Constructive negation without subsidiary trees

Edelmira Pasarella[1 2], Elvira Pino[2], and Fernando Orejas[2]

[1] Dpto de Computación y Tecnología de la Información, Universidad Simón Bolívar, Aptdo Postal 89000, Caracas 1089, Venezuela. edelmira@lsi.upc.es

[2] Dpto de L.S.I., Universidad Politècnica de Catalunya, Campus Nord, Mòdul C6, Jordi Girona 1-3, 08034 Barcelona, Spain. {pino,orejas}@lsi.upc.es

Abstract. In this paper we propose a new operational semantics, called BCN, which is sound and complete with respect to Clark-Kunen's completion for the unrestricted class of Normal Logic Programs. BCN is based on constructive negation and can be seen as an operational semantics for the class of Normal Constraint Logic Programs (NCLP) over the Herbrand universe. The main features of BCN making it a useful operational mechanism are twofold: First, BCN improves the existing proposals because it is more amenable to a practical implementation. The point is that, instead of computing subsidiary trees, the process of constructing answers for negative goals is reduced to a simple symbolic manipulation plus a constraint satisfaction checking process. Essentially, our approach exploits the definition of negative literals in the completion to interpret the constructive negation meta-rule. Second, the way in which BCN is defined makes it an extensible scheme to NCLP over arbitrary constraint domains.

1 Introduction

From a theoretical viewpoint, problems with negation related to the semantic definition of normal programs could be considered solved. On one hand, Clark-Kunen's completion [9] and well-founded semantics [15] provide two alternative approaches to define the (model-based) meaning of programs. On the other hand, the works of Kunen, Fages and, Lucio, Orejas and Pino [5, 11] provide fixpoint operators to compute its logical consequences (i.e. the set of correct answers). Finally, constructive negation, as introduced by Chan [1], Stuckey [14] and Drabent [3], is accepted as a complete and sound mechanisms to describe the operational semantics of the whole class of normal programs. However, from an implementation viewpoint, the proposed operational semantics for constructive negation (see [3, 5, 14] for a representative group of them) are difficult to implement in a practical way. In general, the core of the difficulties for developing practical implementations are the need of (complex) subsidiary computations each time a negative goal is selected during the computation and/or the need of computing disjunctive normal forms.

To be more concrete, most of the proposals share the following procedural interpretation of constructive negation: When a negative goal is selected in a given derivation the construction of a subsidiary derivation tree for its positive counterpart is activated. Then, this tree must be built until some required information is reached [3, 5, 14]. In addition, in some cases, this information must be manipulated to compute disjunctive normal forms to proceed with the original derivation.

In this paper we propose a new sound and complete operational semantics for normal logic programs based on constructive negation called BCN. The main feature of BCN is that, in contrast with the other operational proposals, it is neither based on constructing subsidiary trees for computing answers to negative queries nor on computing disjunctive normal forms in the derivation process. Instead, a new SLD-like direct derivation step is defined to implement the constructive negation meta-rule. Essentially, our method is based on using standard SLD-resolution for positive goals while, for the negative ones, an execution based on program completion is performed. That is, we exploit the definition of negative literals in the completion so that obtaining a procedural interpretation of the constructive negation meta-rule.

The theoretical foundations of this proposal come from a result of Shepherdson [13]. In that paper, Clark-Kunen's semantics based on the completion of programs, is characterized in terms of satisfaction of (equality) constraints. This result provides the ideas to define how to compute answers to negative queries. Being slightly more concrete, answers to a negative goal are computed by activating a simple symbolic manipulation process of its completion definition, plus a constraint satisfaction checking process.

Summarizing, we find two main advantages in our proposal: First, BCN is amenable to practical implementation because it only requires a new (symbolic manipulation) process to deal with negative goals whilst positive goals remain been dealt with as usual. Moreover, this approach enables us to enhance existing practical implementation of logic programming languages with constructive negation in a very modular way. Second, BCN provides an extensible scheme to NCLP over arbitrary constraint domains. Indeed, our proposal is actually an operational semantics for NCLP over the Herbrand universe but it is extensible to other ones by generalizing the results of Shepherdson [13] that we use in Section 3.

The rest of the paper is organized as follows. Section 2 contains some preliminary definitions required in this paper. Section 3 is devoted to introduce the BCN operational semantics and in Section 4 soundness and completeness of BCN with respect to Clark-Kunen's completion are proved. Some comments about the behavior of BCN and how to refine and implement it are given in Section 5. In Section 6, a comparison with other approaches is done. Finally, in Section 7 some conclusions and the ongoing work are presented.

2 Preliminaries

2.1 Notational Conventions

A *countable signature* Σ consists of a pair of sets (FS_Σ, PS_Σ) of function and predicate symbols, respectively, with some arity associated. Any term, atom or first order formula built by using functions and/or predicates from Σ and, also, variables from a fixed countable set X of variable symbols is called $\Sigma(X)$-term, Σ-atoms and Σ-formula, respectively. The Herbrand universe, denoted by H_Σ, is the set of all ground Σ-terms constructed by using Σ-functions. Terms are denoted by t, predicate symbols by p and function symbols by f and g, and the character ℓ is used for literals. The greek letter φ denotes a first order formula. When all the terms occurring in a formula are variables it is called a *flat* formula. $\varphi^\forall$ and $\varphi^\exists$ are the universal and existential closure of φ, respectively. The logical constants are respectively denoted by $\underline{\mathtt{t}}$ and $\underline{\mathtt{f}}$. The set of variables appearing in a term t (resp. formula φ) is denoted $var(t)$ (resp. $var(\varphi)$) and $FV(\varphi))$ denotes the set of free variables occurring in φ. The restriction $\varphi|_Y$ of a formula φ is the formula $\exists x_1, \ldots, \exists x_n \varphi$, where $\{x_1, \ldots, x_n\}$ is $FV(\varphi) \backslash Y$. In general, subscripts and superscripts will be used if needed and a bar is used to denote (finite) sequences of objects. Finally, instead of overloading the relational symbol '=', we use '$\equiv$' for syntactical equality.

2.2 Basic definitions

Throughout this work we assume the existence of an underlying signature Σ so, in general, for the sake of simplicity we do not write it explicitly unless an ambiguity can occur. Constructive negation has been formulated in terms of constraints logic programming (see Stuckey [14] and Drabent [3]). Herein we follow this approach so below we present some needed definitions.

The *free equality theory*, FET_Σ consists of the following axioms:

$\forall x(x = x)$	
$\forall\overline{x}\forall\overline{y}(\overline{x} = \overline{y} \leftrightarrow f(\overline{x}) = f(\overline{y}))$	for each $f \in FS_\Sigma$
$\forall\overline{x}\forall\overline{y}(\overline{x} = \overline{y} \rightarrow (p(\overline{x}) \leftrightarrow p(\overline{y})))$	for each $p \in PS_\Sigma \cup \{=\}$
$\forall\overline{x}\forall\overline{y}(f(\overline{x}) \neq g(\overline{y}))$	for each pair $f, g \in FS_\Sigma$ such that $f \not\equiv g$
$\forall x(x \neq t)$	for each Σ-term t and variable x such that $x \in var(t)$ and $x \not\equiv t$.

Whenever Σ has a finite number of function symbols it is necessary to add the (Weak) Domain Closure Axiom (WDCA) to FET_Σ in order to make it a complete theory.

$$\forall x(\bigvee_{f \in FS_\Sigma} \exists y_1, \ldots, y_{n_f}(x = f(y_1, \ldots, y_{n_f})))$$

An *equality constraint* is an arbitrary first order formula in which the only relational symbol occurring in atoms is the equality. Constraints are denoted by using the letters c and d (possibly with subscripts). Let c and d be constraints. The constraint c is *satisfiable* if and only if $FET_\Sigma \models c^\exists$, c is *more general* than d if and only if $FET_\Sigma \models (d \rightarrow c)^\forall$ and c and d are *equivalent* if and only if $FET_\Sigma \models (d \leftrightarrow c)^\forall$. A *constrained goal* is an expression of the form $\leftarrow\overline{\ell}\Box c$ where $\leftarrow\overline{\ell}$ is a normal flat goal (a conjuction of positive and negative flat literals) and c is a satisfiable constraint. The symbol $\Box$ is a syntactic variant of conjuction used to separate constraints out. Whenever c is $\underline{\mathtt{t}}$, it is omitted in goals. Henceforth when we refer to a goal we mean a constrained goal. In addition, given a goal $\leftarrow\overline{\ell}_1, \underline{\ell}, \overline{\ell}_2\Box c$, it means that the literal ℓ is selected in that goal. *Programs* are finite sets of normal clauses where the heads are flat atoms and the bodies are constrained goals, i.e. clauses are of the form $p(\overline{x}) :- \overline{\ell}\Box c$. Whenever in the body of a rule $\overline{\ell}$ is empty it represents the truth constant $\underline{\mathtt{t}}$. Also the set of all the definitions of predicate p in program P is defined as $def_P(p(\overline{x})) \equiv \{p(\overline{x}) :- \overline{\ell}^i\Box c^i \mid i = 1, \ldots, m\}$.

For giving a declarative semantics to a program P, Clark [2] proposed completing the definition of those predicates in P. *Clark's completion of* P, $Comp(P)$, consists of the free equality theory FET_Σ join with P^*, where P^* is the set of predicate completion formulas. The *predicate completion formula of predicate* $p \in PS_\Sigma$ *in* P is defined using $def_P(p(\overline{x}))$ as follows:

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

where $\overline{y}^i$ are the variables appearing in c^i and $\overline{\ell}^i$ which do not belong to $\overline{x}$. If $def_P(p(\overline{x})) \equiv \emptyset$ then $\forall\overline{x}(p(\overline{x}) \leftrightarrow \underline{\mathtt{f}})$ is included in P^*. $Comp(P)$ was originally interpreted in two-valued logic but Kunen [9] interpreted it in three-valued logic and hereafter it is considered the standard declarative meaning of normal logic programs, thus we assume three-valued completion.

3 The BCN operational semantics

In this section we introduce a new operational semantics to interpret constructive negation called BCN (**B**asic **C**onstructive **N**egation). As we pointed out in the introduction, the main feature of our proposal is that, in contrast with the usual existing resolution procedures [3, 5, 14], BCN is not based on constructing subsidiary trees for computing answers to negative queries. Instead, a new direct derivation step is defined to implement (at the high level) the constructive negation mechanism.

The idea behind the operational semantics we propose came from a reformulation, given by Drabent in [3], of Lemma 4.1 and Theorem 6 of Shepherdson in [13] in which Clark-Kunen's semantics based on the completion of programs is characterized in terms of satisfaction of constraints. Below we present a simplified version useful enough for our purposes.

Proposition 1. *Given a program P and a goal $\leftarrow\overline{\ell}\square c$:*

1. $Comp(P) \models_3 (F_n(\overline{\ell}) \rightarrow \neg\overline{\ell})^\forall$ *for every* $n \in \mathbb{N}$
2. *If* $Comp(P) \models_3 (c \rightarrow \neg\overline{\ell})^\forall$ *then* $FET_\Sigma \models (c \rightarrow F_n(\overline{\ell}))^\forall$ *for some* $n \in \mathbb{N}$
3. *If* $Comp(P) \models_3 (c \rightarrow \overline{\ell} \wedge d)^\forall$ *then* $FET_\Sigma \models (c \rightarrow T_n(\overline{\ell} \wedge d))^\forall$ *for some* $n \in \mathbb{N}$

where $T_n(\varphi)$ and $F_n(\varphi)$ are a couple of constraints defined for any formula φ without $\leftrightarrow$ and for any natural number n as follows:

(a) *If c is a constraint: $T_n(c) \equiv c$ and $F_n(c) \equiv \neg c$.*
(b) *For an atomic formula $p(\overline{x})$ such that $\forall\overline{x}(p(\overline{x}) \leftrightarrow \bigvee_{i=1}^{m} \exists\overline{y}^i(c^i \wedge \overline{\ell}^i)) \in P^*$:*

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

$$T_{n+1}(p(\overline{x})) \equiv \bigvee_{i=1}^{m} \exists\overline{y}^i(c^i \wedge T_n(\overline{\ell}^i))$$

$$F_{n+1}(p(\overline{x})) \equiv \bigwedge_{i=1}^{m} \forall\overline{y}^i(\neg c^i \vee F_n(\overline{\ell}^i))$$

(c) For non-atomic formulas:

$$\begin{array}{ll}
T_n(\neg\varphi) \equiv F_n(\varphi) & F_n(\neg\varphi) \equiv T_n(\varphi) \\
T_n(\varphi_1 \wedge \varphi_2) \equiv T_n(\varphi_1) \wedge T_n(\varphi_2) & F_n(\varphi_1 \wedge \varphi_2) \equiv F_n(\varphi_1) \vee F_n(\varphi_2) \\
T_n(\varphi_1 \vee \varphi_2) \equiv T_n(\varphi_1) \vee T_n(\varphi_2) & F_n(\varphi_1 \vee \varphi_2) \equiv F_n(\varphi_1) \wedge F_n(\varphi_2) \\
T_n(\varphi_1 \to \varphi_2) \equiv F_n(\varphi_1) \vee T_n(\varphi_2) & F_n(\varphi_1 \to \varphi_2) \equiv T_n(\varphi_1) \wedge F_n(\varphi_2) \\
T_n(\forall\overline{x}\varphi) \equiv \forall\overline{x}T_n(\varphi) & F_n(\forall\overline{x}\varphi) \equiv \exists\overline{x}F_n(\varphi)
\end{array}$$

In addition, due to some technicalities, we define the following. For all $n \in \mathbb{N}$*:* $T_n(\underline{\mathtt{t}}) \equiv F_n(\underline{\mathtt{f}}) \equiv \underline{\mathtt{t}}$ *and* $T_n(\underline{\mathtt{f}}) \equiv F_n(\underline{\mathtt{t}}) \equiv \underline{\mathtt{f}}$

The above proposition establishes the basic requirement for obtaining soundness and completeness with respect to Clark-Kunen's completion semantics. Statement 1 of Proposition 1 establishes that, given a goal $\leftarrow\overline{\ell}\square c$, it is sound to propose $F_n(\overline{\ell}) \wedge c, n > 0$, if satisfiable, as computed answer. Moreover, Statement 2 guarantees that this is enough for obtaining completeness. Then, we use the definition of F_n to compute failure answers but, in contrast, positive answers are computed as in SLD-resolution or, strictly speaking, as in the CLP-paradigm. Thereby, our proposal can be seen as an extension of SLD-like resolutions. As a consequence, it is useful for enhancing existing practical implementations of logic programming languages with constructive negation in a modular way. Notice that $F_n(\ell), n \geq 0$, are syntactical objects that can be obtained from ℓ by using its inductive definition, leaving the satisfaction-checking problem to a proper constraint solver procedure. Finally, Statement 3 is convenient in order to prove completeness.

Summing up the above ideas, we define the BCN operational semantics as follows: Let P be a program. A *BCN-derivation step* is obtained by applying one of the following derivation rules:

(R1) A goal $G' \equiv \leftarrow\overline{\ell}_1, \overline{\ell}, \overline{\ell}_2\square d'$ is *BCN-derived*$^+$ from a goal $G \equiv \leftarrow\overline{\ell}_1, \underline{p(\overline{x})}, \overline{\ell}_2\square c$ with respect to P, if there exists a (renamed apart) clause $p(\overline{x}) :- \overline{\ell}\square d$ in $def_P(p(\overline{x}))$ such that the constraint $d' \equiv c \wedge d$ is satisfiable.

(R2) A goal $G' \equiv \leftarrow\overline{\ell}_1, \overline{\ell}_2\square F_k(p(\overline{x})) \wedge c$ is *BCN-derived*$^-$ from a goal $G \equiv \leftarrow\overline{\ell}_1, \underline{\neg p(\overline{x})}, \overline{\ell}_2\square c$ with respect to P if there exists $k > 0$ such that the constraint $F_k(p(\overline{x})) \wedge c$ is satisfiable.

A *BCN-derivation* is a succession of BCN-derivation steps. Henceforth the notation $G \overset{n}{\rightsquigarrow} G'$ means that the goal G' is derived from G in n steps of derivation. A finite BCN-derivation $G \overset{n}{\rightsquigarrow} G'$ is a *successful BCN-derivation* if $G' \equiv \leftarrow\square c$ and $c|_{var(G)}$ is its *BCN-computed answer*. Finally, a goal $G \equiv \leftarrow\overline{\ell}\Box c$ is a *BCN-failed goal* if $FET_\Sigma \models (c \rightarrow F_k(\overline{\ell}))^\forall$ for some $k > 0$.

The first consequence of the above SLD-like definition is that, as SLD, BCN is independent of the selection rule. A *selection rule* is a function selecting a literal in a goal and BCN is independent of the selection rule used. As Lloyd in [10], we first prove the following switching lemma.

Lemma 1 (Switching lemma). *Let P be a program, G a goal and $G_0 \equiv G \overset{j-1}{\rightsquigarrow} G_{j-1}, G_j, G_{j+1} \overset{n-(j+1)}{\rightsquigarrow} G_n \equiv \leftarrow\Box c_n$ a successful BCN-derivation of G w.r.t P where $G_{j-1} \equiv \ell_1, \ldots, \ell_{i-1}, \ell_i, \ell_{i+1}, \ldots, \ell_{r-1}, \ell_r, \ell_{r+1}, \ldots, \ell_m \Box c$ and the selected literal in G_{j-1} is ℓ_i. Then there exists a successful BCN-derivation of G w.r.t P, $G_0 \equiv G \overset{j-1}{\rightsquigarrow} G_{j-1}, G'_j, G'_{j+1} \overset{n-(j+1)}{\rightsquigarrow} G_n \equiv \leftarrow\Box c'$ where the selected literal in G_{j-1} is ℓ_r and $FET_\Sigma \models (c \leftrightarrow c')^\forall$.*

We omit the proof of the above lemma which can be done by standard techniques (see, for instance, Lloyd [10]).

Theorem 1 (Independence of the selection rule). *Let P be a program, G a goal. Suppose there exists a successful BCN-derivation of G w.r.t P with computed answer c. Then using any selection rule R there exists a successful BCN-derivation with computed answer c_R such that $FET_\Sigma \models (c \leftrightarrow c_R)^\forall$.*

Proof. To apply the switching lemma repeatedly. $\Box$

4 Soundness and completeness

In this section we present the soundness and completeness results for the BCN operational semantics with respect to Clark-Kunen's semantics. Soundness proceeds in a very similar way to the corresponding proof for SLD-resolution, even when negative literals are selected.

Theorem 2 (Soundness of the BCN operational semantics). *Let P be a program and $G \equiv \leftarrow\overline{\ell}\Box c$ be a goal.*

1. *If G is a BCN-failed goal then $Comp(P) \models_3 (c \rightarrow \neg\overline{\ell})^\forall$*
2. *If there exists a successful BCN-derivation $\leftarrow\overline{\ell}\Box c \overset{n}{\rightsquigarrow} \leftarrow\Box c'$ then $Comp(P) \models_3 (c' \rightarrow \overline{\ell} \wedge c)^\forall$.*

Proof.

1. Suppose $G \equiv \leftarrow\overline{\ell}\Box c$ BCN-fails. This means that $FET_\Sigma \models (c \rightarrow F_k(\overline{\ell}))^\forall$, for some $k > 0$. Then, by Statement 1 of Proposition 1, we can conclude that $Comp(p(\overline{x})) \models_3 (c \rightarrow \neg\overline{\ell})^\forall$ $\Box$

We omit the proof of part 2 of soundness theorem which can be done by standard techniques (see, for instance, Lloyd [10]).

Now we prove completess of BCN operational semantics. For this purpose the definition of T_k provides a useful theoretical foundation for proving completeness of BCN-derivations with respect to Clark-Kunen's semantics.

Theorem 3 (Completeness of the BCN operational semantics). *Let P be a program and $G \equiv \leftarrow\overline{\ell}\Box d$ a goal and c a satisfiable constraint.*

1. *If $Comp(P) \models_3 (d \rightarrow \neg\overline{\ell})^\forall$, then G is a BCN-failed goal.*
2. *If there exists a constraint c such that $Comp(P) \models_3 (c \rightarrow \overline{\ell}\Box d)^\forall$ then there exist BCN-computed answers $c_1, \ldots, c_n$, $n > 0$ of $\leftarrow\overline{\ell}\Box d$ with respect to P, such that $FET_\Sigma \models (c \rightarrow \bigvee_{i\equiv1}^{n} c_i)^\forall$.*

Proof.

1. Directly from Statement 2 of Proposition 1. So, $FET_\Sigma \models (d \rightarrow F_k(\overline{\ell}))^\forall$ for some $k > 0$.
2. (Sketch) If $Comp(P) \models_3 (c \rightarrow \overline{\ell} \wedge d)^\forall$ then, by Statement 3 of Proposition 1, $FET_\Sigma \models T_k((c \rightarrow \overline{\ell} \wedge d)^\forall)$ for some k. At this point, using induction on k, it can be proved that for each $k \geq 0$, if $T_k(\overline{\ell} \wedge d)$ is satisfiable, then there exist BCN-computed answers $d_1, \ldots, d_m$, $m > 0$ of $\leftarrow\overline{\ell}\Box d$ with respect to P such that $FET_\Sigma \models (T_k(\overline{\ell} \wedge d) \rightarrow \bigvee_{i\equiv1}^{m} d_i)^\forall$. So, $FET_\Sigma \models (c \rightarrow \bigvee_{i\equiv1}^{m} d_i)^\forall$ $\Box$

5 Comments about refining BCN

In this section we slightly discuss some features of BCN that must be taken into account when implementing it. For this purpose we begin by introducing the notion of BCN-derivation tree.

5.1 BCN-derivation tree

Let P be a program and G a goal. A *BCN-computation tree* of G with respect to P is a tree that satisfies the following properties:

(a) Each node is a goal and the root is G.

(b) If $\leftarrow\overline{\ell}_1, \underline{\ell}, \overline{\ell}_2 \square c$ is an internal node and ℓ is a positive (resp. negative) literal, its children are all those goals BCN-derived$^+$ (resp. BCN-derived$^-$) from it.

In a BCN-computation tree each path is a BCN-derivation. In addition in such trees can occur three kind of distinguished nodes:

1. A *successful node* is a leave of the form $\leftarrow\square c$ corresponding to the last goal in a successful derivation.
2. A *failure node* is a leave of the form $\leftarrow\overline{\ell}_1, \underline{p(\overline{x})}, \overline{\ell}_2 \square c$ corresponding to a failed BCN-goal.
3. An *incessant node* is a leave of the form $\leftarrow\overline{\ell}_1, \underline{\neg p(\overline{x})}, \overline{\ell}_2 \square c$ corresponding to a BCN-derivation suspended forever.

The notion of incessant node require more explanation. Let us see the following example:

Example 1. Consider the program $P \equiv \{p(x) :- \neg p(x), \neg r(x)\}$. Then, the BCN-computation tree of $\leftarrow\neg p(\overline{x})$ with respect to P is

$$\leftarrow\underline{\neg p(\overline{x})}$$

That above node does not have any child because $F_n(p(\overline{x}))$ is unsatisfiable for all $n > 0$ so the derivation remains as an endless embedded computation looking for some satisfiable constraint. This is the reason why this kind of nodes and derivations are called incessant nodes.

Since BCN is complete with respect to the Clark-Kunen's semantics we can state that this kind of derivation corresponds to an infinite computation given by the semantics of the considered program such as occurs in the program $\{p(\overline{x}) :- p(\overline{x})\}$.

Apart from that kind of infinite computation, another kind of infiniteness can occur in BCN-derivation trees. Whenever a negative literal $\neg p(\overline{x})$ is selected in a node of a BCN-computation tree if there are computed answers for it, a *breadth infinite* subtree will be obtained. In order to analyze this behavior it is necessary to introduce a property of constraints $F_n(p(\overline{x})),\ n \geq 0$.

Proposition 2 (Monotonicity of F_n). *For all $n \geq 0$ and all flat literal $\ell(\overline{x})$*

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

The proof of the above proposition follows by induction on n. Now, let us proceed considering the next two examples.

Example 2. Consider the program $P_1 \equiv \{even(x) :- \Box x = 0;\ even(x) :- \neg even(y) \Box x = suc(y)\}$ and the goal $G_1 \equiv \leftarrow \neg even(x)$ with respect to P_1.

Example 3. Consider the program $P_2 \equiv \{p(x) :- \Box x = a\}$ and the goal $G_2 \equiv \leftarrow \neg p(\overline{x})$ with respect to P_2.

In both cases one obtains breadth infinite BCN-computation trees for G_i with respect to P_i, $i \in \{1, 2\}$. In the first example this corresponds to the number of possible answers but in the second one, there is only one answer, $x \neq a$. In general one of the problems is that (R2) does not determine when a "new" answer is equivalent to the last computed one. For instance, in Example 2 , for all $i > 0 :\ F_{2*i}(even(x))$ is equivalent to $F_{2*i+1}(even(x))$, thus in the tree each answer occurs twice. From an implementation point of view this behavior could be undesirable. Luckily, thanks to the monotonicity of F_k, this process of generation of answers can be controlled by refining the rule (R2) as follows:
(R2') Departing from a goal $G \equiv \leftarrow \overline{\ell}_1, \neg p(\overline{x}), \overline{\ell}_1 \Box c$, two possible derivation steps can be done. Let $k \equiv min\{n \in \mathbb{N} \mid c \wedge F_k(p(\overline{x}))$ is satisfiable$\}$.

(a) $\leftarrow \overline{\ell}_1, \overline{\ell}_2 \Box c \wedge F_k(p(\overline{x}))$
(b) $\leftarrow \overline{\ell}_1, \neg p(\overline{x}), \overline{\ell}_2 \Box d$, being $d \equiv c \wedge \neg F_k(p(\overline{x}))$ and satisfiable.

Using this rule in the process of BCN-computation tree generation, implies that each node with a negative literal selected has, at most, two children: those generated by (R2')(a) and (R2')(b). Moreover, the rule (R2') provides a mechanism to obtain answers to negative queries one by one. It can be proved that with this refinement soundness and completeness are preserved but rare branches are added to the tree. Then, the righ refinement in order to eliminate these branches is given by using $d' \equiv d \wedge \neg T_k(p(\overline{x}))$ instead of constraint d in (R2'). Furthermore, by this way it is possible to stop the answer generation process when there is no new ones, such is the case of Example 3. Accordingly, (R2') abolishes the breadth infiniteness from the BCN-computation trees.

5.2 Implementation issues

In the following we give some ideas about how to implement BCN in a practical way. Since there are well-known implementations of definite constraint logic programming languages we rather focus the implementation of the negative part of BCN. In this sense, most of the computations lie on the evaluation of the functions F_k and T_k, so, we propose an implementation scheme to maintain the necessary information so that it would be possible to take advantage of previous evaluations of these functions.

Let $\Sigma \equiv (PS_\Sigma, FS_\Sigma)$ be the signature of the program P. For each $p \in PS_\Sigma$ we define the *last computed constraint tuple*, $\mathcal{C}_P(p) \equiv \langle T, k_T, F, k_F \rangle$ where $k_T, k_F \geq 0$ and, T and F correspond to $T_{k_T}(p(\overline{x}))$ and $F_{k_F}(p(\overline{x}))$, respectively. The initial state of $\mathcal{C}_P(p)$ is $\langle \underline{\mathtt{f}}, 0, \underline{\mathtt{f}}, 0 \rangle$ corresponding to the evaluation of $T_0(p(\overline{x})$ and $F_0(p(\overline{x})$. Going into details, in a practical implementation the information corresponding to the tuple $\mathcal{C}_P(A)$ can be maintained with the usual information of A in the symbol table (predicate descriptor).

Notice that whenever $F_k(A)$ is required it could be returned $F_{k'}(A)$ being $k' > k$ instead of it. This decision is supported by the monotonicity property since it guarantees soundness and completeness of this refinement with respect to Clark-Kunen's semantics.

6 Comparison with other approaches

In this section we relate our proposal to some of the most well-known ones in the area. A common factor of BCN as compared with other operational semantics is that it looks simpler than them. The fact that BCN is SLD-like and no disjunctive normal form has to be computed gives rise to some simplifications in implementing it. Indeed, in spite of the various proposals to define an operational semantics for constructive negation, to our knowledge, none of them have improved in a substantial way the other ones from an implementation viewpoint. To be more concrete, most of the proposals share the following procedural interpretation of constructive negation: when a negative goal is selected in a given derivation the construction of a subsidiary derivation tree for its positive counterpart is activated. Then, this tree must be built until some required information for proceeding with the original derivation is reached. For instance, in Drabent's approach [3], besides other significant features, we can say that this information must provide a constraint to prune the tree so that it becomes a finitely failed tree and, in this way, obtaining a failure answer. The proposal of Fages [5] is a bit more involved to describe since two trees are concurrently constructed, but, essentially, the same kind of information is required. That is, a failure answer to proceed with the original derivation. In the case of Stuckey's proposal [14], the subsidiary tree is truncated for obtaining some information required to build a disjunctive normal first order formula which is (logically) equivalent to the original goal. Then, one can use each of the components of the disjunction to proceed with the original derivation. In the rest of this section we compare more precisely BCN with the three above-mentioned approaches.

The resolution procedure proposed by Drabent, SLDFA-resolution, generalizes the (SLDNF) standard notion of finitely failed trees. In an SLDFA-finitely failed tree of rank k the children of an internal node $\leftarrow\overline{\ell}_1, \underline{\neg p(\overline{x})}, \overline{\ell}_2 \square c$ are the goals $\leftarrow\overline{\ell}_1, \overline{\ell}_2 \square d_1; \ldots; \leftarrow\overline{\ell}_1, \overline{\ell}_2 \square d_m, \quad m \geq 0$, whenever there exist $c_1, \ldots, c_n, \quad n \geq 0$, which are some SLDFA-computed answers for $\leftarrow p(\overline{x}) \square c$ of rank $< k$ and $FET_\Sigma \models (c \rightarrow c_1 \vee \ldots \vee c_n \vee d_1 \vee \ldots \vee d_m)^\forall$. Notice that the process of computing an appropriate $c_1 \vee \ldots \vee c_n \vee d_1 \vee \ldots \vee d_m$ (covering) of c is indeterministic and to implement it, it is necessary to provide some re-computation process of the covering in order to reach the corresponding SLDFA-finitely failed. Moreover, a disjunctive normal form has to be computed.

Stuckey's and Fages's proposals [14, 5] have in common the use of frontiers in the derivation trees defined by each of them, respectively. In general, a *frontier* in a derivation tree is a finite set of nodes in the tree, excluding the root, such that every derivation in the tree is either finitely failed or passes through exactly one node of the frontier. Relating this notion with our proposal, intuitively one can see that what the computation of a $F_k(p(\overline{x}))$ for some $k > 0$ does is to consider a frontier of the (non-constructed) BCN-computation tree of the goal $\leftarrow p(\overline{x})$ in which the deepest node in the frontier is at level k, hence, in some way, the computation of all the $F_k(p(\overline{x}))$, $k > 0$ corresponds to consider the answers given by selecting some frontiers as Stuckey's method does.

Stuckey's method generalizes to NCLP programs the constructive negation for logic programs introduced by Chan [1]. In this approach finitely failed trees are not defined, instead, this method is based on selecting frontiers of derivation trees to compute answers for negative goal so that the selection rule is extended to choose frontiers too. Summarizing this method, when a negative literal $\neg p(\overline{x})$ is selected in a goal G, the construction of a (partial) derivation tree for $p(\overline{x})$ is activated. When the frontier, F, chosen by the selection rule is available in the tree, its construction is stopped and the disjunctive normal form associated to the negated disjunction of nodes in F is computed. Then, for each component of that disjunctive form the goal G has a child.

In implementing Stuckey's scheme there are some hard problems to be addressed. Firstly the need of computing a disjunctive normal form each time that a negative derivation step has to be done (as in Drabent's method). Also, the implementation of the selection rule is overloaded since it is in charge of selecting frontiers. Moreover the manipulation of frontiers by itself complicates any implementation. Besides, when the chosen frontier is not a singleton a proliferation of subgoals can occur in the original derivation tree depending on the resulting normal disjunctive form obtained. In contrast, the BCN operational semantics's derivation rules are defined for standard goals without the need of considering more explicitly quantified formulas than the corresponding ones to the constraints. In the BCN operational semantics, a negative derivation step activates a symbolic manipulation process whose result is yet a constraint, we mean an answer for the negative literal selected, therefore there is no proliferation of goals in the considered BCN-computation tree.

Fages proposes an scheme called constructive negation by pruning for NCLP programs. This approach relies on a frontier calculus which can be seen, at a low abstraction level, as a pruning mechanism over standard SLD-derivation trees.

The idea is that given the goal $G \equiv \leftarrow \overline{\ell}_1, \underline{\neg p(\overline{x})}, \overline{\ell}_2 \square c$ two SLD-derivation trees are constructed concurrently, T_1, for the goal G but selecting another literal and T_2 for $\leftarrow p(\overline{x}) \square c$. When a successful derivation is found in T_2, T_1 is pruned with the negation of the computed answer obtained from the successful derivation in T_2.

In implementing this scheme we find two main problems. First, the problems of constructing derivation trees have to be dealt with twice. Second, in presence of nesting of negations, actually, it would be necessary to maintain more than two derivation trees and, in fact, the propagation of the pruning process would not be a trivial process. Even considering a powerful capacity of concurrency this problem remains.

To finish this section we remark the fact that the independence of the selection rule of BCN, in contrast with the other mentioned proposals, eliminates one source of incompleteness in implementing it without imposing any restriction on the class of programs or literals to be selected. Indeed, the requirement of a fair selection rule in SLDNF-resolution implies, in general, the need of having a breadth-first strategy to select literals and this could imply a less efficient implementation. SLDFA-resolution and Stuckey's method would have this problem since they also have the fairness requirement for the selection function. Finally we have to say that in the BCN operational semantics, as in SLD-resolution's implementations, still remains the possibility of incompleteness associated to the selection of the clause to be used for positive steps of derivations.

7 Conclusions and ongoing work

We have developed an operational semantics for NCLP over the Herbrand universe. The main contribution herein is that it provides an scheme of constructive negation amenable for practical implementations. In effect, since the similarity of the BCN operational semantics with SLD-resolution and the successful implementations of CLP languages, concretely of constraint solvers [6, 7] we believe many of the features of implementing the BCN operational semantics are overcome yet so the responsibility of the success is almost transferred to the process of computing the F_k, $k > 0$.

In conclusion, we found the following main advantages in our proposal. First, BCN is amenable to practical implementation since it is not based on a lazy unfolding of completion formulas [4, 12] that would require complex procedures for manipulating them. Indeed, only formulas as complex as the completion ones need to be manipulated. Moreover, the constraint satisfaction checking process can be left to a proper constraint solver procedure, and, being BCN close to the logical semantics based on the completion and SLD-like, this makes BCN easy to implement. Second, BCN provides a method for enhancing existing practical implementations of logic programming languages with constructive negation in a very modular way. Third, BCN provides an extensible scheme to NCLP over arbitrary constraint domains. Indeed, it could be an expected extension since constructive negation as presented by Stuckey [14] was thought to fit with NCLP. From this viewpoint, our proposal is actually an operational semantics for NCLP over the Herbrand universe but it is extensible to other ones by generalizing the results of Shepherdson [13] that we use in Section 3.

Regarding the related ongoing work, on one hand, such as we said above, the BCN operational semantics is extensible to others domains. The details of this assertion are been addressed in an incoming work but the idea consists on using the scheme proposed by Jaffar, Maher, Marriot and Stuckey [8]. Thus, we think the first step in extending BCN is to generalize its theoretical basis, that is, the Proposition 1, to any domain constraint $\mathcal{X}$. On the other hand, we are working on refining BCN in order to detect and study those features that could be useful in developing a practical implementation.

Acknowledgments We are grateful to Paqui Lucio for her valuable comments and discussions on the topic of this paper. This work was supported by the venezuelan program *Desarrollo de Investigadores* (CONICIT/Universidad Simón Bolívar), the spanish project HEMOSS (CICYT ref. TIC98-0949-C02-01) and *Grup de Recerca Consolidat* (CIRIT 1997SGR-00051).

References

1. D. Chan. Constructive negation based on the completed database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conf. and Symp. on Logic Programmming*. The MIT Press, 1988.
2. K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press. New York, 1978.
3. W. Drabent. What is a failure? An approach to constructive negation. *Acta Informática*, 32:27–59, 1995.
4. S. Etalle and F. Teusink. A compositional semantics for normal open programs. In *Proc. Int. Conf. and Symp. on Logic Programming'96*. The MIT Press, 1996.
5. F. Fages. Constructive negation by pruning. *Journal of Logic Programming*, 32:85–118, 1997.
6. T. Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, (37):95–138, 1998.
7. J. Jaffar and M. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, (19/20):503–581, 1994.
8. J. Jaffar, M. Maher, K. Marriot, and P. Stukey. The semantics of constraint logic programs. *The Journal of Logic Programming*, (37):1–46, 1998.
9. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.
10. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
11. 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.
12. T. Sato and F. Motoyoshi. A complete top-down interpreter for first order programs. In *Proceedings of International Logic Programming Simposium*, pages 35–53, 1991.
13. J.C. Shepherdson. Language and equality theory in logic programming. Technical Report PM-91-02, University of Bristol, 1991.
14. P. J. Stuckey. Negation and constraint logic programmming. *Information and Computation*, 118:12–23, 1995.
15. van Gelder, A., Ross K., and Schlipf J. The well-founded semantics for general logic programs. *Journal of ACM*, 38:620–650, 1991.