The BCN Prototype: An Implementation of Constructive Negation *

J. Álvez, P. Lucio
Dpto de L.S.I., Univ. Pais Vasco
Paseo Manuel de Lardizabal, 1
20080-San Sebastián, Spain.
{jibalgij,jiplucap}@si.ehu.es

F. Orejas, E. Pasarella †, E. Pino
Dpto de L.S.I., Univ. Politécnica Catalunya
Campus Nord, Modul C6, Jordi Girona 1-3,
08034-Barcelona, Spain.
{orejas,edelmira,pino}@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.

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

†Additional address for this author: Dpto de Computación y Tecnologia de la Información, Univ. Simón Bolívar, Aptdo 89000, Caracas 1080, Venezuela. email: edelmira@ldc.usb.ve

Keywords

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

1. INTRODUCTION

The idea of constructive negation was introduced by Chan in [5] and extended, by Chan (in [6]) and by Drabent (in [10]), to a complete and sound operational semantics for the whole class of normal logic programs. Fages (in [11]) and Stuckey (in [22]) provided generalizations of constructive negation to the framework of Constraint Logic Programming (CLP for short, see [13] 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 [6]) and Stuckey (in [22]) 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. [10] and [11]) 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 following, we will present an example that explains how these approaches work.

Let P be the following program:

$$\begin{array}{l} \mathtt{p(a)}. \\ \mathtt{p(f(X,Y))} :- \mathtt{p(X)},\ \neg\, \mathtt{p(Y)}. \\ \mathtt{p(f(X,Y))} :- \neg\, \mathtt{p(X)},\ \mathtt{p(Y)}. \end{array} \qquad (1)$$

over the signature $\{\mathtt{a}\backslash 0, \mathtt{f}\backslash 2\}$ and assume that we want to solve the query

$$\neg \mathtt{p(Z)}$$

Figure 1 represents the subsidiary computation tree for $\mathtt{p(Z)}$ and figure 2 represents the pre-computation tree for this query according to the complex-goals approach. In particular, the successor of node (**1**) (i.e. node (**2**)) in figure 2 comes from the negation of the first-level frontier of node

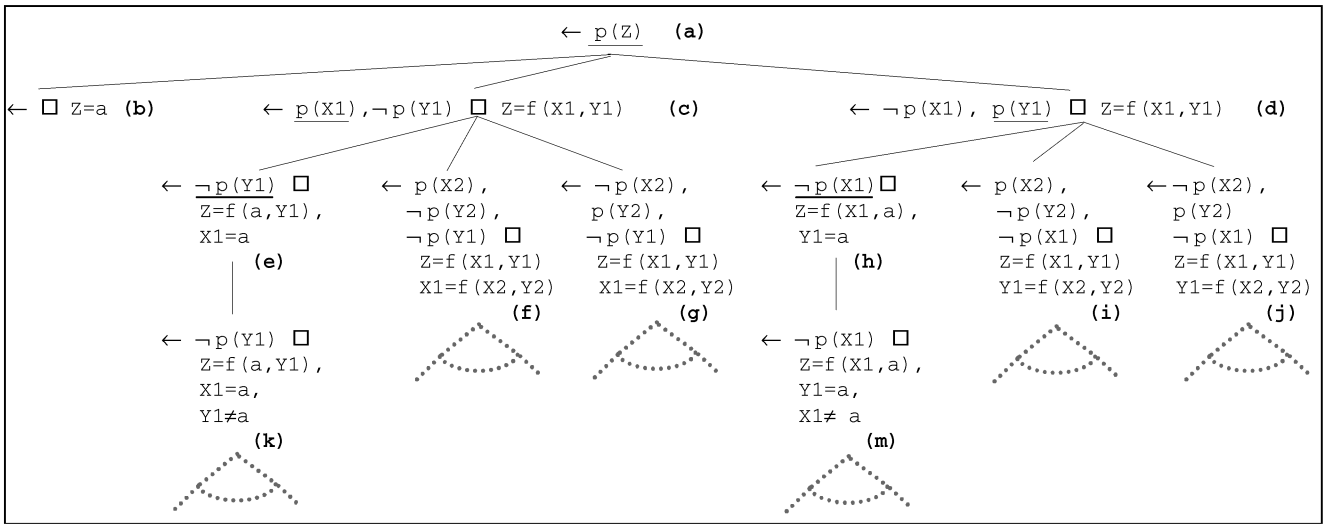

Figure 1: Subsidiary tree

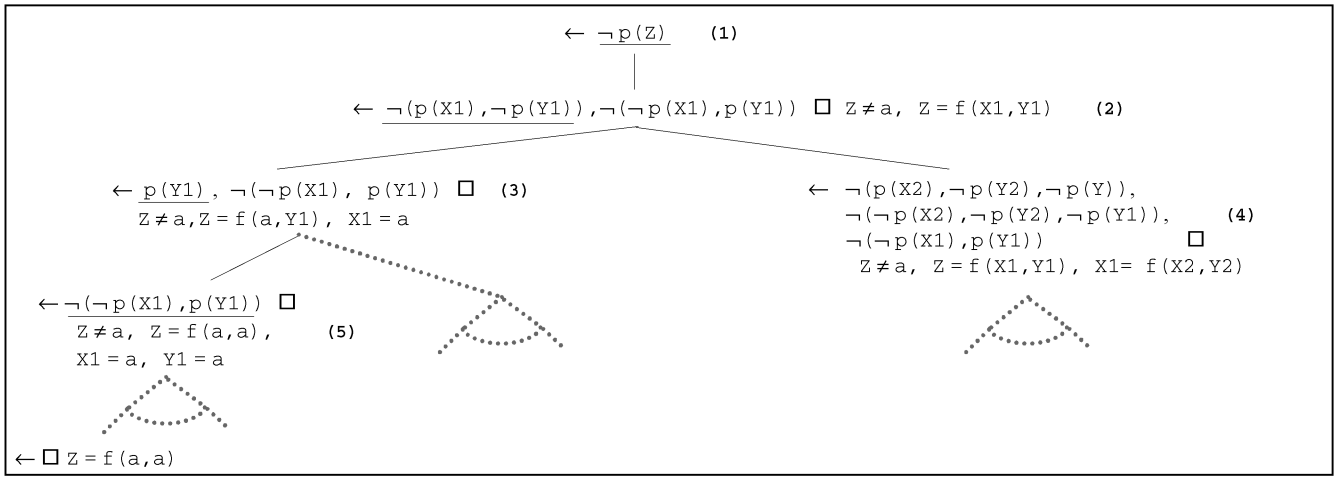

Figure 2: A *complex-goal* pre-computation tree

(**a**) in figure 1 (i.e. nodes (**b**), (**c**) and (**d**)). In general, we could have (non-deterministically) chosen a different frontier in the subsidiary tree, but in this case, we have made the simplest choice consisting of the immediate successors of (**a**).

The computation of the successors of node (**2**) in figure 2 is a little bit more involved. First, we have to choose a complex subgoal of (**2**). In this case we have chosen $\neg(\mathtt{p(X1)}, \neg\mathtt{p(Y1)})$. Then, if this subgoal is negative, we have to compute a frontier for its positive counterpart. This frontier may consist of nodes (**e**), ((**f**) and (**g**) in fig. 1, if we select for resolution the literal $\mathtt{p}(\mathtt{X}_1)$. Next, we have to compute the successors of (**2**) by substituting the chosen subgoal by the negation of that frontier. However, the result is not yet a complex-goal since it includes some (inner) disjunctions. Then, using repeatedly the transformational rule:

$$\neg\exists\mathtt{X}\exists\mathtt{Y}(\mathtt{c} \wedge \mathtt{G}) \longmapsto \neg\exists\mathtt{X}\exists\mathtt{Y}(\mathtt{c}) \vee \exists\mathtt{X}\exists\mathtt{Y}(\mathtt{c} \wedge \neg\mathtt{G})$$

together with distribution and quantifier elimination we can arrive to a disjunction of eight complex-goals. However, six of them can be discarded since they can be shown to be unsatisfiable because they include the constraint:

$$\neg(\mathtt{Z} = \mathtt{a}) \wedge \neg\exists\mathtt{X}\exists\mathtt{Y}(\mathtt{Z} = \mathtt{f}(\mathtt{X}, \mathtt{Y}))$$

Therefore, the two remaining complex goals (nodes (**3**) and (**4**)) are the successors of (**2**). This process would continue until arriving to goals consisting only of equality constraints.

It may be noticed that, along the computation, goals become more and more complex, especially by nested negation (see (**4**) for instance). 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.

It may also be noticed that this 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 frontiers to be used in a given

computation, according to [22], can also be chosen nondeterministically. However, defining the frontiers in terms of the immediate successors in the subsidiary tree, as done in the example above, seems to be a reasonable choice.

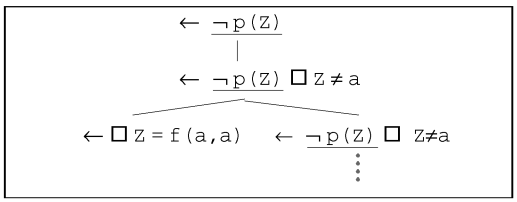

Figure 3: A *fail-answers* pre-computation tree

In the case of the fail-answers approach, things work differently. In this approach, 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. For instance, in the previous example, the success **(b)** of Figure 1 gives rise to the first child in the main computation tree of Figure 3. Then, since the negation of the constraints of the first level frontier (nodes **(b)**, **(c)** and **(d)**) is unsatisfiable, we cannot obtain any answer from that frontier and we have to compute another one. As a consequence, the computation of the subsidiary tree would continue. This computation can stop when arriving at the frontier consisting of nodes **(b)**, **(k)**, **(f)**, **(g)**, **(m)**, **(i)** and **(j)**. In this frontier, the conjunction of the negation of the associated constraints is satisfiable and equivalent to the answer `Z=f(a,a)`.

It may be noticed that goals **(e)** and **(h)** require a sub-subsidiary computation tree. It may also be noticed that this approach, especially the approach of [11], 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 involves recursion through negated atoms.

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. [21]). 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 [1]. It was extended, in [4], to the CLP setting. To solve a negative literal, our schemes work similarly (but not identically) to the negative programs which are obtained (in [1, 4]) by transformation. In [1], the presence of universal quantification in goals prevents to achieve a complete goal computation mechanism. This problem is carried out in [4] 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.

Let us give a preliminary explanation of our approach through the previous program (1). Shepherdson's operators give the following bottom-up scheme for the definition of the success-answers of the literal $\neg\mathtt{p(Z)}$ (or equivalently, the fail-answers of $\mathtt{p(Z)}$):

$$\begin{aligned} F_1(\mathtt{p(Z)}) &\equiv \quad \underline{\mathtt{f}} \qquad (\underline{\mathtt{f}} \text{ denotes falsehood}) \\ F_{k+1}(\mathtt{p(Z)}) &\equiv \quad \exists \mathtt{X1}\exists \mathtt{Y1}(\ \mathtt{Z} = \mathtt{f(X1,Y1)} \wedge \\ &\qquad\qquad (F_k(p(\mathtt{X1})) \vee T_k(p(\mathtt{Y1}))) \wedge \\ &\qquad\qquad (T_k(p(\mathtt{X1})) \vee F_k(p(\mathtt{Y1})))\ \ ) \end{aligned}$$

The computation of this scheme requires the same computational effort than the negation of the first level frontier for $\mathtt{p(Z)}$. However, using this scheme, as a consequence of the monotonicity of the operators, we avoid the above mentioned repetitions of symbolic transformations and satisfiability checks. That is, the success-answers of $\neg\mathtt{p(Z)}$ are bottom-up obtained incrementally from the scheme as follows (we show the next two iterations):

$$\begin{aligned} F_2(\mathtt{p(Z)}) &\equiv \exists \mathtt{X1}\exists \mathtt{Y1}(\mathtt{Z} = \mathtt{f(X1,Y1)} \wedge \\ &\qquad\qquad (\underline{\mathtt{f}} \vee \mathtt{Y1} = \mathtt{a}) \wedge (\mathtt{X1} = \mathtt{a} \vee \underline{\mathtt{f}})) \\ &\equiv \mathtt{Z} = \mathtt{f(a,a)} \\ F_3(\mathtt{p(Z)}) &\equiv \exists \mathtt{X1}\exists \mathtt{Y1}(\mathtt{Z} = \mathtt{f(X1,Y1)} \wedge \\ &\qquad\qquad (\mathtt{X1} = \mathtt{f(a,a)} \vee \mathtt{Y1} = \mathtt{a}) \wedge \\ &\qquad\qquad (\mathtt{X1} = \mathtt{a} \vee \mathtt{Y1} = \mathtt{f(a,a)})) \\ &\equiv \mathtt{Z} = \mathtt{f(a,a)} \vee \mathtt{Z} = \mathtt{f(f(a,a),f(a,a))} \end{aligned}$$

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. [10, 11]) this is due to their non-deterministic formulation. In the case of complex-goals (cf. [6, 22, 4]), 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 $\forall\overline{z}(c \vee B)$.

In order to show how failure detection works, let us add two clauses to program (1):

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

$$\mathtt{r(f(a,a))}.$$

Then, the goal $\leftarrow\neg\mathtt{p(Z)}, \neg\mathtt{q(Z)}$ has exactly one success-answer $\mathtt{Z} = \mathtt{f(a,a)}$ and, after it, the computation fails. A BCN-computation tree is shown in Figure 4. The literal $\neg\mathtt{p(Z)}$

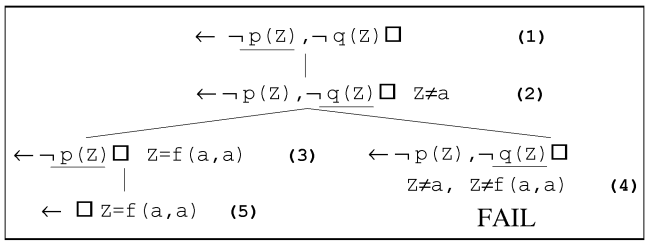

Figure 4: BCN-computation tree

has no success at the first iteration. Then, $\mathtt{Z} = \mathtt{a}$ is the only fail-answer of $\neg\mathtt{p(Z)}$ at the first iteration. Hence, the goal (**1**) does not fail, but it is restricted by $\mathtt{Z} \neq \mathtt{a}$ to give the goal (**2**). Next, selecting $\neg\mathtt{q(Z)}$, there are neither success- nor fail-answers at the first iteration. Hence the goal remains unchanged and both iteration-counters are incremented. To simplify we have supposed that the same literal is selected again. At the second iteration $\mathtt{Z} = \mathtt{f(a,a)}$ is a success, then the branch is split into the goals (**3**) and (**4**). The latter branch is for trying with higher-iterations success of $\neg\mathtt{p(Z)}$. The answer (**5**) comes from $F_2(\mathtt{p(Z)}) \equiv \mathtt{Z} = \mathtt{f(a,a)}$. The goal (**4**) fails because $\neg\mathtt{p(Z)}$ has not new success at the second iteration, but each answer of the goal-constraint $\mathtt{Z} \neq \mathtt{a} \wedge \mathtt{Z} \neq \mathtt{f(a,a)}$ is a second-iteration fail-answer of $\neg\mathtt{p(Z)}$.

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

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, like $\overline{x}$ as abbreviation of the n-tuple of variables $x_1, \ldots, x_n$. Concatenation of tuples is denoted by the infix $\cdot$ operator, i.e. $\overline{x}\cdot\overline{y}$ represents the concatenation of $\overline{x}$ and $\overline{y}$.

We classify (possibly negated) atoms into (dis)equations and literals depending on the predicate symbol they use: the equality for the former and any uninterpreted predicate symbol for the latter. 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 *substitution* σ is a mapping from a finite set of variables, called its domain, into the set of terms. It is assumed that σ behaves as the identity for the variables outside its domain. The *most general unifier* of a set of terms $\{s_1, \ldots, s_n\}$, denoted $mgu(s_1, \ldots, s_n)$, is an idempotent substitution σ such that $\sigma(s_i) \equiv \sigma(s_j)$ for all $i, j \in 1..n$ and for any other substitution θ with the same property, $\theta \equiv \sigma'\cdot\sigma$ holds for some substitution σ'. For tuples, $mgu(\overline{s}^1, \ldots, \overline{s}^m)$ is an abbreviation of $\sigma_1\cdot\cdots\cdot\sigma_n$ where $\sigma_i \equiv mgu(s_i^1, \ldots s_i^m)$ for all $i \in 1..n$.

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 constrained 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 $:-\overline{\ell}(\overline{y}) \,\square\, b(\overline{x}\overline{y}, \overline{w})$ 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:

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

To define the semantics of a Σ-program P, Clark [7] 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 *free equality theory* [2] $FET(\Sigma)$ together with the set P^* of the predicate completion formulas for every $p \in PS_\Sigma$. Then, the standard declarative meaning of normal logic programs is $Comp(P)$ interpreted in three-valued logic (cf. [14]).

The theoretical foundations of our proposal comes from a result of Shepherdson ([21]) 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 ([21]). These operators provide a bottom-up scheme for computing the success- and fail-answers of a given flat literal.

DEFINITION 1. *The operators* T_k *and* F_k *are inductively defined, with respect to a* Σ*-program* P*, as follows:*

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

$$T_k(\overline{\ell}^1 \wedge \overline{\ell}^2) \equiv T_k(\overline{\ell}^1) \wedge T_k(\overline{\ell}^2) \quad T_k(\neg p(\overline{x})) \equiv F_k(p(\overline{x}))$$
$$F_k(\overline{\ell}^1 \wedge \overline{\ell}^2) \equiv F_k(\overline{\ell}^1) \vee F_k(\overline{\ell}^2) \quad F_k(\neg p(\overline{x})) \equiv T_k(p(\overline{x}))$$
$$T_k(\underline{\mathtt{t}}) \equiv \underline{\mathtt{t}} \qquad F_k(\underline{\mathtt{t}}) \equiv \underline{\mathtt{f}}$$ ■

[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.[7]) or the *first-order theory of finite trees.*

A key result for our work is the following Theorem 1, which is a simple consequence of Theorem 6 and Lemma 4.1 in [21].

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* $k \in \mathbb{N}$ ■

In particular, $Comp(P) \models_3 (T_k(\overline{\ell}) \rightarrow \overline{\ell})^\forall$ and $Comp(P) \models_3 (F_k(\overline{\ell}) \rightarrow \neg\overline{\ell})^\forall$ hold for every $k \in \mathbb{N}$. 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 over such literal. Different immediate consequence operators, providing bottom-up semantics for normal logic programs, have been proposed (cf. [3, 12, 14, 15, 22]). Intuitively, the equality constraint $T_k(\ell)$ represents the k-success of ℓ, whereas $F_k(\ell)$ gives the k-failures of ℓ. As a result, the operators T and F are monotonic and coherent, in the following sense:

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

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

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

(iii) $FET(\Sigma) \models (T_n(\ell(\overline{x})) \rightarrow \neg F_k(\ell(\overline{x})))^\forall$ *for all* $k \in \mathbb{N}$

(iv) $FET(\Sigma) \models (F_n(\ell(\overline{x})) \rightarrow \neg T_k(\ell(\overline{x})))^\forall$ *for all* $k \in \mathbb{N}$

Proof. The four items follow − by an easy induction on n − from Definition 1. ■

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. [9, 16], for instance). It is known that the decidability of FET(Σ) is a non-elementary problem (cf. [23]). 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. [9]) 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.* ■

The following transformation rules are used to obtain collapsing (dis)equations. Any equation can be transformed into an equivalent conjunction of collapsing equations (in particular, $\underline{\mathtt{f}}$), by repeatedly applying of the following three rules:
$(E1)$ $f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n) \longmapsto t_1 = s_1 \wedge \cdots \wedge t_n = s_n$
$(E2)$ $f(t_1,\ldots,t_n) = g(s_1,\ldots,s_n) \longmapsto \underline{\mathtt{f}}$ if $f \not\equiv g$
$(E3)$ $x = t \longmapsto \underline{\mathtt{f}}$ if x occurs in t.
Similarly, any disequation can be transformed into a disjunction of collapsing disequations (in part., $\underline{\mathtt{t}}$) by the rules:
$(D1)$ $f(t_1,\ldots,t_n) \neq f(s_1,\ldots,s_n) \longmapsto t_1 \neq s_1 \vee \cdots \vee t_n = s_n$
$(D2)$ $f(t_1,\ldots,t_n) \neq g(s_1,\ldots,s_n) \longmapsto \underline{\mathtt{t}}$ if $f \not\equiv g$
$(D3)$ $x \neq t \longmapsto \underline{\mathtt{t}}$ if x occurs in t.

In order to deal with universal quantification, we use the transformation rule (UD) in Figure 5. The rule (UD) is correct (w.r.t. the theory FET_Σ of any signature Σ) provided that each variable x_i is either a fresh variable or a w_j that does not occur in the term t_i.

$$\forall \overline{v}(\overline{x} \neq \overline{t}(\overline{w},\overline{v}) \vee \varphi(\overline{w},\overline{v})) \longmapsto$$
$$\forall \overline{v}^1(\overline{x} \neq \overline{t}(\overline{w},\overline{v}^1)) \vee \exists \overline{v}^1(\overline{x} = \overline{t}(\overline{w},\overline{v}^1) \wedge \forall \overline{v}^2 \varphi(\overline{w},\overline{v}))$$
$$\text{where } \overline{v}^1 \equiv free(\overline{t}) \cap \overline{v} \text{ and } \overline{v}^2 \equiv \overline{v} \setminus \overline{v}^1$$

Figure 5: Transformation Rule (UD)

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
$\mathtt{x_1} = \_\mathtt{A},\ \mathtt{x_2} = \_\mathtt{B},\ \mathtt{x_3} = \mathtt{g}(\_\mathtt{A}),\ \_\mathtt{A} \neq \mathtt{a}, \_\mathtt{A} \neq \_\mathtt{B},\ \_\mathtt{A} \neq \mathtt{f}(*\mathtt{C}, \_\mathtt{B})$
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 [8], where only infinite signatures are considered. However, for finite signatures, an answer can be unsatisfiable. For example, $\exists w(x = w \wedge w \neq a \wedge w \neq g(a) \wedge \forall v_1(w \neq g(g(v_1))))$ 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.*

Proof. The equational part of an answer is always satisfiable. Our satisfiability test only looks up the disequational part of the input answer. It must decide if there exists some possible assignment to the variables $\overline{w}$ satisfying all the disequations. The initial domain of each w_j, called $Dom(w_j)$ is determined by the signature. We can represent it using the anonymous variable "_". For example, over the signature $\Sigma \equiv \{a/0, g/1, f/2\}$, the initial domain of any $w_i \in \overline{w}$ is given by $Dom(w_i) = \{a, g(\_), f(\_,\_)\}$ The checking is made in two steps. In the first step, it only takes into account the disequations (of the input answer) whose right-hand term has neither existential variables nor repetitions of universal variables. These are of the form $\forall \overline{v}(w_i \neq s(\overline{v}))$ where every $v_j \in \overline{v}$ occurs (at most) once in $s(\overline{v})$. We refine the initial domain $Dom(w_j)$ for each w_j, in order to eliminate the values that not satisfy these disequations. To do that we use the unfolding technique. For example, with the above initial domain for w_1 and the disequation $\forall v(w_1 \neq f(a, v))$, the domain is refined to $Dom(w_1) = \{a, g(\_), f(f(\_,\_),\_), f(g(\_),\_)\}$. Notice also that $w_1 \neq a$ is in the class of the considered disequations. Hence, if it belongs to the input answers, then $Dom(w_1)$ becomes $\{g(\_), f(f(\_,\_),\_), f(g(\_),\_)\}$. Once all these disequations has been applied, each $Dom(w_j)$ can be empty, finite (a non-empty set of ground terms) or infinite (when it contains at least one anonymous variable). Obviously, if $Dom(w_j)$ is empty for some w_j, then the answer is unsatisfiable and the test is finished. Otherwise, it is easy to realize that every disequation such that

- it has not been treated yet (in the first step), and
- it involves at least one variable with infinite domain

is satisfiable with independence of the possible assignments to its variables with finite domain. Therefore, if $Dom(w_j)$ is infinite for all w_j, then the answer is satisfiable and the checking is stopped. Otherwise, the set

$$Fin(\overline{w}) \equiv \{w_i | Dom(w_i) \text{ is finite (and non-empty)}\}$$

is a non-empty subset of $\overline{w}$. In the second step, we only deal with the variables in $Fin(\overline{w})$ and the disequations of the form

$$\forall \overline{v}(w_j \neq s(\overline{v}, w_{1_1}, \ldots, w_{i_k}))$$

where $\{w_j, w_{1_1}, \ldots, w_{i_k}\} \subset Fin(\overline{w})$. Therefore, the problem is a very simple finite domain constraint solving problem (CSP): to decide if there exists some substitution σ such that $\sigma(w_j) \in Dom(w_j)$ for each $w_j \in Fin(\overline{w})$ and σ satisfies all these disequations.

In particular, for infinite signature, if the first step were performed, then every domain would remain infinite. Notice that any answer contains a finite number of disequations. Therefore, as explained above, the answer 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 $\overline{z}$.

Proof. (i) A conjunction of n answers for the variables $\overline{x}$:

$$\bigwedge_{i=1}^{n} \exists \overline{w}^i (\overline{x} = \overline{t}^i(\overline{w}^i) \wedge \bigwedge \forall \overline{v}\ (w_j \neq s_k(\overline{w}^i, \overline{v})))$$

is performed by unification of the tuples of terms $\overline{t}^i(\overline{w}^i)$. If the most general unifier does not exist, the result is $\underline{\mathtt{f}}$. Otherwise, the *mgu* σ is applied. Then, we obtain a constraint of the form

$$\exists \overline{w} (\overline{x} = \overline{t}(\overline{w}) \wedge \bigwedge_{j} \forall \overline{v}\ (u_j(\overline{w}) \neq s_j(\overline{w}, \overline{v})))$$

where $\overline{w}$ collects all the variables $\overline{w}^i$ occurring in the constraint. Then, each disequation is reduced to a disjunction of collapsing disequations (by rules $(D1) - (D3)$):

$$\exists \overline{w} (\overline{x} = \overline{t}(\overline{w}) \wedge \bigwedge_{j} \forall \overline{v}\ (w_{j_1} \neq r_{j_1}(\overline{w}, \overline{v}) \vee \cdots \vee w_{j_k} \neq r_{j_k}(\overline{w}, \overline{v})))$$

Now, by the transformation rule (UD) of Figure 5, we transform each universal disjunction of collapsing disequations as follows:

$$\exists \overline{w} (\overline{x} = \overline{t}(\overline{w}) \wedge \bigwedge \bigvee \exists \overline{v}^1 (\overline{w} = \overline{s}(\overline{w}, \overline{v}^1) \wedge \forall \overline{v}^2\ (w_{j_i} \neq r_{j_i}(\overline{w}, \overline{v}^2)))$$

where, in some disjuncts, the equational part for variables w could be empty. By distribution and lifting the disjunction:

$$\bigvee \exists \overline{w} \big(\overline{x} = \overline{t}(\overline{w}) \wedge \underbrace{\bigwedge \exists \overline{v}^1 (\overline{w} = \overline{s}(\overline{w}, \overline{v}^1) \wedge \forall \overline{v}^2 (w_{j_i} \neq r_{j_i}(\overline{w}, \overline{v}^2)))}_{\varphi}\big)$$

It suffices to transform the inner conjunction φ into

$$\exists \overline{v}^1 (\overline{w} = \overline{s}'(\overline{w}, \overline{v}^1) \wedge \bigwedge \forall \overline{v}^2\ (w_{j_i} \neq r_{j_i}(\overline{w}, \overline{v}^2)))$$

by collecting equations for $\overline{w}$. We use unification where there are two o more equations on the same variable w_i. Finally, by substitution on the terms $\overline{t}(\overline{w})$, we obtain the following disjunction of answers for $\overline{x}$:

$$\bigvee \exists \overline{w} \cdot \overline{v}^1 (\overline{x} = \overline{t}'(\overline{w} \cdot \overline{v}^1) \wedge \bigwedge \forall \overline{v}^2\ (w_{j_i} \neq r_{j_i}(\overline{w}, \overline{v}^2)))$$

(ii) The negation of an answer for $\overline{x}$:

$$\forall \overline{w} (\overline{x} \neq \overline{t}(\overline{w}) \vee \bigvee \exists \overline{v}^j (w_j = s(\overline{w}, \overline{v}^j)))$$

is equivalent (by the rule (UD) and a renaming) to

$$\exists \overline{w}' (\overline{x} = \overline{w}' \wedge \forall \overline{v} (\overline{w}' \neq \overline{t}(\overline{v}))) \ \vee$$
$$\exists \overline{w} (\overline{x} = \overline{t}(\overline{w}) \wedge \bigvee \exists \overline{v}^j (w_j = s(\overline{w}, \overline{v}^j)))$$

The second disjunct (lifting the inner existential disjunction and substituting w_j in $t(\overline{w})$) is already a disjunction of answers for $\overline{x}$. To transform the first one into a disjunction of answers for $\overline{x}$, it is enough to apply (just as in (i)) the transformation rule (UD) of Figure 5 to $\forall \overline{v} (\overline{w}' \neq \overline{t}(\overline{v}))$ and then to substitute the equations on variables $\overline{w}'$ on the right-hand side of $\overline{x} = \overline{w}'$.

(iii) The instantiation of the variables $\overline{x}$ by terms $\overline{t}'(\overline{z})$ in an answer for $\overline{x}$:

$$\exists \overline{w} (\overline{x} = \overline{t}(\overline{w}) \wedge \bigwedge \forall \overline{v} (w_j \neq s(\overline{w}, \overline{v}))) [\overline{t}'(\overline{z})/\overline{x}]$$

is performed on the basis of $\mu \equiv mgu(\overline{t}(\overline{w}), \overline{t}'(\overline{z}))$. If such *mgu* does not exist, then the result is $\underline{\mathtt{f}}$. Otherwise, it is equivalent to

$$\exists \overline{w} (\mu_1 \wedge \bigwedge \forall \overline{v} ((w_j \neq s(\overline{w}, \overline{v}))\mu_2)$$

where $\mu_1 \equiv \mu \restriction \overline{z}$ and $\mu_2 \equiv \mu \restriction \overline{w}$. To obtain a disjunction of answers for $\overline{z}$ we firstly apply μ_2. Then, disequations are transformed into disjunctions of collapsing disequations. Finally, we split the universal variables using the transformation rule (UD) as in (i) and (ii). ■

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

lation 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:*

$$\begin{aligned} \mathcal{O}_1(p(\overline{x})) &\equiv \bigvee \exists \overline{w} (a(\overline{x}, \overline{w})) \\ \mathcal{O}_{k+1}(p(\overline{x})) &\equiv \bigvee \exists \overline{w} \Big(a(\overline{x}, \overline{w}) \wedge \bigwedge_{j=1}^{n} \varphi_j^{[p,k]}(\overline{w}) \Big) \end{aligned} \quad (1)$$

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.

Proof. The schemes for both operators are obtained transforming their definition (see Definition 1). For the operator T, since $T_k(\underline{\mathtt{t}}) = \underline{\mathtt{t}}$ for any k, it suffices to consider separately the clauses with an empty tuple of literals (equivalent to $\underline{\mathtt{t}}$):

$$T_1(p(\overline{x})) \equiv \bigvee_{\overline{t}^i \equiv \underline{\mathtt{t}}} \exists \overline{w}^i (\overline{x} = \overline{t}^i(\overline{w}^i))$$

$$T_{k+1}(p(\overline{x})) \equiv \bigvee_{\overline{t}^i \not\equiv \underline{\mathtt{t}}} \exists \overline{y}^i \cdot \overline{w}^i (\overline{x} = \overline{s}^i(\overline{w}^i) \wedge \overline{y}^i = \overline{r}^i(\overline{w}^i) \wedge T_k(\overline{\ell}^i(\overline{y}^i)))$$

$$\equiv \bigvee_{\overline{\ell}^i \not\equiv \underline{\mathtt{t}}} \exists \overline{w}^i (\overline{x} = \overline{s}^i(\overline{w}^i_j) \wedge \bigwedge_j T_k(l^i(\overline{y}^i_j))[\overline{r}^i(\overline{w}^i)/\overline{y}^i])$$

With regard to the operator F, we first apply the rule (UD) (Fig. 5) to the definition of $F_{k+1}(p(\overline{x}))$ (Def. 1), this yields:

$$F_{k+1}(p(\overline{x})) \equiv \bigwedge_{i=1}^{m} \big(\forall \overline{w}^{i1} (\, \overline{x} \neq \overline{t}^i(\overline{w}^{i1})) \vee \exists \overline{w}^{i1} (\, \overline{x} = \overline{t}^i(\overline{w}^{i1}) \wedge \forall \overline{w}^{i2} (F_k(\overline{\ell}^i(\overline{y}^i))[\overline{r}^i(\overline{w}^i)/\overline{y}^i])) \big)$$

where each $\forall \overline{w}^{i1}(\overline{x} \neq \overline{t}^i(\overline{w}^{i1}))$ is the negation of one answer for $\overline{x}$, that is transformed into a disjunction of answers for $\overline{x}$. Then we have a formula of the form:

$$F_{k+1}(p(\overline{x})) \equiv \bigwedge_{i=1}^{m} \big(\bigvee \exists \overline{w}^{i1} (\, a(\overline{x}, \overline{w}^{i1})) \vee \exists \overline{w}^{i1} (\quad \overline{x} = \overline{t}^i(\overline{w}^{i1}) \wedge \forall \overline{w}^{i2} (F_k(\overline{\ell}^i(\overline{y}^i))[\overline{r}^i(\overline{w}^i)/\overline{y}^i])) \big)$$

Then, by distribution, there is a disjunct of the form

$$\bigwedge_{i=1}^{m} \big(\bigvee \exists \overline{w}^{i1} (a(\overline{x}, \overline{w}^{i1})) \big)$$

which is equivalent to $F_1(p(\overline{x}))$, since $F_0(\ell) \equiv \underline{\mathtt{f}}$ for any ℓ. Now, by distribution and conjunction of answers we are able to transform it into a disjunction of answers for $\overline{x}$:

$$F_1(p(\overline{x})) \equiv \bigvee \exists \overline{w} (a(\overline{x}, \overline{w}))$$

The remaining disjuncts (after the above distribution) are conjunctions of formulas of both forms: $\exists \overline{w}(a(\overline{x}, \overline{w}))$ and $\exists \overline{w}(\overline{x} = \overline{t}(\overline{w}) \wedge \forall \overline{v}(F_k(\overline{\ell}(\overline{y}))[\overline{r}(\overline{w}, \overline{v})/\overline{y}]))$, with at least one of the second type. Therefore, performing conjunction of answers for $\overline{x}$, we get that $F_{k+1}(p(\overline{x}))$ for $k \geq 1$ has the following form:

$$\bigvee \exists \overline{w} \big(a(\overline{x}, \overline{w}) \wedge \bigwedge_j \forall \overline{v} (F_k(\overline{\ell}^j(\overline{y}^j))[\overline{r}^j(\overline{w}, \overline{v})/\overline{y}^j]) \big)$$

where the variables $\overline{v}$ of some (or even all) members in the internal conjunction may be empty. In this case, each of them is equivalent to

$$F_k(\ell^j_1(\overline{y}^j))[\overline{r}^j(\overline{w})/\overline{y}^j] \vee \cdots \vee F_k(\ell^j_n(\overline{y}^j))[\overline{r}^j(\overline{w})/\overline{y}^j]$$

Then, distributing and lifting the disjunction to the outermost disjunction of the scheme, we obtain that, for $k \geq 1$:

$$F_{k+1}(p(\overline{x})) \equiv \bigvee \exists \overline{w} \big(a(\overline{x}, \overline{w}) \wedge \bigwedge_j \varphi^{[p,k]}_j(\overline{w}) \big)$$

where each $\varphi^{[p,k]}_j(\overline{w})$ is of the form either $F_k(\ell^j_i(\overline{y}))[\overline{t}(\overline{w})/\overline{y}]$ or $\forall \overline{v}(F_k(\overline{\ell}^j(\overline{y}^j))[\overline{r}^j(\overline{w}, \overline{v})/\overline{y}^j])$. ■

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* $\{\mathtt{0} \backslash 0, \mathtt{s} \backslash 1\}$*-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 $\varphi^{[p,k]}_j(\overline{w})$ of (1) can be split into $\varphi^{[p,k-1]}_j(\overline{w}) \vee \varphi^{[p,=k]}_j(\overline{w})$ ■

Proof. We distinguish the two cases of $\varphi^{[p,k]}_j(\overline{w})$. If it has the form *(i)* $\mathcal{O}_k(\ell(\overline{y}))[\overline{t}(\overline{w})/\overline{y}]$, then

$$\mathcal{O}_k(\ell(\overline{y}))[\overline{t}(\overline{w})/\overline{y}] \equiv \mathcal{O}_{k-1}(\ell(\overline{y}))[\overline{t}(\overline{w})/\overline{y}] \vee \mathcal{O}_{=k}(\ell(\overline{y}))[\overline{t}(\overline{w})/\overline{y}]$$

holds as a direct consequence of the induction hypothesis. In the case of *(ii)*

$$\varphi^{[p,k]}_j(\overline{w}) \equiv \forall \overline{v} \Big(F_k(\overline{\ell}(\overline{y}))[\overline{t}(\overline{w}, \overline{v})/\overline{y}] \Big) \equiv$$

$$\neg\exists\overline{v}\Big(\underbrace{\neg F_{k-1}(\overline{\ell}(\overline{y}))[\overline{t}(\overline{w},\overline{v})/\overline{y}]}_{\psi}\ \wedge\ \underbrace{\neg F_{=k}(\overline{\ell}(\overline{y}))[\overline{t}(\overline{w},\overline{v})/\overline{y}]}_{\eta}\Big)$$

In the previous iteration k, since $\varphi_j^{[p,k-1]}(\overline{w}) \equiv \neg\exists\overline{v}(\psi)$, we have transformed ψ into a disjunction of answers for $\overline{w}\cdot\overline{v}$. At the current step $k+1$ we solve η, which gives another disjunction of answers for $\overline{w}\cdot\overline{v}$. Since

$$\varphi_j^{[p,=k]}(\overline{w}) \equiv \varphi_j^{[p,k]}(\overline{w}) \wedge \neg\varphi_j^{[p,k-1]}(\overline{w}) \equiv \neg\exists\overline{v}(\psi\wedge\eta)\wedge\exists\overline{v}(\psi)$$

Using conjunction and negation of answers, the variables $\overline{v}$ are eliminated and $\varphi_j^{[p,=k]}(\overline{w})$ is reduced to a disjunction of answers for $\overline{w}$. ■

Then, to compute $\mathcal{O}_{=k+1}(p(\overline{x}))$, the first member of each $\varphi_j^{[p,k]}(\overline{w})$ (that is $\varphi_j^{[p,k-1]}(\overline{w})$) 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 (just explained) 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. The procedural mechanism that computes a given goal, not only must obtain all its correct answers, but also must detect failure. In spite of the bottom-up nature of the answers 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, by means of the construction of a computation tree for an arbitrary given goal. 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 [20]. 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 answers

$$\bigvee_{i=1}^{m}\exists\overline{w}^i(a_i(\overline{x},\overline{w}^i))$$

We write $\underline{\mathsf{SolvedForm}}(c(\overline{x}))\equiv\underline{\mathtt{t}}$ for $m=1$ and $a_1\equiv\underline{\mathtt{t}}$, and $\underline{\mathsf{SolvedForm}}(c(\overline{x}))\equiv\underline{\mathtt{f}}$ for $m=0$.

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.

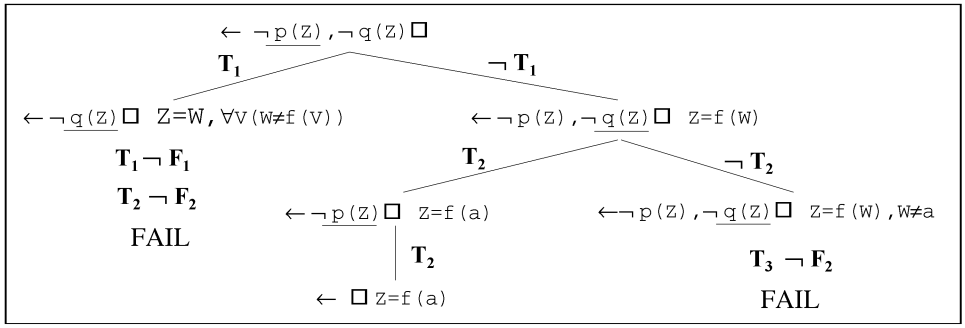

Figure 6: A finite BCN-computation tree

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* $\mathtt{Z = f(a)}$. *In fact,* $\forall \mathtt{Z}(\mathtt{Z = f(a)} \leftrightarrow (\neg\, \mathtt{p(Z)}, \neg\, \mathtt{q(Z)}))$ *is a logical consequence of program completion. Figure 6 shows a finite BCN-computation tree. The operators written in the edges of the tree of Figure 6 are applied to the literal that is just above marked as selected* [3]. *The first T-iteration for the 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 the constraint

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

and $\neg F_2(\neg\, \mathtt{q(Z)}) \equiv \mathtt{Z = f(a)}$ *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. In that case the answers for goals involving positive literals are obtained in a different order but, by completeness of the SLD-resolution (w.r.t. a fair selection rule), 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 there is no clause that can be applied to the selected literal. This happens whenever the conjunction of a with the constraint of any clause with head $p(\overline{x})$ is unsatisfiable. It is very 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.* ■

[3] Remember that $T_k(\neg\varphi) \equiv F_k(\varphi)$ and $F_k(\neg\varphi) \equiv T_k(\varphi)$.

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 $Comp(P) \models_3 (a' \rightarrow \overline{\ell} \wedge a)^{\forall}$.*

Proof. It is easy to check, by induction on the construction of computation trees, that

$$Comp(P) \models_3 (\bigvee_{i=1}^{r} (\overline{m}^i \wedge a_i) \leftrightarrow (\overline{m} \wedge a_0))^{\forall}$$

holds for any pre-computation tree such that its root is $\leftarrow \overline{m} \,\square\, a_0$ and $\{\leftarrow \overline{m}^i \,\square\, a_i \mid i \in 1..r\}$ $(r \geq 1)$ is the (finite) collection of all its leaves. In particular,

$$Comp(P) \models_3 ((\overline{m}^i \wedge a_i) \rightarrow (\overline{m} \wedge a_0))^{\forall}$$

holds for any $i \in 1..r$. The statement 2 easily follows from this fact by induction in the length of the derivation.

To prove the statement 1, let $\{\leftarrow \overline{\ell}^i \,\square\, a_i \mid i \in 1..r\}$ $(r \geq 1)$ be the (finite) collection of all leaves of the failure tree for G. Hence, $FET(\Sigma) \models (a_i \rightarrow F_{k_i}(\overline{\ell}^i))^{\forall}$ for each $i \in 1..r$ and some $k_i \in I\!N$. Then, by monotonicity of the operator F (see Proposition 1):

$$Comp(P) \models_3 ((\overline{\ell}^i \wedge a_i) \rightarrow (\overline{\ell}^i \wedge F_k(\overline{\ell}^i)))^{\forall}$$

where $k \equiv max\{k_i | i \in 1..r\}$. Therefore

$$Comp(P) \models_3 (\bigvee_{i=1}^{r} (\overline{\ell}^i \wedge a_i) \rightarrow \bigvee_{i=1}^{r} (\overline{\ell}^i \wedge F_k(\overline{\ell}^i)))^{\forall}$$

Additionally, by Theorem 1, we have that

$$Comp(P) \models_3 (F_k(\overline{\ell}^i) \rightarrow \neg\overline{\ell}^i)^\forall$$

where $F_k(\overline{\ell}^i)$ is a equality constraint, so it is a two-valued formula. Then,

$$Comp(P) \models_3 (\bigvee_{i=1}^{r} (\overline{\ell}^i \wedge a_i) \rightarrow \underline{\mathtt{f}})^\forall$$

Hence, $Comp(P) \models_3 ((\overline{\ell} \wedge a) \leftrightarrow \underline{\mathtt{f}})^\forall$,

$$Comp(P) \models_3 (a \rightarrow \neg\overline{\ell})^\forall \quad \blacksquare$$

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 $FET(\Sigma) \models (c \rightarrow \bigvee_i a_i)^\forall$.*

Proof. We first prove the statement 2. By Theorem 1, $FET(\Sigma) \models (c \rightarrow (T_k(\overline{\ell}) \wedge a))^\forall$ holds for some $k \in I\!N$. Since c is satisfiable, $(T_k(\overline{\ell}) \wedge a)$ must be satisfiable. Hence, it suffices to prove the following fact:

Fact 2. If $FET(\Sigma) \models (T_k(\overline{\ell}) \wedge a)^\exists$, then there exist $n > 0$ computed answers $a_1, \ldots, a_n$ for $\leftarrow\overline{\ell} \square a$ (w.r.t. P) such that $FET(\Sigma) \models ((T_k(\overline{\ell}) \wedge a) \rightarrow \bigvee_{i=1}^{n} a_i)^\forall$.

Suppose that $\overline{\ell} \equiv \ell_1, \ldots, \ell_m$. The proof is made by induction on m. The base case ($m \equiv 0$) trivially holds. For the induction step we assume, without loss of generality, that ℓ_1 is the selected literal. Since $FET(\Sigma) \models (T_k(\ell_1) \wedge a)^\exists$, then there exist $r \geq 1$ answers $a'_1, \ldots, a'_r$ such that

$$\underline{\mathsf{SolvedForm}}(a \wedge T_k(\ell_1)) \equiv \bigvee_{i=1}^{r} a'_i$$

Therefore, the goal G has r children of the form $\leftarrow\overline{\ell}' \square a'_i$ where $\overline{\ell}' \equiv \ell_2, \ldots, \ell_m$. Then

$$FET(\Sigma) \models ((T_k(\ell_1) \wedge a) \leftrightarrow \bigvee_{i=1}^{r} a'_i)^\forall$$

Since $FET(\Sigma) \models (T_k(\overline{\ell}') \wedge \bigvee_{i=1}^{r} a'_i)^\exists$, there exists a non-empty set $J \subseteq \{1, \ldots, m\}$ such that

$$FET(\Sigma) \models ((T_k(\overline{\ell}) \wedge a) \leftrightarrow \bigvee_{j \in J} (T_k(\overline{\ell}') \wedge a'_j))^\forall$$

and $FET(\Sigma) \models (T_k(\overline{\ell}') \wedge a'_j)^\exists$ holds for all $j \in J$. By the induction hypothesis, Fact 2 holds for each $(T_k(\overline{\ell}') \wedge a'_j)$ with $j \in J$. Then, there exist a set of computed answers $\{a^j_1, \ldots, a^j_{k_j} \mid k_j > 0\}$ for the goal $\leftarrow\overline{\ell}' \square a'_j$ such that

$$FET(\Sigma) \models ((T_k(\overline{\ell}') \wedge a'_j) \rightarrow \bigvee_{h=1}^{k_j} a^j_h)^\forall$$

holds for every $j \in J$. Therefore $\{\leftarrow \square a^j_h \mid j \in J,\ h \in 1..k_j\}$ is a non-empty collection of computed answers for $\leftarrow\overline{\ell} \square a$ such that

$$FET(\Sigma) \models ((T_k(\overline{\ell}) \wedge a) \rightarrow \bigvee_{j \in J} \bigvee_{h=1}^{k_j} a^j_h)^\forall.$$

Thus, the Fact 2 and the statement 2 of the Theorem hold.

Finally, we prove the statement 1. By Theorem 1, for some $k \in I\!N$: $FET(\Sigma) \models (a \rightarrow F_k(\overline{\ell}))^\forall$ Then, in the computation tree for G, there is no branch finished by a computed answer, since by the contrary this computed answer must be unsatisfiable by application of the Theorems 2 and 1. Now, let us suppose that the computation tree for G has an infinite branch that is formed by goals

$$\leftarrow\overline{\ell}^0 \square a_0,\ \leftarrow\overline{\ell}^1 \square a_1,\ \leftarrow\overline{\ell}^2 \square a_2,\ \ldots$$

where $\overline{\ell}^0 \equiv \overline{\ell}$, $a_0 \equiv a$ and each $\overline{\ell}^i \subseteq \overline{\ell}$ is non-empty. By construction of the tree, $FET(\Sigma) \models (a_{i+1} \rightarrow a_i)^\forall$ holds for all $i \in I\!N$. By coherence of the operators T and F (see Proposition 1), $FET(\Sigma) \models (a_i \rightarrow \neg T_n(\overline{\ell}))^\forall$ for all $i \in I\!N$. Therefore, there is some $i \in I\!N$ such that the constraint $a_i \wedge T_n(\ell)$ is unsatisfiable for any $\ell \in \overline{\ell}^i$ and any $n \in I\!N$. This means − by construction of the tree and fairness − that a_i should be successively strengthened (in the considered branch) with $\neg F_n(\ell_j)$ for every $\ell_j \in \overline{\ell}^i$ and increasing $n \in I\!N$. Hence, there exists some j such that

$$FET(\Sigma) \models (a_j \rightarrow (a_0 \wedge \neg F_k(\ell_{i_1}) \wedge \cdots \wedge \neg F_k(\ell_{i_{m_j}})))^\forall$$

where $\ell_{i_1}, \ldots, \ell_{i_{m_j}} \subseteq \overline{\ell}^j \subseteq \overline{\ell}$. This is a contradiction, since a_j should be unsatisfiable because $a_0 \equiv a$ and $FET(\Sigma) \models (a \rightarrow F_k(\overline{\ell}))^\forall$. As a result, the computation tree for G has neither a leaf with a computed answer, nor an infinite branch. Therefore, it must be a failure tree. $\blacksquare$

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 drackwards. 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 [6]. Consider the following program[4]:

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

With the delay technique, the goal $\leftarrow$ `p(X),¬ 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

[4]Of course, $f^{99}(a)$ denotes $\underbrace{f(\ldots(f}_{99}(a))\ldots)$

Program	Goal	Time (ms.)	# Answers

`p(a).`
`p(f(X,Y)) :- p(X), ¬ p(Y).`
`p(f(X,Y)) :- ¬ p(X), p(Y).`
	← ¬p(Z).	26
		130
		433
			100
			500
			1500

`even(0).` `even(s(X)):- ¬ even(X).`	← ¬even(Z).	151 1474 6532	25 50 75
`less(0,s(Y)).` `less(s(X),s(Y)) : −less(X,Y).`	← ¬less(Z, s^5(0)), less(Z, s^{15}(0)).	42	10
	← ¬less(Z, s^{10}(0)), less(Z, s^{100}(0)).	2079	90
	← ¬less(Z1,Z2).	21 42 78	50 75 100

`sum(0,X,X).`
`sum(s(X),Y,s(Z)):-`
`sum(X,Y,Z).`
	← ¬sum(Z1,Z2,Z3)	21
		42
		83
			50
			75
			100

`even_by_sum(X) : −sum(Y,Y,X).`	← ¬even_by_sum(Z).	229 2870 16468	10 20 30

`symmetric(a).`
`symmetric(g(X)):- symmetric(X).`
`symmetric(f(X,Y)):- mirror(X,Y).`
`mirror(a,a).`
`mirror(g(X),g(Y)):- mirror(X,Y).`
`mirror(f(X,Y),f(Z,W)):- mirror(X,W),`
`mirror(Y,Z).`
	← ¬symmetric(Z)	26
		130
		281
		433
		624
			100
			500
			1000
			1500
			2000

`member(X,[X|_]).` `member(X,[_|L]) : −member(X,L).`	← ¬disjoint(L1,L2).	16 130 990	100 500 1500

`disjoint([],_).`
`disjoint([E|EL],L) : −¬member(E,L),`
`disjoint(EL,L).`
	← ¬disjoint(L,[0]), maxlist(L, s^5(0)).	125
		672
		2677
			100
			500
			1500

`gt(0,X,X).`
`gt(X,0,X).`
`gt(s(X),s(Y),s(Z)) : −gt(X,Y,Z).`
	← ¬maxlist(L,s(_)).	3624
		3652
		3659
			100
			500
			1500

`maxlist([],0).` `maxlist([E|L],NE) : −maxlist(L,EAux), gt(E,EAux,NE).`	← ¬maxlist(L,Z).	3661 3687 3718	100 500 1500

Figure 7: Some experimental results

because the second iteration for the second literal gives the constraint ∃V (X = 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 ¬ 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 a prototype (`http://www.sc.ehu.es/jiwlucap/BCN.html`) in Sictus Prolog v.3.10.1 and the results obtained seem very promising. In Figure 7 you can find a table describing some 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 third column means the milliseconds of CPU-time to produce the number of answers specified by the immediate cell in the fourth column.

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 ([5]) and Barták ([2]) 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 [17] 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 [18], where the use of abstract interpretation is discussed in this context. Finally, quite recently we have learned about [19] 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. REFERENCES

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

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

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

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

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