Constructive Negation by Bottom-up Computation of Literal Answers *

Javier Álvez
Dpto de L.S.I. de la U.P.V.
P. Manuel de Lardizabal, 1,
20080-San Sebastian, Spain.
jibalgij@si.ehu.es

Paqui Lucio
Dpto de L.S.I. de la U.P.V.
P. Manuel de Lardizabal, 1,
20080-San Sebastian, Spain.
jiplucap@si.ehu.es

Fernando Orejas
Dpto de L.S.I. de la U.P.C.
Campus Nord, Modul C6,
Jordi Girona 1-3,
08034-Barcelona, Spain.
orejas@lsi.upc.es

ABSTRACT

In this paper, we present a new proposal for an efficient implementation of constructive negation. In our approach the answers for a literal are bottom-up computed by solving equality constraints, instead of by handling frontiers of subsidiary computation trees. The required equality constraints are given by Shepherdson's operators which are, in a sense, similar to bottom-up immediate consequence operators. However, in order to make the procedure efficient two main techniques are applied. First, we restrict our constraints to a class of success-answers (resp. fail-answers) which are easy to manipulate and to solve (or to prove their unsatisfiability). And, second, we take advantage of the monotonic nature of Shepherdson's operators to make the procedure incremental and to avoid recalculations that are typical in frontiers-based methods. Then, goal computation is made in the usual top-down CLP scheme of collecting the answers for the selected literal into the constraint of the goal. The procedural mechanism for constructive negation is designed not only to generate every correct answer of a goal, but also to detect failure. That is, in spite of the bottom-up nature of the calculation of literal answers, goal computation is not necessarily infinite. The operational semantics that makes use of these ideas, called BCN, is sound and complete with respect to three-valued program completion for the whole class of normal logic programs. A prototype implementation of this approach has been developed and the experimental results are very promising.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; F.3.2 [**Semantics of Programming Languages**]: Operational Semantics

General Terms

Languages, Theory, Experimentation

Keywords

constructive negation, operational semantics, bottom-up operators, implementation, equality constraint solving.

1. INTRODUCTION

The idea of constructive negation was introduced by Chan in [6] and extended, by Chan (in [7]) and by Drabent (in [11]), to a complete and sound operational semantics for the whole class of normal logic programs. Fages (in [12]) and Stuckey (in [23]) provided generalizations of constructive negation to the framework of Constraint Logic Programming (CLP for short, see [14] for a survey). From the operational point of view, these approaches are based on subsidiary computation trees and frontiers. That is, when a negative literal $\neg A$ is selected, a subsidiary computation tree for its positive counterpart A is activated. A frontier is a set $\{G_1, \ldots, G_n\}$ of goals containing exactly one goal from each non-failed branch of the computation tree. Since $\neg A$ is equivalent to the frontier negation $\neg(G_1 \vee \cdots \vee G_n)$, the problem is how to transform this formula into a suitable one for proceeding with the original derivation. Chan (in [7]) and Stuckey (in [23]) transform the frontier negation into a disjunction of *complex-goals*, which are goals of the form $\forall \overline{z}(c \vee B)$ where c is a constraint and B is a disjunction of complex-goals and literals. However, Drabent's and Fages's proposals (resp. [11] and [12]) keep the notion of normal goal. They do not negate the whole frontier, but only the constraints (not the literals) involved in the frontier goals, producing the so-called *fail-answers*.

In the *complex-goals approach*, the computation mechanism produces the nesting of both quantification and negation. This is particularly the case in programs including recursion through negated goals. Two side effects of this increasing complexity are, on one hand, that the kind of computations needed to compute new goals becomes also increasingly involved. On the other, the most obvious implementation of this computation process involves the continuous repetition of the same transformation steps. However, avoiding this repetition may be a very difficult task. Moreover the computation process involves two kinds of nondeterminism. On one hand, the selection of the subgoal to be solved is, as usual, nondeterministic. On the other, the

*This work has been partially supported by the Spanish Project TIC 2001-2476-C03.

SAC '04, March 14-17, 2004, Nicosia, Cyprus

frontiers to be used in a given computation, according to [23], can also be chosen nondeterministically.

In the case of the *fail-answers approach*, things work differently. Each subsidiary computation step intends to get either: **(i)** a success, or **(ii)** a frontier such that the negation of the associated constraints is satisfiable. In the former case, the negation of the produced success is used to restrict the goal of the main computation. In the latter case, it obtains an answer for the selected literal of the main computation. This approach, especially the proposal of [12], is more amenable for practical implementation than the complex-goals approach, since one can avoid manipulating these goals. However, it is still a problem how to avoid the repetition of the same computations, especially in the case where the predicate definitions involve recursion through negative literals. Besides, the operational semantics proposed in [11] and [12] are non-deterministic in computational steps which have not an obvious efficient implementation.

In this paper, we propose a new approach for constructive negation. Our proposal is based on the bottom-up computation of the set of answers for a given literal, by applying Shepherdson's operators (cf. [22]). We perform once, at compilation time, the schemes for computing the successive iterations of these operators. Using these schemes, we avoid the above mentioned repetitions of symbolic transformations and satisfiability checks. Exploiting the monotonicity of the operators, the answers for a literal are incrementally obtained from its scheme. Besides, the form of the schemes and their solving method are based on a particular class of equality constraints which can be easier handled than general equality constraints.

The "compile-time" nature of our schemes makes our proposal closer to the so-called *intensional negation* that was introduced in [2]. It was extended, in [5], to the CLP setting. To solve a negative literal, our schemes work similarly (but not identically) to the negative programs which are obtained (in [2, 5]) by transformation. In [2], the presence of universal quantification in goals prevents to achieve a complete goal computation mechanism. This problem is carried out in [5] where a complete operational semantics is provided, although the transformed programs have complex-goals as clause bodies. This compilative complex-goals approach allows a more incremental goal computation process than the previous complex-goal approach. However, the problems of increasingly involved goal computation process (caused by nesting of complex-goals) and repeated calculations still remain.

Our proposal keeps the notion of normal goal. Universal quantification affecting literals is restricted to the schemes and it is exclusively caused by clauses with (at least) one variable in its body that does not occur in its head. The incremental resolution of these schemes allows us to provide a complete goal computation mechanism. The procedural mechanism for computing a CLP normal goal combines the bottom-up calculated answers for each individual literal into the global constraint (of the goal), in order to obtain the goal answers. For generating all correct answers, it suffices to keep iteration-counters for each literal and a fair selection rule. Nevertheless, a goal can fail after the computation of zero o more answers. To support failure we use a computation rule that works as follows (informally):

> *If the selected literal has no new success-answer, then every solution of the goal-constraint could be a fail-answer of the literal. In this case, the goal is a failure leaf. Otherwise, the goal-constraint is restricted to the non fail-answers of the selected literal at the current iteration.*

By means of this rule (technically *(C2)* in Def. 4), our procedural mechanism is able to finish computations (whenever termination is semantically expected) under the proviso of fairness. Our completeness result only assumes the classical notion of fairness in the selection of the literal, which is the unique nondeterminism in computations. It is difficult to talk over computation termination of the above explained proposals. For the fail-answers proposals (cf. [11, 12]) this is due to their non-deterministic formulation. In the case of complex-goals (cf. [7, 23, 5]), the rule $\forall\overline{z}(c \vee B) \longmapsto \forall\overline{z}(c) \vee \forall\overline{z}(c \vee B)$ that is used to extract information from complex-goals, often gives raise to superfluous infinite computation of $\forall\overline{z}(c \vee B)$.

We have implemented a prototype that is available in `http://www.sc.ehu.es/jiwlucap/BCN.html`. The experimental results are very encouraging.

For the lack of space, we omit here some technical details (in particular, proofs). The interested reader is referred to our technical report [1].

Outline of the paper. Section 2 contains preliminary definitions and notation. Section 3 is devoted to the constraint solving method that we use in the bottom-up computation of literal answers. That includes the notion of answer, the basic operations for constraint handling and the incremental solving of schemes. In section 4 we introduce the procedural mechanism for constructive negation, together with the soundness and completeness results. In Section 5 we summarize some conclusions and experimental results.

2. PRELIMINARIES

We deal with the usual syntactic objects of first-order languages. These are function and predicate symbols (in particular, the equality symbol $=$), terms and formulas. Terms are variables, constants and function symbols applied to terms. Formulas are the logical constants $\underline{\mathtt{t}}$ and $\underline{\mathtt{f}}$, predicate symbols applied to terms, and composed formulas with connectives $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ and quantifiers $\forall, \exists$. To avoid confusion, we use the symbol $\equiv$ for the metalanguage equality.

A bar is used to denote tuples, or finite sequences, of objects. For example, the n-tuple of variables $x_1, \ldots, x_n$ is denoted by $\overline{x}$. The concatenation of tuples $\overline{x}$ and $\overline{y}$ is denoted by $\overline{x}{\cdot}\overline{y}$.

If t_1, t_2 are terms, then $t_1 = t_2$ is called an *equation* and $t_1 \neq t_2$ (as abbreviation of $\neg(t_1 = t_2)$) is called a *disequation*. If $\overline{t}$ and $\overline{t}'$ are n-tuples of terms then $\overline{t} = \overline{t}'$ abbreviates $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n$ and $\overline{t} \neq \overline{t}'$ abbreviates $t_1 \neq t'_1 \vee \cdots \vee t_n \neq t'_n$.

A *literal* is an atom $p(\overline{t})$ (called *positive* literal) or its negation $\neg p(\overline{t})$ (called *negative* literal), where p is an n-ary predicate symbol (different from equality) and $\overline{t}$ an n-tuple of terms. By a *flat literal*, we mean that $\overline{t}$ is an n-tuple of variables.

An *equality constraint* is an arbitrary first-order formula such that equality ($=$) is the only predicate symbol occurring in atoms.

Let α be a syntactic object (term, equality constraint, formula, literal, etc), $free(\alpha)$ is the set of all variables occurring free in α. We write $\alpha(\overline{x})$ to denote that $free(\alpha) \subseteq \overline{x}$.

Let φ be a formula, in particular an equality constraint, and $Q \in \{\exists, \forall\}$, then φ^Q denotes the existential/universal quantification of φ in all its free-variables.

A *basic constraint*, denoted by $b(\overline{x}, \overline{w})$, is a conjunction of equations of the form $\overline{x} = t(\overline{w})$, where $\overline{x}$ and $\overline{w}$ are disjoint tuples of pairwise distinct variables. In the sequel $b(\_, \_)$ is used as a metavariable for basic constraints − over an specific pair of variable tuples, when necessary.

A *goal* is an expression of the form $\leftarrow \overline{\ell} \,\square\, c$ where $\overline{\ell}$ is a conjunction of positive and negative flat literals and c is an equality constraint. A *basic goal* is a goal $\leftarrow \overline{\ell} \,\square\, b$ where b is a basic constraint. As usual in CLP, the symbols comma (,) and box ($\square$) are syntactic variants of conjunction, respectively used to separate literals and constraints. Whenever b is $\underline{\mathtt{t}}$ or $\overline{\ell}$ is empty or $\underline{\mathtt{t}}$, they are omitted in goals.

A *normal clause* is an expression $p(\overline{x}) :- \overline{\ell}(\overline{y}) \,\square\, b(\overline{x}\cdot\overline{y}, \overline{w})$ where the flat atom $p(\overline{x})$ is called its *head*, the basic goal $:-\overline{\ell}(\overline{y}) \,\square\, b(\overline{x}\cdot\overline{y}, \overline{w})$ is called its *body*, and the disjoint tuples of variables $\overline{x}, \overline{y}, \overline{w}$ are related in the basic constraint $b(\overline{x}\cdot\overline{y}, \overline{w})$, since it has the form $\overline{x} = \overline{t}(\overline{w}) \wedge \overline{y} = \overline{t}'(\overline{w})$.

Programs are finite sets of normal clauses. Every program P is built from symbols of a *signature* $\Sigma \equiv (FS_\Sigma, PS_\Sigma)$ of function and predicate symbols, respectively, and variables from X. We use the term Σ-*program* whenever the signature is relevant. Given a program P and a predicate symbol p, the set $def_P(p(\overline{x}))$ consists of all the clauses in P with head predicate p. For simplicity, we assume that all clauses with the same head predicate (namely p) use the same head variables (namely $\overline{x}$) and different body variables. It is easy to see that every classical[1] normal logic program can be rewritten as one of our programs.

EXAMPLE 1. *The classical normal* $\{\mathtt{a}\backslash\mathtt{0}, \mathtt{f}\backslash\mathtt{1}\}$*-program:*

$\mathtt{p(f(X))} :- \mathtt{p(X)}, \neg\, \mathtt{q(f(X))}.$
$\mathtt{q(a)} :- \mathtt{q(a)}.$
$\mathtt{q(X)} :- \neg\, \mathtt{r(X)}.$
$\mathtt{r(f(a))}.$

is rewritten as:

$\mathtt{p(X)} :- \mathtt{p(Y_1)}, \neg\, \mathtt{q(Y_2)} \,\square\, \mathtt{X = f(W), Y_1 = W, Y_2 = f(W)}.$
$\mathtt{q(X)} :- \mathtt{q(Y_1)} \,\square\, \mathtt{X = a, Y_1 = a}.$
$\mathtt{q(X)} :- \neg\, \mathtt{r(Y_2)} \,\square\, \mathtt{X = W, Y_2 = W}.$
$\mathtt{r(X)} :- \square\, \mathtt{X = f(a)}.$ ■

To define the semantics of a Σ-program P, Clark [8] proposed to complete the definition of the predicates in P. The *predicate completion formula of a predicate* $p \in PS_\Sigma$ such that

$$def_P(p(\overline{x})) \equiv \{p(\overline{x}) :- \overline{\ell}^{\,i}(\overline{y}^{\,i}) \,\square\, b^i(\overline{x}\cdot\overline{y}^{\,i}, \overline{w}^{\,i}) \mid i \in 1..m\}$$

is the sentence:

$$\forall\overline{x}(p(\overline{x}) \leftrightarrow \bigvee_{i=1}^{m} \exists\overline{y}^{\,i}\cdot\overline{w}^{\,i}(b^i(\overline{x}\cdot\overline{y}^{\,i}, \overline{w}^{\,i}) \wedge \overline{\ell}^{\,i}(\overline{y}^{\,i})))$$

In particular, for $m = 0$ (or $def_P(p(\overline{x})) \equiv \emptyset$) the above disjunction becomes $\underline{\mathtt{f}}$. Hence, the formula is equivalent to $\forall\overline{x}(\neg p(\overline{x}))$. The *Clark's completion of a program* P, namely $Comp(P)$, consists of the set P^* of the predicate completion formulas for every $p \in PS_\Sigma$ together with the *free equality theory* [2] $FET(\Sigma)$. Then, the standard declarative meaning of normal logic programs is $Comp(P)$ interpreted in three-valued logic (cf. [15]).

The theoretical foundations of our proposal comes from a result of Shepherdson (cf. [22]) characterizing Clark-Kunen's completion semantics in terms of satisfaction of equality constraints. In Definition 1 we recall the bottom-up operators that were introduced by Shepherdson ([22]). These operators provide a bottom-up scheme for computing the success- and fail-answers of a given flat literal (in general, a conjuction of them).

DEFINITION 1. *The iterations of the operators* T *and* F *are inductively defined as follows (w.r.t. a* Σ*-program* P*):*

- *For any atom* $p(\overline{x})$ *such that* $p \in PS_\Sigma$ *and*

  $def_P(p(\overline{x})) \equiv \{p(\overline{x}) :- \overline{\ell}^{\,i}(\overline{y}^{\,i}) \,\square\, b^i(\overline{x}\cdot\overline{y}^{\,i}, \overline{w}^{\,i}) \mid i \in 1..m\}$:

  $T_0(p(\overline{x})) \equiv F_0(p(\overline{x})) \equiv \underline{\mathtt{f}}$

  $T_{k+1}(p(\overline{x})) \equiv \bigvee_{i=1}^{m} \exists\overline{y}^{\,i}\cdot\overline{w}^{\,i}(b^i(\overline{x}\cdot\overline{y}^{\,i}, \overline{w}^{\,i}) \wedge T_k(\overline{\ell}^{\,i}(\overline{y}^{\,i})))$

  $F_{k+1}(p(\overline{x})) \equiv \bigwedge_{i=1}^{m} \forall\overline{y}^{\,i}\cdot\overline{w}^{\,i}(\neg b^i(\overline{x}\cdot\overline{y}^{\,i}, \overline{w}^{\,i}) \vee F_k(\overline{\ell}^{\,i}(\overline{y}^{\,i})))$

- *For any* $k \in \mathbb{N}$*:*

  $T_k(\overline{\ell}^{\,1} \wedge \overline{\ell}^{\,2}) \equiv T_k(\overline{\ell}^{\,1}) \wedge T_k(\overline{\ell}^{\,2})$ $\quad T_k(\neg p(\overline{x})) \equiv F_k(p(\overline{x}))$

  $F_k(\overline{\ell}^{\,1} \wedge \overline{\ell}^{\,2}) \equiv F_k(\overline{\ell}^{\,1}) \vee F_k(\overline{\ell}^{\,2})$ $\quad F_k(\neg p(\overline{x})) \equiv T_k(p(\overline{x}))$

  $T_k(\underline{\mathtt{t}}) \equiv \underline{\mathtt{t}}$ $\qquad\qquad\qquad\qquad$ $F_k(\underline{\mathtt{t}}) \equiv \underline{\mathtt{f}}$ ■

A key result for our work is the following Theorem 1, which is a simple consequence of Theorem 6 and Lemma 4.1 in [22].

THEOREM 1. *Let* P *be a* Σ*-program,* $\overline{\ell}$ *a conjunction of literals and* c, d *constraints, then the following two facts hold:*

(i) $Comp(P) \models_3 (c \rightarrow (\overline{\ell} \wedge d))^\forall$ *if and only if* $FET(\Sigma) \models (c \rightarrow (T_k(\overline{\ell}) \wedge d))^\forall$ *for some* $k \in \mathbb{N}$

(i) $Comp(P) \models_3 (c \rightarrow (\neg\overline{\ell} \vee d))^\forall$ *if and only if* $FET(\Sigma) \models (c \rightarrow (F_k(\overline{\ell}) \vee d))^\forall$ *for some* $k \in \mathbb{N}$ ■

Roughly speaking, we call a k-success (resp. k-failure) of a literal to any answer (resp. failure-answer) belonging to the k-iteration of some immediate consequence operator. Different immediate consequence operators, providing bottom-up semantics for normal logic programs, have been proposed (cf. [4, 13, 15, 16, 23]). Intuitively, the equality constraint $T_k(\ell)$ represents the k-success of ℓ, whereas $F_k(\ell)$ gives the k-failures of ℓ. It is very easy to prove (by induction) that the operators T and F are monotonic in the following sense:

PROPOSITION 1. **(Monotonicity)** *Let* P *be* Σ*-program and* $\ell(\overline{x})$ *a flat literal, then for all* $n \in \mathbb{N}$*:*

[1] We say "classical" to distinguish Logic Programming (LP) concepts (goal, program, etc) from CLP concepts

[2] also known as *Clark's equational theory* (cf.[8]) or the *first-order theory of finite trees.*

(i) $FET(\Sigma) \models (T_n(\ell(\overline{x})) \rightarrow T_{n+1}(\ell(\overline{x})))^{\forall}$

(ii) $FET(\Sigma) \models (F_n(\ell(\overline{x})) \rightarrow F_{n+1}(\ell(\overline{x})))^{\forall}$. ■

3. BOTTOM-UP COMPUTATION OF LITERAL ANSWERS

The crucial aspect for practical implementation is how to compute literal answers, using Shepherdson's operators, in an efficient incremental manner. The CLP goal-derivation process has to combine the answers for a selected literal with the answers for the remaining literals of the goal. Hence, the choice of a notion of answer affects the class of equality constraints to be handled along the computations. The decidability of FET(Σ) has been proved by different methods (cf. [10, 17], for instance). It is known that the decidability of FET(Σ) is a non-elementary problem (cf. [24]). However, our proposal deals with a particular class of equality constraints, called *answers*, that could be more efficiently solved than general equality constraints. As a consequence, our constraint solving method is different from general decision methods (cf. [10]) which usually combine quantifier elimination with a set of transformational rules. Instead, we only need procedures for combining answers (by conjunction, negation or instantiation) and to check answer satisfiability. In addition, answers should be user friendly, in order to be displayed as goal answers. In this section, we introduce the notion of answer and the basic operations for handling constraints along computations. Finally, we introduce the schemes for T and F and show how to solve them efficiently.

3.1 Constraints Handling

Our notion of answer is based on the following class of equations and disequations.

DEFINITION 2. *An (dis)equation is called* collapsing *whenever (at least) one of its terms is a variable.* ■

DEFINITION 3. *An* answer for the variables $\overline{x}$ *is either a constant (*$\underline{\mathtt{t}}$*,* $\underline{\mathtt{f}}$*) or a formula* $\exists\overline{w}(a(\overline{x},\overline{w}))$ *where* $a(\overline{x},\overline{w})$ *is a conjunction of both*

- *collapsing equations of the form* $x_i = t(\overline{w})$*, and*
- *universally quantified collapsing disequations of the form* $\forall\overline{v}(w_j \neq s(\overline{w},\overline{v}))$*, where the term* s *is not a single variable in* $\overline{v}$ *and* w_j *does not occur in* s*.*

where each x_i *occurs at most once.* ■

An example of answer for x_1, x_2, x_3 is:

$$\exists w_1 \exists w_2 (\ x_1 = w_1 \ \wedge\ x_2 = w_2 \ \wedge\ x_3 = g(w_1) \ \wedge\ w_1 \neq a \ \wedge\ w_1 \neq w_2 \ \wedge\ \forall v(w_1 \neq f(v, w_2))\ )$$

which is represented, in Prolog-like notation, by

$$\mathtt{x_1 = \_A,\ x_2 = \_B,\ x_3 = g(\_A),\ \_A \neq a, \_A \neq \_B,\ \_A \neq f(*C, \_B)}$$

where traditional Prolog-variables of the form $\_\langle char\rangle$ represent existential variables, whereas new variables of the form $*\langle char\rangle$ are associated to universal variables. It is obvious that every answer can be represented in this Prolog-like notation. Notice that, for any answer, the scope of each universal variable is one single disequation and there is no restriction about repetition of existential, neither universal, variables.

Answers are *solved forms* in the sense that their satisfiability is easily decidable. In the case of infinite signatures, an answer (different from $\underline{\mathtt{f}}$) is always satisfiable. In fact, a similar (but less user friendly) kind of solved form is used in [9], where only infinite signatures are considered. However, for finite signatures, an answer can be unsatisfiable. For example the following answer

$$\exists w(x = w \wedge w \neq a \wedge w \neq g(a) \wedge \forall v_1(w \neq g(g(v_1))))$$

is unsatisfiable for the signature $\{a/0, g/1\}$.

PROPOSITION 2. *Answer satisfiability can be checked without transforming the input answer. Moreover, with respect to an infinite signature, any answer (different from* $\underline{\mathtt{f}}$*) is satisfiable.* ■

There are other three operations which are basic for solving the constraints generated by Shepherdson operators:

PROPOSITION 3.

(i) A conjunction of answers for $\overline{x}$ *can be transformed into an equivalent disjunction of answers for* $\overline{x}$*.*

(ii) The negation of an answer for $\overline{x}$ *can be transformed into an equivalent disjunction of answers for* $\overline{x}$*.*

(iii) The instantiation of the variables $\overline{x}$ *by terms* $\overline{t}'(\overline{z})$ *in an answer* a *for* $\overline{x}$ *(denoted* $a[\overline{t}'(\overline{z})/\overline{x}]$*) can be transformed into an equivalent disjunction of answers for* $\overline{z}$*.* ■

3.2 Operator Schemes: Incremental Solving

Now, we show how literal answers can be computed in an efficient, incremental and lazy way. There are three aspects that are crucial for efficiency purposes. First, both operators $\mathcal{O} \in \{T, F\}$ are monotonic (see Prop. 1). If we denote by $\mathcal{O}_{=n}(p(\overline{x}))$ the answers that are obtained exactly in the step n, then

$$\mathcal{O}_{k+1}(p(\overline{x})) \equiv \mathcal{O}_k(p(\overline{x})) \vee \mathcal{O}_{=k+1}(p(\overline{x}))$$

Hence, at the $k+1$-iteration step, we calculate $\mathcal{O}_{=k+1}(p(\overline{x}))$. The previously obtained answers (given by $\mathcal{O}_k(p(\overline{x}))$) have being loaded, as part of the predicate description of p, and we do not recalculate them. Second, in order to avoid some symbolic transformations and satisfiability checks that are common to all iteration-steps, we obtain and load (at compilation time) the operators schemes where such operations are already performed. In particular, the first iteration $\mathcal{O}_1(p(\overline{x}))$ is once calculated at compilation time. Third, these schemes are in disjunctive form (see Lemma 1) to allow the partial solving of each disjunct. In fact, each disjunct is solved until one satisfiable answer (for a literal) is obtained. Then, in the Prolog-style, the calculated answer can be displayed to the user or passed to the procedural mechanism that is computing a goal (collection of literals). The unsolved part of this scheme (also in disjunctive form) is left to be treated, in the same lazy way, when more answers would be demanded (by the user or by the goal computation process).

LEMMA 1. *Let* P *be a* Σ*-program and* $p \in PS_\Sigma$*. The iterations of* $\mathcal{O} \in \{T, F\}$ *(with respect to* P*) can be computed*

by schemes of the form:

$$\mathcal{O}_1(p(\overline{x})) \equiv \bigvee \exists\overline{w}(a(\overline{x},\overline{w}))$$

$$\mathcal{O}_{k+1}(p(\overline{x})) \equiv \bigvee \exists\overline{w}\Big(a(\overline{x},\overline{w}) \wedge \bigwedge_{j=1}^{n} \varphi_j^{[p,k]}(\overline{w})\Big) \quad (1)$$

where each $\varphi_j^{[p,k]}(\overline{w})$ *has one of the following two forms:*

(i) $\mathcal{O}_k(\ell(\overline{y}))[\overline{t}(\overline{w})/\overline{y}]$

(ii) $\forall\overline{v}\Big(F_k(\overline{\ell}(\overline{y}))[\overline{t}(\overline{w},\overline{v})/\overline{y}]\Big)$ *where* $\overline{v}$ *is non-empty.* ■

Notice that $\mathcal{O}_k(\ell(\overline{y}))$ (in part. $F_k(\ell(\overline{y}))$) gives disjunctions of answers for $\overline{y}$, and the instantiation $[\overline{t}(\overline{w})/\overline{y}]$ (resp. $[\overline{t}(\overline{w},\overline{v})/\overline{y}]$) transforms them into disjunctions of answers for $\overline{w}$ (resp. $\overline{w}\cdot\overline{v}$). We would like to remark that universal quantification with literals in its scope (option *(ii)* in Lemma 1) exclusively appears in the F-scheme of atoms that are defined by (at least) one classical normal clause with a fresh variable in its body. In the following example, this is the case of `even`, but it is not the case of `plus`.

EXAMPLE 2. *For the* $\{\mathtt{0}\backslash\mathtt{0}, \mathtt{s}\backslash\mathtt{1}\}$*-program:*

$\mathtt{plus}(\mathtt{0},\mathtt{X},\mathtt{X}).$

$\mathtt{plus}(\mathtt{s}(\mathtt{X}_1),\mathtt{X}_2,\mathtt{s}(\mathtt{X}_3)) :- \mathtt{plus}(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3).$

$\mathtt{even}(\mathtt{X}) : -\mathtt{plus}(\mathtt{Y},\mathtt{Y},\mathtt{X}).$

the F-schemes are :

$F_1(\mathtt{plus}(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3)) \equiv$
$\exists\overline{\mathtt{W}}(\mathtt{X}_1 = \mathtt{s}(\mathtt{W}_1) \wedge \mathtt{X}_2 = \mathtt{W}_2 \wedge \mathtt{X}_3 = \mathtt{W}_3 \wedge \forall\mathtt{V}(\mathtt{W}_3 \neq \mathtt{s}(\mathtt{V}))) \quad \vee$
$\exists\overline{\mathtt{W}}(\mathtt{X}_1 = \mathtt{0} \wedge \mathtt{X}_2 = \mathtt{W}_1 \wedge \mathtt{X}_3 = \mathtt{W}_2 \wedge \mathtt{W}_1 \neq \mathtt{W}_2)$

$F_{k+1}(\mathtt{plus}(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3)) \equiv$
$\exists\overline{\mathtt{W}}(\mathtt{X}_1 = \mathtt{s}(\mathtt{W}_1) \wedge \mathtt{X}_2 = \mathtt{W}_2 \wedge \mathtt{X}_3 = \mathtt{s}(\mathtt{W}_3) \wedge F_k(\mathtt{plus}(\overline{\mathtt{Y}}))[\overline{\mathtt{W}}/\overline{\mathtt{Y}}])$

$F_1(\mathtt{even}(\mathtt{X})) \equiv \underline{\mathtt{f}}$

$F_{k+1}(\mathtt{even}(\mathtt{X})) \equiv \exists\mathtt{W}(\mathtt{X} = \mathtt{W} \wedge \forall\mathtt{V}(F_k(\mathtt{plus}(\overline{\mathtt{Y}}))[\mathtt{V},\mathtt{V},\mathtt{W}/\overline{\mathtt{Y}}]))$

Hence, to compute the answers of any literal of the form $\neg\mathtt{even}(\mathtt{t})$ *universal quantification must be handled, but it is not required for literals of the form* $\neg\mathtt{plus}(\mathtt{t}_1,\mathtt{t}_2,\mathtt{t}_3)$. ■

EXAMPLE 3. *For the program of Example 1, we obtain the following F-schemes:*

$F_1(\mathtt{p}(\mathtt{X})) \equiv \quad \exists\mathtt{W}(\mathtt{X} = \mathtt{W} \wedge \forall\mathtt{V}(\mathtt{W} \neq \mathtt{f}(\mathtt{V})))$

$F_{k+1}(\mathtt{p}(\mathtt{X})) \equiv \quad \exists\mathtt{W}(\mathtt{X} = \mathtt{f}(\mathtt{W}) \wedge F_k(\mathtt{p}(\mathtt{Y}_1))[\mathtt{W}/\mathtt{Y}_1]) \vee$
$\exists\mathtt{W}(\mathtt{X} = \mathtt{f}(\mathtt{W}) \wedge T_k(\mathtt{q}(\mathtt{Y}_2))[\mathtt{f}(\mathtt{W})/\mathtt{Y}_2])$

$F_1(\mathtt{q}(\mathtt{X})) \equiv \quad \underline{\mathtt{f}}$

$F_{k+1}(\mathtt{q}(\mathtt{X})) \equiv \quad (\mathtt{X} = \mathtt{a} \wedge F_k(\mathtt{q}(\mathtt{Y}_1))[\mathtt{a}/\mathtt{Y}_1] \wedge T_k(\mathtt{r}(\mathtt{Y}_2))[\mathtt{a}/\mathtt{Y}_2])$
$\vee \exists\mathtt{W}(\mathtt{X} = \mathtt{W} \wedge \mathtt{W} \neq \mathtt{a} \wedge T_k(\mathtt{r}(\mathtt{Y}_2))[\mathtt{W}/\mathtt{Y}_2])$ ■

We can show, by induction on k, how to compute only $\mathcal{O}_{=k+1}(p(\overline{x}))$ avoiding to recalculate $\mathcal{O}_k(p(\overline{x}))$. For $k = 0$, $\mathcal{O}_{=1}(p(\overline{x})) \equiv \mathcal{O}_1(p(\overline{x}))$ since $\mathcal{O}_0(p(\overline{x})) \equiv \underline{\mathtt{f}}$. Assuming the induction hypothesis (for k), it is easy to prove the following fact:

Fact 1. The formulas $\varphi_j^{[p,k]}(\overline{w})$ of (1) can be split into $\varphi_j^{[p,k-1]}(\overline{w}) \vee \varphi_j^{[p,=k]}(\overline{w})$ ■

Then, to compute $\mathcal{O}_{=k+1}(p(\overline{x}))$, the first member $\varphi_j^{[p,k-1]}(\overline{w})$ of each $\varphi_j^{[p,k]}(\overline{w})$ has been calculated yet, as a subformula of $\mathcal{O}_k(p(\overline{x}))$. Hence, each internal conjunction of Lemma 1(1) can be written as:

$$\bigwedge_{j=1}^{n} (\varphi_j^{[p,k-1]}(\overline{w}) \vee \varphi_j^{[p,=k]}(\overline{w}))$$

By distribution, each one gives a disjunction of formulas of the form:

$$\varphi_{i_1}^{[p,e_1]}(\overline{w}) \wedge \varphi_{i_2}^{[p,e_2]}(\overline{w}) \wedge \cdots \wedge \varphi_{i_n}^{[p,e_n]}(\overline{w})$$

The collection of disjuncts such that $e_j \equiv k-1$ for all $j \in 1..n$ generates $\mathcal{O}_k(p(\overline{x}))$. Hence, to calculate $\mathcal{O}_{=k+1}(p(\overline{x}))$ we discard all these disjuncts. The remaining ones produce answers for $\overline{w}$ which, by substitution in the corresponding $a(\overline{x},\overline{w})$ (of (1)), give the new answers for $\overline{x}$.

4. THE PROCEDURAL MECHANISM

Now, we present how the bottom-up computation of literal answers can be managed by a top-down goal computation process that successively collects the literals' answers into the constraint of the current goal. In spite of the bottom-up nature of the answer calculation, the procedural mechanism is in charge of detecting when a goal should fail. In this section, we define the procedural mechanism, called BCN operational semantics. Our formulation provides a uniform treatment for positive and negative literals. As we will explain later (in Remark 1), there is no problem to use the new mechanism only when the selected literal is negative, whereas the positive ones are left to SLD-resolution. Indeed, a preliminary work in this direction was presented in [21]. In example 4, we show how the BCN operational semantics works to compute goal answers and also to detect failure. Finally, we provide the soundness and completeness results.

4.1 The BCN Operational Semantics

The notion of computation tree is relative to a program and a selection rule that chooses a literal in the current goal.

In order to define the computation tree, we associate to each literal ℓ two counters: $k_T(\ell)$ and $k_F(\ell)$. They respectively mean the iteration of the operator (resp. T or F) that has to be computed in the next selection of the literal ℓ. The nodes of a computation tree are pairs $(G, K(G))$, where K is a function that associates values to both counters of each literal in G. For initialization, the constant function `cons1` associates the value 1 to both counters of every literal.

The expression $\underline{\mathsf{SolvedForm}}(c(\overline{x}))$ denotes the solved form of the equality constraint $c(\overline{x})$, that is a disjunction of (satisfiable) answers $\bigvee_{i=1}^{m} \exists\overline{w}^i(a_i(\overline{x},\overline{w}^i))$. For $m = 1$ and $a_1 \equiv \underline{\mathtt{t}}$ the disjunction is $\underline{\mathtt{t}}$, and for $m = 0$ it is $\underline{\mathtt{f}}$.

DEFINITION 4. *A* BCN-computation tree *for a* Σ*-goal* G*, with respect to a* Σ*-program* P *and a selection rule* R*, is a tree which root is* $(G, \mathtt{cons1}(G))$ *and for each node with goal*

$$G' \equiv \leftarrow\overline{\ell}^1, \underline{\ell(\overline{x})}, \overline{\ell}^2 \,\square\, a(\overline{x},\overline{w})$$

where $\ell(\overline{x})$ *is the selected literal and* $(k_T(\ell(\overline{x})), k_F(\ell(\overline{x})))$ *is associated by* $K(G')$ *to values* (n^+, n^-)*:*

(C1) *If* $\underline{\mathsf{SolvedForm}}(\exists\overline{w}(a(\overline{x},\overline{w}))\wedge T_{n^+}(\ell(\overline{x})))\not\equiv\underline{\mathtt{f}}$, *then it is of the form* $\bigvee_{i=1}^{m}\exists\overline{w}^i a_i(\overline{x},\overline{w}^i)$. *Hence,* G' *has one child for each* $i\in 1..m$, *with goal* $G_i\equiv\leftarrow\overline{\ell}^1,\overline{\ell}^2\square a_i(\overline{x},\overline{w}^i)$. *Each* $K(G_i)$ *is identical to* $K(G')$ *except that* ℓ *has no associated information (it does not appear in* G_i*). Besides, if* $\underline{\mathsf{SolvedForm}}(\exists\overline{w}(a(\overline{x},\overline{w}))\wedge\neg T_{n^+}(\ell(\overline{x})))\equiv\bigvee_{j=1}^{m'}\exists\overline{w}^j a'_j(\overline{x},\overline{w}^i)\not\equiv\underline{\mathtt{f}}$, *then* G' *has also one child for each* $j\in 1..m'$ *with goal* $G'_j\equiv\leftarrow\overline{\ell}^1,\ell,\overline{\ell}^2\square a'_j(\overline{x},\overline{w}^j)$ *and* $K(G'_j)$ *is identical to* $K(G')$ *except that* $k_T(\ell(\overline{x}))$ *is updated to be* $n^+ + 1$.

(C2) *Otherwise − if case (C1) is not applied − there are the following two possible cases:*

(C2a) *If* $\underline{\mathsf{SolvedForm}}(\exists\overline{w}(a(\overline{x},\overline{w}))\wedge\neg F_{n^-}(\ell(\overline{x})))\equiv\underline{\mathtt{f}}$, *then* G' *is a* failure leaf.

(C2b) *If* $\underline{\mathsf{SolvedForm}}(\exists\overline{w}(a(\overline{x},\overline{w}))\wedge\neg F_{n^-}(\ell(\overline{x})))\not\equiv\underline{\mathtt{f}}$, *then it is of the form* $\bigvee_{i=1}^{m}\exists\overline{w}^i a_i(\overline{x},\overline{w}^i)$. *Thus,* G' *has one child for each* $i\in 1..m$, *with goal* $G_i\equiv\leftarrow\overline{\ell}^1,\ell,\overline{\ell}^2\square a_i(\overline{x},\overline{w}^i)$. *Each* $K(G_i)$ *results by respectively updating in* $K(G')$ *the counters* $k_T(\ell(\overline{x}))$ *and* $k_F(\ell(\overline{x}))$ *to* $n^+ + 1$ *and* $n^- + 1$. ■

In other words, when a literal ℓ is selected in a goal $\leftarrow\overline{\ell}\square a$, we try to get the success-answers of ℓ by applying the rule *(C1)*. When it applies, the goal children are $\leftarrow\overline{\ell}\setminus\{\ell\}\square a_i$ $(i\in 1..m)$. where $a_1\vee\cdots\vee a_m$ is the disjunction of answers produced by the solver. Besides, the computation tree could have more branches, keeping the literal ℓ, for computing higher iterations of $T(\ell)$. Notice that these other branches do not exist whether every $\overline{x}$ satisfying $\exists\overline{w}(a(\overline{x},\overline{w}))$ is also a success-answer of $\ell(\overline{x})$ at the n^+ iteration-step. When the rule *(C1)* can not be applied, we try to detect failure with the rule *(C2)*. In the case of the rule *(C2a)* the goal fails, whereas *(C2b)* behaves as an incremental failure detection.

Definition 5. *Any finite branch of a computation tree for* G *which ends by a leaf of the form* $\leftarrow\square a$ *represents a* successful derivation *and the constraint* a *is a* computed answer *for* G. *A* failure tree *is a finite tree such that every leaf is a failure.* ■

Now, we give an example of computation that produces one answer and then fails.

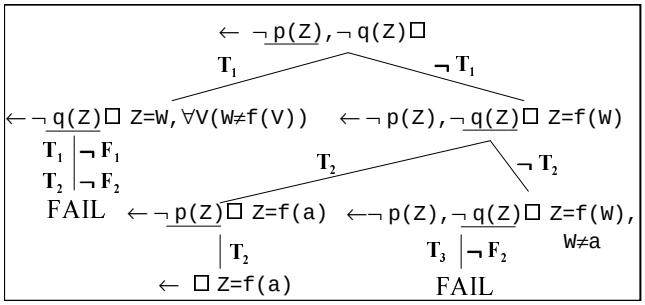

Figure 1: A finite BCN-computation tree

Example 4. *Consider the program of Example 1 and 3. The goal*

$$\leftarrow\neg\,\mathtt{p(Z)},\neg\,\mathtt{q(Z)}$$

produces a unique answer $\mathtt{Z=f(a)}$, *since*

$$\forall\mathtt{Z}(\mathtt{Z=f(a)}\leftrightarrow(\neg\,\mathtt{p(Z)},\neg\,\mathtt{q(Z)}))$$

is a logical consequence of program completion. The Figure 1 shows a finite BCN-computation tree. The operators written in the edges of the tree of Figure 1 are applied to the literal that is just above marked as selected. It is worthwhile to remember that $T_k(\neg\varphi)\equiv F_k(\varphi)$ *and* $F_k(\neg\varphi)\equiv T_k(\varphi)$. *The first* T*-iteration for selected literal yields*

$$T_1(\neg\,\mathtt{p(Z)})\equiv\exists\mathtt{W}(\mathtt{Z=W}\wedge\forall\mathtt{V}(\mathtt{W}\neq\mathtt{f(V)}))$$

and, therefore, $\neg T_1(\neg\,\mathtt{p(Z)})\equiv\exists\mathtt{W}(\mathtt{Z=f(W)})$. *Hence, the computation tree is split into two branches. In the left branch*

$$T_1(\neg\,\mathtt{q(Z)})\equiv\underline{\mathtt{f}}$$

therefore failure detection is intended. Since

$$F_1(\neg\,\mathtt{q(Z)})\equiv\underline{\mathtt{f}}$$

both counters are updated, but the goal does not change (conjunction with $\underline{\mathtt{t}}$*). Next, the conjunction of the goal constraint with*

$$T_2(\neg\,\mathtt{q(Z)})\equiv\mathtt{Z=f(a)}$$

is unsatisfiable. Since $\neg F_2(\neg\,\mathtt{q(Z)})$ *is also* $\mathtt{Z=f(a)}$, *failure is detected. In the right branch, the first iteration of both operators for* $\neg\,\mathtt{q(Z)}$ *increases both counters (as before). In the next two steps*

$$T_2(\neg\,\mathtt{q(Z)})\equiv\mathtt{Z=f(a)}$$

and

$$T_2(\neg\,\mathtt{p(Z)})\equiv\exists\mathtt{W}(\mathtt{Z=f(W)}\wedge\forall\mathtt{V}(\mathtt{W}\neq\mathtt{f(V)}))$$

Thus, the expected answer $\mathtt{Z=f(a)}$ *is generated. There is not additional branch because*

$$\mathtt{Z=f(a)}\wedge\neg\,T_2(\neg\,\mathtt{p(Z)})$$

is unsatisfiable. In the rightmost branch, the third iteration of T *for the selected literal does not produce any new answer. Then, the conjunction of both constraints:*

$$\exists\mathtt{W}(\mathtt{Z=f(W)}\wedge\mathtt{W}\neq\mathtt{a})$$

$$\neg F_2(\neg\,\mathtt{q(Z)})\equiv\mathtt{Z=f(a)}$$

is unsatisfiable. Therefore the goal fails. ■

Remark 1. *The presented procedural mechanism can be also used to implement an extension of SLD-resolution for normal logic programs. That is, we can apply it only when the selected literal is negative, whereas SLD-resolution is applied to positive literals. By completeness of the SLD-resolution, that is equivalent to use the operator* T. *In SLD-resolution, a goal* $\leftarrow\overline{\ell}\square a$ *with selected positive literal* $p(\overline{x})$ *should be a failure leaf if the conjunction of* a *with the constraint of any clause with head* $p(\overline{x})$ *is unsatisfiable. It is easy to see that this is equivalent to* $FET(\Sigma)\models(a\rightarrow F_1(\ell))^{\forall}$. *Hence, a particular case of our failure condition holds.* ■

4.2 Soundness and Completeness

The BCN operational semantics is sound and complete with respect to the three-valued interpretation of program completion for the whole class of normal logic programs. In the following soundness result, computation is relative to some selection rule.

THEOREM 2. *Let be a Σ-program P and a Σ-goal $G \equiv \leftarrow \overline{\ell} \square a$, then*

1. *If G has a failure tree, then $Comp(P) \models_3 (a \rightarrow \neg\overline{\ell})^{\forall}$.*
2. *If there is a successful derivation for G with computed answer a', then $Comp(P) \models_3 (a' \rightarrow \overline{\ell} \wedge a)^{\forall}$.* ■

For completeness the classical notion of *fairness* is needed. A selection rule is *fair* if and only if every literal that appears in an infinite branch of a computation tree is eventually selected.

THEOREM 3. *Let be a Σ-program P and a Σ-goal $G \equiv \leftarrow \overline{\ell} \square a$. Then, for any fair selection rule:*

1. *If $Comp(P) \models_3 (a \rightarrow \neg\overline{\ell})^{\forall}$ then the computation tree for G is a failure tree.*
2. *If there exists a satisfiable constraint c such that $Comp(P) \models_3 (c \rightarrow \overline{\ell} \wedge a)^{\forall}$, then there exist $n > 0$ computed answers $a_1, \ldots, a_n$ for G such that*

$$FET(\Sigma) \models (c \rightarrow \bigvee_i a_i)^{\forall}$$ ■

5. CONCLUSIONS

Constructive negation subsumes the *negation as failure* (NAF) rule and, at the same time, solves the floundering problem of NAF. With regard to the wellknown technique of delaying each negative literal until it would be grounded (then, NAF could be used) we would like to point out two drawbacks. First, it can produce infinite computation when constructive negation finitely fails and, second, it is not necessarily more efficient. The latter was also remarked in [7]. Consider the following program:

```
p(a).
q(f^99(a)).
r(Z):- ¬ s(Z).
p(f(X)):- p(X).
q(Y):- q(f(Y)).
s(g(V)).
```

where $f^{99}(a)$ denotes $\overbrace{f(\ldots(f}^{99}(a))\ldots)$. With the delay technique, the goal

$$\leftarrow \mathtt{p(X)}, \neg\, \mathtt{r(X)}.$$

causes an infinite computation that successively obtains a ground term from the first subgoal and a failure from the second one. Nevertheless, our procedural mechanism finitely fails because the second iteration for the second literal gives the constraint $\exists \mathtt{V}\ (\mathtt{X} = \mathtt{g(V)})$. Besides, the following goal:

$$\leftarrow \mathtt{q(X)}, \neg\, \mathtt{r(X)}.$$

fails with the delay technique, but constructive negation works more efficiently. In fact, if we select the literal $\neg\, \mathtt{r(X)}$, then, it is restricted by a strong constraint:

$$\leftarrow \mathtt{q(X)} \square \exists \mathtt{V}\ (\mathtt{X} = \mathtt{g(V)})$$

that immediately produces the failure. By delaying the second subgoal, failure requires the construction of 100 failure-trees.

In this paper we have provided the basic ideas for designing a sound, complete and efficient implementation of constructive negation. Actually, we have implemented (in Sictus Prolog v.3.8.5) a prototype which is available in `http://www.sc.ehu.es/jiwlucap/BCN.html`. The obtained results seem very promising.

Goal	100 a.	500 a.	1500 a.
$\leftarrow \neg$ disjoint(L1,L2)	16	130	990
$\leftarrow \neg$ disjoint(L,[0]), maxlist(L,$s^5(0)$)	125	672	2677
$\leftarrow \neg$ maxlist(L,s(_))	3624	3652	3659
$\leftarrow \neg$ maxlist(L,Z)	3661	3687	3718

Figure 2: Some experimental results

In Figure 2 you can find a table describing some representative experiments conducted with the prototype on a Pentium IV at 1.7 GHz. We have taken measurements with the function `statistic/2` of Sicstus Prolog. The cells show the milliseconds of CPU-time that is taken to produce the first `100`, `500` and `1500` answers for the left-hand specified goal. The considered program is:

```
memb(X,[X|_]).
memb(X,[_|L]):- memb(X,L).

disjoint([],_).
disjoint([E|L1],L2):- ¬ memb(E,L2),
                       disjoint(L1,L2).

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

maxlist([],0).
maxlist([E|L],NE):- maxlist(L,EAux)
                    gt(E,EAux,NE).
```

Notice the fresh variable `EAux` in the body of the last clause, it causes unavoidable universal quantification of literals in the F-scheme for `maxlist`.

We are aware that it is difficult to asses the value of these experiments in terms of the absolute time spent by the prototype to produce (some) answers. Instead, a comparative study with respect to other implementations would have been more adequate. The problem is that the existing experience in implementing negation (beyond negation as failure) in logic programming is, to our knowledge, very limited. In particular, Chan ([6]) and Barták ([3]) have implemented constructive negation for the special case of finite computation trees. This restriction is quite strong and causes that the examples used to test the implementation are computationally very simple. As a consequence, the results obtained by our implementation and by Barták are quite similar (it was impossible to obtain Chan's implementation). Moreno and Muñoz in [18] discuss how to incorporate negation in a Prolog compiler. But the paper essentially discusses ways to avoid using constructive negation. On the other hand, the paper leaves opened the problem of how constructive negation can be implemented to use it when is unavoidable. A

similar problem happens in [19], where the use of abstract interpretation is discussed in this context. Finally, quite recently we have learned about [20] where an implementation of constructive negation is proposed. However, the proposal seems very preliminary. No proof of soundness or completeness is provided and, actually, some examples have led us to think that this implementation is not yet fully correct.

6. ADDITIONAL AUTHORS

Edelmira Pasarella[(1)(2)] and Elvira Pino[(1)]
[(1)]Dpto de L.S.I., Univ. Politécnica de Catalunya, Campus Nord, Modul C6, Jordi Girona 1-3, 08034 Barcelona, Spain.
email: {edelmira,pino}@lsi.upc.es
[(2)]Dpto de Comp. y Tecn. de la Inf., Univ. Simón Bolívar, Aptdo 89000, Caracas 1080, Venezuela.
email: edelmira@ldc.usb.ve

7. REFERENCES

[1] J. Álvez, P. Lucio, F. Orejas, E. Pasarella, and E. Pino. The BCN prototype: An implementation of constructive. Technical Report UPV/EHU/LSI/TR 12-2003, Fac. de Informática de San Sebatián, November 2003.

[2] M. Barbuti, P. Mancarella, D. Pedreschi, and F. Turini. A tranformational approach to negation in logic programming. *Journal of Logic Programming*, 8:201–228, 1990.

[3] R. Barták. Constructive negation in clp(h). Technical Report No 98/6,, Dept. of Theoretical Computer Science, Charles Univ., Prague, July 1998.

[4] A. Bossi, M. Fabris, and M. C. Meo. A bottom-up semantics for constructive negation. In P. V. Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP '94)*, pages 520–534. MIT Press, 1994.

[5] P. Bruscoli, F. Levi, G. Levi, and M. C. Meo. Compilative constructive negation in constraint logic programs. In S. Tison, editor, *Proc. of the Trees in Algebra and Programming 19th Int. Coll.(CAAP '94)*, volume 787 of *LNCS*, pages 52–67. Springer-Verlag, 1994.

[6] D. Chan. Constructive negation based on the completed database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. of the 5th Int. Conf. and Symp. on Logic Progr.*, pages 111–125. MIT Press, 1988.

[7] D. Chan. An extension of constructive negation and its application in coroutining. In E. Lusk and R. Overbeek, editors, *Proc. of the NACLP'89*, pages 477–493. MIT Press, 1989.

[8] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322, New York, 1978. Plenum Press.

[9] A. Colmerauer and T.-B.-H. Dao. Expresiveness of full first order constraints in the algebra of finite and infinite trees. In *6th Int. Conf. of Principles and Practice of Constraint Programming CP'2000*, volume 1894 of *LNCS*, pages 172–186, 2000.

[10] H. Common. Disunification: A survey. In J. Lassez and G. Plotkin, editors, *Essays in Honour of Alan Robinson*, 1991.

[11] W. Drabent. What is failure? an approach to constructive negation. *Acta Informática*, 32:27–59, 1995.

[12] F. Fages. Constructive negation by pruning. *Journal of Logic Programming*, 32(2):85–118, 1997.

[13] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[14] J. Jaffar and J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19,20:503–581, 1994.

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

[16] P. Lucio, F. Orejas, and E. Pino. An algebraic framework for the definition of compositional semantics of normal logic programs. *Journal of Logic Programming*, 40:89–123, 1999.

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

[18] J. Moreno-Navarro and S. Muñoz. How to incorporate negation in a prolog compiler. In V. Santos and E. Pontelli, editors, *Practical Applications Declarative Languages PADL'2000*, number 1753 in LNCS, pages 124–140, 2000.

[19] S. Muñoz, J. J. Moreno, and M. Hermenegildo. Efficient negation using abstract interpretation. In R.Nieuwenhuis and A. Voronkov, editors, *Proc. of the Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2001)*, number 2250 in LNAI, 2001.

[20] S. Muñoz and J. J. Moreno-Navarro. Constructive negation for prolog: A real implementation. In *Proc. of the Joint Conference on Declarative Programming AGP'2002*, pages 39–52, 2002.

[21] E. Pasarella, E. Pino and F. Orejas. Constructive negation without subsidiary trees. In *Proc. of the 9th Internatonal Workshop on Functional and Logic Programming, WFLP'2000, Benicassim, Spain.* Also available as Technical Report LSI-00-44-R of LSI Department, Univ. Politécnica de Catalunya, 2000.

[22] J. Shepherdson. Language and equality theory in logic programming. Technical Report No. PM-91-02, University of Bristol, 1991.

[23] P. J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.

[24] S. Vorobyov. An improved lower bound for the elementary theories of trees. In *Automated Deduction CADE-13 LNAI 110*, pages 275–287. Springer, 1996.