Constructive Negation for Prolog: A Real Implementation

Susana Muñoz and Juan José Moreno-Navarro

LSIIS, Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo s/n, Boadilla del Monte
28660 Madrid, Spain *
{susana,jjmoreno}@fi.upm.es

Abstract. Logic Programming has been advocated as a language for system specification, specially those involving logical behaviors, rules and knowledge. However, modeling problems involving negation, quite natural in many cases, has some limitations if Prolog is used as the specification/implementation language. The restrictions do not come from the theoretical viewpoint, where the user can find many different models with the corresponding semantics, but from practical implementation issues. The negation capabilities supported by current Prolog systems are rather limited from the constructive point of view. In this paper, we refine and propose some extensions to the method of constructive negation, providing the complete theoretical algorithm, and we also discuss about implementation issues providing a preliminary implementation.

Keywords Constructive Negation, Negation in Logic Programming, Constraint Logic Programming, Implementations of Logic Programming.

1 Introduction

From its very beginning Logic Programming has been advocated to be either a programming language and a specification language. It seems to be natural to use Logic Programming for specifying/programming systems involving logical behaviors, rules and knowledge. However, this idea has a strong limitation: the use of negation. Probably, negation is the most significant aspect of logic that was not included from the outset. This is due to the fact that dealing with negation involves significant additional complexity. Nevertheless the use of negation is very natural and plays an important role in many cases, for instance management of constraints in databases, program composition, manipulation and transformation, default reasoning, etc.

This restriction cannot be perceived from the theoretical point of view because there are many alternative ways to understand and incorporate negation into Logic Programming. The related problems start already at the semantic level

* This work was partly supported by the Spanish MCYT project TIC2000-1632.

and the different proposals (negation as failure (*naf*), stable models, well founded semantics, explicit negation, etc.) differ not only in expressiveness but also in semantics. However, the negation techniques supported by current Prolog compilers are rather limited, restricted to Negation as failure under the Fitting/Kunen semantics [1] (sound only under some circumstances) that is a built-in or library in most Prolog compilers (Quintus, SICStus, Ciao, BinProlog, etc.), and the "delay technique" (applying negation as failure only *when* the variables of the negated goal become ground, which is sound but incomplete due to the possibility of floundering) that is present in Nu-Prolog, Gödel, and Prolog systems which implement delays (most of those above).

Among all the mentioned proposals, constructive negation is probably the most promising because it is proved to be sound and complete and its semantics is fully compatible with the Prolog one. In fact, constructive negation was announced in early versions of the Eclipse Prolog compiler, but was removed from the last releases. The reasons seem to be related with some technical problems with the use of coroutining (risk of floundering) and the management of constrained answers.

The goal of this paper is to describe a description of constructive negation in an algorithmic way, i.e. making explicit the details needed for an implementation. We want also to discuss the pragmatic ideas needed to provide a concrete and real implementation. Early results for a concrete implementation extending the Ciao Prolog compiler are presented.

The rest of the paper is organized as follows. Section 2 describe in detail our constructive negation algorithm. It tells how to obtain the *frontier* of a goal (Section 2.1), how to prepare it to be negated (Section 2.2) and finally how to negate it (Section 2.3). Section 3 talks about implementation issues: the code expansion (Section 3.1), the disequality constraints (Section 3.2) that are needed, optimizations (Section 3.3), examples (Section 3.4) and some experimental results (Section 3.5). Finally we conclude and sketch some future work.

2 Constructive Negation

Most of the papers devoted to constructive negation deal with semantical aspects. In fact, only the original papers by Chan gave some hints about a possible implementation based on coroutining, but the technique was just sketched. When we have tried to reconstruct it we have found several problems including floundering (in fact it seems to be the reason why constructive negation was removed from recent Eclipse versions) and the management of constrained answers. Our claim is that it is not possible to overcome this problems in an easy and efficient way. Thus, we decided to design an implementation from scratch. One of our additional requirements is that we want to use a standard Prolog implementation, so we will avoid implementation-level manipulations.

We first start with the definition of a frontier and how to manage it for negating the corresponding formula.

2.1 Frontier

We first start with Chan's definition of frontier (in fact, the formal definition is due to Stuckey [2]).

Definition 1. Frontier

A frontier of a goal G is a finite set of nodes in the derivation tree such that every derivation of G is either finitely failed or passes through exactly one frontier node.

What is missing is a method to generate the frontier. Up to now we are using the simplest possible frontier: the frontier of depth 1 obtained by doing all possible single steps of SLD resolution. Simple inspection of the applicable clauses can do this[1]. Additionally, built-in based goals have a special treatment (moving conjunctions into disjunctions, disjunctions into conjunction, eliminating double negations, etc.)

Definition 2. Depth-one frontier

- *If $G \equiv (G_1; G_2)$ then $Frontier(G) \equiv Frontier(G_1) \vee Frontier(G_2)$.*
- *If $G \equiv (G_1, G_2)$ then $Frontier(G) \equiv Frontier(G_1) \wedge Frontier(G_2)$ and then we have to apply the distributive property of DeMorgan to keep the disjunction of conjunctions format.*
- *If $G \equiv p(\overline{X})$ and predicate p/n is defined by N clauses:*

$$p(\overline{X}^1) : -C'_1.$$
$$p(\overline{X}^2) : -C'_2.$$
$$\ldots$$
$$p(\overline{X}^3) : -C'_N.$$

The frontier of the goal has the format: $Frontier(G) \equiv \{C_1 \vee C_2 \vee \ldots \vee C_N\}$, where each C_i is the join of the conjunction of subgoals C'_i plus the equalities that are needed to get the unification between the variables of $\overline{X}$ and the corresponding terms of $\overline{X}^i$.

Consider, for instance, the following code:

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

The frontier for the goal $odd(Y)$ is the following:

$$Frontier(odd(Y)) = \{(Y = s(0)) \vee (Y = s(s(X)) \wedge odd(X))\}$$

[1] Nevertheless, we plan to improve it by using abstract interpretation and detecting the degree of evaluation of a term that the execution will generate.

To obtain the negation of G is enough to negate the frontier formula. This is done by negating each component of the disjunction of all implied clauses (that form the frontier) and combining the results.

The solutions of $cneg(G)$ are the solutions of the combination (conjunction) of one solution of each of the N conjunctions C_i. From now we are going to explain how to negate a single conjunction C_i.

2.2 Preparation

Before negating a conjunction obtained from the frontier we have to simplify, organize, and normalize it:

- **Simplification of the conjunction**. If one of the terms of C_i is trivially equivalent to *true* (e.g. $X = X$) we can eliminate it from C_i, and if one of the terms is trivially *failing* (e.g. $X \neq X$) we can simplify $C_i \equiv fail$. This simplification is carried during the process of generation of terms of the frontier.
- **Organization of the conjunction**. We make three groups with the components of C_i to divide them in equalities, disequalities and the rest of subgoals. Then we obtain $C_i \equiv \overline{I} \wedge \overline{D} \wedge \overline{R}$ where $\overline{I}$ is the set of equalities, $\overline{D}$ is the set of disequalities (that appear explicitly in C_i') and $\overline{R}$ is the rest of subgoals.
- **Normalization of the conjunction**. The set of variables of the goal is called $GoalVars$. The set of free variables of $\overline{R}$ is called $RelVars$.
  - **Elimination of redundant variables and equalities**. If $I_i \equiv X = Y$ where $Y \notin GoalVars$ then we have now the formula $(I_1 \wedge \ldots \wedge I_{i-1} \wedge I_{i+1} \wedge \ldots \wedge I_{NI} \wedge \overline{D} \wedge \overline{R})\sigma$ where $\sigma = \{Y/X\}$, i.e. the variable Y is substituted by X in the entire formula.
  - **Elimination of irrelevant disequalities**. The set of variables of $GoalVars$ and the variables that appear in $\overline{I}$ are called jointly $ImpVars$. The disequalities D_i which contain any variable that wasn't in $ImpVars$ neither $RelVars$ are irrelevant and must be eliminated.

2.3 Negation of the formula

To obtain all solutions of C_i and to negate them is not possible because C_i can have an infinite number of solutions. So we have to use the general constructive negation algorithm.

We consider that $ExpVars$ is the set of variables of $\overline{R}$ that aren't in $ImpVars$, i.e. $RelVars$ except the variables of $\overline{I}$ of the normalized formula.

First step: **Division of the formula**
We have to divide C_i in its sub-parts:

$$C_i \equiv \overline{I} \wedge \overline{D}_{imp} \wedge \overline{R}_{imp} \wedge \overline{D}_{exp} \wedge \overline{R}_{exp}$$

where $\overline{D}_{exp}$ are the disequalities in $\overline{D}$ with variables in $ExpVars$ and $\overline{D}_{imp}$ are the rest of disequalities, $\overline{R}_{exp}$ are the goals of $\overline{R}$ with variables in $ExpVars$ and $\overline{R}_{imp}$ are the rest of goals, and $\overline{I}$ are the equalities.

Therefore the constructive negation of the divided formula will be:

$$\begin{array}{l} \neg C_i \equiv \neg \overline{I} \vee \\ (\overline{I} \wedge \neg \overline{D}_{imp}) \vee \\ (\overline{I} \wedge \overline{D}_{imp} \neg \overline{R}_{imp}) \vee \\ (\overline{I} \wedge \overline{D}_{imp} \wedge \overline{R}_{imp} \wedge \neg (\overline{D}_{exp} \wedge \overline{R}_{exp})) \end{array}$$

It is not possible to separate $\overline{D}_{exp}$ and $\overline{R}_{exp}$ because they contain free variables and their negation cannot be done separately. The answers of the negations will be the answers of the negation of the equalities, the answers of the negation of the disequalities without free variables, the answers of the negation of the subgoals without free variables and the answers of the negation of the rest of subgoals of the conjunctions (the ones with free variables). Each of them will be obtained as follow:

Step2: **Negation of subformulas**

- **Negation of $\overline{I}$**. We have $\overline{I} \equiv I_1 \wedge \ldots \wedge I_{NI} \equiv$

$$\exists\, \overline{Z}_1\ X_1 = t_1 \wedge \ldots \wedge \exists\, \overline{Z}_{NI}\ X_{NI} = t_{NI}$$

  where $\overline{Z}_i$ are the variables of the equality I_i that aren't included in $GoalVars$ (i.e. that aren't quantified and therefore are free variables). When we negate this conjunction of equalities we obtain the constraint

$$\underbrace{\forall\, \overline{Z}_1\ X_1 \neq t_1}_{\neg\ I_1} \vee \ldots \vee \underbrace{\forall\, \overline{Z}_{NI}\ X_{NI} \neq t_{NI}}_{\neg\ I_{NI}} \equiv \bigvee_{i=1}^{NI} \forall\, \overline{Z}_i X_i \neq t_i$$

  This constraint is the first answer of the negation of C_i that contains NI solutions.
- **Negation of $\overline{D}_{imp}$**. If we have $N_{D_{imp}}$ disequalities $\overline{D}_{imp} \equiv D_1 \wedge \ldots \wedge D_{N_{D_{imp}}}$ where $D_i \equiv \forall\, \overline{W}_i\ \exists\, \overline{Z}_i\ Y_i \neq s_i$ where Y_i is a variable of $ImpVars$, s_i is a term without variables in $ExpVars$, $\overline{W}_i$ are universally quantified variables that are neither in the equalities [2], nor in the rest of the goals of $\overline{R}$ because then it will be a disequality of $\overline{D}_{exp}$. Then we will get $N_{D_{imp}}$ new solutions with the format:

$$\begin{array}{l} \overline{I} \wedge \neg D_1 \\ \overline{I} \wedge D_1 \wedge \neg D_2 \\ \ldots \\ \overline{I} \wedge D_1 \wedge \ldots \wedge D_{N_{D_{imp}}-1} \wedge \neg D_{N_{D_{imp}}} \end{array}$$

[2] of course, there are no universally quantified variables into an equality

where $\neg\ D_i \equiv \exists\ \overline{W}_i\ Y_i = s_i$. The negation of an universally quantification turns on existentially quantification and the quantification of free variables of $\overline{Z}_i$ get lost because they are unified with the evaluation of the equalities of $\overline{I}$. Then we will get $N_{D_{imp}}$ new answers.

- **Negation of $\overline{R}_{imp}$.** If we have $N_{R_{imp}}$ subgoals $\overline{R}_{imp} \equiv R_1 \wedge \ldots \wedge R_{N_{R_{imp}}}$. Then we will get new answers from each of the conjunctions:

$$\overline{I} \wedge \overline{D}_{imp} \wedge \neg\ R_1$$
$$\overline{I} \wedge \overline{D}_{imp} \wedge R_1 \wedge \neg\ R_2$$
$$\ldots$$
$$\overline{I} \wedge \overline{D}_{imp} \wedge R_1 \wedge \ldots \wedge R_{N_{R_{imp}}-1} \wedge \neg\ R_{N_{R_{imp}}}$$

  where $\neg\ R_i \equiv cneg(R_i)$. It is again the constructive negation that is applied over R_i recursively.
- **Negation of $\overline{D}_{exp} \wedge \overline{R}_{exp}$.** This conjunction can't be disclosed because the negation of $\exists\ \overline{V}_{exp}\ \overline{D}_{exp} \wedge \overline{R}_{exp}$, where $\overline{V}_{exp}$, gives universal quantifications: $\forall\ \overline{V}_{exp}\ cneg(\overline{D}_{exp} \wedge \overline{R}_{exp})$. Now the complete algorithm of constructive negation must be applied again. The set $GoalVars$ that we are going to consider therefore is the former set $ImpVars$. Variables of $\overline{V}_{exp}$ are considered as free variables. When solutions of $cneg(\overline{D}_{exp} \wedge \overline{R}_{exp})$ will be obtained, then we will reject solutions with equalities with variables of $\overline{V}_{exp}$. When there will be a disequality with any of these variables, e.g. V, the variable will be universally quantified in the disequality. This is the treatment to negate the negation of a goal but there is a detail that wasn't consider in former approaches and that is necessary to obtain a sound implementation: the existence of universally quantified variables in $\overline{D}_{exp} \wedge \overline{R}_{exp}$ by the iterative application of the method. So, what we are really negating is a subgoal of the form: $\exists\ \overline{V}_{exp}\ \overline{D}_{exp} \wedge \overline{R}_{exp}$. Here we will provide the last group of answers that come from:

$$\overline{I} \wedge \overline{D}_{imp} \wedge \overline{R}_{imp} \wedge \forall\ \overline{V}_{exp}\ \neg\ (\overline{D}_{exp} \wedge \overline{R}_{exp})$$

3 Implementation Issues

Once we have described the theoretical algorithm, including relevant details, we supply important aspects for a practical implementation, including how to compute the frontier and the management of answer constraints.

3.1 Code Expansion

The first thing we have to do to implement this technique is obtaining the frontier of a goal. For this purpose is necessary that the code of the implied predicates is available during execution to construct the frontier. It is possible to handle the code of clauses during the execution thanks to the package system

of Ciao [CH00] that let us to expand the code at run time. Therefore we have been able to evaluate and execute predicates and provide their code at the same time to calculate their *Frontiers* at any step of the algorithm. The expansion is implemented in the package *cneg.pl* that is incorporated in the declaration of the module that is going to be expanded (i.e. where there are goals that are negations).

A simple example would be the module *mod1.pl* that export the predicate *odd*/1 and the predicate $not_o dd$/1 that is semantically the negation of *odd*/1:

```
:- module(mod1,[odd/1,not_odd/1],[cneg]).

odd(s(0)).
odd(s(s(X))) :- odd(X).

not_odd(X) :- cneg(odd(X)).
```

The loading of the package *cneg.pl* means that the compiler works with a expanded code added to the previous one:

```
stored_clause(odd(s(0)),[]).
stored_clause(odd(s(s(X))),[odd(X)]).
```

where information about structure of code is stored to be used by the negation algorithm. Now the execution is able to compute the frontier we describe above $\{(Y = s(0)) \vee (Y = s(s(X)) \wedge odd(X))\}$

3.2 Disequality constraints

An instrumental step in order to manage negation is to be able to handle disequalities between terms such as $t_1 \neq t_2$. Prolog implementations typically include only the built-in predicate `/==` /2 which can only work with disequalities if both terms are ground and simply succeeds in the presence of free variables. A "constructive" behavior must allow the "binding" of a variable with a disequality. On the other hand, the negation of an equation $X = t(\overline{Y})$ produces the universal quantification of the free variables in the equation, unless a more external quantification affects them. The negation of such an equation is $\forall\ \overline{Y}\ X \neq t(\overline{Y})$. As we explained in [4], the inclusion of disequalities and constrained answers has a very low cost. It incorporates negative normal form constraints instead of bindings and the decomposition step can produce disjunctions. More precisely, the normal form of constraints is:

$$\underbrace{\bigwedge_i (X_i = t_i)}_{\text{positive information}} \quad \wedge \quad \underbrace{(\bigwedge_j \forall\ \overline{Z}_j^1\ (Y_j^1 \neq s_j^1) \vee \ldots \vee \bigwedge_l \forall\ \overline{Z}_l^n\ (Y_l^n \neq s_l^n))}_{\text{negative information}}$$

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).

Using some normalization rules we can obtain a normal form formula from any initial formula. 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 [5] and differs from Chan's representation where only disjunctions are used[3].

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 [6] (e.g. in Sicstus Prolog, or in Eclipse where they are called meta-structures). These variables let us keep associated information with each variable during the unification 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. Basically, a list of list of pairs (variable, term) is used. 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 bound 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') satisfies the constraint associated with Y. A conveniently defined 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.

We have defined a predicate `=/= /2` [4], used to check disequalities, in a similar way to explicit unification (`=`). Each constraint is a disjunction of conjunctions of disequalities that are implemented as a list of lists of terms as T_1/T_2 (that represents the disequality $T_1 \neq T_2$). When a universal quantification is used in a disequality (e.g., $\forall Y\ X \neq c(Y)$) the new constructor `fA/1` is used (e.g., `X / c(fA(Y))`).

The first list is used to represent disjunctions while the inside list represents the conjunction of disequalities. We focus on the variable X.

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

SUBGOAL	ATTRIBUTE	CONSTRAINT
`not_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=/=5`	$[[X/4], [X/5]]$	$X \neq 4 \vee X \neq 5$
`X =/= 5; (X=/=6, X=/=Y)`	$[[X/4], [X/6, X/Y]]$	$X \neq 4 \vee (X \neq 6 \wedge X \neq Y)$
`forall (Y, X =/= s (Y)) )`	$[[X/s(fA(Y))]]$	$\forall Y.X \neq Y$

3.3 Optimization

Compact information. We have decided to represent negative information in a compact way providing lesser solutions from the negation of $\overline{I}$. The advantage is that if we ask for all solutions using backtracking we are cutting the search tree by offering all the solutions together in one only answer. For example we can offer a simple answer for

```
?- cneg(p(X,Y,Z,W)).

[[X/0, Y/s(Z)], [X/Y], [X/Z], [X/W], [X/s(0)], [Z/0]] ? ;
no
```

(that is equivalent to $(X \neq 0 \wedge Y \neq s(Z)) \vee X \neq Y \vee X \neq Z \vee X \neq W \vee X \neq s(0) \vee Z \neq 0$) instead of returning six answers:

```
?- cneg(p(X,Y,Z,W)).

[X/0, Y/s(Z)] ? ;
[X/Y] ? ;
[X/Z] ? ;
[X/W] ? ;
[X/s(0)] ? ;
[Z/0] ? ;
no
```

Pruning subgoals. We have cut the search tree of the generation of the frontiers with a double action over the ground subgoals: removing soon the subgoals whose failure we are able to detect, and simplifying the subgoals that we can reduce to true. Suppose we have a predicate $p/2$ defined as

```
p(X,Y):- greater(X,Y),
         q(X,Y,Z),
         r(Z).
```

We can imagine that $q/3$ and $r/1$ are predicates defined by several clauses with a complicated implementation. If we would like to obtain the frontier of $p(s(0), s(s(0)))$ to negate this goal, we would get:

$$Frontier(p(s(0), s(s(0)))) \equiv$$
$$X = s(0) \wedge Y = s(s(0)) \wedge greater(X,Y) \wedge q(X,Y,Z) \wedge r(Z) \equiv$$
$$greater(s(0), s(s(0)) \wedge q(s(0), s(s(0), Z) \wedge r(Z) \equiv$$
$$fail \wedge q(s(0), s(s(0), Z) \wedge r(Z) \equiv$$
$$fail$$

Now we would have to expand the code of the subgoals of the frontier to all the combination (disjunction) of the code of their clauses and the result will be a very complicated and long to check frontier. However we optimize the process evaluating the ground terms. In this case $greater(s(0), s(s(0))$ fails and therefore it is no necessary to continue with the generation of the frontier because the result is reduced to fail (i.e. the negation of $p(s(0), s(s(0)))$ will be trivially true). The opposite example means a simplification:

$$Frontier(p(s(s(0)), s(0))) \equiv$$
$$X = s(s(0)) \wedge Y = s(0) \wedge greater(X,Y) \wedge q(X,Y,Z) \wedge r(Z) \equiv$$
$$greater(s(s(0)), s(0) \wedge q(s(s(0)), s(0), Z) \wedge r(Z) \equiv$$
$$true \wedge q(s(s(0)), s(0), Z) \wedge r(Z) \equiv$$
$$q(s(s(0)), s(0), Z) \wedge r(Z)$$

Constraint simplification. Another way of optimization is the simplification of the constraints. During all the process of negating a goal the variables of the frontier are constrained. Sometimes the constraints are satisfiable and we can eliminate them, other times the constraints can be reduced to fail and we can stop the evaluation of the negation because the result is true.

We intend to reduce a constraint F to success or failure. Constraints that are managed are formulas of the form:

$$F \equiv \bigvee_i \bigwedge_j \forall\, \overline{Z}_j^i \; (Y_j^i \neq s_j^i)$$

We are going to transform this formula. Firstly we will obtained the Prenex form [7] extracting the universal variables with different names to the head of the formula applying logic rules:

$$F \equiv \forall \overline{x} \bigvee_i \bigwedge_j (Y_j^i \neq s_j^i)$$

Using the distributive property:

$$F \equiv \forall \overline{x} \bigwedge_k \bigvee_l (Y_l^k \neq s_l^k)$$

The formula can be separated into subformulas that are simple disjunctions of disequalities :

$$F \equiv \bigwedge_k \forall \overline{x} \bigvee_l (Y_l^k \neq s_l^k) \equiv F_1 \wedge ... \wedge F_n$$

Now we have to evaluate each single formula F_k. The first step will be to substitute the existential quantified variables (those that do not belong to $\overline{x}$) by Skolem constants s_j^i that will keep the equivalence without loosing generality:

$$F_k \equiv \forall \overline{x} \bigvee_l (Y_l^k \neq s_l^k) \equiv \forall \overline{x} \bigvee_l (Y_{Skl}^k \neq s_{Skl}^k)$$

Then we transform it into:

$$F_k \equiv \neg \exists\, \overline{x} \neg (\bigvee_l (Y_{Skl}^k \neq s_{Skl}^k)) \equiv \neg Fe_K$$

The meaning of F_k is the negation of the meaning of Fe_k. Then the next goal is evaluating Fe_k:

$$Fe_k \equiv \exists\, \overline{x} \neg (\bigvee_l (Y_{Skl}^k \neq s_{Skl}^k))$$

Resolving the negations we obtain the result through simple unifications of the variables of $\overline{x}$:

$$Fe_k \equiv \exists\, \overline{x} \bigwedge \neg (Y_{Skl}^k \neq s_{Skl}^k) \equiv \exists\, \overline{x} \bigwedge (Y_{Skl}^k = s_{Skl}^k)$$

Therefore we get the truth value of the F_k from the negation of the value of Fe_k and finally the value of F is the conjunction of the values of all the F_k. If F succeeds then we remove the constraint because is redundant and we continue with the negation process. If it fails then the negation directly succeeds.

3.4 Examples

The interesting side of this implementation is that it returns constructive results from a negative question. Let us start with a simple example involving predicate *boole*/1.

```
boole(0).
boole(1).
```

```
?- cneg(boole(X)).
[[X/1,X/0]] ? ;
no
```

More interesting examples are those that have an infinite number of solutions.

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

```
?- cneg(positive(X)).

[[X/s(fA(_A)),X/0]] ? ;
X = s(_A),
[[_A/s(fA(_B)),_A/0]] ? ;
X = s(s(_A)),
[[_A/s(fA(_B)),_A/0]] ? ;
X = s(s(s(_A))),
[[_A/s(fA(_B)),_A/0]] ?
yes
```

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

greater(s(X),0):-
        number(X).
greater(s(X),s(Y)):-
        greater(X,Y).
```

```
?- cneg(greater(X,Y)).

[[Y/0,Y/s(fA(_A))]] ? ;

[[Y/s(fA(_A))]],
[[X/s(fA(_B))]] ? ;

X = s(_A),
Y = 0,
[[_A/s(fA(_B)),_A/0]] ? ;

X = s(s(_A)),
Y = 0,
[[_A/s(fA(_B)),_A/0]] ? ;

X = s(s(s(_A))),
Y = 0,
[[_A/s(fA(_B)),_A/0]] ? ;

X = s(s(s(s(_A)))),
Y = 0,
[[_A/s(fA(_B)),_A/0]] ?

yes
```

3.5 Experimental results

We have firstly measured the execution times in milliseconds for the previous examples when using negation as failure ($naf/1$) and constructive negation ($cneg/1$). A '-' in a cell means that the negation as failure is not applicable. All measurements were made using Ciao Prolog[4] 1.5 on a Pentium II at 350 Mhz. The results are shown in Table 1. We have added a first column with the runtime of the evaluation of the goal that is negated in the other columns and a last column with the ratio that measures the speedup of the *naf* technique w.r.t. the constructive negation.

Using **naf** instead of **cneg** results in slight speed-ups near 1.06 in average for ground calls with few recursive calls. So the possible slow-down from the constructive negation is not so high as we could expect for these examples. Furthermore the results are rather similar. But the same goals with data that involves many recursive calls give us speed-ups near 14.69 in average and increasing exponentially with the number of recursive calls. Additionally, there are, of course, many goals that cannot be negated using the *naf* technique and that are resolved using constructive negation.

[4] The negation system is coded as a library module ("package" [CH00]), which includes the corresponding syntactic and semantic extensions (i.e. Ciao's attributed variables). Such extensions apply locally within each module which uses this negation library.

goals	Goal	naf(Goal)	cneg(Goal)	ratio
boole(1)	2049	2099	2069	0.98
boole(8)	2070	2170	2590	1.19
positive(s(s(s(s(s(0))))))	2079	1600	2159	1.3
positive(s(s(s(s(0)))))	2079	2139	2060	0.96
greater(s(s(0)),s(0))	2110	2099	2100	1.00
greater(s(0),s(s(0)))	2119	2129	2089	0.98
average				1.06
positive($s^{500000}(0)$)	2930	2949	41929	14.21
positive($s^{1000000}(0)$)	3820	3689	81840	22.18
greater($s^{500000}(0)$,$s^{500000}(0)$)	3200	3339	22370	7.70
average				14.69
boole(X)	2080	-	3109	
positive(X)	2020	-	7189	
greater(s(s(0)),X)	2099	-	6990	
greater(X,Y)	7040	-	7519	
queens(s(s(0)),Qs)	6939	-	9119	

Table 1. Runtime comparation

4 Conclusion and Future Work

After making some preliminary experiments with the constructive negation technique following Chan's description, we realized that the algorithm needed some additional explanations and modifications.

Once we have specified the algorithm in a detailed way we proceed to provide a real, complete and consistent implementation. The result we have reported are very encouraging because we have proved that it is possible to extend Prolog with a constructive negation module in a relatively cheap way. However, the algorithm is inherently inefficient, so it is quite important to take care of possible optimizations and we are working to improve the efficiency of the implementation. Among them we can mention a more accurate selection of the frontier based on the demanded form of argument in the vein of [8]). Another future work is to incorporate our algorithm into the WAM machine level.

In any case we will probably not be able to provide a efficient enough implementation of constructive negation. That is the reason why we do not intend to use it neither for all cases of negation nor for negating goals directly.

Our goal is to design and implement a practical negation operator and incorporate it into a Prolog compiler. In [4,9] we studied systematically what we understood to be the most interesting existing proposals: negation as failure (*naf*) [10], use of delays to apply *naf* in a secure way [11], intensional negation [12,13], and constructive negation [14,15,16,17,2]. As none of them can satisfy our requirements of completeness and efficiency, we proposed to use a combination of these techniques and the information from a static analysis of the program could be used to reduce the cost of selecting among techniques [9]. So we avoid the

inefficiency of the constructive negation. However we still need it because is the only one that is sound and complete for any kind of goals. For example, if we observe the goals of Table 1 the strategy will obtain all ground negation using the *naf* technique and only would use constructive negation for the goals with variables where it is impossible to use *naf*.

We are testing the implementation trying to improve the code and our intention is to include it in the next distribution of Ciao Prolog [5].

References

1. Kunen, K.: Negation in logic programming. Journal in Logic Programming **4** (1987) 289–308
2. Stuckey, P.: Negation and constraint logic programming. In: Information and Computation. Volume 118. (1995) 12–33
3. Cabeza, D., Hermenegildo, M.: A New Module System for Prolog. In: International Conference on Computational Logic, CL2000. LNCS, Springer-Verlag (2000)
4. Muñoz, S., Moreno, J.: How to incorporate negation in a prolog compiler. In E. Pontelli, V.S.C., ed.: 2nd International Workshop PADL'2000. Volume 1753 of LNCS., Boston, MA (USA), Springer (2000) 124–140
5. Moreno-Navarro, J.: Default rules: An extension of constructive negation for narrowing-based languages. In: Proc. ICLP'94, The MIT Press (1994) 535–549
6. Carlsson, M.: Freeze, indexing, and other implementation issues in the wam. In: ICLP, The MIT Press (1987) 40–58
7. : Mathematical Logic. Association for Symbolic Logic (1967)
8. Moreno-Navarro, J.: Extending constructive negation for partial functions in lazy narrowing-based languages. (1996)
9. Muñoz, S., Moreno, J., Hermenegildo, M.: Efficient negation using abstract interpretation. In R.Nieuwenhuis, Voronkov, A., eds.: Logic for Programming, Artificial Intelligence and Reasoning, La Habana (Cuba) (2001)
10. Clark, K.L.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases, New York, NY, Plenum Press (1978) 293–322
11. Naish, L.: Negation and Control in Prolog. In: LNCS. Number 238, Springer-Verlag (1985)
12. Barbuti, R., Mancarella, D., Pedreschi, D., Turini, F.: Intensional negation of logic programs. Lecture notes on Computer Science **250** (1987) 96–110
13. Barbuti, R., Mancarella, D., Pedreschi, D., Turini, F.: A transformational approach to negation in logic programming. JLP **8** (1990) 201–228
14. Chan, D.: Constructive negation based on the complete database. In: Proc. Int. Conference on LP'88, The MIT Press (1988) 111–125
15. Chan, D.: An extension of constructive negation and its application in coroutining. In: Proc. NACLP'89, The MIT Press (1989) 477–493
16. Drabent, W.: What is a failure? An approach to constructive negation. Acta Informatica. **33** (1995) 27–59
17. Stuckey, P.: Constructive negation for constraint logic programming. In: Proc. IEEE Symp. on Logic in Computer Science. Volume 660., IEEE Comp. Soc. Press (1991)

[5] http://www.clip.dia.fi.upm.es/Software