i.e. $var(t^i) \cap var(t^j) = \emptyset$ for $i \neq j$).

This clause covers the cases where there is no definition for the predicate in the original program, and it must be included in the new program. For the rest of the clauses of the negated predicates some additional concepts are needed:

Definition: *Critical pair*
We say that a program has a critical pair t in $\{l_1, \ldots, l_r\} \subseteq \{1, \ldots, m\}$ if

$$m.g.u.(t^{l_1}, \ldots, t^{l_r}) = \sigma \text{ and } t = t^{l_1}\sigma$$

The definition is well know in term rewriting and, intuitively, detects the terms for which there are more than one clause applicable. In those cases, all the bodies of the applicable clauses must be negated together.

For each critical pair t in $\{l_1, \ldots, l_r\}$ of the program we generate the following clause:

$$not\_\_p(t) \ :- negate\_body(var(t), (G_{l_1}; \ldots; G_{l_r})).$$

where the *negate_body* function negates a body clause (see below). There is no rule if all the G_{l_j} are empty. Notice that the formula $p(t) \longleftrightarrow G_{l_1} \vee \ldots \vee G_{l_r}$ is part of the completed program.

Now, we are in a position to transform each of the clauses of the program. Each clause

$p(t^i) : -G_i.$

generates one of the following clauses for not_p:

- $not\_p(t^i) : -t^i = / = t, negate\_body(var(t^i), G_i).$

  if there is a critical pair t involving clause i.
- $not\_p(t) : -negate\_body(var(t^i), G_i).$ otherwise. There is no clause if G_i is empty.

The effect of *negate_body* is easy to explain. It just negates the body and introduces universal quantifications when they are needed:

- $negate\_body(V, G) = negate(G)$ if $var(G) \subset V$
- $negate\_body(V, G) = \texttt{for\_all}([Y_1, \ldots, Y_k], negate(G))$
  if $var(G) - V = \{Y_1, \ldots, Y_k\} \neq \emptyset$.

The function *negate* can be defined inductively: it moves conjunction (`,`) into disjunction (`;`), disjunction into conjunction, equality into disequalities and vice versa.

The effect of *negate* over a single predicate call needs a further discussion. In principle it is possible to define $negate(q(s))$ by any of the previous methods: negation as failure (`naf (q (s))`), constructive negation (`cneg (q (s))`) or the transformed predicate (`not__q (s)`). The last one will be used in case of recursive calls. The decision will be fixed by the negation strategy of the compiler that will be discussed in the next section.

The transformation has some similarities with the one proposed in [5]. Although we have only found the paper very recently, we can mention some differences. [5] proposal has a much more simple formulation and some optimization covered by our detection of critical pairs are not taken into account. The result is that much more universally quantified goals are generated and the programs contains a lot of trivial constraints (i.e. they are trivially true or false, as $X = a \ \wedge \ X \neq a$, or $X = a \ \vee \ X \neq a$).

Let us discuss the application of the method in a couple of examples. Consider the fragment of the program:

```
less (0, s(Y)).
less (s(X), s(Y)) :- less (X, Y).
```

First of all, we need to compute the complement of the terms in the head of the clauses, with respect to the variable pair (W, Z). We have:

- $Comp(0, s(Y)) = (W \neq 0, \forall\, Y(Z \neq s(Y)))$
- $Comp(s(X), s(Y)) = (\forall\, X(W \neq s(X)), \forall\, Y(Z \neq s(Y)))$

The complement clause is:

```
not__less (W, Z) :- W =/= 0, fA (X, W =/= s (X)), fA (Y, Z =/= s (Y)).
```

There are no critical pairs, so the transformed clause is (the first one has no body):

```
not__less (s (X), s (Y)) :- not__less (X, Y).
```

The second example is also well known, and includes free variables in the body:

```
parent(john,mary).      ancestor(X,Y) :- parent(X,Y).
parent(john,peter).     ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y)
parent(peter,susan).
```

The transformation of the predicate ancestor has no complement clause. The first clause and the second clause have an obvious critical pair (X, Y). The clause for it is:

```
not__ancestor (X, Y) :- cneg (parent (X,Y)),
                        for_all ([Z], (not__ancestor (X, Z);
                                       cneg (parent (Z,Y))).
```

Notice that we have used `cneg` as the way to negate the predicate `parent`. It is safe because `parent` has always a finite number of solutions. In the case we can infer that a call to $\neg parent$ is ground, we can use `naf` instead of `cneg`. Notice that the call inside the `for_all` goal has the first argument ground for sure.

In principle, we need to transform each of the clauses, including the constraint $(X, Y) = / = (X, Y)$ in their bodies, but it is trivially unsatisfiable and we can omit the clauses.

5.2 Implementation of universal quantification

The efficient implementation of universally quantified goals is not an easy task. However, we are only interested in a particular use of this quantification: that which comes from the previous transformation.

There are some other approaches to implement some kind of universal quantification, although they are rather limited:

1. Nu-Prolog [20] and Göedel [13] include universally quantified goals, but they are executed when all the variables are ground. Obviously it has the risk of floundering.
2. Voronkov [22] has studied the use of bounded quantifications over finite sets.

Our implementation is based on two ideas:

1. A universal quantification of the goal Q over a variable X succeeds when Q succeeds without binding (or constraining) X.
2. A universal quantification of Q over X is true if Q is true for all possible values for the variable X.

The second point can be combined with the first one in order to get an implementation. Instead of generating all possible values (which is not possible in the presence of a constructor of arity greater than 0) we can generate all the possible esqueletons of values, using new variables. The simplest possibility is to include all the constants and all the constructors applied to fresh variables. Now, the universal quantification is tested for all this terms, using the new variables in the quantification.

In order to formalize this concept, we need the notion of **covering**.

Definition: *Covering of the Herbrand Universe*
A covering is any set of terms $\{t_1, \ldots, t_n\}$ such that:

- For every i, j with $i \neq j$, t_i and t_j do not superpose, i.e. there is no ground substitution σ with $t_i\sigma = t_j\sigma$.
- For all ground term s of the Herbrand Universe there exists i and a ground substitution σ with $s = t_i\sigma$.

The simplest covering is a variable $\{X\}$. If the program only uses natural numbers, the following sets are coverings: $\{0, s(X)\}$, $\{0, s(0), s(s(X))\}$ $\{0, s(0), s(s(0)), s(s(s(X)))\}$, ...

The example also gives us the hint about how to incrementally generate coverings. We depart from the simplest covering X. From one covering we generate the next one choosing one term and one variable in this term. The term is removed and then we add all the terms obtained replacing the variable by all the possible instances of that element.

In order to fix a strategy to select the term and the variable we use a Cantor's diagonalization[2] to explore the domain of a set of variables. It is a breadth first strategy to cover every element of the domain. The previous concepts extend trivially in the case of tuple of elements of the Herbrand Universe, i.e. several variables.

The implementation of the `for_all ([X1, ..., Xn], Q)` predicate follows the previous ideas. We start with the initial covering $\{(X_1, \ldots, X_n)\}$.

The actual covering is checked with the predicate Q. This means that for each element $\overline{t}$ in the covering we execute Q replacing the variables $(X_1, \ldots, X_n)$ by $\overline{t}$. We have two possibilities:

1. Q succeeds in all the cases without any bind of the variables introduced by the covering. Then the universal quantification is true.
2. Q fails in a case without any attempt to use a quantified variable. Then the universal quantification is false for sure.
3. Q fails in at least one of the cases attempting to use a quantified variable. The next covering is generated and the process continues recursively.

There are two important details that optimize the execution. The first one is that in order to check if there are bindings in the covering variables, it is better to replace them by new constants that do not appear in the program. In other words, we are using "Skolem constants".

The second optimization is much more useful. Notice that the coverings grow up incrementally, so we only need to check the most recently included terms. The other ones have been checked before and there are no reason to do it again.

As an example, consider the sequence of coverings for the goal $\forall\ X, Y, Z\ p(X, Y, Z)$ in a program using only natural numbers. `Sk (i)`, with i a number represents the ith Skolem constant.

$C_1 = [(Sk(1), Sk(2), Sk(3))]$
$C_2 = [\underline{(0, Sk(1), Sk(2))}, \underline{(s(Sk(1)), Sk(2), Sk(3))}]$
$C_3 = [\underline{(0, 0, Sk(1))}, \underline{(0, s(Sk(1)), Sk(2))}, (s(Sk(1)), Sk(2), Sk(3))]$
$C_4 = [\underline{(0, 0, 0)}, \underline{(0, 0, s(Sk(1)))}, (0, s(Sk(1)), Sk(2)), (s(Sk(1)), Sk(2), Sk(3))]$
$C_5 = [(0, 0, 0), \underline{(0, 0, s(0))}, \underline{(0, 0, s(s(Sk(1))))}, (0, s(Sk(1)), Sk(2)), (s(Sk(1)), Sk(2), Sk(3))]$
$C_6 = \ldots$

In each step, only two elements need to be checked, those that appear underlined. The rest are part of the previous covering and they do not need to be checked again. Again, the authors can supply details of the code [19].

Let us show some examples of the use of the `for_all` predicate, indicating the covering found to get the solution. We are still working only with natural numbers:

```
| ?- for_all ([X], even (X)).
     no
```

with covering $\{0, \underline{s(0)}, s(s(Sk(1))\}$.

```
| ?- for_all ([X], X =/ a).
     yes ?  .
```

with covering $\{Sk(1)\}$.

```
| ?- for_all ([X], less (0, X) -> less (X, Y)).
     Y = s (s(_A)) ?.
```

[2]This is a method to enumerate $I\!N^m$. It ensures that all elements are visited in a finite number of steps.

with covering $\{0, s(Sk(1))\}$.

Actually, this solution does not guarantee completeness of the query evaluation process. There are some cases when the generation of coverings does not find one which is correct or incorrect. Nevertheless, this solution fails to work properly in very particular cases. Remember that we are not interested in giving the user an universal quantification operator, but just to implement the code coming from the tranformation of a negated predicate.

5.3 Behaviour of the application of the technique

Here we have some examples coming from a running session that show the behaviour of the transformation technique:

```
|?- not_even (s(s(0))).
   no
|?- not_even (s(s(s(0)))).
   yes ?;
   no
|?- not_even(X).
   X =/= 0, X =/= s (s (_A)) ? ;
   X=s(s(Y)),Y=/=0,Y=/=s(s(_A)) ? ;
   ⋮
|?- not_less (0, s(X)).
   no
|?- not_less (s(X), 0).
   true ?  ;
   no
```

```
|?- not_less(s(X), X).
   ^C
   Prolog interruption (h for help)?  a
   {Execution aborted}
|?- not_ancestor (mary, peter).
   yes ?  ;
   no
|?- not_ancestor (john, X).
   no
|?- not_ancestor (peter, X).
   X = john ?  ;
   X = mary ?  ;
   X = joe ?  ;
   X = peter ?  ;
   no
```

The divergence of the goal `not_less(s(X), X)` is of the same nature of the divergence of `less (X, s (X))` and is related to the incompleteness of Prolog implementations.

In any case, our implementation provides only sound results, although there are cases where we cannot provide any result.

6 The compiler strategy

Once we have described the main implemented methods, we can discuss the most important part: the combination of these techniques in order to get a system to handle negation.

What we need is a strategy that the compiler can use to generate code for a negated goal. The strategy is fixed by the information of the different program analyses. Notice that the strategy also ensures the soundness of the method: if the analysis is correct, the precondition to apply a technique is ensured, so the results is sound.

Given a (sub)goal of the form $\neg G(\overline{X})$ the compiler produces one of the following codes:

1. If the analysis of the program ensures that $G(\overline{X})$ is ground then simple negation as failure is applied, i.e. it is compiled to `naf` $(G(\overline{X}))$. Since floundering is undecidable, the analysis only provides an approximation of the cases where negation as failure can be applied safely. This means that maybe we are avoiding to use the technique even in cases that it could work properly.
2. Otherwise, the compiler generates a new program replacing the goal by $G(\overline{X})$ and adding a delay directive to get ground variables in $\overline{X}$ before the call. Then, the

compiler applies the elimination of delays technique. If the analysis and the program transformation are able to remove the delay (maybe moving the goal), use the outcoming program but replace $G(\overline{X})$ by `naf` $(G(\overline{X}))$ as before. Again, the approximation of the analysis could forbid us to apply constructive negation in cases it should give a sound result.

3. Otherwise, look for the result of the finiteness analysis. If it ensures that $G(\overline{X})$ has a finite number of solutions, then the compiler can use simple constructive negation, transforming the negated goal into `cneg` $(G(\overline{X}))$.
4. Otherwise, the compiler uses the intensional negation approach. Some negated predicates are generated and the goal is replaced by $negate(G(\overline{X}))$.

   During this process new negated goal can appear and the same compiler strategy is applied to each of them.

The strategy is incomplete, in the sense that the last step does not ensure to produce a result. However, if a result is got, it is correct.

Up to now, we have applied this strategy manually (using the real results of the analyzers) and a good collection of programs with negation has been checked. As a future work, we plan to modify the CIAO compiler in order to implement this strategy.

7 Conclusion

We have presented a collection of techniques, more or less well known in logic programming, that can be used togheter in order to produce a system that can handle negation efficiently. Although we do not claim to invent any "new method" to handle negation, to our knowledge it is one of the first serious attempts to include such proposals on the incorporation of negation into a Prolog compiler.

Our main contribution is to use the information of a program analyzer to design a strategy to organize the use of the negation components.

Some other contributions of the paper are the management of disequality constraints using attributed variables and the new compact way to handle constraint normal forms, which has a number of advantages.

The transformation approach in term of disequality constraint is another important point, because it solves some of the problems of intensional negation in a more efficient way than [5].

Finally, the approach to compute universally quantified goals was sketched in [4], but the concrete implementation needs to solve a lot of technical difficulties, what makes our implementation more than a student exercise.

The results of the practical experimentation are quite acceptable on time. However, we cannot compare with existing methods, so the runtimes are not useful.

As a future work, we plan to modify the compiler in order to produce a version of CIAO with negation. It will give us a real measure of what important is the information of the analyzer to help our strategy. On the other hand, there are still some unsolved problems. The most important is the detection of the cases where the universal quantification does not work. Probably, in such these cases we will need to use full constructive negation, hard to implement and, probably, not very efficient. The work of [11] could help on this task. In any case, it will be the last resource to be used.

References

[1] K. R. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. 3, pp. 493–574, Elsevier, New York, 1990.

[2] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality Analysis of Prolog. In *I. S. on Logic Programming*, pages 457–471. The MIT Press, 1994.

[3] R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. Intensional Negation of Logic Programs. *Springer LNCS*, 250:96–110, 1987.

[4] R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. A Transformational Approach to Negation in L P. *J. Logic Programming*, 8(3):201–228, 1990.

[5] P. Bruscoli, F. Levi, G. Levi, and M.C. Meo. Compilative Constructive Negation in Constraint Logic Programs. *CAAP'94, Springer LNCS*, 1994.

[6] F. Bueno. *The CIAO Multiparadigm Compiler: A User's Manual*, 1995.

[7] M. Carlsson. Freeze, Indexing and other Implementation Issues in the WAM. In *ICLP'97*, pp. 40–58. The MIT Press, 1987.

[8] D. Chan. Constructive Negation Based on the Complete Database. In *Proc. ICLP'88*, pp. 111–125. The MIT Press, 1988.

[9] D. Chan. An Extension of Constructive Negation and its Application in Coroutining. In *Proc. NACLP'89*, pages 477–493. The MIT Press, 1989.

[10] K. L. Clark. Negation as failure. In J. Minker H. Gallaire, editor, *Logic and Data Bases*, pp. 293–322, New York, NY, 1978.

[11] W. Drabent. What is a Failure? An Approach to Constructive Negation. *Acta Informatica*, 33:27–59, 1995.

[12] M. García de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. In *ILPS'95*, pp. 417–431. The MIT Press, 1995.

[13] P.M. Hill, J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[14] K. Kunen. Negation in Logic Programming. *J. of Logic Programming*, 4:289–308, 1987.

[15] P. López-García, M. Hermenegildo, S. Debray, and N. W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 ILPS'97*. The MIT Press, 1997.

[16] J. W. Lloyd. *Foundations of Logic Programing, 2nd edition*. Springer, 1987.

[17] K. Muthukumar, M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *J. Logic Programming*,13(2/3):315–347,1992.

[18] J.J. Moreno-Navarro. Default rules: An Extension of Constructive Negation for Narrowing-based Languages. In *ICLP'94*, pages 535–549. The MIT Press, 1994.

[19] S. Munoz. Algunas técnicas para el tratamiento de información negativa en Prolog. Master's thesis, Facultad de Informática, UPM, 1997.

[20] L. Naish. Negation and Quantifiers in NU-Prolog. In *ICLP'86*, 1986.

[21] P. Stuckey. Negation and Constraint Logic Programming. *Information and Computation*, 118:12-23, 1995.

[22] A. Voronkov. Logic Programming with Bounded Quantifiers. In A. Voronkov, editor, *First Russian Conference on Logic Programming*, Springer LNAI 592, pp. 486–514, 1992.