A Real Implementation for Constructive Negation

Susana Muñoz Juan José Moreno-Navarro
susana@fi.upm.es jjmoreno@fi.upm.es

LSIIS, Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo s/n Boadilla del Monte
28660 Madrid, Spain **

Abstract. Logic Programming has been advocated as a language for system specification, especially for those involving logical behaviours, rules and knowledge. However, modelling problems involving negation, which is quite natural in many cases, is somewhat limitated if Prolog is used as the specification / implementation language. These restrictions are not related to theory viewpoint, where users can find many different models with their respective semantics; they concern practical implementation issues. The negation capabilities supported by current Prolog systems are rather limited, and there is no correct and complete implementation. In this paper, we refine and propose some extensions to the method of constructive negation, providing the complete theoretical algorithm. Furthermore, we also discuss 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 both a programming language and a specifi cation language. It is natural to use Logic Programming for specifying/programming systems involving logical behaviours, rules and knowledge. However, this idea has a severe limitation: the use of negation. Negation is probably the most signifi cant aspect of logic that was not included from the outset. This is due to the fact that dealing with negation involves signifi cant additional complexity. Nevertheless, the use of negation is very natural and plays an important role in many cases, for instance, constraints management in databases, program composition, manipulation and transformation, default reasoning, natural language processing, 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 problems really start at the semantic level, and the different proposals (negation as failure (*naf*), stable models, well-founded semantics, explicit negation, etc.) differ not only as to expressiveness but also as to semantics. However, the negation techniques supported by current Prolog compilers are rather limited, restricted to

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

negation as failure under Fitting/Kunen semantics [9] (sound only under some circumstances usually not checked by compilers) which 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), which is present in Nu-Prolog, Gödel, and Prolog systems that implement delays (most of the above).

Of all the proposals, constructive negation [5, 6] is probably the most promising because it has been proven to be sound and complete, and its semantics is fully compatible with Prolog's. Constructive negation was, in fact, announced in early versions of the Eclipse Prolog compiler, but was removed from the lastest releases. The reasons seem to be related to some technical problems with the use of coroutining (risk of floundering) and the management of constrained solutions.

The goal of this paper is to give an algorithmic description of constructive negation, i.e. explicitly stating the details needed for an implementation. We also intend 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. We assume some familiarity with constructive negation techniques and Chan's papers.

The remainder of the paper is organized as follows. Section 2 details our constructive negation algorithm. It explains how to obtain the *frontier* of a goal (Section 2.1), how to prepare the goal for negation (Section 2.2) and, finally, how to negate the goal (Section 2.3). Section 3 discusses implementation issues: code expansion (Section 3.1), required disequality constraints (Section 3.2), optimizations (Section 3.3), examples (Section 3.5) and some experimental results (Section 3.4). Finally, we conclude and outline some future work.

2 Constructive Negation

Most of the papers addressing constructive negation deal with semantic aspects. In fact, only the original papers by Chan gave some hints about a possible implementation based on coroutining, but the technique was only outlined. When we tried to reconstruct this implementation we came across several problems, including the management of constrained answers and floundering (which appears to be the main reason why constructive negation was removed from recent versions of Eclipse). It is our beief that this problems cannot be easily and efficiently overcome. Therefor, we decided to design an implementation from scratch. One of our additional requirements is that we want to use a standard Prolog implementation (to be able to reuse thousands of existing Prolog lines and maintain their efficiency), so we will avoid implementation-level manipulations. This is coherent with the usual Prolog definition of *naf*:

```
naf (P), P, !, fail.
naf (P).
```

We start with the definition of a frontier and how it can be managed to negate the respective formula.

2.1 Frontier

Firstly, Chan's definition of frontier (we actually owe the formal definition to Stuckey [17]).

Definition 1. Frontier

A frontier of a goal G is the disjunction of 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. So far we have used the simplest possible frontier: the frontier of depth 1 obtained by taking all the possible single SLD resolution steps. This can be done by a simple inspection of the applicable clauses can do this[1]. Additionally, built-in based goals receive 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 DeMorgan's distributive property to retain the disjunction of conjunctions format.*
- *If* $G \equiv p(\overline{X})$ *and predicate* p/m *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 union of the conjunction of subgoals* C'_i *plus the equalities that are needed to unify the variables of* $\overline{X}$ *and the respective 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 as follows:

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

To get the negation of G it suffices 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. Now we are going to explain how to negate a single conjunction C_i. This is done in two phases: *Preparation* and *Negation of the formula*.

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

2.2 Preparation

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

- **Simplification of the conjunction**. If one of the terms of C_i is trivially equivalent to *true* (e.g. $X = X$), we can eliminate this term from C_i. Symmetrically,if one of the terms is trivially $fail$ing (e.g. $X \neq X$), we can simplify $C_i \equiv fail$. The simplifi cation phase can be carried out during the generation of frontier terms.
- **Organization of the conjunction**. Three groups are created containing the components of C_i, which are divided into equalities ($\overline{I}$), disequalities ($\overline{D}$), and other subgoals ($\overline{R}$). Then, we get $C_i \equiv \overline{I} \wedge \overline{D} \wedge \overline{R}$.
- **Normalization of the conjunction**. Let us classify the variables in the formula. 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 now have 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**. *ImpVars* is the set of variables of *GoalVars* and the variables that appear in $\overline{I}$. The disequalities D_i that contain any variable that was neither in *ImpVars* nor in *RelVars* are irrelevant and should be eliminated.

2.3 Negation of the formula

It is not feasible, to get all solutions of C_i and to negate their disjunction because C_i can have an infi nite 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 are not in *ImpVars*, i.e. *RelVars*, except the variables of $\overline{I}$ in the normalized formula.

First step: **Division of the formula**
C_i is divided into:

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

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

Therefore, the constructive negation of the divided formula is:

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

It is not possible to separate $\overline{D}_{exp}$ and $\overline{R}_{exp}$ because they contain free variables and they cannot be negated 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 other subgoals of the conjunctions (the ones with free variables). Each of them will be obtained as follows:

Second step: **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 are not included in *GoalVars* (i.e. that are not quantified and are therefore free variables). When we negate this conjunction of equalities we get 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_1 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 other goals of $\overline{R}$ because otherwise $\overline{R}$ would be a disequality of $\overline{D}_{exp}$. Then we will get $N_{D_{imp}}$ new solutions with the format:

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

  where $\neg\, D_i \equiv \exists\, \overline{W}_i\ Y_i = s_i$. The negation of a universal quantification turns into an existential quantification and the quantification of free variables of $\overline{Z}_i$ gets lost, because the variables 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:

$$\begin{array}{l}\overline{I} \wedge \overline{D}_{imp} \wedge \neg\, R_1\\ \overline{I} \wedge \overline{D}_{imp} \wedge R_1 \wedge \neg\, R_2\\ \ldots\\ \overline{I} \wedge \overline{D}_{imp} \wedge R_1 \wedge \ldots \wedge R_{N_{R_{imp}}-1} \wedge \neg\, R_{N_{R_{imp}}}\end{array}$$

  where $\neg\, R_i \equiv cneg(R_i)$. Constructive negation tis again applied over R_i recursively.

[2] There are, of course, no universally quantified variables in an equality

– **Negation of** $\overline{D}_{exp} \wedge \overline{R}_{exp}$. This conjunction cannot be disclosed because of the negation of $\exists\, \overline{V}_{exp}\ \overline{D}_{exp} \wedge \overline{R}_{exp}$, where $\overline{V}_{exp}$ gives universal quantifications: $\forall\, \overline{V}_{exp}\ cneg(\overline{D}_{exp} \wedge \overline{R}_{exp})$. The entere constructive negation algorithm must be applied again. Note that the new set *GoalVars* is the former set *ImpVars*. Variables of $\overline{V}_{exp}$ are considered as free variables. When solutions of $cneg(\overline{D}_{exp} \wedge \overline{R}_{exp})$ are obtained some can be rejected: solutions with equalities with variables in $\overline{V}_{exp}$. If there is a disequality with any of these variables, e.g. V, the variable will be universally quantified in the disequality. This is the way to negate the negation of a goal, but there is a detail that was not considered in former approaches and that is necessary to get 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

Having described the theoretical algorithm, including important details, we now discuss important aspects for a practical implementation, including how to compute the frontier and manage answer constraints.

3.1 Code Expansion

The first issue is how to get the frontier of a goal. For this purpose, the code of the predicates involved needs to be available during execution to construct the frontier. It is possible to handle the code of clauses during the execution thanks to the Ciao package system [3], which allows the code to be expanded at run time. Therefore, we have been able to evaluate and execute predicates and also provide their code at to calculate their *Frontiers* at any step of the algorithm. The expansion is implemented in the *cneg.pl* package which is included 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 *mod*1.*pl* that exports the predicate $odd/1$ and the predicate $not\_odd/1$ that, semantically, is 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 *cneg.pl* package means that the compiler works with an expanded code added to the previous code:

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

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

Note that a similar, but less efficient, behaviour can be emulated using metaprogramming facilities, available in most Prolog compilers.

3.2 Disequality constraints

An instrumental step for managing negation is to be able to handle disequalities between terms such as $t_1 \neq t_2$. The typical Prolog resources for handling these disequalities are limited to the built-in predicate `/== /2`, which needs both terms to be ground because it always succeeds in the presence of free variables. It is clear that a variable needs to be bound with a disequality to achieve a "constructive" behaviour. Moreover, when an equation $X = t(\overline{Y})$ is negated, the free variables in the equation must be universaly quantified, unless affected by a more external quantification, i.e. $\forall \overline{Y}\ X \neq t(\overline{Y})$ is the correct negation. As we explained in [12], 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. Our proposal for normal form 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 $X_i = t_i$, no s_k^r is Y_k^r and the universal quantification could be empty (leaving a simple disequality).

It is easy to see that some normalization rules can be defined for a normal form formula from any initial formula. The redefinition of the unification algorithm to manage constrained variables is also a simple exercise.

[10] introduces this very compact way to represent a normal form constraint. Chan's representation uses only disjunctions and they are dealt with by means of backtracking. The main advantage of our normal form is that the search space is drastically reduced.

To include disequalities into a Prolog compiler, we need to just reprogram unification. This can be done using attributed variables [4] (available in several Prolog versions, e.g. in Sicstus Prolog, or in Eclipse, where they are called meta-structures). These variables allow us to keep information associated with each variable (in an attibute that is a term) during the unification, which can be used to dynamically control the constraints.

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

1. X is a free variable and t is not a variable with a negative constraint: just bind X to t,
2. X is a free variable or bound to a term t' and t is a variable Y with a negative constraint: check whether X (or, equivalently, t') satisfies the constraint associated with Y. A conveniently defined predicate `satisfy` is used for this purpose,

3. X is bound to a term t' and t is a term (or a variable bound to a term): use the classical unification algorithm.

A Prolog predicate `=/= /2` [12] has been defined, used to check disequalities, similarly to explicit unification (`=`). Each constraint is a disjunction of conjunctions of disequalities that are implemented as a list of lists of terms like T_1/T_2 (which 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 internal list represents the conjunction of disequalities.

Let us show some examples involving variable X:

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 Optimizing the algorithm and the implementation

Our constructive negation algorithm and the implementation techniques admit some additional optimizations that can improve the runtime behaviour of the system. Basically, the optimizations rely on the compact representation of information, as well as the early detection of successful or failing branches.

Compact information. In our system, negative information is represented quite compactly, providing fewer solutions from the negation of $\overline{I}$. The advantage is twofold. On the one hand constraints contain more information and failing branches can be detected earlier (i.e. the search space could be smaller). On the other hand, if we ask for all solutions using backtracking, we are cutting the search tree by offering all the solutions together in a single answer. For example, we can offer a simple answer for the negation of a predicate p (the code for p is skipped):

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

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

(which 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 upon backtracking:

```
?- 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. The frontiers generation search tree can be cut with a double action over the ground subgoals: removing the subgoals whose failure we are able to detect early on, and simplifying the subgoals that can be reduced to true. Suppose we have a predicate $p/2$ defined as

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

where $q/3$ and $r/1$ are predicates defined by several clauses with a complex computation. To negate the goal $p(s(0),s(s(0)))$, its frontier is computed:

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

The next step is to expand the code of the subgoals of the frontier to the combination (disjunction) of the code of all their clauses, and the result will be a very complicated and hard to check frontier. However, the process is optimized by evaluating ground terms. In this case, $greater(s(0),s(s(0))$ fails and, therefore, it is not 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 is 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. During the whole process for negating a goal,the frontier variables are constrained. In cases where the constraints are satisfiable, they can be eliminated and where the constraints can be reduced to fail, the evaluation can be stopped with result *true*.

We focus on the negative information of a normal form constraint F:

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

Firstly, the Prenex form [15] can be obtained by extracting the universal variables with different names to the head of the formula, applying logic rules:

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

and 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 :

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

Each single formula F_k can be evaluated. The first step will be to substitute the existentially quantified variables (variables that do not belong to $\overline{x}$) by Skolem constants s_j^i that will keep the equivalence without losing 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 it can be transformed 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;

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

Solving the negations, the result is obtained 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 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 F_k. If F succeeds, then the constraint is removed because it is redundant and we continue with the negation process. If it fails, then the negation directly succeeds.

3.4 Experimental results

This section reports some experimental results from our prototype implementation. First of all, we show the behaviour of the implementation in some simple examples.

3.5 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
```

Another simple example obtained from [17] gives us the following answers:

```
p(a,b,c).
p(b,a,c).
p(c,a,b).

proof1(X,Y,Z):-
    X =/= a,
    Z = c,
    cneg(p(X,Y,Z)).
```

```
?- proof1(X,Y,Z).

Z = c,
X/b,X/a ? ;

Z = c,
Y/a,X/a ? ;

no
```

[17] contains another example showing how a constructive answer ($\forall T\ X \neq s(T)$) is provided for the negation of an undefi ned goal in Prolog:

```
p(X):- X = s(T), q(T).

q(T):- q(T).

r(X):- cneg(p(X)).
```

```
?- r(X).

X/s(fA(_A))
```

Examples that have an infi nite number of solutions are more interesting.

```
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
```

3.6 Implementation measures

We have firstly measured the execution times in milliseconds for the above examples when using negation as failure ($naf/1$) and constructive negation ($cneg/1$). A '-' in a cell means that negation as failure is not applicable. All measurements were made using Ciao Prolog[3] 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 positive 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. constructive negation.

Using **naf** instead of **cneg** results in small ratios around 1.06 on average for ground calls with few recursive calls. So, the possible slow-down for constructive negation is not so high as we might expect for these examples. Furthermore, the results are rather similar. But the same goals with data that involve many recursive calls yield ratios near 14.69 on average w.r.t **naf**, increasing exponentially with the number of recursive calls. There are, of course, many goals that cannot be negated using the *naf* technique and that are solved using constructive negation.

4 Conclusion and Future Work

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

Having given a detailed specification of algorithm in a detailed way we proceed to provide a real, complete and consistent implementation. The result, we have reported

[3] The negation system is coded as a library module ("package" [3]), which includes the respective 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(s(0))))))	2079	2139	2060	0.96
greater(s(s(s(0))),s(0))	2110	2099	2100	1.00
greater(s(0),s(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(s(0))),X)	2099	-	6990	
greater(X,Y)	7040	-	7519	
queens(s(s(0)),Qs)	6939	-	9119	

Table 1. Runtime comparison

are very encouraging, because we have proved that it is possible to extend Prolog with a constructive negation module relatively inexpensively. Nevertheless, it is quite important to address possible optimizations, and we are working to improve the efficiency of the implementation. These include a more accurate selection of the frontier based on the demanded form of argument in the vein of [11]). Other future work is to incorporate our algorithm at the WAM machine level.

In any case, we will probably not be able to provide an efficient enough implementation of constructive negation, because the algorithm is inherently inefficient. This is why we do not intend to use it either for all cases of negation or for negating goals directly.

Our goal is to design and implement a practical negation operator and incorporate it into a Prolog compiler. In [12, 13] we systematically studied what we understood to be the most interesting existing proposals: negation as failure (*naf*) [7], use of delays to apply *naf* securely [14], intensional negation [1, 2], and constructive negation [5, 6, 8, 16, 17]. As none of them can satisfy our requirements of completeness and efficiency, we propose to use a combination of these techniques, where the information from static program analyzers could be used to reduce the cost of selecting techniques [13]. So, in many cases, we avoid the inefficiency of constructive negation. However, we still need it because it is the only method that is sound and complete for all kinds of goal. For example, looking at the goals in Table 1, the strategy will obtain all ground negation using the *naf* technique and would only use constructive negation for the goals with variables where it is impossible to use *naf* .

We are testing the implementation and trying to improve the code, and our intention is to include it in the next version of Ciao Prolog [4].

References

1. R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. Intensional negation of logic programs. *LNCS*, 250:96–110, 1987.
2. R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *JLP*, 8(3):201–228, 1990.
3. D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.
4. M. Carlsson. Freeze, indexing, and other implementation issues in the wam. In *ICLP*, pages 40–58. The MIT Press, 1987.
5. D. Chan. Constructive negation based on the complete database. In *Proc. Int. Conference on LP'88*, pages 111–125. The MIT Press, 1988.
6. D. Chan. An extension of constructive negation and its application in coroutining. In *Proc. NACLP'89*, pages 477–493. The MIT Press, 1989.
7. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322, New York, NY, 1978. Plenum Press.
8. W. Drabent. What is a failure? An approach to constructive negation. *Acta Informatica.*, 33:27–59, 1995.
9. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.
10. J. J. Moreno-Navarro. Default rules: An extension of constructive negation for narrowing-based languages. In *Proc. ICLP'94*, pages 535–549. The MIT Press, 1994.
11. J. J. Moreno-Navarro. Extending constructive negation for partial functions in lazy narrowing-based languages. *ELP*, 1996.
12. S. Muñoz and J. J. Moreno-Navarro. How to incorporate negation in a prolog compiler. In E. Pontelli and V. Santos Costa, editors, *2nd International Workshop PADL'2000*, volume 1753 of *LNCS*, pages 124–140, Boston, MA (USA), 2000. Springer-Verlag.
13. S. Muñoz, J. J. Moreno-Navarro, and M. Hermenegildo. Efficient negation using abstract interpretation. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence and Reasoning*, number 2250 in LNAI, pages 485–494, La Habana (Cuba), 2001. LPAR 2001.
14. L. Naish. *Negation and control in Prolog*. Number 238 in Lecture Notes in Computer Science. Springer-Verlag, New York, 1986.
15. J. R. Shoenfield. *Mathematical Logic*. Association for Symbolic Logic, 1967.
16. P. Stuckey. Constructive negation for constraint logic programming. In *Proc. IEEE Symp. on Logic in Computer Science*, volume 660. IEEE Comp. Soc. Press, 1991.
17. P. Stuckey. Negation and constraint logic programming. In *Information and Computation*, volume 118(1), pages 12–33, 1995.

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