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 dealing with logical behaviors, rules and knowledge. Problems involving negation are quite common in system specification and include natural language processing, program optimization, transformation and composition, diagnosis, etc. However, their representation is rather limited using Prolog 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 pretty constrained, and a programmer cannot find a correct and complete implementation available.

Therefore, our goal is to achieve a real Prolog implementation of constructive negation. Studying the original papers, we found many operational problems in the methods proposed.

In this paper, we refine and propose some extensions to the classical method of constructive negation, providing the complete theoretical and operational algorithm. Furthermore, we also discuss implementation issues providing a preliminary implementation and also an optimized one to negate goals that have a finite number of solutions. These are, to our knowledge, the first available implementation of sound and complete constructive negation in Prolog.

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

1 Introduction

The beginning of logic is tied with that of scientific thinking (Sterling and Shapiro 1987). Logic provides a precise language for representing a goal, knowledge and assumptions. Moreover, logic allows consequences to be deduced from premises to study the truth or falsity of statements from the truth or falsity of others.

At the inception of computers construction-related difficulties were clearly so dominant that the language for expressing problems and instructing the computer how to solve them was designed from the perspective of computer engineering alone.

The use of logic directly as a programming language is called *Logic Programming*. Very close to this kind of programming is *Functional Programming*, which uses mathematical functions to program. Both constitute *Declarative Programming*, whose reasoning is closer to human reasoning (based on logic predicates or mathematical functions) than the conventional machine-based *Imperative Programming*.

The development of the formal foundations of logic programming began in the late 1970s, especially with the works (van Emden and Kowalski 1976; Clark 1978; Reiter 1978). Further progress in this direction was achieved in the early 1980s, leading to the appearance of the first book on the foundations of logic programming (Lloyd 1987). The selection of logic programming as the underlying paradigm for the Japanese Fifth Generation Computer Systems Project led to the rapid proliferation of various logic programming languages.

Round about 1980, *non-monotonic reasoning* entered computer science and began to constitute a new field of active research. It was originally initiated because *Knowledge Representation* and *Common-Sense Reasoning* using classical logic had reached its limits. Formalisms like classical logic are inherently monotonic and they seem to be too weak and, therefore, inadequate for such reasoning problems (Dix et al. 1997).

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 behaviors, 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, negative queries (search of false information), etc.

There are many ways of understanding and incorporating negation into Logic Programming), the problems really start at the semantic level, where the different proposals differ not only as to semantics but also as to expressiveness. 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 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 (Börger 1987)), which is present in Nu-Prolog, Gödel, and Prolog systems that implement delays (most of the above).

1.1 Related Work: Operational Approaches

Solving the negation of a ground literal is an easy task using negation as failure (Shepherdson 1984; Shepherdson 1985), but what about non-ground negative lit-

[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.

erals?. Most procedural semantics of logic programs guarantees completeness for only "non-floundered" queries, whose evaluation does not involve the selection of a non-ground negative literal (Bol and Degerstedt 1993; Chen and Warren 1996; Przymusinski 1989a; Ross 1992). Similar restrictions are placed by bottom-up methods of query evaluation such that negation can be implemented by set difference (Naughton and Ramakrishnan 1991; Stuckey and Sudarshan 1993).

From a user's point of view, it is desirable to get constructive answers for non-ground negative literals. This would be, similar behavior to positive queries, but we are introducing the concept of disequality as the negation of an equality (the unification of a variable with a term). In other words, we are dealing with constraints over the Herbrand Universe.

In terms of the mechanisms to solve a non-ground negative literal, several different techniques have been developed,varying as to the degree to which the corresponding positive literal is evaluated. We can cite a some of them:

- In(Chan 1988; Chan 1989; Stuckey 1991), a negative literal is solved using Clark's completed definitions at run time, possibly with partial evaluation. Quantified complex formulas have to be transformed into disjunctive normal forms and be dealt with explicitly. We call classical constructive negation to the one proposed in (Chan 1989)[2]. The first approach (Chan 1988) was a simple variant for negating goals that have a finite number of solutions [3]. Since quantified disequations must appear in any scheme for constructive negation that is applicable to all logic programs, there is no impediment to extending the syntax of bodies and goals to include disequations, and this is exactly what Chan describes. Thus modified, logic programs begin to look like constraint logic programs over the Herbrand Universe with constraints of the form $s = t$ and $\forall Z\, s \neq t$. Similarly, answers are conjunctions of these constraints. Along this line, Stuckey (Stuckey 1995) proposes that the natural formulation of constructive negation is in terms of constraint logic programming. We will also use constraints. When a goal has a finite number of answers, Chan's scheme (Chan 1988) and Stuckey's approach (Stuckey 1991) will both return for the negation a finite set of (treating the sets as disjunctions) obtained from the negation of the answers of the positive goal. Chan's approach to constructive negation is quite entrenched in the Herbrand Universe. Both SLD-CNF resolution and (of course) the constraint simplification procedure are specific to the Herbrand Universe. Our approach follows Chan's scheme, since it is solely applicable to the Herbrand Universe, to implement a negation subsystem for Prolog, not for general CLP languages.
- Other schemes for constructive negation that do not introduce quantified negations apply only to specific kinds of programs. For example, (Foo et al. 1988) gives a scheme for constructive negation for function-free programs

[2] In this work, we propose an optimization of classical constructive negation (see Section 2) to implement a general, sound and complete operational semantics for constructive negation.
[3] Taking (Chan 1988) as a basis, we have implemented an efficient version of constructive negation that can only be used for negating goals with a finite number of answers (see Section 5).

(DATALOG). By deducing a finite *relevant type* for each predicate, they can ensure that a negative subgoal is always ground by simply grounding the variables appearing in the subgoal using predicates representing the deduced type. Wallace (Wallace 1987) also gives a scheme for function-free programs including (unquantified) disequations, where the number of constants is infinite.

- Sato and Motoyoshi (Sato and Motoyoshi 1991) defined a top-down interpreter for logic programs with negation, based on unfolding. They provide a completeness result with respect to Kunen's three-valued semantics (Kunen 1987) similar to Stuckey's (Stuckey 1995). The interpreter they describe revolves around a method of generalized quantifier elimination specific to the Herbrand Universe and does not appear readily extensible to other domains. As it is based on unfolding, its breadth-first nature makes it appear difficult to implement efficiently.
- Przymusinski (Przymusinski 1989b) first studied constructive negation under perfect model semantics and developed SLSC-resolution for the constructive negation of stratified programs. (Liu et al. 1999) presents SLG_{CN} resolution for effective query evaluation of function-free programs with constructive negation under well-founded semantics. Termination is guaranteed due to the reduction of constraint answers to a normal form, redundant answer elimination and tabled evaluation.
  Like SLG resolution (Chen and Warren 1996), SLG_{CN} resolution is formalized directly in such a way that repeated computation is avoided in a subgoal. This is achieved by delaying only ground negative literals. It remains a challenge to extend SLG_{CN} resolution to general cases where non-ground negative literals may have to be delayed, while avoiding any repetition of computation in a subgoal. Delaying non-ground negative literals directly means that constructive negation can be applied even if the constraints and bindings of variables in a negative literal are not completely determined. This achieves the same effect as the notion of frontiers in (Drabent 1995; Fages 1997) and the TU-forests in (Damásio 1996). The open problem is how to control the delaying of non-ground negative literals and the iterated propagation of constraints to avoid repeated computation.
- Another approach (Maluszynski and Näslund 1989) is aimed directly at allowing negative goals to return computed answers. A positive derivation for G returns γ such that $Comp(P) \models \forall G\gamma$, this approach attempts to let a negative derivation for G return θ such that $Comp(P) \models \forall \neg G\theta$. θ is called a fail substitution. Unfortunately, (Lassez and Marriot 1987) have shown that answers to negative queries cannot generally be represented by finite positive information alone, thus this approach must sometimes return an infinite number of fail substitutions.
- In (Drabent 1995; Fages 1997; Maluszynski and Näslund 1989), substitutions called *fail answers* are generated for variables in a negative literal $\neg A$ based upon a *frontier* of the positive literal A. This is a powerful technique, since A does not have to be completely evaluated before an answer for $\neg A$ is de-

rived. Drabent (Drabent 1995) described SLSFA-resolution for constructive negation under well-founded semantics. SLSC and SLSFA resolutions require infinite failure and are, therefore, not suitable for effective query evaluation. Since a subgoal can have many different frontiers, there is an implementation problem regarding how to control the derivation tree of a subgoal and the choice of frontiers [4]. Drabent proposes a similar approach to (Maluszynski and Näslund 1989) but allows quantified disequations as answers, combining the ideas of fail answers with constructive negation. His scheme has a form of completeness with respect to Kunen's semantics. Drabent defines SLDFA resolution over the Herbrand universe. Drabent's scheme (Drabent 1995) defines SLDFA resolution for Herbrand Universe domains. It can be viewed as a generalization of Chan's first approach (Chan 1988) in which only successful answers to a negative sub-derivation are used in constructing answers to the negative goal. Drabent extends Chan's approach as follows. While (Chan 1988) is only applicable if the negative sub-derivation tree is finite, and in this case the frontier selected is always the success nodes themselves, (Drabent 1995) considers only selecting some finite subset of the success nodes.
The essential step in Drabent's method is as follows: the answers to a negative subgoal $(\neg G \| c)$ are over approximated by taking the success nodes $c_1, \cdots, c_n$ of some frontier to the goal $(G \| true)$ and constructing "answers" $d_1, \cdots, d_n$ such that $c \rightarrow c_1 \vee \cdots \vee c_n \vee d_1 \vee \cdots \vee d_k$. The over approximations can be safely used in determining falsity of formulas. The goal $(c|(\neg G), H)$ is false if each goal $(d_i|H)$ is false.
Drabent shows soundness and completeness results for his approach with respect to Kunen's semantics. Because of the approximation steps, the notion of a SLDFA tree is quite complicated in this scheme, and the completeness results rely on considering all possible SLDFA trees for a particular goal.
In particular, to get a sound and complete implementation of his scheme, Drabent must include some form of iterative deepening or re-execution of negative subgoals to ensure that eventually an accurate enough approximation is obtained to ensure failure for all false goals.

- Apart from (Chan 1988; Chan 1989), in (Bossu and Siegel 1985; Chan 1988; Damásio 1996; Khabaza 1984; Przymusinski 1989b), constraint answers of a negative literal are derived by taking the negation of the disjunction of all the answers of its positive counterpart.
  The work in (Damásio 1996) is a systematic study of constructive negation in tabled query evaluation under well-founded semantics. It extends tabulated resolution for well-founded semantics in (Bol and Degerstedt 1993) with constructive negation. Approximate constraint answers are derived for non-ground negative literals involved in recursion through negation. Due to iterated approximations, constructive negation may be repeatedly applied to the same non-ground negative literal using slightly different answers, caus-

[4] This is why we have not choose this method for our implementations

ing repeated computation inside a subgoal. Although theoretical results of soundness and search space completeness are established, independently of the constraint domain used, pragmatic control issues such as redundant answer elimination, termination and repeated computation are not incorporated into the formalization, for constraint logic programs with function symbols. In particular, no termination result is given for tabled query evaluation with constructive negation.

- Warren (Warren 1992) developed a Prolog meta interpreter for constructive negation that was executed using an OLDT implementation. Constraint answers of negative literals are represented using anti-subsumption constraints. Thus, it is not sound in general. Recursion through negation is not supported and this implementation does not handle constraints with universal and existential quantifications properly.
- There are many papers related to Clark's (Clark 1978) program completion, see (Lloyd 1987; Apt 1990), some of them (Barbuti et al. 1987; Barbuti et al. 1990) oriented to getting a program that is a transformation of an original program P, which also introduces the "only if" part of the predicate definitions (i.e., interpreting implications as equivalences). Programs only contain the "if" halves of the definitions of their predicates. This is the reason why only positive information can be deduced. To deduce negative information, the "only-if" halves of the definitions must be added in a process called program completion. This is the well-known *Intensional Negation*[5]. Sato and Tamaki (Sato and Tamaki 1984) also use a transformation method based on continuation passing. It transforms universally quantified implications, of the form $\forall x (F_1 \rightarrow F2)$, to Prolog programs with disequality constraints. One special case of quantified implication is negation, $\forall x (F_1 \rightarrow \mathbf{f}) = \neg F_1$, where **f** means false.

According to Clark's Equality Theory, CET, (Clark 1978) we can distinguish two groups of schemes for constructive negation:

- Those assuming that the Herbrand Universe has an infinite number of functors (Chan 1988; Wallace 1987; Sato and Motoyoshi 1991). They use the infinite number of functions symbols to simplify handling disequations. In this case, each disequation is independent in the sense that we can determine whether a conjunction of equations and disequations are satisfiable by examining whether each disequation is individually satisfiable with respect to the equations. For this group, Clark's Equality Theory (CET) (Clark 1978) is a complete axiomatization (Maher 1988).
- Those assuming a finite number of functors ((Maluszynski and Näslund 1989; Foo et al. 1988)). They use domain closure to derive positive information from "purely" negative information but are not as generally applicable as

[5] Intensional negation, which is very difficult to implement because of the use of universally quantified goals, is quite promising in efficiency terms. It uses the transformational approach, so it is a "compile-time" technique and significant efficiency is expected. This approach is also studied by the authors of this paper and it will be presented in future publications

the first group. In this case, CET must be extended with a domain closure axiom see (Maher 1988) to arrive at a complete axiomatization. The work of (Drabent 1995) encompasses both groups by using the appropriate complete axiomatization.

From the semantics point of view, most of the previous work on constructive negation with the notable exceptions of (Damásio 1996; Przymusinski 1989b), uses Clark's completion as the respective declarative semantics. It is known, however, that Clark's completion has various drawbacks (Przymusinski and Warren 1992). Although, well-founded semantics (van Gelder et al. 1991) has been accepted as a more natural and robust semantics for logic programs, we are interested in a negation that is compatible with Prolog so that we can use its compilers and take advantage of all the libraries and programs already made using Prolog. This it why we will also work with the Clark's completion (3-valued version), as the first group of approaches described above, in order to have a compatible semantics with ordinary Prolog programs.

By characterizing constructive negation in the CLP framework, Stuckey (Stuckey 1995) extends its applicability to many more languages (in particular, to the domain that Chan describes) constructs a more efficient procedure by treating both the equations and disequations similarly, and provides a completeness result. For the scheme defined by Stuckey (Stuckey 1995), a single derivation tree captures all the information about a goal.

As we have seen in these section the problem of handling "non-ground" literals is tackled from the operational point of view by various techniques of *constructive negation*. Some of them more oriented to achieve an implementation: *intensional negation* (Barbuti et al. 1987; Barbuti et al. 1990), Chan's *constructive negation* (Chan 1988; Chan 1989; Stuckey and Sudarshan 1993), fail substitutions (Maluszynski and Näslund 1989), fail answers (Drabent 1995), etc. From a theoretical viewpoint, it would be enough to implement Chan's approach (because of its completeness results (Stuckey and Sudarshan 1993))[6].

1.2 Motivation

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 proved to be sound and complete, and its semantics is fully compatible with Prolog's one. 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, complete and effective implementation.

[6] This is the reason why we have chosen it as starting point for our work

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 our 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. In order to avoid an excessively long paper, we assume some familiarity with constructive negation techniques (Chan 1988; Chan 1989).

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 or negation as instantiation (Pierro and Drabent 1996), where many papers discuss the theoretical aspects but not implementation details. The reason is that 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) where a bottom-up computation of literal answers is discussed (no execution times are reported but it is easy to deduce inefficiency both in terms of time and memory).

We have started this paper with an introduction about the area, about the previous operational approaches and the ones that are related with our proposal. The remainder of the paper is organized as follows. Section 2 details our constructive negation algorithm into three steps. It explains:

- how to obtain the *frontier* of a goal that is a compact expression which contains all its answers (Section 2.1),
- how to prepare the frontier for being negated (Section 2.2) and finally,
- how to negate the prepared frontier (Section 2.3).

For each step some details about the implementation[7] are given.

Section 3 discusses important additional implementation issues:

- code expansion that is used to obtain the frontier (Section 3.1),
- required disequality constraints that are used to provide negative information (Section 3.2) and
- optimizations of the algorithm and of the implementation (Section 3.3). This is an essential point dealing with constructive negation due to its efficiency problems.

[7] Full code implementation is available in http://babel.ls.fi.upm.es/~susana/code/negation/cneg/. It works on Ciao-Prolog 1.10#8. In other releases built-in predicates can change their position in the libraries and the code of Inge should be adapted easily. If any problem appears contact the authors or the Ciao support team.

Section 4 provides some experimental data based on examples and measures and Section 5 talks about a variant of our implementation for negating goals that have a finite number of solutions. We have called this variant *Finite Constructive Negation* and also experimental results are provided. Finally, we conclude and outline some future work in Section 6.

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 expressiveness or simplicity of programs.

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

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 just 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. Additionally, 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 negation.

We use Kunen's 3-valued completion semantics that was defined by Stukey (Stuckey 1995).

After defining the operational aspects of an implementation step, we provide some additional details regarding our concrete implementation. Remember that full code can be obtained from our website.

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)). We use de classical definition of derivation tree as a ordered parse tree that represents as nodes the derivations from a goal according to the rules of a program.

Definition 1
Frontier

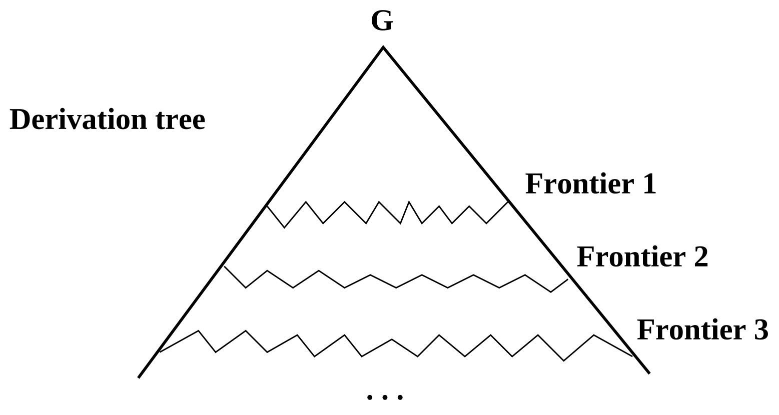

Fig. 1. Frontiers of the goal G

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*.

In the figure 1 there is a representation of different frontiers in a derivation tree of a goal G. 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 the first step of all the possible SLD resolutions. This can be done by a simple inspection of the clauses of the program.[8] 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:

$$\begin{array}{l} p(\overline{t}^1) :- C'_1. \\ \ldots \\ p(\overline{t}^N) :- C'_N. \end{array}$$

  The frontier of the goal, G, has the format: $Frontier(G) \equiv C_1 \vee C_2 \vee \ldots \vee C_N$, where each C_i, with $i \in 1 \ldots N$, is the conjunction of two elements:
  (1) the equalities that are needed to unify the m terms of $\overline{X}$ and the m terms $\overline{t}^i$ (from the head of the clauses). These equalities can be simplified into a

[8] 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 necessary at execution time.

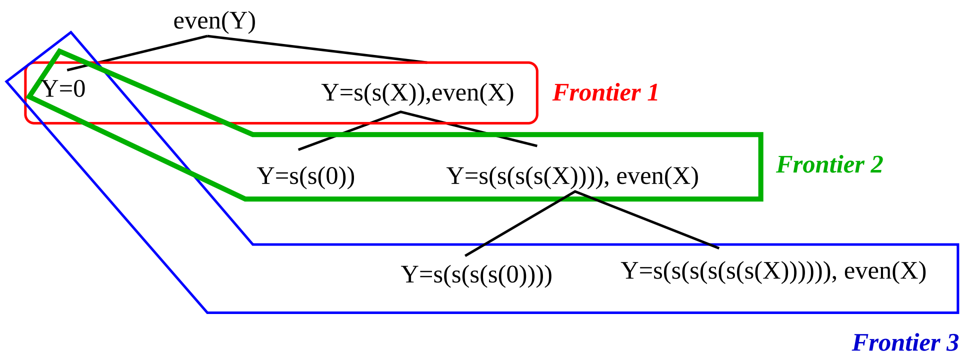

Fig. 2. First frontiers of the goal *even*(Y)

set of simple equalities in between the variables of $\overline{X}$ and the corresponding terms into $\overline{t}^i$.
(2) the conjunction of subgoals C'_i (from the bodies of the clauses)
That is $C_i = (\overline{X} = \overline{t}^i) \wedge C'_i$.

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:

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

The frontier for the goal *even*(Y) is as follows:

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

Indeed, if we continue with the instantiation of the resolution tree we find new frontiers more "instantiated" that are all equivalent to the initial goal. See Figure 2 where three frontiers of the goal *even*(Y) are represented.

Another example is:

```
parent(bob,mary).
parent(mary,joan).

grandparent(Y,X):-
    parent(Y,Z),
    parent(Z,X).
```

The frontier of the goal *grandparent*(Y, X) has only one element in the disjunction:

$$Frontier(grandparent(Y, X)) = \{\exists Z(parent(Y, Z) \wedge parent(Z, X))\}$$

Implementation details: A predicate *frontier*/2 has been implemented. The call *frontier*(G, *Frontier*) obtains the frontier (list of lists of subgoals, i.e. disjunction of conjunctions of subgoals) of the goal G. It is simple achieved by using the code expansion described in section 3.1, that provides predicates code at execution time. Of course, the complexity of this implementation is exponential. □

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, 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.[9]

- **Simplification of the conjunction**. If one of the terms of C_i (one of the elements of the conjunction 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 $C_i \equiv \overline{I} \wedge \overline{D} \wedge \overline{R}$. If we remember the examples that we have provided for frontier:
  $Frontier(even(Y)) = \{C_1 \vee C_2\} = \{(Y = 0) \vee (Y = s(s(X)) \wedge even(X))\}$
  $Frontier(grandparent(Y, X)) = \{C_1'\} = \{\exists Z(parent(Y, Z) \wedge parent(Z, X))$

  We can organize the conjunctions C_1, C_2 and C_1' following the division into equalities, disequalities and rest as follow:
  $C_1 \equiv (Y = 0) \wedge \emptyset \wedge \emptyset$
  $C_2 \equiv \exists X((Y = s(s(X))) \wedge \emptyset \wedge (even(X)))$
  $C_1' \equiv \exists Z(\emptyset \wedge \emptyset \wedge (parent(Y, Z) \wedge parent(Z, X)))$

  Implementation details: We call *GoalVars* the set of variables of G. We have provided a predicate *organization_conj*/5 that receives in a call $organization\_conj(Conj, G, I, D, R)$ the conjunction *Conj* and the goal G. It returns in I the values that are assigned to the variables of *GoalVars* in the equalities that are in *Conj*, in D the list of the disequalities and in R the rest of

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

subgoals of the conjunction Conj. Arguments I, D and R are lists representing conjunctions of equalities, disequalities and other subgoals respectively. The first argument represents conjunctions using a pair $(\mathit{Head}, \mathit{Body})$, where Head is the unification of the goal with the head of a clause (from where equalities, i.e. unifications, can be obtained), and Body is the list that represents the conjunction of subgoals of the body of a clause. This is a comfortable way of representing the idea of frontier in the implementation. In our example, in order to negate the goal $\mathit{even}(Y)$, we call this predicate twice with each of the two conjunctions of the frontier.
First call: $\mathit{organization\_conj}((\mathit{even}(0), []), \mathit{even}(Y), I, D, R)$ returns $I = [Y = 0]$, $D = []$ and $R = []$.
Second call: $\mathit{organization\_conj}((\mathit{even}(s(s(X))), [\mathit{even}(X)]), \mathit{even}(Y), I, D, R)$ returns $I = [Y = s(s(X))]$, $D = []$, and $R = [\mathit{even}(X)]$.
To negate the goal $\mathit{grandparent}(Y, X)$, that is listed above, we call to $\mathit{organization\_conj}((\mathit{grandparent}(A, B), [\mathit{parent}(A, C), \mathit{parent}(C, B)]),$ $\mathit{grandparent}(Y, X), I, D, R)$, returning $I = [X = B, Y = A]$, $D = []$, and $R = [\mathit{parent}(A, C), \mathit{parent}(C, B)]$.
There are equalities when there are explicit unifications in the body of the clauses or when there are implicit unifications in the head of the clauses. Disequalities only appear if they are explicitly placed in the body of the clauses. □

- **Normalization of the conjunction**. Let us classify the variables in the formula. As we have already defined, the set of variables of the goal, G, is called $\mathit{GoalVars}$. The set of variables of a goal G, $\mathit{GoalVars}$, plus the set of variables that appear in the equalities I is called $\mathit{ImpVars}$. The set of free variables of $\overline{R}$ is called $\mathit{RelVars}$. We consider that $\mathit{ExpVars}$ is the set of variables of $\overline{R}$ that are not in $\mathit{ImpVars}$, i.e. $\mathit{RelVars}$, except the variables of $\overline{I}$ in the normalized formula.

  — **Elimination of redundant variables and equalities**. If $I_i \equiv X = Y$, where $Y \notin \mathit{GoalVars}$, 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**. $\mathit{ImpVars}$ is the set of variables of $\mathit{GoalVars}$ and the variables that appear in $\overline{I}$. Disequalities D_i that contain any variable that was neither in $\mathit{ImpVars}$ nor in $\mathit{RelVars}$ are irrelevant and should be eliminated.

Implementation details: Before starting with the two steps of the negation we should obtain the sets of variables that we will need for the algorithm. We have implemented a predicate $\mathit{normalization\_conj}/9$ that in a call $\mathit{normalization\_conj}(I, D, R, \mathit{GoalVars}, \mathit{In}, \mathit{Dn}, \mathit{Rn}, \mathit{ImpVars}, \mathit{ExpVars})$ receives the equalities (I), disequalities (D), rest of subgoals (R) and list of variables of $\mathit{GoalVars}$ and returns the updated equalities (In), disequalities (Dn) and rest of subgoals (Rn removing useless variables), and the sets of variables $\mathit{ImpVars}$ and $\mathit{ExpVars}$.

In the example with *grandparent* described above, the call $normalization\_conj([X = B, Y = A], [], [parent(A, C), parent(C, B)], [Y, X], In, Dn, Rn, ImpVars, ExpVars)$ returns $In = []$, $Dn = []$, $Rn = [parent(Y, C), parent(C, X)]$, $ImpVars = [Y, X]$, and $ExpVars = [C]$. □

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.

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

If we consider for example one of the conjunctions of the frontier of $even(Y)$ that is $C_2 \equiv (Y = s(s(X))) \wedge \emptyset \wedge (even(X))$ we can divide it into:

$$\begin{array}{lllllllll}
\overline{I} & \wedge & \overline{D}_{imp} & \wedge & \overline{R}_{imp} & \wedge & \overline{D}_{exp} & \wedge & \overline{R}_{exp} \\
\exists X((Y = s(s(X))) & \wedge & \emptyset & \wedge & even(X) & \wedge & \emptyset & \wedge & \emptyset)
\end{array}$$

Another example comes from the frontier of $parent(Y, X)$ that is the conjunction $C'_1 \equiv \emptyset \wedge \emptyset \wedge (\exists Z(parent(Y, Z) \wedge parent(Z, X))$ that can be divided into:

$$\begin{array}{lllllllll}
\overline{I} & \wedge & \overline{D}_{imp} & \wedge & \overline{R}_{imp} & \wedge & \overline{D}_{exp} & \wedge & \overline{R}_{exp} \\
\exists Z(\emptyset & \wedge & \emptyset & \wedge & \emptyset & \wedge & \emptyset & \wedge & (parent(Y, Z) \wedge parent(Z, X)))
\end{array}$$

Therefore, the constructive negation of the divided formula is:

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

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). How to obtain each of this answers is described in the second step.
Implementation details: We provide a predicate *divide_formula*/4. A call to the goal $divide\_formula(F, ExpVars, Fimp, Fexp)$ receives a formula F that is a list of subgoals, and a list of variables *ExpVars* and provide in the lists of subgoals *Fimp* all the subgoals from F without any variable from *ExpVars* and in *Fexp* the list

with the rest of subgoals of F (the ones that contain any variable of *ExpVars*). This predicate is called twice: *divide_formula*(D, *ExpVars*, *Dimp*, *Dexp*), to split the set of disequalities and, *divide_formula*(R, *ExpVars*, *Rimp*, *Rexp*), to split the set of the rest of subgoals.

In the example of the negation of *even*(Y), the first conjunction of the frontier has empty sets D and R. In the second conjunction D is also empty, so the call *divide_formula*([], [], *Dimp*, *Dexp*) returns in both sets the empty list. For the second conjunction the call *divide_formula*([*even*(X)], [], *Rimp*, *Rexp*) returns *Rimp* = [*even*(X)] and *Rexp* = []. Note that the set of free variables is empty.

In the example that negates *grandparent*(Y, X), there are only one conjunction and the first call to *divide_formula*([], [C], *Dimp*, *Dexp*) returns the empty list in both variables.

The second call *divide_formula*([*parent*(Y, C), *parent*(C, X)], [C], *Rimp*, *Rexp*) returns *Rimp* = [] and *Rexp* = [*parent*(Y, C), *parent*(C, X)].

These are simple examples, and although in general is less frequent to find non-empty disequalities sets, the regular case is a conjunction with free variables, subgoals with them and subgoals without them (i.e. non empty sets *Rimp*, *Rexp*). $\square$

Second step: **Negation of subformulas**

- **Negation of $\overline{I}$.** The conjunction of equalities is of the form $\overline{I} \equiv I_1 \wedge \ldots \wedge I_{NI} \equiv \exists \overline{Z}_1 X_1 = t_1 \wedge \ldots \wedge \exists \overline{Z}_{NI} X_{NI} = t_{NI}$ 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 therefore are 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. In the example that we have seen that $I \equiv \exists X\ Y = s(s(X))$, its negation is: $\overline{I} \equiv \forall X\ Y \neq s(s(X))$.
  Implementation details: The predicate *negate_I*/3 implements this task. The call to the goal *negate_I*(I, *GoalVars*, *LSol*) receives the list of equalities and list of variables of *Goalvars* and provides in *LSol* the list of solutions that represents the disjunction of the disequalities obtained. There are as many solutions of the negation as elements in the disjunction.
  During the negation of the set of equalities of the first conjunction of the frontier of the goal *even*(Y), the subgoal *negate_I*([$Y = 0$], [Y], *LSol*) is called and returns *LSol* = [$Y = / = 0$]. $\square$

- **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 [10], 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.
Implementation details: The predicate *negate_Dimp*/2 implements the negation of these disequalities. Each call to *negate_Dimp*(*Dimp*, *Sol*) provides as many solutions in *Sol* (as equalities) as disequalities there are in *Dimp*. For example *negate_Dimp*([$X = / = 3$, $Y = / = 5$], *Sol*) returns two answers with the solutions $X = 3$ and $Y = 5$. Indeed, solutions are returned as lists of one only element for implementation reasons because to return the complete solution of the negation, *SolC*, the negation of *Dimp* should be combined with the positive equalities (it is done by a simple *append*(I, *Sol*, *SolC*)). $\square$

- **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 applied over R_i recursively. In the example of C_2 from the frontier of *even*(Y) we will provide the answer: $Y = s(s(X)) \wedge \neg even(X)$.
Implementation details: The predicate *negate_Rimp*/2 provides the negation of the subgoals of *Rimp*. For example, in the negation of the second conjunction of the frontier of *even*(Y), from the call *negate_Rimp*([*even*(X)], *Sol*) the solution *cneg*(*even*(X)) is obtained that later is combined with the positive equalities and disequalities, and the solution [$Y = s(s(X))$, *cneg*(*even*(X))] is provided. Before returning the answer, all its subgoals are combined and in this case the new execution of *cneg* will provide results that will be combined with the equality. The backtracking will give as many answers (going deeper and deeper in the recursive calls) as we want. $\square$
- **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

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

negated goal. Of course it is possible to produce infinitely many answers to a negated goal.
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:

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

In the conjunction C_1' from the frontier of $grandparent(Y, X)$ we obtain the answer: $\forall Z \neg(parent(Y, Z) \wedge parent(Z, X))$.
Implementation details: We have implemented *negate_Dexp_Rexp*/3. A call to $negate\_Dexp\_Rexp(DRexp, ImpVars, ExpVars, Sol)$ receives the set of disequalities and the rest of subgoals with free variables, *DRexp*, the list of variables of the equalities, *ImpVars*, and the list of free variables *ExpVars*. It returns a solution *Sol* that is the negation of the conjunction of subgoals of *DRexp*. In the example of negating $grandfather(Y, X)$, the call to $negate\_Dexp\_Rexp([parent(Y, C), parent(C, X)], [Y, X], [C], Sol)$ returns the single list, $Sol = [cneg\_aux((parent(Y, C), parent2(C, X)), [Y, X], [C])]$. The predicate *cneg_aux*/3 is equivalent to the constructive negation, *cneg*/1, but with two additional arguments, $cneg\_aux(Goal, GoalVars, UnivVars)$, that are the list of existential variables of the goal that is going to be negated and the list of the universal variables (that comes from the negation of the free variables). This detail of taking into account the universal variables in the recursive calls to the negation is one of the keys to achieve a real constructive implementation. It is also necessary to join to this solution the equalities and disequalities without free variables before providing an answer, with a call $append(I\_Dimp\_Rimp, Sol, SolC)$. $\square$

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 including negation). The loading of the *cneg*.pl package means that the compiler works with an expanded code added to the previous code. For each clause, $H : -B_1, \cdots, B_n$, of the input program, a new fact $\mathit{stored\_clause}(H, [B_1, \cdots, B_n])$ is added.

Consider a module *even* where the predicate *even*/1 and its negation are defined:

```
:- module(even,[even/1,not_even/1],[cneg]).

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

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

At compilation time this input program is expanded with these two new facts of the predicate *stored_clause*/2:

```
stored_clause(even(0),[]).
stored_clause(even(s(s(X))),[even(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 $\{(Y = 0) \vee (Y = s(s(X)) \wedge \mathit{even}(X))\}$

Note that a similar, but less efficient, behavior 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" behavior. 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 conjunction 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 (Holzbaur 1992; Carlsson 1987) are used which associate a data structure to the variable. Most of the Prolog extensions related to constraints are implemented using attributed variables. An attributed variable is a variable that apart from its level of instantiation (ground, partially instantiated, free) has an additional element that is an attributed and that accompany it during the whole evaluation. The attribute is a formula where additional information about the variable can be stored. In general, this information used to be a simple or complex constraint with a particular form. The formula that is stored here at the attribute of any variable is a normal form constraint:

$$\underbrace{(\bigwedge_j \forall \overline{Z}_j^1 (Y_j^1 \neq s_j^1) \vee \ldots \vee \bigwedge_l \forall \overline{Z_l}^n (Y_l^n \neq s_l^n))}_{\text{negative information}}$$

Additionally, 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)))`. We refer the interested reader to check the details in (Muñoz and Moreno-Navarro 2000).

Let us show some examples involving variable X where we show the corresponding attribute that represents the constraint of each subgoal[11]:

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)`	$X = / = 5;$	$X \neq 5 \vee$
	$(X = / = 6, X = / = Y)$	$(X \neq 6 \wedge X \neq Y)$
`forall([Y], X =/= s(Y))`	$X = / = s(fA(Y))$	$\forall Y \cdot X \neq s(Y)$

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 behavior 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):

[11] The predicate forall/2 implements the universal quantification. I.e. $forall(L, E)$ quantify universally the set of variables L in the expression E

```
?- 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 backtracking 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 frontier 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 by

```
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:

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

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 only one 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:

$Frontier(p(s(s(0)), s(0))) \equiv$
Step 1 $X = s(s(0)) \wedge Y = s(0) \wedge greater(X, Y) \wedge q(X, Y, Z) \wedge r(Z) \equiv$
Step 2 $greater(s(s(0)), s(0)) \wedge q(s(s(0)), s(0), Z) \wedge r(Z) \equiv$
Step 3 $true \wedge q(s(s(0)), s(0), Z) \wedge r(Z) \equiv$
Step 4 $q(s(s(0)), s(0), Z) \wedge r(Z)$

Constraint simplification. During the whole process for negating a goal,the frontier variables (variables of the frontier of the goal that is negated) 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:

$F \equiv \bigvee_i \bigwedge_j \forall \overline{Z}^i_j (Y^i_j \neq s^i_j)$

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:

$F \equiv \forall \overline{x} \bigvee_i \bigwedge_j (Y^i_j \neq s^i_j)$

and using the distributive property (notice that subindexes are different):

$F \equiv \forall \overline{x} \bigwedge_k \bigvee_l (Y^k_l \neq s^k_l)$

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

$F \equiv \bigwedge_k \forall \overline{x} \bigvee_l (Y^k_l \neq s^k_l) \equiv F_1 \wedge \cdots \wedge F_n$

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:

$F_k \equiv \forall \overline{x} \bigvee_l (Y^k_l \neq s^k_l) \equiv \forall \overline{x} \bigvee_l (Y^k_{Skl} \neq s^k_{Skl})$

Then it can be transformed into:

$F_k \equiv \neg \exists \overline{x} \neg (\bigvee_l (Y^k_{Skl} \neq s^k_{Skl})) \equiv \neg Fe_K$

The meaning of F_k is the negation of the meaning of Fe_k;

$Fe_k \equiv \exists \overline{x} \neg (\bigvee_l (Y^k_{Skl} \neq s^k_{Skl}))$

Solving the negations, the result is obtained through simple unifications of the variables of $\overline{x}$:

$Fe_k \equiv \exists \overline{x} \bigwedge \neg (Y^k_{Skl} \neq s^k_{Skl}) \equiv \exists \overline{x} \bigwedge (Y^k_{Skl} = s^k_{Skl})$

Let's follow a couple of example to understand better the simplification. A formula $F = F_1$ (i.e. $n = 1$, so $F \equiv \forall \overline{x} \bigwedge_k \bigvee_l (Y^k_l \neq s^k_l) \equiv F_1 \equiv \forall \overline{x} \bigvee_l (Y^1_l \neq s^1_l)$) in a program that handles the constant 0 and the constructors $p/1$ and $q/2$:

$F_1 \equiv \forall X_1, X_2((Y \neq 0) \vee (Z \neq q(0, X_2)) \vee (W \neq p(X_1)))$
$F_1 \equiv \forall X_1, X_2((Sk_1 \neq 0) \vee (Sk_2 \neq q(0, X_2)) \vee (Sk_3 \neq p(X_1)))$
$F_1 \equiv \neg \exists X_1, X_2 \neg ((Sk_1 \neq 0) \vee (Sk_2 \neq q(0, X_2)) \vee (Sk_3 \neq p(X_1))) \equiv \neg Fe_1$
$Fe_1 \equiv \exists X_1, X_2((Sk_1 = 0) \vee (Sk_2 = q(0, X_2)) \vee (Sk_3 = p(X_1))) \equiv false$
$F \equiv F_1 \equiv \neg Fe_1 \equiv true$

Another formula $F = F_1$ in a program that handles the constant 0 and the constructors $s/1$:

$F_1 \equiv \forall X(X \neq s(Y))$

$F_1 \equiv \forall X(X \neq s(Sk_1))$
$F_1 \equiv \neg\exists X \neg(X \neq s(Sk_1)) \equiv \neg Fe_1$
$Fe_1 \equiv \exists X(X = s(Sk_1)) \equiv true$
$F \equiv F_1 \equiv \neg Fe_1 \equiv false$

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 behavior 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[12] 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

[12] 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(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(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 obviously 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 for the positive goal that is going to be negated. So, our analysis is conservative.

We use a combination of two analyzes to determine if a goal G can be negated with *cnegf*/2: the non-failure analysis (if G does not fail) (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 case in which the non-failure analysis succeeds (the goal does not fail). So the finite upper cost analysis is enough in practice. We test these analyzes 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 maximum number N of solutions of a goal G (we can use the library predicate *findnsols*/4 that obtains a maximum of N solutions of a goal) 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 (if we have obtained exactly N solutions) we do not know if there are infinite solutions or if the number of solutions is greater than N, so in the first case we can apply this technique but in the second case it is not possible.

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) $G \equiv S_1 \vee \cdots \vee S_n \equiv (S_1^1 \wedge \cdots \wedge S_1^{N1}) \vee \cdots \vee (S_n^1 \wedge \cdots \wedge S_n^{Nn})$

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*

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:

```
?- 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. We also wanted to provide a set of examples using negation (see the appendix).

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. This is the reason why we cannot relate it with previous works. 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)). Another possible future work is to incorporate our algorithm at the WAM machine level.

In any case, we will probably not be able to provide an admissible efficient 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 distribute it in the next version of Ciao Prolog [13].

References

Alferes, J. J., Damásio, 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.

Apt, K. R. 1990. Logic programming. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. 3. Elsevier, New York, 493–574.

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.

Bol, R. and Degerstedt, L. 1993. Tabulated resolution for well founded semantics. In *Proc. International Logic Programming Symposium*. MIT Press, Cambridge, Massachuset.

Börger, E. 1987. Unsolvable decision problems for prolog programs. *LNCS 270*, 3–48.

Bossu, G. and Siegel, P. 1985. Saturation, nonmonotonic reasoning and the closed world assumption. *Artificial Intellicence 25,* 1, 13–63.

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

Carlsson, M. 1987. Freeze, indexing, and other implementation issues in the wam. In *ICLP*. The MIT Press, 40–58.

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.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of ACM 43,* 1, 20–74.

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

Damásio, C. 1996. Paraconsistent extended logic programming with constraints. Ph.D. thesis, Dept. de Informática, Universidade Nova de Lisboa.

Dix, J., Pereira, L. M., and Przymusinski, T. C. 1997. Prolegomena to logic programming and non-monotinic reasoning. In *Non-Monotonic Extensions of Logic Programming, Selected papers from NMELP'96*, J. Dix, L. Pereira, and T. C. Przymusinski, Eds. Lecture Notes in Artificial Intelligence, vol. 1216. Springer-Verlag, 1–36.

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

Fages, F. 1997. Constructive negation by prunning. *Journal of Logic Programming 32,* 2, 85–118.

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

Foo, N., Rao, A., Taylor, A., and Walker, A. 1988. Deduced relevant types and constructive negation. In *5th. International Conference and Symposium on Logic Programming*. 126–139.

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.

Holzbaur, C. 1992. Metastructures vs. attributed variables in the contex of extensible unification. *Implementation and LP LNC S631*, 260–268.

Khabaza, T. 1984. Negation as failure and parallelism. In *Symposium on Logic Programming*. IEEE, 70–75.

Kunen, K. 1987. Negation in logic programming. *Journal of Logic Programming 4*, 289–308.

Lassez, J. L. and Marriot, K. 1987. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning 3*.

Liu, J. Y., Adams, L., and Chen, W. 1999. Constructive negation under the well-founded semantics. *Journal of Logic Programming 38*, 3, 295–330.

Lloyd, J. W. 1987. *Foundations of Logic Programing*. Springer-Verlag, Berlin, Germany. 2nd edition.

Maher, M. J. 1988. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceeding os the 3rd IEEE Symp. on Logic in Computer Science*. 348–357.

Maluszynski, J. and Näslund, T. 1989. Fail substitutions for negation as failure. In *North American Conference on Logic Programming*, E. L. Lusk and R. A. Overbeek, Eds. 461–476.

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

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 *Logic for Programming, Artificial Intelligence and Reasoning*, R. Nieuwenhuis and A. Voronkov, Eds. Number 2250 in LNAI. LPAR 2001, 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.

Naughton, J. F. and Ramakrishnan, R. 1991. Bottom-up evaluation of logic programs. In *Computational Logic: Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin, Eds. MIT Press, Cambridge, MA, 640–700.

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

Przymusinski, T. C. 1989a. Every logic program has a natural stratification an d an iterated least fixed point model. In *Symposium on Principles of Database-Systems*. ACM SIGACT-SIGMOD-SIGART, 11–21.

Przymusinski, T. C. 1989b. On constructive negation in logic programming. In *North American Conference on Logic Programming*.

Przymusinski, T. C. and Warren, D. S. 1992. Well founded semantics: Theory and implementation. Draft.

Reiter, R. 1978. On close world data bases. *Logic and Databases*, 55–76.

Ross, K. A. 1992. A procedural semantics for well founded negation in logic programs. *Journal of Logic Programming 13*, 1, 1–22.

Sato, T. and Motoyoshi, F. 1991. A complete top-down interpreter for first order programs. In *International Logic Programming Symposium*. 35–53.

Sato, T. and Tamaki, H. 1984. Transformational logic program synthesis. In Proc. of the International Conference on 5th Generation Computer Systems FGCS84. 195–201.

Shepherdson, J. C. 1984. Negation as failure: A comparison of clark's completed data base and reiter's closed world assumption. *Journal of Logic Programming*, 51–79.

Shepherdson, J. C. 1985. Negation as failure ii. *Journal of Logic Programming*, 185–202.

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

Sterling, L. and Shapiro, E. 1987. *The Art of Prolog*. The MIT Press.

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.

Stuckey, P. and Sudarshan, S. 1993. Well-founded ordered search. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*. Vol. 761. LNCS.

van Emden, M. H. and Kowalski, R. A. 1976. The semantics of predicate logic as a programmig language. *Journal of the ACM 23,* 4, 733–742.

van Gelder, A., Ross, K. A., and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM 38,* 3, 620–650.

Wallace, M. 1987. Negation by constraints: A sound and efficient implementation of negation in deductive databases. In *IEEE Symposium on Logic Programming*. 253–263.

Warren, D. S. 1992. *The XOLDT System*. SUNY, Stoney Brook.

Appendix: Negation Examples

This appendix provides a set of examples in logic programming using negation in a constructive way. They are taken from papers and recompilation from other authors. They are all also available at

http://babel.ls.fi.upm.es/˜susana/code/negation/cneg/examples_cneg.pl.

Numbers

Some predicates related to natural numbers (the ones that have been used for the measures) are included:

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

digit(0).
digit(s(0)).
digit(s(s(0))).
digit(s(s(s(0)))).
digit(s(s(s(s(0))))).
digit(s(s(s(s(s(0)))))).
digit(s(s(s(s(s(s(0))))))).
digit(s(s(s(s(s(s(s(0)))))))).
digit(s(s(s(s(s(s(s(s(0))))))))).

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

Some results related to *boole*/1 are:

```
?- boole(X).

X = 0 ? ;

X = 1 ? ;

no
?- cneg(boole(X)).

X=/=1,X=/=0 ? ;

no
```

Another example negating a subgoal with *greater*/2:

```
?- digit(X), cneg(greater(X,s(s(s(0))))).

X = 0 ? ;

X = s(0) ? ;

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

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

no
```

Queens

This is the implementation in Prolog of the classical problem of the N queens. The goal queens(N,Qs) returns in Qs the column where we must place each of N queens in a checkerboard of NxN assuming each of them is in a different row. For example : $queens(s(s(s(s(0)))), [s(s(0)), s(s(s(s(0)))), s(0), s(s(s(0)))])$ means that the 4 queens are placed in positions (1,2),(2,4),(3,1) and (4,3).

```
queens(N, Qs):-
        queens_list(N, Ns),
        queens1(Ns, [], Qs).     % To place, placed, result

queens1([], Qs, Qs).
queens1([X|Unplaced], Placed, Qs):-
        select(Q, [X|Unplaced], NewUnplaced),
        no_attack(Q, Placed),
        queens1(NewUnplaced, [Q|Placed], Qs).

no_attack(Q, Safe):- no_attack1(Safe, Q, s(0)).

no_attack1([], _Queen, _Nb).
no_attack1([Y|Ys], Queen, Nb):-
add(Y,Nb,YNb),
        Queen =/= YNb,
  subst(Y,Nb,NbY),
        Queen =/= NbY,
        add(Nb,s(0),Nb1),
        no_attack1(Ys, Queen, Nb1).

select(X, [X|Ys], Ys).
select(X, [Y|Ys], [Y|Zs]):-
        select(X, Ys, Zs).

add(0,X,X).
add(s(X),Y,s(Z)):-
add(X,Y,Z).

subst(Z,X,Y):-
greater(Z,X),
add(X,Y,Z).
subst(Z,X,neg):-
greater(X,Z).
subst(X,X,0).
```

The set of answers that we get after negating the goal $queens(s(s(0)), L)$ is:

```
?- cneg(queens(s(s(0)),L)).

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

```
L = [s(_A),s(s(0))],
_A=/=0, _A=/=s(0) ? ;

L = [s(0),s(s(0))] ? ;

L = [s(s(0)),s(_A)],
_A=/=0 ? ;

L = [s(s(0)),s(0)] ?

no
```

In this example we see that as existential as universal variables can appear at the solutions. E.g. at the first answer there are two universal variables. The formal interpretation of this answer is $\forall A, B\,(L \neq [s(A), s(s(0))] \wedge L \neq [s(s(0)), s(B)])$. In the second answer there is an existential variable. The formal interpretation of this answer is $\exists A(L = [s(A), s(s(0))] \wedge A \neq 0 \wedge A \neq s(0))$.

Even and Odd

A different program to compute even and odd numbers:

```
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 instantiated answers:

```
?- odd(X).
```

```
X = s(0) ? ;

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

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

...
```

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 cycles. We define that a node is save if it is not connected to the node "null" (see Figure 3):

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

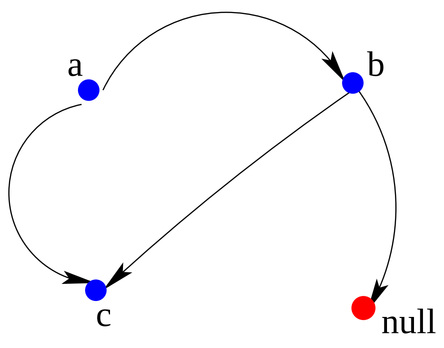

Fig. 3. Graph

```
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=/=null,X=/=a,X=/=b ? ;

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 provide the following answers:

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

X=/=a,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
```