Implementation Results in Classical Constructive Negation

Cambiar el título: Implementing Classical Constructive Negation

SUSANA MUNOZ-HERNANDEZ and JUAN JOSÉ MORENO-NAVARRO

DLSIIS, Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo s/n Boadilla del Monte
28660 Madrid, Spain ∗
E-mail: susana@fi.upm.es — jjmoreno@fi.upm.es

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Logic Programming has been advocated as a language for system specification, especially for those involving logical behaviours, rules and knowledge. However, modeling problems involving negation, which is quite natural in many cases, is somewhat limited 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 constrained, and there is no a correct and complete implementation available. In this paper, we refine and propose some extensions to the classical method of constructive negation, providing the complete theoretical algorithm. Furthermore, we also discuss implementation issues providing a preliminary implementation and also an optimized one to negate predicates with a finite number of solutions.

KEYWORDS: Constructive Negation, Negation in Logic Programming, Constraint Logic Programming, Implementations of Logic Programming, Optimization.

Reescribir. Énfasis en resultados de implementación

1 Introduction

From its very beginning Logic Programming has been advocated to be both a programming language and a specification 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 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, constraints management in databases, program composition, manipulation and transformation, default reasoning, natural language processing, etc.

Although this restriction cannot be perceived from the theoretical point of view (because there are many ways to understand and incorporate negation into Logic Programming), the problems really start at the semantic level, where the different

Reformular y ampliar.

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[1] compilers are rather limited, restricted to negation as failure under Fitting/Kunen semantics (Kunen 1987) (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 (Chan 1988; Chan 1989) (that we will call *classical* constructive negation) 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 latest 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. We are trying to fill a long time open gap in this area (remember that the original papers are from late 80s) facing the problem of providing a correct implementation.

Desarrolar más

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. We are combining several different techniques: implementation of disequality constraint, program transformation, efficient management of constraints on the Herbrand universe, etc. While many of them are relatively easy to understand (and the main inspiration are, of course, in papers on theoretical aspects of constructive negation including Chan's ones) the main novelty of this work is the way we combine by reformulating constructive negation aspect in an implementation oriented way. In fact, results for a concrete implementation extending the Ciao Prolog compiler are presented. Due to space limitations, we assume some familiarity with constructive negation techniques (Chan 1988; Chan 1989).

Esto también puede ampliarse

On the side of related work, unfortunately we cannot compare our work with any existing implementation of classical constructive negation in Prolog (even with implementations of other negation techniques, like intensional negation (Barbuti et al. 1987; Bruscoli et al. 1994; Muñoz et al. 2004) or negation as instantiation (Pierro and Drabent 1996) where many papers discuss the theoretical aspects but not implementation details) because we have not found in the literature any reported practical realization. However, there are some very interesting experiences: notably XSB prototypes implementing well-founded semantics ((Alferes et al. 1995)). Less interesting seem to be the implementation of constructive negation reported in (Barták 1998) because of the severe limitations in source programs (they cannot contain free variables in clauses) and the prototype sketched in (Alvez et al. 2004)

[1] We understand Prolog as depth-first, left to right implementation of SLD resolution for Horn clause programs, ignoring, in principle, side effects, cuts, etc.

where a botton-up computation of literal answers is discussed (no execution times are reported but it is easy to deduce inneficiency both in terms of time and memory).

Desarrolar algo mas

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) and optimizations (Section 3.3). Section 4 provides some experimental results and Section 5 talks about a variant of our implementation for negating goals that have a finite number of solutions. Finally, we conclude and outline some future work in Section 6.

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 belief that these problems cannot be easily and efficiently overcome. Therefore, we decided to design an implementation from scratch. One of our additional requirements is that we want to use a standard Prolog implementation to enable that Prolog programs with negation could reuse libraries and existing Prolog code. Additionaly, we want to maintain the efficiency of these Prolog programs, at least for the part that does not use negation. In this sense we will avoid implementation-level manipulations that would delay simple programs without negations.

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

2.1 Frontier

Firstly, we present Chan's definition of frontier (we actually owe the formal definition to Stuckey (Stuckey 1995)).

Detallar esto un poco

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*.

Poner un ejemplo de frontera

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 clauses of

the program.[2] 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: $p(\overline{X}^1) : -C'_1 \cdot$ $\ldots$ $p(\overline{X}^N) : -C'_N \cdot$

  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 $\overline{X}^i$.

The definition is an easy adaptation of Chan's one, but it is also a simple example of the way we attack the problem, reformulating yet defined concepts in an implementation oriented way.
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:

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

Pintar en forma de arbol

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. That is, $\neg G \equiv \neg Frontier(G) \equiv \neg C_1 \wedge \ldots \wedge \neg C_N$.

Therefore, the solutions of $cneg(G)$ are the result of the combination (conjunction) of one solution of each $\neg C_i$. So, 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*.

2.2 Preparation

Before negating a conjunction obtained from the frontier, we have to simplify, organize, and normalize this conjunction. The basic ideas are present in (Chan 1988) in a rather obscure way. In Chan's papers, and then in Stuckey's one, it is simply stated that the conjunction is negated using logic standard techniques. Of course,

[2] Nevertheless, we plan to generate the frontier in a more efficient way by using abstract interpretation over the input program for detecting the degree of evaluation of a term that will be necesary at execution time.

it is true but it is not so easy in the middle of a Prolog computation because we have not access to the whole formula or predicate we are executing.[3]

- **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* (e.g. $X \neq X$), we can simplify $C_i \equiv$ *fail*. The simplification 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, G, 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.

Aquí hace falta un ejemplo de cómo se aplican estas cosas

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 infinite number of solutions. So, we have to use the classical 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:
$\neg C_i \equiv \neg\overline{I} \vee (\overline{I} \wedge \neg\overline{D}_{imp}) \vee (\overline{I} \wedge \overline{D}_{imp} \wedge \neg\overline{R}_{imp}) \vee \overline{I} \wedge \overline{D}_{imp} \wedge \overline{R}_{imp} \wedge \neg(\overline{D}_{exp} \wedge \overline{R}_{exp}))$

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

[3] unless we use metaprogramming techniques that we try to avoid for efficiency reasons

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:

Otro ejemplo

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_i that contains NI components.
- **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 [4], 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:
  $\overline{I} \wedge \neg D_1$
  $\overline{I} \wedge D_1 \wedge \neg D_2$
  $\ldots$
  $\overline{I} \wedge D_1 \wedge \ldots \wedge D_{N_{D_{imp}}-1} \wedge \neg D_{N_{D_{imp}}}$
  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:
  $\overline{I} \wedge \overline{D}_{imp} \wedge \neg R_1$
  $\overline{I} \wedge \overline{D}_{imp} \wedge R_1 \wedge \neg R_2$
  $\ldots$
  $\overline{I} \wedge \overline{D}_{imp} \wedge R_1 \wedge \ldots \wedge R_{N_{R_{imp}}-1} \wedge \neg R_{N_{R_{imp}}}$
  where $\neg R_i \equiv cneg(R_i)$. Constructive negation is again applied over R_i recursively using this operational semantics.
- **Negation of** $\overline{D}_{exp} \wedge \overline{R}_{exp}$. This conjunction cannot be separated 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 entire constructive negation algorithm must be applied again. Notice the recursive application of constructive negation. However, the previous steps could have generated an answer for the original negated goal. Of course it is possible to produce infinitely many answer to a negated goal.

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

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 obtain 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. That is, we are really negating a subgoal of the form: $\exists \overline{V}_{exp} \overline{D}_{exp} \wedge \overline{R}_{exp}$. Its negation is $\forall \overline{V}_{exp} \neg(\overline{D}_{exp} \wedge \overline{R}_{exp})$ and therefore, we will provide the last group of answers that comes from:

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

Más ejemplillos

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. It is possible to handle the code of clauses during the execution thanks to the Ciao package system (Cabeza and Hermenegildo 2000), which allows the code to be expanded at run time. 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).

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

Ejemplo: De hecho la primera versión lo tenia y lo quitamos por falta de espacio. Recuperalo.

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 universally 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 (Muñoz and Moreno-Navarro 2000), the inclusion of disequalities and constrained answers has a very low cost. From the theoretical point of view, it incorporates negative normal form constraints (in the form of conjuntion of equalities plus conjunction of disjunctions of possibly universally quantified disequations) instead of simple bindings as the decomposition step can produce disjunctions. In the implementation side,

attributed variables are used which associate a data structure, containing a normal form constraint, to any variable: $\underbrace{(\bigwedge_j \forall \overline{Z}_j^1 (Y_j^1 \neq s_j^1) \vee \ldots \vee \bigwedge_l \forall \overline{Z}_l^n (Y_l^n \neq s_l^n))}_{\text{negative information}}$

Additionaly, a Prolog predicate `=/= /2` has been defined, used to check disequalities, similarly to explicit unification (`=`). Each constraint is a disjunction of conjunctions of disequalities. A universal quantification in a disequality (e.g., $\forall Y\, X \neq c(Y)$), is represented with a new constructor `fA`/1 (e.g., `X =/= c(fA(Y))`). Due to the lack of space, we refer the interested reader to check the details in (Muñoz and Moreno-Navarro 2000).

Ejemplos. Aquí puede expandirse algo. También estaba en la primera versión y está comentado en

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 thanks to our constraint normal form, providing fewer solutions from the negation of $\overline{I}$ and $\overline{D}_{imp}$. 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 because it is no relevant for the example):

```
?- 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 the formula $(X \neq 0 \wedge Y \neq s(Z)) \vee X \neq Y \vee X \neq Z \vee X \neq W \vee (X \neq s(0) \wedge Z \neq 0)$ that can be represented in our constraint normal form and, therefore, managed by attributes to the involved variables), instead of returning the six equivalent 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
```

In this case we get the whole disjunction at once instead of getting it by backtraking step by step. The generation of compact formulas in the negation of subformulas (see above Second step) is used whenever possible (in the negation of $\overline{I}$ and the negation of $\overline{D}_{imp}$). The negation of $\overline{R}_{imp}$ and the negation of $(\overline{D}_{exp} \vee \overline{R}_{exp})$ can have infinite solutions whose disjunction would be impossible to compute. So, for these cases we construct incrementally the solutions using backtracking.

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:

$Frontier(p(s(0), s(s(0)))) \equiv$
$Step1 \quad X = s(0) \wedge Y = s(s(0)) \wedge greater(X, Y) \wedge q(X, Y, Z) \wedge r(Z) \equiv$
$Step2 \quad greater(s(0), s(s(0))) \wedge q(s(0), s(s(0)), Z) \wedge r(Z) \equiv$
$Step3 \quad fail \wedge q(s(0), s(s(0)), Z) \wedge r(Z) \equiv$
$Step4 \quad fail$

The first step is to expand the code of the subgoals of the frontier to the combination of the code of all their clauses (disjunction of conjunctions in general but only one conjunction in this case because $p/2$ is defined by one only clause), and the result will be a very complicated and hard to check frontier. However, the process is optimized by evaluating ground terms (Step 2). 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 of a successful term in the third step:

$Frontier(p(s(s(0)), s(0))) \equiv$
$Step1 \quad X = s(s(0)) \wedge Y = s(0) \wedge greater(X, Y) \wedge q(X, Y, Z) \wedge r(Z) \equiv$
$Step2 \quad greater(s(s(0)), s(0)) \wedge q(s(s(0)), s(0), Z) \wedge r(Z) \equiv$
$Step3 \quad true \wedge q(s(s(0)), s(0), Z) \wedge r(Z) \equiv$
$Step4 \quad q(s(s(0)), s(0), Z) \wedge r(Z)$

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 (Shoenfield 1967) 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 (notice that subindexes are different):

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

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

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

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 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.

4 Experimental results

Our prototype is a simple library that is added to the set of libraries of Ciao Prolog. Indeed, it is easy to port the library to other Prolog compilers. The only requirement is that attributed variables should be available.

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

4.1 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 (Stuckey 1995) 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
```

(Stuckey 1995) contains another example showing how a constructive answer ($\forall T\, X \neq s(T)$) is provided for the negation of an undefined 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)) ?
yes
```

Notice that if we would ask for a second answer, then it will loop according to

the Prolog resolution. An example with an infinite number of solutions is 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
```

4.2 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. Some goals were executed a number of times to get a significant measurement. All of them were made using Ciao Prolog[5] 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.

5 Finite Constructive Negation

The problem with the constructive negation algorithm is of course efficiency. It is the price that it has to be paid for a powerful mechanism that negates any kind of goal. Thinking of Prolog programs, many goals have a finite number of solutions (we are considering also that this can be discovered in finite time, of course). There is a simplification of the constructive negation algorithm that we use to negate these goals. It is very simple in the sense that if we have a goal $\neg G$ where the solution of the positive subgoal G is a set of n solutions like $\{S_1, S_2, \cdots, S_n\}$, then we can consider these equivalences: $\neg G \equiv \neg(S_1 \vee S_2 \vee \cdots \vee S_n) \equiv (\neg S_1 \wedge \neg S_2 \wedge \cdots \wedge \neg S_n)$

Of course, these solutions are a conjunction of unifications (equalities) and disequality constraints. As described in section 3.2, we know how to handle and negate this kind of information. The implementation of the predicate $cnegf/1$ is something akin to

[5] The negation system is coded as a library module ("package" (Cabeza and Hermenegildo 2000)), 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(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(500000)	2930	2949	41929	14.21
positive(1000000)	3820	3689	81840	22.18
greater(500000,500000)	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 comparation*

```
cnegf(Goal):-
  varset(Goal,GVars), % Getting variables of the Goal
  setof(GVars,Goal,LValores),!, % Getting the solutions
  cneg_solutions(GVars,LValores). % Negating solutions
cnegf(_Goal). % Without solutions, the negation succeeds
```

where *cneg_solutions*/2 is the predicate that negates the disjunction of conjunctions of solutions of the goal that we are negating. It works as described in section 2.3, but it is simpler, because here we are only negating equalities and disequalities.

We get the set of variables, *GVars*, of the goal, *Goal*, that we want to negate (we

use the predicate *varset*/2). Then we use the *setof*/3 predicate to get the values of the variables of *GVars* for each solution of *Goal*. For example, if we want to evaluate $cnegf(boole(X))$, then we get $varset(boole(X), [X])$, $setof([X], boole(X), [[0], [1]])$ (i.e. $X = 0 \vee X = 1$) and *cneg_solutions*/2 will return $X \neq 0 \wedge X \neq 1$.

If we have the goal $p(X, Y)$, which, has two solutions $X = a$, $Y = b$ and $X = c$, $Y = d$, then, in the evaluation of $cnegf(p(X, Y))$, we will get $varset(p(X, Y), [X, Y])$, $setof([X, Y], p(X, Y), [[a, b], [c, d]])$ (i.e. $(X = a \wedge Y = b) \vee (X = c \wedge Y = d)$) and *cneg_solutions*/2 will return the four solutions $(X \neq a \wedge X \neq c) \vee (X \neq a \wedge Y \neq d) \vee (Y \neq b \wedge X \neq c) \vee (Y \neq b \wedge Y \neq d)$.

5.1 Analysis of the number of solutions

The optimization below is very intuitive but, perhaps, the main problem is to detect when a goal is going to have a finite number of solutions. To get sound results, we are going to use this technique (finite constructive negation) just to negate the goals that, we are sure do not have infinite solutions. So, our analysis is conservative.

We use a combination of two analyses to determine if a goal G can be negated with *cnegf*/2: the non-failure analysis (if G does not fail) and the analysis of upper cost (López-García et al. 1997) (if G has an upper cost inferior to infinite). Both are implemented in the Ciao Prolog precompiler (Hermenegildo et al. 1999). Indeed, finite constructive negation can handle the negation of failure that is success. So the finite upper cost analysis is enough in practice. We test these analyses at compilation time and then, when possible, we directly execute the optimized version of constructive negation at compilation time.

It is more complicated to check this at execution time although we could provide a rough approximation. First, we get a maximun number N of solutions of G (we can use the library predicate *findnsols*/4) and then we check the number of solutions that we have obtained. If it is less than N, we can assure that G has a finite number of solutions and otherwise we do not know.

5.2 Experimental results

We have implemented a predicate *cnegf*/1 to negate the disjunction of all the solutions of its argument (a goal). The implementation of this predicate takes advantage of backtracking to obtain only the information that we need to get the first answer. Then, if the user asks for another answer, the backtracking gets information enough to provide it. Accordingly, we avoid the complete evaluation of the negation of all the solutions first time round. We negate the subterms only when we need to provide the next solution. In this sense, if we have the goal G (where each S_i is the conjunction of Ni equalities or disequalities) $G \equiv S_1 \vee \cdots \vee S_n \equiv (S_1^1 \wedge \cdots \wedge S_1^{N1}) \vee \cdots \vee (S_n^1 \wedge \cdots \wedge S_n^{Nn})$ and we then want to obtain $\neg G$, then we have $\neg G \equiv \neg(S_1 \vee S_2 \vee \cdots \vee S_n) \equiv (\neg S_1 \wedge \neg S_2 \wedge \cdots \wedge \neg S_n) \equiv \neg(S_1^1 \wedge \cdots \wedge S_1^{N1}) \wedge \cdots \wedge \neg(S_n^1 \wedge \cdots \wedge S_n^{Nn}) \equiv (\neg S_1^1 \vee \cdots \vee \neg S_1^{N1}) \wedge \cdots \wedge (\neg S_n^1 \vee \cdots \vee \neg S_n^{Nn}) \equiv (\neg S_1^1 \wedge \cdots \wedge \neg S_n^1) \vee \cdots \vee (\neg S_1^{N1} \wedge \cdots \wedge \neg S_n^{Nn})$ we begin calculating just the first answer of the negation that will be $(\neg S_1^1 \wedge \cdots \wedge \neg S_n^1)$, and the rest will be calculated if necessary using backtracking.

Let us present a simple example:

goals	Goal	cneg(Goal)	cnegf(Goal)	ratio
boole(1)	821	831	822	1,01
positive(s(s(s(0))))	811	1351	860	1,57
greater(s(0),s(s(0)))	772	1210	840	1,44
positive($s^{500000}(0)$)	1564	8259	3213	2,57
positive($s^{7500000}(0)$)	2846	12445	3255	3,82
greater($s^{50000}(0)$,$s^{50000}(0)$)	1240	66758	30112	2,21
boole(X)	900	1321	881	1,49
greater(s(s(s(0))),X)	990	1113	1090	1,02
queens(s(s(s(0))),Qs)	1481	50160	1402	35,77

Table 2. *Runtime comparison of finite constructive negation*

```
?- member(3,[X,Y,Z]).
X = 3 ? ;
Y = 3 ? ;
Z = 3 ? ;
no
```

```
?- cnegf(member(3,[X,Y,Z])).
X=/=3, Y=/=3, Z=/=3 ?;
no
```

We get the symmetric behavior for the negation of the negation of the initial query

```
?- cnegf(cnegf(member(3,[X,Y,Z]))).
X = 3 ? ;
Y = 3 ? ;
Z = 3 ? ;
no
```

We checked some time results in Table 2. The results are much more significant for more complicated goals. Indeed, the more complicated the code of a predicate is, the more inefficient its classical constructive negation (*cneg*) is. However, finite constructive negation (*cnegf*) is independent of code complexity. Finite constructive negation depends on the complexity of the solutions obtained for the positive goal and, of course, the number of solutions of this goal.

6 Conclusion and Future Work

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

Having given a detailed specification of the algorithm in a detailed way we proceed to provide a real, complete and consistent implementation. To our knowledge it is the first reported work of a running implementation of constructive negation in Prolog from its definition in 1988. The results we have reported are very encouraging, because we have proved that it is possible to extend Prolog with a constructive negation module relatively inexpensively and overall without any delay in Prolog programs that are not using this negation. 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 (Moreno-Navarro 1996)). The full version of the paper will provide more details as well as these additional optimizations. Another possible future work is to incorporate our algorithm at the WAM machine level.

Reescribir

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 (Muñoz and Moreno-Navarro 2000; Muñoz et al. 2001) we systematically studied what we understood to be the most interesting existing proposals: negation as failure (*naf*) (Clark 1978), use of delays to apply *naf* securely (Naish 1986), intensional negation (Barbuti et al. 1987; Barbuti et al. 1990), and constructive negation (Chan 1988; Chan 1989; Drabent 1995; Stuckey 1991; Stuckey 1995). 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 (Muñoz et al. 2001). So, in many cases, we avoid the inefficiency of classical constructive negation. However, we still need it because it is the only method that is sound and complete for any kind of goals. For example, looking at the goals in Table 1, the strategy will obtain all ground negations using the *naf* technique and it would only use classical constructive negation for the goals with variables where it is impossible to use *naf* . Otherwise, the strategy will use finite constructive negation (*cnegf*) for the three last goals of Table 2 because the positive goals have a finite number of solutions.

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 [6].

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

Appendix: Negation Examples

One of the problems related to negation in logic programming is that there are no real logic programs which use (constructive) negation. The reason is that the absent of negation have oblied logic programmers to avoid negation is their programs loosing expresiveness or simplicity of programs.

Our approach is supported by our implementation. So, we have also provided a set of examples in logic programming using negation in a constructive way. They provide from papers and recopilation from other authors.

Even and Odd

A different program to compute even and odd numbers:

¿Estos ejemplos son los usados para las medidas?

```
sum(0,X,X).
sum(s(X),Y,s(Z)):- sum(X,Y,Z).

even(X):- sum(Y,Y,X).

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

Odd numbers can be generated as constraints:

```
?- odd(X).

X=/=s(fA(_A)),X/0 ? ;

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

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

...
```

Or including types in the definition:

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

odd(X):- number(X), cneg(even(X)).
```

We can provide instanciated answers:

```
?- odd(X).

X = s(0) ? ;

X = s(s(s(0))) ? ;

X = s(s(s(s(s(0))))) ? ;

...
```

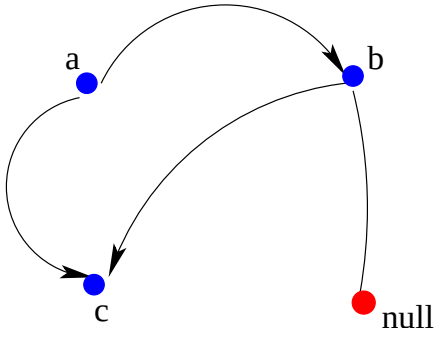

Fig. 1. Graph

Insert

Insert elements in a list without repetitions:

```
member(X,[X|Ys]).
member(X,[Y|Ys]):- member(X,Ys).

insert(X,Xs,[X|Xs]):- cneg(member(X,Xs)).
insert(X,Xs,Xs):- member(X,Xs).
```

It provides interesting constructive answers:

```
?- insert(X,[3,4],L).

L = [X,3,4],
X=/=3,X=/=4 ? ;

L = [3,4],
X = 3 ? ;

L = [3,4],
X = 4 ? ;

no

?- insert(X,[Y],L2).

L2 = [X,Y],
X=/=Y ? ;

L2 = [X],
Y = X ? ;

no
```

Graphs

We represent an oriented graph using *next*/2. The predicate *path*/2 succeeds if there is a path between two nodes without cicles. We define that a node is save if it is not connected to the node "null" (see Figure 1):

```
next(a,b).
next(a,c).
next(b,c).
next(b,null).

path(X,X).
path(X,Y):- X=/=Y, next(X,Z),path(Z,Y).

save(X):- cneg(path(X,null)).
```

So, we can ask for the nodes that are saved:

```
?- save(X).

X=/=b,X=/=a,X=/=d ? ;

no
```

Bartak

The following example was introduced by Bartak in (Barták 1998). In http://kti.ms.mff.cuni.czJ bartak/html/negation.html there is a prototype for constructive negation restricted to the case of finite computation trees.

```
p(a,f(Z)):- t(Z).
p(f(Z),b):- t(Z).
t(c).
```

Our implementation probide ~~the following answers:~~

provide

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

X=/=f(fA(_A)) ;

X=/=a,
Y=/=b ? ;

X = f(_A),
Y = b,
_A=/=c ? ;

Y=/=f(fA(_A)),
X=/=f(fA(_B)) ? ;

Y=/=b,Y=/=f(fA(_A)) ? ;

X = a,
Y = f(_A),
_A=/=c ? ;

no
```

Symmetric

The next program computes symmetric (non-symmetric) terms of the signature f2/2,f1/1,o/0.

```
symmetric(o).
symmetric(f1(X)):- symmetric(X).
symmetric(f2(X,Y)):- mirror(X,Y).

mirror(o,o) .
mirror(f1(X),f1(Y)):- mirror(X,Y).
mirror(f2(X,Y),f2(Z,W)):- mirror(X,W),mirror(Y,Z).
```

The query obtains the list of non symmetric terms:

```
?- cneg(symmetric(Z)).

Z=/=f2(fA(_B),fA(_A)),Z=/=o,Z=/=f1(fA(_C)) ? ;

Z = f2(_A,_),
_A=/=f2(fA(_C),fA(_B)),_A=/=o,_A=/=f1(fA(_D)) ? ;

Z = f2(_A,_B),
_A=/=f1(fA(_E)),_A=/=o,
_B=/=f2(fA(_D),fA(_C)) ? ;

Z = f2(f2(_A,_),f2(_,_)),
_A=/=f2(fA(_C),fA(_B)),_A=/=o,_A=/=f1(fA(_D)) ? ;

Z = f2(f2(_D,_),f2(_,_A)),
_D=/=f1(fA(_E)),_D=/=o,
_A=/=f2(fA(_C),fA(_B)) ? ;

Z = f2(f2(f2(_A,_),_),f2(_,f2(_,_))),
_A=/=f2(fA(_C),fA(_B)),_A=/=o,_A=/=f1(fA(_D)) ? ;

...
```

Duplicates

Definition of a predicate that checks or generates a list with duplicated elements from (Chan 1988).

```
member(X,[X|_]).
member(X,[_|Y]):- member(X,Y).

has_duplicates([X|Y]):- member(X,Y).
has_duplicates([_|Y]):- has_duplicates(Y).

list_of_digits([ ]).
list_of_digits([X|Y]):- digit(X),list_of_digits(Y).

digit(1).
digit(2).
digit(3).
```

The following question is an example of generate and test:

```
?- L=[_,_,_], cneg(has_duplicates(L)), list_of_digits(L).

L = [1,2,3] ? ;

L = [1,3,2] ? ;

L = [2,1,3] ? ;

L = [2,3,1] ? ;

L = [3,1,2] ? ;

L = [3,2,1] ? ;

no
```

Disjoint

The intuitive definition of disjoint lists includes a negation:

```
disjoint([],_).
disjoint([X|L1],L2):- cneg(member(X,L2)), disjoint(L1,L2).
```

With the above definition of $list_of_digits/1$ we can make the following queries:

```
?-  disjoint([1,2,3],[X,Y]).

Y=/=3,Y=/=1,Y=/=2,
X=/=3,X=/=1,X=/=2 ? ;

no
?- list_of_digits([X,Y]),disjoint([1,2,3],[X,Y]).

no
?- list_of_digits([X,Y]),disjoint([1,2],[X,Y]).

X = 3,
Y = 3 ? ;

no
```

References

Alferes, J. J., Damsio, C. V., and Pereira, L. M. 1995. A logic programming system for non-monotonic reasoning. In *Journal of Automated Reasoning*. Vol. 14(1). 93–147.

Alvez, J., Lucio, P., Orejas, F., Pasarella, E., and Pino, E. 2004. Constructive negation by bottom-up computation of literal answers,. In *Proceedings of the 2004 ACM Symposium on Applied Computing*. ACM, vol. 2. Nicosia (Cyprus), 1468–1475.

Barbuti, R., Mancarella, D., Pedreschi, D., and Turini, F. 1987. Intensional negation of logic programs. *LNCS 250*, 96–110.

Barbuti, R., Mancarella, D., Pedreschi, D., and Turini, F. 1990. A transformational approach to negation in logic programming. *JLP 8*, 3, 201–228.

Barták, R. 1998. Constructive negation in clp(h). Tech. Report 98/6, Department of Theoretical Computer Science, Charles University, Prague. July.

Bruscoli, P., Levi, F., Levi, G., and Meo, M. C. 1994. Compilative Constructive Negation in Constraint Logic Programs. In *Proc. of the 19th International CAAP '94*, S. Tyson, Ed. LNCS, vol. 787. Springer-Verlag, Berlin, 52–67.

Cabeza, D. and Hermenegildo, M. 2000. A New Module System for Prolog. In *CL2000*. Number 1861 in LNAI. Springer-Verlag, 131–148.

Chan, D. 1988. Constructive negation based on the complete database. In *Proc. Int. Conference on LP'88*. The MIT Press, 111–125.

Chan, D. 1989. An extension of constructive negation and its application in coroutining. In *Proc. NACLP'89*. The MIT Press, 477–493.

Clark, K. L. 1978. Negation as failure. In Logic and Data Bases, H. Gallaire and J. Minker, Eds. 293–322.

Drabent, W. 1995. What is a failure? An approach to constructive negation. *Acta Informatica. 33*, 27–59.

Hermenegildo, M., Bueno, F., Puebla, G., and López-García, P. 1999. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 ICLP*. MIT Press, Cambridge, MA, 52–66.

Kunen, K. 1987. Negation in logic programming. *JLP 4*, 289–308.

López-García, P., Hermenegildo, M., Debray, S., and Lin, N. W. 1997. Lower bound cost estimation for logic programs. In *1997 International Logic Programming Symposium*. MIT Press.

Moreno-Navarro, J. J. 1996. Extending constructive negation for partial functions in lazy narrowing-based languages. *ELP*.

Muñoz, S., Mariño, J., and Moreno-Navarro, J. J. 2004. Constructive intensional negation. In *Proceedings of the 7th International Symposiun in Functional and Logic Programming, FLOPS'04*. Number 2998 in LNCS. Nara, Japan, 39–54.

Muñoz, S. and Moreno-Navarro, J. J. 2000. How to incorporate negation in a prolog compiler. In *2nd International Workshop PADL'2000*, E. Pontelli and V. S. Costa, Eds. LNCS, vol. 1753. Springer-Verlag, Boston, MA (USA), 124–140.

Muñoz, S., Moreno-Navarro, J. J., and Hermenegildo, M. 2001. Efficient negation using abstract interpretation. In *LPAR 2001*, R. Nieuwenhuis and A. Voronkov, Eds. Number 2250 in LNAI. La Habana (Cuba), 485–494.

Naish, L. 1986. *Negation and control in Prolog*. Number 238 in Lecture Notes in Computer Science. Springer-Verlag, New York.

Pierro, A. D. and Drabent, W. 1996. On negation as instantiation. *Proceedings of The Fifth International Conference on Algebraic and Logic Programming ALP'96*.

Shoenfield, J. R. 1967. *Mathematical Logic*. Association for Symbolic Logic.

Stuckey, P. 1991. Constructive negation for constraint logic programming. In Proc. IEEE Symp. on Logic in Computer Science. *660*.

Stuckey, P. 1995. Negation and constraint logic programming. In Information and Computation. *118(1)*, 12–33.