An Implementation of Constructive Negation [⋆]

J. Álvez[1], P. Lucio[1], F. Orejas[2], E. Pasarella[23], and E. Pino[2]

[1] Dpto de L.S.I., Facultad de Informática, Universidad del País Vasco, Paseo Manuel de Lardizabal, 1, Apdo 649, 20080-San Sebastián, Spain.
[2] Dpto de L.S.I., Universidad Politécnica de Catalunya, Campus Nord, Modul C6, Jordi Girona 1-3, 08034 Barcelona, Spain. {orejas, edelmira, pino}@lsi.upc.es
[3] Dpto de Computación y Tecnología de la Información , Universidad Simón Bolívar, Aptdo Postal 89000, Caracas 1080, Venezuela. edelmira@ldc.usb.ve

Abstract. In this paper, we present a new procedural interpretation for constructive negation which is sound and complete with respect to three-valued program completion. Its main features are twofold: first, it gives a uniform treatment to positive and negative literals in goals; second, it provides an incremental way to detect failure. This mechanism is a refinement of an operational semantics that does not require subsidiary computation trees to compute answers for negative goals. Instead, such answers are built by solving equality constraints which are directly obtained from predicate completion definitions. The constraints generated during derivations constitute a particular subclass of general equality constraints. We provide an implementation based on a specialized solver for this class of constraints.
Keywords: constructive negation, operational semantics, implementation, equality constraint, constraint solver.

1 Introduction

Constructive negation was introduced in [2]. It was extended, in [3, 7], to a complete and sound operational semantics for the whole class of normal logic programs. In [8, 19] these results were generalized to the framework of Constraint Logic Programming (CLP for short, see [10] for a survey). However, from a practical implementation viewpoint, these operational semantics proposals lack an easy procedural interpretation. The main reason is that they are highly non-deterministic in aspects that are crucial for practical implementation purposes. More concretely, each time a negative literal $\neg A$ is selected in a derivation, a subsidiary computation tree for its positive counterpart A is activated in order to obtain some information about $\neg A$ that can be used to proceed with the original derivation. Non-determinism appears − depending on the approach − either in the way for obtaining such information or in how it affects the original derivation tree.
Drabent's proposal [7] requires a non-deterministically defined set S of answers for A, that could allow calculating another set S', such that $S \cup S'$ becomes a covering for the constraint of the original derivation. The answers in S' (if there exists any) are used to follow with the original computation.
Chan (in [3]) and Stuckey (in [19]) use frontiers of the subsidiary computation tree for $\leftarrow A \,\square\, c$. Frontiers are finite set of nodes − which are goals − with exactly one goal belonging to each non-failed branch of the computation tree. The selection rule is in charge of the non-deterministic choice of a frontier in the subsidiary tree. Then the negated disjunction of nodes of this frontier must be manipulated to continue with the original computation. Besides, the formula obtained by this manipulation makes necessary to extend the proposed mechanism to the so-called *complex goals*, which extends the usual notion of goal with explicit quantification.

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

Regarding Fages's proposal [8], the main and the subsidiary tree are concurrently computed. When one answer is obtained, in the subsidiary tree, its negation is used to prune the main tree. In both − the main and the subsidiary tree − could appear new goals with negative literals. That produces a proliferation of concurrently activated computation trees, each one affecting − by pruning − the derivations of some other in a non-deterministic way.
To the best of our knowledge there are no working implementation of constructive negation. We have recently known the preliminary implementation [15] of Muñoz and Moreno-Navarro. It is based on frontiers of subsidiary trees, but it is operationally different from any of the above-mentioned proposals. Hence, its correctness and completeness can not be deduced from the previous results.
We propose a new interpretation of the constructive negation meta-rule that does not require subsidiary computation trees. Conversely, the idea is to construct the answers of a goal by solving equality constraints. Goal computation follows the usual top-down CLP scheme of collecting the answers for the selected literal into the constraint of the goal. The main difference lies on the way for obtaining the answers for the selected literal. For this, we use constraint solving instead of the generation of a subsidiary tree. The answers for a (selected) literal are deterministically obtained in a bottom-up incremental manner. A preliminary work along the same lines was presented in [17]. The theoretical foundations of our approach comes from a result of Shepherdson [18] characterizing Clark-Kunen's completion semantics in terms of satisfaction of equality constraints, which are generated by two mutually recursive operators. Shepherdson's operators (Definition 1) are similar, but not identical, to immediate consequence operators. For that reason one could suspect that our computations should necessarily be infinite. However, this is not the case, as Example 14 shows.
In this paper, we first introduce an operational semantics based on the Shepherdson's characterization. The soundness and completeness of this operational semantics are a consequence of the results in [18]. Then, we refine it to a purely procedural mechanism and we prove the soundness and completeness of this refinement. Our formulation provides a uniform treatment for positive and negative literals and a new incremental way to detect failure. Since it is a purely procedural mechanism, it is amenable for − and very close to − practical implementation. In fact, we have implemented a prototype which makes use of an equality constraint solver that takes advantage of the particular form of the constraints generated during derivations. We have obtained some promising experimental results. We also describe this implementation.

Outline of the paper. Section 2 contains preliminary definitions and notation. Section 3 is devoted to the operational semantics, its foundations and the main related results. We also give examples to explain the main problems for refining it into a purely procedural mechanism. In section 4 we present our refinement for procedural interpretation. The main ideas behind it are explained and we give examples to illustrate its behavior. We also prove its soundness and completeness. Section 5 discusses implementation issues, in particular specialized equality constraint solving and incremental calculation of answers.

2 Preliminaries

We will 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 $\forall, \exists$.
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*. We say that an equation or disequation is *flat* whenever (at least) one of its terms is a variable. If $\overline{t}$ and $\overline{t}'$ are n-tuples of terms then $\overline{t} = \overline{t}'$ abbreviates $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$ and $\overline{t} \neq \overline{t}'$ abbreviates $t_1 \neq t'_1 \vee \ldots \vee t_n \neq t'_n$. 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. To avoid confusion, we will use the symbol $\equiv$ for the metalanguage equality.
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 $mgu(\overline{s})$, 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'.
A *basic constraint*, denoted by $b(\overline{x}, \overline{w})$, is a conjunction of flat 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 *constrained goal* is an expression of the form $\leftarrow \overline{\ell} \,\Box\, c$ where $\overline{\ell}$ is a conjunction of positive and negative flat literals and c is a equality constraint. A *basic constrained goal* is a constrained goal $\leftarrow \overline{\ell} \,\Box\, b$ where b is a basic constraint. As usual in CLP, the symbols comma (,) and box ($\Box$) 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 $p(\overline{x}) :- \overline{\ell}(\overline{y}) \,\Box\, b(\overline{x}\cdot\overline{y}, \overline{w})$ where the flat atom $p(\overline{x})$ is called its *head*, the basic constrained goal $:- \overline{\ell}(\overline{y}) \,\Box\, b(\overline{x}\cdot\overline{y}, \overline{w})$ is called its *body*, and the three 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. It is easy to see that every normal logic program (respect. normal goal) can be rewritten as one of our programs (respect. basic constrained goal). For example, the program given by the two clauses: $even(0)$. and $even(s(x)) :- \neg\, even(x)$., is rewritten as

$even(x) :- \Box\, x = 0.$
$even(x) :- \neg\, even(y) \,\Box\, x = s(w), y = w.$

Given a program P and a predicate symbol p, the set $def_P(p(\overline{x}))$ consists of all clauses in P with head predicate p, that is *the definition of p in P*. For simplicity, we assume that all clauses with the same head predicate (namely p) use the same head variables (namely $\overline{x}$), but the other variables (in bodies) are assumed to be distinct in different clauses.
To define the semantics of a Σ-program P, Clark [4] proposed completing the definition of the predicates in P. The *predicate completion formula of predicate $p \in PS_\Sigma$ (w.r.t. P)* such that $def_P(p(\overline{x})) \equiv \{p(\overline{x}) :- \overline{\ell}^i(\overline{y}^i) \,\Box\, b^i(\overline{x}\cdot\overline{y}^i, \overline{w}^i) \mid i \in 1..m\}$ for some $m \geq 0$ is

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

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* [4] $FET(\Sigma)$ 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. [11]).
The theory FET(Σ) axiomatizes the usual Herbrand FS_Σ-universe. Its decidability, for finite and infinite signatures, has been proved by different methods (cf. [6, 11, 13, 14, 16, 18]). The decidability problem for $FET(\Sigma)$ is (in the worst case) a non-elementary problem (cf. [20]), that means it does not belong to the class $NTIME(n^m)$ for any $m \in \omega$. In a strict sense, a decision method for a Σ-theory $\mathcal{T}$ is an algorithm that, for any Σ-sentence φ as input, it gives the answer "yes" if $\mathcal{T} \models \varphi$ and the answer "no" if $\mathcal{T} \models \neg\varphi$. However, constraints usually are not sentences, since they have free variables. Goals are questions about their free variables, asking for all their possible values. Hence, solving a constraint consists in finding the collection of answers for their free variables, which is given by a logically equivalent constraint that fits some so-called *solved form.* An equality constraint solver transforms any equality constraint – possibly with free variables – into its solved form. The main feature of a solved form is that it is easy to decide its satisfiability. As a consequence, in CLP, solved forms are feasible computed answers and they determine the syntactical form of the constraints along derivations. The CLP goal-derivation process has to combine the solved forms – being computed answers for the selected literal – with the basic constraints inside the body of program clauses – which have also a prefixed form – to produce a new equality constraint for the solver. Therefore, the equality constraint solver can be specialized to a particular class of equality constraints, that could be more efficiently solved than general equality constraints.

3 BCN Operational Semantics

In this section, we define the BCN operational semantics, introduced in [17], that underlies our procedural approach and the main related results. We also explain what are the main problems for a direct procedural interpretation.
The BCN operational semantics is based on the following operators T_n and F_n which associate an equality constraint to each conjunction of literals.

Definition 1. The operators T_n and F_n 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

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

$$T_0(p(\overline{x})) \equiv \underline{\mathtt{f}} \qquad T_{n+1}(p(\overline{x})) \equiv \bigvee_{i=1}^{m} \exists\overline{y}^i\exists\overline{w}^i(b^i(\overline{xy}^i, \overline{w}^i) \wedge T_n(\overline{\ell}^i(\overline{y}^i)))$$

$$F_0(p(\overline{x})) \equiv \underline{\mathtt{f}} \qquad F_{n+1}(p(\overline{x})) \equiv \bigwedge_{i=1}^{m} \neg\exists\overline{y}^i\exists\overline{w}^i(b^i(\overline{xy}^i, \overline{w}^i) \wedge \neg F_n(\overline{\ell}^i(\overline{y}^i)))$$

- For any $n \in I\!N$:

$$T_n(\overline{\ell}^1 \wedge \overline{\ell}^2) \equiv T_n(\overline{\ell}^1) \wedge T_n(\overline{\ell}^2) \qquad T_n(\neg p(\overline{x})) \equiv F_n(p(\overline{x})) \qquad T_n(\underline{\mathtt{t}}) \equiv \underline{\mathtt{t}}$$
$$F_n(\overline{\ell}^1 \wedge \overline{\ell}^2) \equiv F_n(\overline{\ell}^1) \vee F_n(\overline{\ell}^2) \qquad F_n(\neg p(\overline{x})) \equiv T_n(p(\overline{x})) \qquad F_n(\underline{\mathtt{t}}) \equiv \underline{\mathtt{f}} \quad \blacksquare$$

These operators were introduced by Shepherdson ([18]) in order to characterize the (three-valued) logical consequences of $Comp(P)$. An easy consequence of that characterization, enough for our purposes, is the following result:

[4] also known as *Clark's equational theory* (cf.[4]) or the *first-order theory of finite trees.*

Theorem 2. *Let P be a Σ-program, $\overline{\ell}$ a conjunction of literals and c, d constraints, then the following two facts hold:*
(i) $Comp(P) \models_3 (c \rightarrow (\overline{\ell} \wedge d))^{\forall}$ *iff* $FET(\Sigma) \models (c \rightarrow (T_k(\overline{\ell}) \wedge d))^{\forall}$ *for some* $k \in I\!N$
(i) $Comp(P) \models_3 (c \rightarrow (\neg\overline{\ell} \vee d))^{\forall}$ *iff* $FET(\Sigma) \models (c \rightarrow (F_k(\overline{\ell}) \vee d))^{\forall}$ *for some* $k \in I\!N$

Proof. It is simple consequence of Theorem 6 and Lemma 4.1 of [18], which is also reformulated in Lemma 5.2 of [7]. ■

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 $k \in I\!N$. 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. [1, 9, 11, 12, 19]). 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 3. (Monotonicity and Coherence of the operators T and F) *Let P be Σ-program and $\ell(\overline{x})$ a flat literal, then for all $n \in I\!N$:*
(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* $k \in I\!N$
(iv) $FET(\Sigma) \models (F_n(\ell(\overline{x})) \rightarrow \neg T_k(\ell(\overline{x})))^{\forall}$ *for all* $k \in I\!N$

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

The above Theorem 2 establishes the basic requirements for obtaining soundness and completeness with respect to Clark-Kunen's completion semantics of the BCN operational semantics that we are going to define next. We want to point out that this semantics is as far from implementation than close to logical semantics, but it is introduced firstly as a good vehicle for the presentation of our procedural mechanism. In fact, the later will be introduced as a refinement that avoids the computational problems of the BCN operational semantics.

Definition 4. Let P be a Σ-program. A *BCN-derivation step* is obtained by applying the following derivation rule:

(R) A goal $G' \equiv \leftarrow \overline{\ell}^1, \overline{\ell}^2 \,\square\, d$ is *BCN-derived* from a goal $G \equiv \leftarrow \overline{\ell}^1, \underline{\ell(\overline{x})}, \overline{\ell}^2 \,\square\, c$ with respect to P if there exists $k > 0$ such that the equality constraint $d \equiv \overline{T_k(\ell(\overline{x}))} \wedge c$ is satisfiable in $FET(\Sigma)$.

A *BCN-derivation* $G \overset{n}{\rightsquigarrow} G'$ is a succession of n BCN-derivation steps. ■

Remark 5. The rule **(R)**, and also the resulting operational semantics, treats positive and negative literals in exactly the same way. We decided to do that in this paper to simplify the technical details of the presentation. From the implementation point of view, it would be more convenient to treat positive literals in the standard way of SLD-resolution. That means to split the above rule **(R)** into the following two 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 equality constraint $d' \equiv c \wedge d$ is satisfiable in $FET(\Sigma)$.

(R2) A goal $G' \equiv \leftarrow \overline{\ell}^1, \overline{\ell}^2 \,\square\, d$ 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 equality constraint $d \equiv \overline{F_k(p(\overline{x}))} \wedge c$ is satisfiable in $FET(\Sigma)$.

Indeed, a preliminary version of this work that considers the rules **(R1)** and **(R2)**, instead of the rule **(R)**, was presented in [17]. As we will explain in Remark 17, after the procedural mechanism has been presented, 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. Besides, in order to explain the technical details independently of this choice, our examples of goals will not contain any positive literal. ■

Definition 6. A finite BCN-derivation $G \overset{n}{\leadsto} G'$ is a *successful BCN-derivation* if $G' \equiv \leftarrow \Box\, c$. In this case c is called the *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 $k > 0$. ■

Soundness and completeness of the BCN operational semantics can be easily proved on the basis of Theorem 2.

Theorem 7. *Let P be a program and $G \equiv \leftarrow \overline{\ell} \Box\, c$ a goal.*
1. *G is a BCN-failed goal iff $Comp(P) \models_3 (c \rightarrow \neg\overline{\ell})^{\forall}$*
2. *If there exists a successful BCN-derivation for G with computed answer $\leftarrow \Box\, c'$, then $Comp(P) \models_3 (c' \rightarrow \overline{\ell} \wedge c)^{\forall}$*
3. *If there exists a satisfiable equality constraint d such that $Comp(P) \models_3 (d \rightarrow \overline{\ell} \wedge c)^{\forall}$, then there exist a BCN-computed answer d' for G with respect to P, such that $FET(\Sigma) \models (d \rightarrow d')^{\forall}$.*

Proof. The statement 1 is a trivial consequence of Theorem 2. The statement 2 is proved by induction on the length of the derivation, using the following easy consequence of Theorem 2: $Comp(P) \models_3 (T_k(\overline{\ell}) \rightarrow \overline{\ell})^{\forall}$ for every $k \in I\!N$.
The assumption of the statement 3, by Theorem 2, means that $FET(\Sigma) \models (d \rightarrow (T_k(\overline{\ell}) \wedge c))^{\forall}$ holds for some $k \in I\!N$. Let us suppose $\overline{\ell} \equiv \ell_1, \ldots \ell_m$. Since d is satisfiable, $\bigwedge_{j \in J} T_k(\ell_j) \wedge c$ is also satisfiable for any $J \subseteq \{1, \ldots, m\}$. This allows us to build the following derivation $G \overset{m}{\leadsto} \leftarrow \Box\, d'$ where $d' \equiv c \wedge T_k(\overline{\ell})$:

$$\leftarrow \underline{\ell_1}, \ell_2, \ldots \ell_m \Box\, c \overset{1}{\leadsto} \leftarrow \underline{\ell_2}, \ldots \ell_m \Box\, c \wedge T_k(\ell_1) \overset{1}{\leadsto} \ldots \overset{1}{\leadsto} \leftarrow \underline{\ell_i}, \ldots \ell_m \Box\, c \wedge \bigwedge_{j=1}^{i-1} T_k(\ell_j) \overset{1}{\leadsto} \ldots$$
$$\ldots \overset{1}{\leadsto} \leftarrow \underline{\ell_m} \Box\, c \wedge \bigwedge_{j=1}^{m-1} T_k(\ell_j) \overset{1}{\leadsto} \ldots \overset{1}{\leadsto} \leftarrow \Box\, c \wedge T_k(\overline{\ell});$$

Obviously, this construction also works by selecting literals in any other order. ■

The most obvious computational problems of the BCN operational semantics is related to stop the construction of T_k, when the goal is a BCN-failed goal. This requires checking the failure condition of Definition 8(a), that is the satisfiability of a constraint

$$(c \rightarrow (F_k(\ell_1) \vee \ldots \vee F_k(\ell_n)))^{\forall}$$

for some $k \in I\!N$. There are two main problems to do that. First, exhaustive search of such $k \in I\!N$ could be infinite, although the part (b) of Definition 8 could produce one or more children for the node. Second, the equality constraint to be checked involves not only the F_k-constraint for the selected literal, but also for all the literals of the goal. The procedural interpretation – we will present in the next section, – avoids such problems looking for failures in an incremental way. At the same time, it avoids some undesirable infinitudes and repetition of answers that occurs in BCN-computations. In order to illustrate these problems and motivate the procedural mechanism, we next define BCN-computation trees and give two simple examples.

Definition 8. A *BCN-computation tree* for a goal G with respect to P is a tree such that each node is a goal, the root is G and for any node $G' \equiv \leftarrow \overline{\ell} \Box\, c$ with selected literal ℓ_i:

(a) G' is a failure leaf if $FET(\Sigma) \models (c \rightarrow F_k(\overline{\ell}))^{\forall}$ for some $k \in I\!N$.
(b) Otherwise, the children of G' are all the goals BCN-derived from it where the selected literal is ℓ_i. ■

In the following two examples the signature has – as function symbols – the constant 0 (zero) and the unary function s (successor).

Example 9. Let P_1 and P_2 be the following two programs

$$P_1 \equiv q(x) :- \Box\, x = 0 \qquad\qquad P_2 \equiv q(x) :- \neg q(y) \,\Box\, x = 0, y = 0$$

It is easy to see that $T_k(\neg q(x)) \equiv F_k(q(x)) \equiv x \neq 0$ for all $k \geq 1$ w.r.t. both programs. Hence, the goal $\leftarrow \neg q(x)$ has the same BCN-computation tree w.r.t. both programs. This tree has an infinite number of branches with the answer $\leftarrow x \neq 0$. From a practical point of view, an infinite computation could be expected in the case of program P_2. However, with program P_1 it does not seem natural to produce an infinite computation, neither to repeat each iteration the unique answer. ■

Example 10. Consider the program:

$$\begin{array}{l} even(x) :- \Box\, x = 0. \\ even(x) :- \neg\, even(y) \,\Box\, x = s(w), y = w. \end{array}$$

The goal $\leftarrow \neg even(x)$ has a leaf for each satisfiable $T_k(\neg even(x)) \equiv F_k(even(x))$ where $k = 2, 3, \ldots$, since $F_1(even(x))) \equiv \underline{\mathsf{f}}$ and

$$F_{2\times i}(even(x)) \equiv F_{2\times i+1}(even(x)) \equiv \bigvee_{j=1}^{i} x = s^{2\times j-1}(0) \text{ for all } i \geq 1$$

Therefore, the computation tree has an infinite number of branches, as the goal has an infinite number of answers. However, there are two kinds of non-desirable repetitions. The first one is that each pair of consecutive answers are identical. The second is that each new answer includes all the previous ones. The latter happens by monotonicity of the operator F (see Proposition 3). ■

4 The Procedural Interpretation

A procedural mechanism, amenable for practical implementation, has to be related to a feasible equality constraint solver. Along this section we assume an equality constraint solver that transforms any equality constraint with $\overline{x}$ as free variables, into a disjunction of answers for $\overline{x}$. An *answer for the variables* $\overline{x}$ is either one of the logical constants ($\underline{\mathsf{t}}$ or $\underline{\mathsf{f}}$), or an equality constraint of the form $\exists \overline{w}(a(\overline{x}, \overline{w}))$. Although, the concrete syntactical form of $a(\overline{x}, \overline{w})$ is not necessary for the contents of this section (it will be given in Section 5), notice that disequations are required to solve equality constraints such as $\neg\exists y(x = f(y, y))$. We denote by $\underline{\mathsf{SolForm}}(c(\overline{x}))$ the solved form obtained by the solver for the equality constraint $c(\overline{x})$. Whenever $\underline{\mathsf{SolForm}}(c(\overline{x}))$ is not one of the logical constants, then it is a disjunction of answers $\bigvee_{i=1}^{m} \exists \overline{w}^i(a_i(\overline{x}, \overline{w}^i))$ where all variables $\overline{w}^1, \ldots, \overline{w}^m$ are assumed pairwise distinct. For simplicity, we consider $\underline{\mathsf{SolForm}}(c) \equiv \underline{\mathsf{t}}$ to be the particular case of disjunction of answers for $m = 1$ and $a_1 \equiv \underline{\mathsf{t}}$. The solver transforms equality constraints into its solved form in a correct and complete manner with respect to the free equality theory.

Definition 11. We say that a equality constraint solver is *correct and complete with respect to the theory* $FET(\Sigma)$ iff for every equality constraint $c(\overline{x})$ the following two conditions hold:

(a) $\underline{\mathsf{SolForm}}(c(\overline{x}))\equiv \bigvee_{i=1}^{m} \exists\overline{w}^i(a_i(\overline{x},\overline{w}^i))$ iff $FET(\Sigma)\models (c(\overline{x}) \leftrightarrow \bigvee_{i=1}^{m} \exists\overline{w}^i(a_i(\overline{x},\overline{w}^i)))^{\forall}$. In particular, $\underline{\mathsf{SolForm}}(c(\overline{x}))\equiv \underline{\mathtt{t}}$ iff $FET(\Sigma)\models c(\overline{x})^{\forall}$
(b) $\underline{\mathsf{SolForm}}(c(\overline{x}))\equiv \underline{\mathtt{f}}$ iff $FET(\Sigma)\models (\neg c(\overline{x}))^{\forall}$, or equivalently $FET(\Sigma)\models \neg(c(\overline{x})^{\exists})$. ■

In the sequel, we will use $a(\_,\_)$ as a metavariable (over an specific pair of variable tuples, when necessary) and computation trees will be formed by nodes of the form $\leftarrow\overline{\ell}\,\Box\, a(\_,\_)$.
In the rest of this section we refine the BCN operational semantics to a purely procedural mechanism that makes use of the answers produced by the solver and avoids the problems discussed in the previous section. Then, we show some examples of computation tree. Finally, we prove the soundness and completeness of our procedural interpretation.
In order to avoid answer repetition, we use the monotonicity of the operators T and F (see Proposition 3). The idea is, whenever $T_k(\ell(\overline{x}))$ has been used to obtain computed answers for ℓ, the equality constraint of the original goal is strengthened with $\neg T_k(\ell(\overline{x}))$ to look for new answers of the same goal. For instance, consider the goal $\leftarrow\neg even(x)\,\Box\, x\neq 0$ and the program of Example 10. Then, since

$$T_1(\neg even(x))\equiv \underline{\mathtt{f}} \text{ and } (x\neq 0 \wedge T_2(\neg even(\overline{x})))\equiv x=s(0),$$

the computed answer $x=s(0)$ is obtained, and other branch with the goal $\leftarrow\neg even(x)\,\Box\, x\neq 0, x\neq s(0)$ is generated. Now, $x\neq 0\wedge x\neq s(0)\wedge T_3(\neg even(\overline{x}))$ is not satisfiable, but $x\neq 0\wedge x\neq s(0)\wedge T_4(\neg even(\overline{x}))\equiv x=s(s(s(0)))$. That generates this new computed answer and also a new branch with $\leftarrow\neg even(x)\,\Box\, x\neq 0, x\neq s(0), x\neq s(s(s(0)))$ and so on. The obtained computation tree is given in Figure 1.

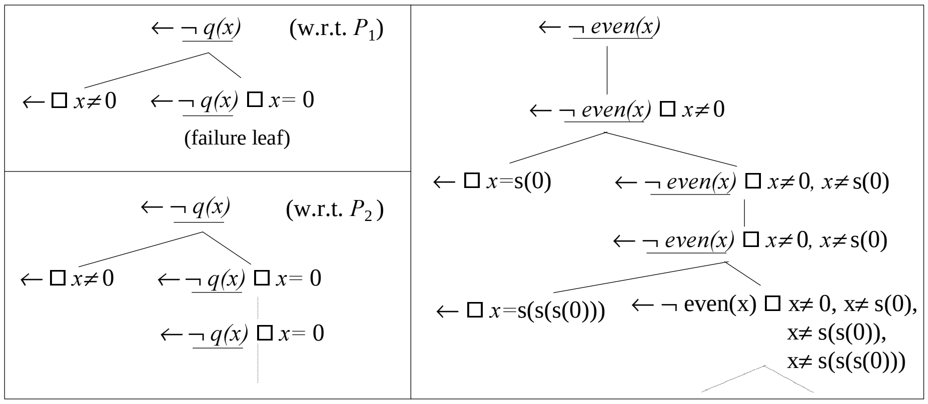


Fig. 1. Computation trees of Examples 9 and 10

In order to achieve failure detection in an incremental way, notice that the following two equality constraints:

$$(c\rightarrow (F_k(\overline{\ell}^1)\vee F_k(\ell(\overline{x}))\vee F_k(\overline{\ell}^2)))^{\forall}$$

$$((c\wedge\neg F_k(\ell(\overline{x})))\rightarrow (F_k(\overline{\ell}^1)\vee F_k(\overline{\ell}^2)))^{\forall}$$

are logically equivalent. Moreover, by coherence of T and F (see Proposition 3), the addition of $\neg F_k(\ell)$ is a coherent strengthening of the goal equality constraint. A failure will be displayed when

$$\underline{\mathsf{SolForm}}((c\wedge\neg F_k(\overline{\ell}^1)\wedge\neg F_k(\ell(\overline{x}))\wedge\neg F_k(\overline{\ell}^2)))\equiv\underline{\mathtt{f}}$$

This is, by Definition 11, equivalent to $FET(\Sigma) \models (\neg c \vee F_k(\overline{\ell}^1) \vee F_k(\ell(\overline{x})) \vee F_k(\overline{\ell}^2))^\forall$, which is, indeed, the required condition. To summarize, in order to converge to the failure condition, we will add the equality constraint $\neg F_k(\ell)$ when ℓ is selected and $T_k(\ell)$ do not provide success. The combination of both techniques serves to bound the anomalous breath-infinitude that was shown in Example 9. That is, for the program P_1 of Example 9 we have that $T_k(\neg q(x)) \equiv x \neq 0$ and $F_k(\neg q(x)) \equiv x = 0$ for all $k \geq 1$. Then, the answer $T_1(\neg q(x)) \equiv x \neq 0$ constitutes one branch. The other branch starts with the goal $\leftarrow \neg q(x) \,\square\, x = 0$. Now, $x = 0 \wedge T_k(\neg q(x))$ is not satisfiable for $k = 2$ (indeed for any $k \geq 2$). So that, since $x = 0 \wedge \neg F_1(\neg q(x))$ is not satisfiable, a failure is detected. However, in the case of the program P_2, the tree construction is identical, except that $x = 0 \wedge \neg F_k(\neg q(x)) \equiv x = 0$ for all k because every $F_k(\neg q(x)) \equiv \underline{\mathtt{f}}$. This gives raise to an infinite branch. Both computation trees are represented in Figure 1.
Now, we will introduce the procedural mechanism directly defining the construction of a computation tree for an arbitrary goal, with respect to a program and a rule that selects, at each step, a literal from the current goal. The goals of a computation tree have answers as – always satisfiable – equality constraints.
From a procedural point of view, it is useful to keep the natural number k corresponding to the k-successes (resp. k-failures) that have to be computed next. This information, denoted by $k_T(\ell)$ (resp. $k_F(\ell)$), is associated to each literal ℓ of any goal inside a computation tree. The collection of all the values associated to both constants for each literal of a goal G is denoted by $K(G)$.

Definition 12. Let P be a Σ-program, R a selection rule, and G a goal. A *computation tree* for G with respect to P and R is a tree whose nodes are pairs $(G', K(G))$ inductively defined as follows. The root is $(G, K_1(G))$ where K_1 associates the value 1 to each k_T and k_F. 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 any $K(G')$ that associates a pair (n^+, n^-) of natural numbers to $(k_T(\ell(\overline{x})), k_F(\ell(\overline{x})))$:

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

(C2b) If $\underline{\mathsf{SolForm}}(\exists \overline{w}(a(\overline{x}, \overline{w})) \wedge \neg F_{n^-}(\ell(\overline{x}))) \equiv \bigvee_{i=1}^m \exists \overline{w}^i a_i(\overline{x}, \overline{w}^i) \not\equiv \underline{\mathtt{f}}$, then it has one child for each $i \in 1..m$, with goal $G_i \equiv \leftarrow \overline{\ell}^1, \ell, \overline{\ell}^2 \,\square\, a_i(\overline{x}, \overline{w}^i)$ and $K(G_i)$ is $K(G')$ where both $k_T(\ell(\overline{x}))$ and $k_F(\ell(\overline{x}))$ are respectively updated to be $n^+ + 1$ and $n^- + 1$. ■

Notice that the constraints in the goals of the tree accumulate $\neg T_n(\ell)$ for every n such that the n-successes of ℓ are obtained in the other branches. Therefore, whenever ℓ is selected again, $a \wedge T_k(\ell)$, is unsatisfiable for all $k < k_T(\ell)$, although it could be satisfiable for some $n > k_T(\ell)$. The role of the constants k_T is to avoid these superfluous calculations of unsatisfiable constraints. Whereas $k_F(\ell)$ is helpful for adding new constraints $\neg F_k(\ell)$ to goals.

Definition 13. Any finite branch of a computation tree for G finished by a leaf of the form $\leftarrow \Box\, a$ 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. ■

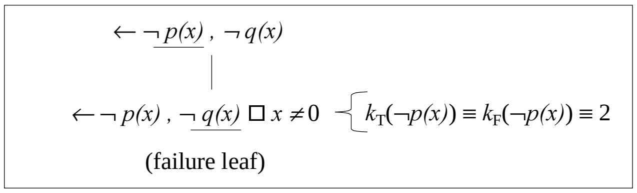
The following example illustrates the construction of failure trees. It uses the same signature with the constant 0 and the unary function s as previous examples.

Example 14. (Inspired by Example 3.12 in [7]). For the program

$$\begin{array}{ll} p(x) : -p(y) \Box\, x = w, y = s(w) & q(x) : -q(y) \Box\, x = 0, y = 0 \\ p(x) : - \Box\, x = 0 & q(x) : -\neg\, r(y) \Box\, x = w, y = w \\ & r(x) : - \Box\, x = 0 \end{array}$$

we have that:

$$\begin{array}{ll} T_k(\neg p(x)) \equiv F_k(p(x)) \equiv \underline{\mathtt{f}} \text{ for all } k \in I\!N & T_k(\neg q(x)) \equiv F_k(q(x)) \equiv \underline{\mathtt{f}} \text{ for all } k \in I\!N \\ F_k(\neg p(x)) \equiv T_k(p(x)) \equiv x = 0 \text{ for all } k \geq 1 & F_k(\neg q(x)) \equiv T_k(q(x)) \equiv \underline{\mathtt{f}} \text{ for all } k \in \{0, 1\} \\ & F_k(\neg q(x)) \equiv T_k(q(x)) \equiv x \neq 0 \text{ for all } k \geq 2 \end{array}$$



$\leftarrow \neg\, \underline{p(x)}\,,\ \neg\, q(x)$
$\leftarrow \neg\, p(x)\,,\ \neg\, \underline{q(x)}\ \Box\, x \neq 0$
$k_T(\neg p(x)) \equiv k_F(\neg p(x)) \equiv 2$
(failure leaf)

Fig. 2. Failure tree of Example 14

The failure tree of Figure 2 for the goal $\leftarrow \neg p(x), \neg q(x)$ results by selecting its first literal, since $T_1(\neg p(x)) \equiv \underline{\mathtt{f}}$ and $\neg F_1(\neg p(x)) \equiv x \neq 0$. Then, $\neg q(x)$ is selected. and $x \neq 0 \wedge T_2(\neg q(x))) \equiv \underline{\mathtt{f}}$ and also $x \neq 0 \wedge \neg F_2(\neg q(x))) \equiv \underline{\mathtt{f}}$. It is easy to check that any other fair selection also produces a failure tree. ■

Now, we prove the soundness and completeness of the introduced procedural mechanism. In both theorems and its proofs, to simplify notation, we write goals in the form $\leftarrow \overline{\ell} \Box\, a$ instead of $\leftarrow \overline{\ell}(\overline{x}) \Box\, a(\overline{x}, \overline{w})$. and, in formulas, we also write a instead of $\exists \overline{w}(a(\overline{x}, \overline{w}))$.

Theorem 15. *(Soundness) Let P be a Σ-program and $G \equiv \leftarrow \overline{\ell} \Box\, a$ be a goal*

1. *If G has a failure tree (with respect to P and some selection rule), then $Comp(P) \models_3 (a \rightarrow \neg \overline{\ell})^{\forall}$.*
2. *If there exists a successful derivation for G (with respect to P and some selection rule) 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 $\leftarrow \overline{m} \Box\, a_0$ and $\{\leftarrow \overline{m}^i \Box\, a_i \mid i \in 1..r\}$ $(r \geq 1)$ is the (finite) collection of all its leaves. Using that, 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 by induction in the length of the derivation. To prove the statement 1, let $\{\leftarrow \overline{\ell}^i \Box\, a_i \mid i \in 1..r\}$ $(r \geq 1)$ be the (finite) collection of all leaves of the failure tree for G. Hence $FET(\Sigma) \models (a_i \rightarrow F_{k_i}(\overline{\ell}^i))^{\forall}$ holds for each $i \in 1..r$ and some $k_i \in I\!N$. Then, by monotonicity of the operator F (Proposition 3), $Comp(P) \models_3 ((\overline{\ell}^i \wedge a_i) \to (\overline{\ell}^i \wedge F_k(\overline{\ell}^i)))^\forall$ where $k \equiv max\{k_i | i \in 1..r\}$. Therefore

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

Additionally, by Theorem 2, we have that $Comp(P) \models_3 (F_k(\overline{\ell}^i) \to \neg\overline{\ell}^i)^\forall$, where $F_k(\overline{\ell}^i)$ is a equality constraint, so it is a two-valued formula. Then $Comp(P) \models_3 (\bigvee_{i=1}^{r}(\overline{\ell}^i \wedge a_i) \to \underline{\mathbf{f}})^\forall$. Hence $Comp(P) \models_3 ((\overline{\ell} \wedge a) \leftrightarrow \underline{\mathbf{f}})^\forall$ or equivalently $Comp(P) \models_3 (a \to \neg\overline{\ell})^\forall$. ■

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

Theorem 16. *(Completeness) Let P be a Σ-program and $G \equiv \leftarrow\overline{\ell} \,\square\, a$ a goal. Then, for any fair selection rule:*

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

Proof. We first prove the statement 2. By Theorem 2, $FET(\Sigma) \models (c \to (T_k(\overline{\ell}) \wedge a))^\forall$ 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: 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 $FET(\Sigma) \models ((T_k(\overline{\ell}) \wedge a) \to \bigvee_{i=1}^{n} a_i)^\forall$.

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 exists $r \geq 1$ answers $a'_1, \ldots, a'_r$ such that $\underline{\mathsf{SolForm}}(a \wedge T_k(\ell_1)) \equiv \bigvee_{i=1}^{r}(a'_i)$. Therefore, the goal G has r children of the form $\leftarrow\overline{\ell}' \,\square\, a'_i$ where $\overline{\ell}' \equiv \ell_2, \ldots, \ell_m$. By completeness of the solver $FET(\Sigma) \models ((T_k(\ell_1) \wedge a) \leftrightarrow \bigvee_{i=1}^{r} a'_i)^\forall$. Since $FET(\Sigma) \models (T_k(\overline{\ell}') \wedge \bigvee_{i=1}^{r} a'_i)^\exists$, then 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, the above Fact holds for each $(T_k(\overline{\ell}') \wedge a'_j)$ with $j \in J$. Then, there exist $k_j > 0$ computed answers $a^j_1, \ldots, a^j_{k_j}$ for $\leftarrow\overline{\ell}' \,\square\, a'_j$ such that $FET(\Sigma) \models ((T_k(\overline{\ell}') \wedge a'_j) \to \bigvee_{h=1}^{k_j} a^j_h)^\forall$ 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

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

Hence the above Fact and the statement 2 of the Theorem hold.
Finally, we prove the statement 1. By Theorem 2, $FET(\Sigma) \models (a \to F_k(\overline{\ell}))^\forall$ for some $k \in I\!N$. Then, in the computation tree for G, there is no branch finished by a computed answer. This is because supposing the existence of a branch ended by a computed answer − and using the Theorems 15 and 2 − we have the contradiction that such computed answer must be unsatisfiable. 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, $FET(\Sigma) \models (a_{i+1} \to a_i)^\forall$ holds for all $i \in I\!N$. This means − by coherence of the operators T and F (Proposition 3) − that $FET(\Sigma) \models (a_i \to \neg T_n(\overline{\ell}))^\forall$ for all $i \in I\!N$. Therefore − by construction of the tree, completeness of the solver, and fairness in the selection − there is some $i \in I\!N$, such that the constraints $a_i \wedge T_n(\ell)$ 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, by correctness of the solver, there exists some j such that $FET(\Sigma) \models (a_j \rightarrow (a_0 \wedge \neg F_k(\ell_{i_1})) \wedge \ldots \wedge \neg F_k(\ell_{i_{m_j}})))^{\forall}$ 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. ■

Remark 17. 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 $\leftarrow\overline{\ell} \,\Box\, a$ 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. ■

5 Implementation

The crucial matter for practical implementation is to solve T_k- and F_k-constraints in an efficient incremental manner. For that, we have to fix a notion of *answer* (or solved form, see Section 2), that fulfills two main requirements: its satisfiability must be easily decidable and they should be user friendly (to be displayed as goal answers). In this section, we first introduce a notion of answer that satisfies both conditions. Second, we explain the basic operations for handling answers and the guidelines for efficient incremental solving of T_k- and F_k-constraints. Finally, we provide some experimental results.
Roughly speaking, answers consists of equations and disequations with some syntactical restrictions:

Definition 18. 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
– flat equations of the form $x_i = t(\overline{w})$, and
– universal 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 occurs in s.
where each x_i occurs at most once. ■

For instance, an answer for the variable x is:

$$\exists w_1 \exists w_2 (x = f(w_1, w_2) \wedge w_1 \neq g(w_2) \wedge \forall v_1 (w_2 \neq f(w_1, v_1))) \qquad (1)$$

We represent it, in Prolog-like notation, by $\mathtt{x} = \mathtt{f(\_A,\_B)}$, $\mathtt{\_A} \neq \mathtt{g(\_B)}$, $\mathtt{\_B} \neq \mathtt{f(\_A,*D)}$ 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. Notice that the scope of each universal variable is one single disequation and there is no restriction about repetition of existential, neither universal, variables. It is obvious that every answer can be represented in the above explained Prolog-like notation.
In the case of infinite signature, an answer is always satisfiable. In fact, a similar (but less user friendly) kind of solved form is used in [5], where only infinite signatures are considered. However, for finite signature, an answer can be unsatisfiable. A simple example of unsatisfiable answer for the signature $\{a/0, f/2, g/1\}$ is:

$$\exists w(x = w \wedge w \neq a \wedge \forall v_1 \forall v_2 (w \neq f(v_1, v_2)) \wedge \forall v_1 (w \neq g(v_1)) \qquad (2)$$

We have implemented an answer satisfiability check that, in a first step, determines whether each variable w_j has a (possibly empty) finite or an infinite number of candidate values. To do that, only a part of the disequations have to be taken into account. If the number of candidates is infinite for all w_j, then the answer is satisfiable. If it is empty for some w_j, then the answer is unsatisfiable. Otherwise, this first step obtains the set W_{fin} of all variables w_j which have a finite set of candidates values and a representation of these candidate sets. In a second step, this information is used to finish the check, by considering only the (remainder) disequations such that they involves more that one variable in W_{fin}. For instance, the satisfiability of the above two examples of answers are decided at the first step. In the example (1) both variables w_1 and w_2 have an infinite set of candidate values. In (2), the set for w is empty.
Now, we are going to explain the other three basic operations – conjunction, negation and instantiation – for handling answers:

- The conjunction of (two or more) answers for the variables $\overline{x}$ produces a disjunction of answers for $\overline{x}$. It is made by unification of the tuples of terms equating identical x_i. If this most general unifier does not exists, the result is $\underline{\mathtt{f}}$. Otherwise this *mgu* is applied to all equations and disequations. Then, disequations are flattened. The resulting universal disjunctions of flat disequations (on the variables $\overline{w}$) are transformed, by the transformation rule (UD) of Figure 3, into a disjunction of flat equations and disequations where universal variables are local to one single disequation. Finally, we distribute and lift the disjunction.
- The negation of an answer for $\overline{x}$ has the form $\forall\overline{w}(\overline{x} \neq \overline{t}(\overline{w}) \vee \bigvee \exists\overline{v}^j(w_j = s(\overline{w},\overline{v}^j)))$ which is equivalent 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 internal existential-disjunction and substituting w_j in $t(\overline{w})$) is 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 the transformation rule (UD) (of Figure 3) to $\forall\overline{v}(\overline{w}' \neq \overline{t}(\overline{v}))$ just as above. Hence, we obtain a disjunction of answers for $\overline{x}$.
- The instantiation $\exists\overline{w}(\overline{x} = \overline{t}(\overline{w}) \wedge \bigwedge \forall\overline{v}(w_j = s(\overline{w},\overline{v})))[\overline{t}'(\overline{z})/\overline{x}]$ produces a disjunction of answers for $\overline{z}$. It is performed on the basis of $\mu \equiv mgu(\overline{t}(\overline{w}),\overline{t}'(\overline{z}))$. It is transformed into $\exists\overline{w}(\mu_1 \wedge \bigwedge \forall\overline{v}((w_j \neq s(\overline{w},\overline{v}))\mu_2)$ where $\mu_1 \equiv \mu \upharpoonright \overline{z}$ and $\mu_2 \equiv \mu \upharpoonright \overline{w}$. To obtain a disjunction of answers for $\overline{z}$, it suffices to apply μ_2, to flat the disequations, and to split the universal variables using the transformation rule (UD) (of Figure 3).

$$\boxed{\begin{array}{r}\forall\overline{v}(w_j \neq t(\overline{w},\overline{v}) \vee d(\overline{w},\overline{v})) \longmapsto \forall\overline{v}^1(w_j \neq t(\overline{w},\overline{v}^1)) \vee \exists\overline{v}^1(w_j = t(\overline{w},\overline{v}^1) \wedge \forall\overline{v}^2 d(\overline{w},\overline{v})) \\ \text{where } \overline{v}^1 \equiv free(\overline{t}) \cap \overline{v} \text{ and } \overline{v}^2 \equiv \overline{v} \setminus \overline{v}^1\end{array}}$$

Fig. 3. Transformation Rule (UD)

The solving of the T_{k+1}- and F_{k+1}-constraints needs the previously constructed T_k- and F_k-constraints for the same literals. Hence, to improve efficiency, $T_k(p(\overline{x}))$ and $F_k(p(\overline{x}))$ must be loaded as part of the p's predicate description in the symbol table, for some prefixed symbols p and for the last k computed. Besides this, there are three other aspects that are crucial for efficiency. First, we solve T_k- and F_k-constraints in a incremental way. That is, we do not recalculate previously obtained answers (remember that both operators are monotonic). Second, we load a constraint scheme, in order to avoid calculations that are common for all steps. Third, we solve once the shared sub-constraints of a constraint and we load the solution of sub-constraints that are shared by two consecutive steps. The last two aspects are important in the solving of F_k. The constraint scheme for $T_{k+1}(p(\overline{x}))$ is

$$\bigvee_{i=1}^{m} \exists\overline{y}^i \exists\overline{w}^i (\overline{x} = \overline{t}^i(\overline{w}^i) \wedge \overline{y}^i = \overline{s}^i(\overline{w}) \wedge \bigwedge_j T_k(\ell^i_j(\overline{y}^i)))$$

where each $T_k(\ell^i_j(\overline{y}^i))$ has been loaded as a disjunction of answers for $\overline{y}^i$:

$$\bigvee_i \exists\overline{z}(a_i^{(k)}(\overline{y}^i, \overline{z})) \vee \bigvee_i \exists\overline{z}(a_i^{(<k)}(\overline{y}^i, \overline{z}))$$

which is spitted into the answers $a_i^{(k)}$ of step k and the answers $a_i^{(<k)}$ of the previous steps. $\bigwedge_j T_k(\ell^i_j(\overline{y})))$ could be transformed, by distribution, in a disjunction of conjunction of answers for $\overline{y}$. But, not all of them are new. Hence, we take only the disjuncts that includes at least one $a_i^{(k)}$. In this way, lifting disjunction, we obtain a constraint of the form

$$\bigvee \exists\overline{y}\exists\overline{w}(\overline{x} = \overline{t}(\overline{w}) \wedge \overline{y} = \overline{t}'(\overline{w}) \wedge \bigwedge_i \exists\overline{z}(a_i(\overline{y}, \overline{z}))))$$

Therefore, we perform the conjunction of answers for $\overline{y}$ and the instantiation of $\overline{y}$ by terms $\overline{t}'(\overline{w})$. This gives a disjunction of answers for $\overline{w}$. Lifting this disjunction, and substituting equations on $\overline{w}$ into the terms $\overline{t}(\overline{w})$, we obtain a disjunction of answers for $\overline{x}$.
The incremental solving of F_k-constraints is technically more intricate. To simplify the presentation, let us consider a definition of $p(\overline{x})$ given by two clauses of the form:

$$p(\overline{x}) :- \overline{\ell}^1(\overline{y}^1) \,\square\, \overline{x} = \overline{t}^1(\overline{w}^1), \overline{y}^1 = \overline{s}^1(\overline{w}^1)$$

$$p(\overline{x}) :- \overline{\ell}^2(\overline{y}^2) \,\square\, \overline{x} = \overline{t}^2(\overline{w}^2_1), \overline{y}^2 = \overline{s}^2(\overline{w}^2_1, \overline{w}^2_2)$$

and assume that every variable appearing in the $\overline{y}^1$-equations of the first clause also appears in its $\overline{x}$-equations, whereas the second clause have new variables $\overline{w}^2_2$ in the $\overline{y}^2$-equations. These are the two kinds of clauses that are significant in the solving of an F_k-constraint. Hence, we hope that the reader can easily infer (from the following explanation for the above two clauses) how our solver works with m clauses. At the first step $F_1(p(\overline{x})) \equiv c^1_1 \wedge c^2_1$ where $c^1_1 \equiv \neg\exists\overline{w}^1(\overline{x} = \overline{t}^1(\overline{w}^1))$ and $c^2_1 \equiv \neg\exists\overline{w}^2_1(\overline{x} = \overline{t}^2(\overline{w}^2_1))$. The constraints c^1_1 and c^2_1 are both the negation of an answer for $\overline{x}$. Hence, they are solved and loaded as a disjunction of answers for $\overline{x}$. To solve the constraint $F_1(p(\overline{x}))$ we use distribution, disjunction-lifting and conjunction of answers for $\overline{x}$. At the next step[5], $F_{=2}(p(\overline{x})) \equiv (c^1_1 \wedge c^2_2) \vee (c^2_1 \wedge c^1_2) \vee (c^1_2 \wedge c^2_2)$ where

$$c^1_2 \equiv \exists\overline{w}^1(\overline{x} = \overline{t}^1(\overline{w}^1) \wedge F_{=1}(\overline{\ell}^1)[\overline{s}^1(\overline{w}^1)/\overline{y}^1])))$$
$$c^2_2 \equiv \exists\overline{w}^2_1(\overline{x} = \overline{t}^2(\overline{w}^2_1) \wedge \forall\overline{w}^2_2(F_{=1}(\overline{\ell}^2)[\overline{s}^2(\overline{w}^2_1, \overline{w}^2_2)/\overline{y}^2])))$$

are the new sub-constraints of step 2 that are once solved and loaded for the step 3. Now, for $k \geq 3$, the scheme of $F_{=k}$ is

$$(c^1_1 \wedge c^2_k \wedge \neg c^2_{k-1}) \vee (c^2_1 \wedge c^1_k) \vee (c^1_k \wedge c^2_k \wedge \neg c^2_{k-1}) \vee (c^1_k \wedge c^2_{k-1}) \vee (c^1_{k-1} \wedge c^2_k \wedge \neg c^2_{k-1})$$

where c^1_1, c^2_1 are loaded since the first step and c^1_{k-1} and c^2_{k-1} has been solved and loaded in the previous step, whereas the new sub-constraints of step k are:

$$c^1_k \equiv \exists\overline{w}^1(\overline{x} = \overline{t}^1(\overline{w}^1) \wedge F_{=k-1}(\overline{\ell}^1)[\overline{s}^1(\overline{w}^1)/\overline{y}^1])$$

$$c^2_k \equiv \exists\overline{w}^2_1(\overline{x} = \overline{t}^2(\overline{w}^2_1) \wedge \forall\overline{w}^2_2((F_{=k-1}(\overline{\ell}^2) \vee \ldots \vee F_{=1}(\overline{\ell}^2))[\overline{s}^2(\overline{w}^2_1, \overline{w}^2_2)/\overline{y}^2])))$$

[5] In what follows, $F_{=k}(p(\overline{x}))$ denotes the incremental part of $F_k(p(\overline{x}))$ w.r.t. $F_{k-1}(p(\overline{x}))$ and c^j_k denotes the new sub-constraint generated at step k from the clause j.

Notice that the above scheme for $F_{=k}$ includes several shared sub-constraints (e.g. $c^2_k \wedge \neg c^2_{k-1}$) that are solved once. Moreover, there are calculations that are common to all steps and we use the scheme to avoid repetitions. For instance, every conjunction of the form $c^1_n \wedge c^2_m$ (in $F_{=k}$) such that the terms $\overline{t}^1(\overline{w}^1)$ and $\overline{t}^2(\overline{w}^2_1)$ are not unifiable, is equivalent to $\underline{\mathtt{f}}$. In this case, the corresponding disjunct is removed from the scheme of $F_{=k}$. Finally, we want to remark that our strategy for obtaining a disjunction of answers is lazy, in the sense that a constraint is partially solved until a single satisfiable answer is obtained. Then, in the Prolog-style, it is displayed to the user, who can ask for more answers. In this case, the remainder constraint will be treated in the same lazy way. Of course, only satisfiable answers are displayed.

We have implemented a prototype of the BCN procedural mechanism in Sicstus Prolog v.3.8.x. We have executed it on a Pentium II at 233 MHz taking measurements with the function statistic/2 of Sicstus Prolog. In Figure 4, we show some experimental results for the program of Example 10 and also for the following programs[6]:

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

(2) even_number_of_f(a).
    even_number_of_f(f(X,Y)) :- even_number_of_f(X), ¬ even_number_of_f(Y).
    even_number_of_f(f(X,Y)) :- ¬ even_number_of_f(X), even_number_of_f(Y).

(3) 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).
```

Atom	$T_{=3}$	$T_{=4}$	$T_{=5}$	$T_{=6}$	$T_{\_}$	$F_{=3}$	$F_{=4}$	$F_{=5}$	$F_{=6}$	$F_{\_}$
`even(X)`	60/1	60/0	50/1	50/0	...	0/0	60/1	60/0	50/1	...
`sum(X,Y,Z)`	50/1	60/1	60/1	0/0	...	0/2	0/2	50/2	60/2	...
`even_by_sum(X)`	50/1	50/1	50/1	60/1	...	50/1	60/0	60/1	60/0	...
`even_number_of_f(X)`	60/2	50/10	110/ 326	53260/ 228826	...	50/1	60/11	110/ 325	147890/ 228827	...
`symmetric(X)`	60/4	50/14	110/ 184	12690/ 33674	...	60/12	60/39	160/ 120	380/ 363	...

Fig. 4. Some Experimental Results

Each item of the array in Figure 4 means "milliseconds of CPU-time / number of answers'". More concretely, the first element specify the milliseconds required for computing all new satisfiable answers that are obtained at the considered step of the operator. These answers are, depending on the above-specified operator ($T_{\_}$ or $F_{\_}$), success or fail answers for the left-hand atom. The second element, says how many satisfiable answers are exactly calculated in that time. We have prefered to show the time required to compute the whole set of answers, but remember that answers are displayed as soon as they are obtained.

[6] It is worthy to notice the existential variable `Y` in the third clause of program `(1)`, the recursion through negation in program `(2)`, and the big search space in programs `(2)` and `(3)`.

References

1. A. Bossi, M. Fabris, and M. C. Meo. A bottom-up semantics for constructive negation. In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP '94)*, pages 520–534. MIT Press, 1994.
2. D. Chan. Constructive negation based on the completed database. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 111–125, Seatle, 1988. ALP, IEEE, The MIT Press.
3. D. Chan. An extension of constructive negation and its application in coroutining. In E. Lusk and R. Overbeek, editors, *Proceedings of the NACLP'89*, pages 477–493. The MIT Press, 1989.
4. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322, New York, 1978. Plenum Press.
5. A. Colmerauer and T.-B.-H. Dao. Expresiveness of full first order constraints in the algebra of finite and infinite trees. In *6th International Conference of Principles and Practice of Constraint Programming CP'2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 172–186, 2000.
6. H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.
7. W. Drabent. What is failure? an approach to constructive negation. *Acta Informática*, 32:27–59, 1995.
8. F. Fages. Constructive negation by pruning. *Journal of Logic Programming*, 32(2):85–118, 1997.
9. M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.
10. J. Jaffar and J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19,20:503–581, 1994.
11. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.
12. 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.
13. M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of the 3rd IEEE Symp. on Logic in Computer Science*, pages 348–357, 1988.
14. A. I. Malcev. Axiomatizable classes of locally free algebras. In B. F. Wells, editor, *The Metamathematics of Algebraic Systems (Collected Papers: 1936-1967)*, volume 66, chapter 23, pages 262–281. North-Holland, 1971.
15. S. Muñoz and J. J. Moreno-Navarro. Constructive negation for prolog: A real implementation. In *Proc. of the Joint Conference on Declarative Programming APPIA-GULP-PRODE'2002*, pages 39–52, 2002.
16. P. Nickolas. The representation of anwers to logical queries. In *Proceedings of the 11th Australian Computer Science Conference*, pages 246–255, 1988.
17. 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.
18. J.C. Shepherdson. Language and equality theory in logic programming. Technical Report No. PM-91-02, University of Bristol, 1991.
19. P. J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.
20. S. Vorobyov. An improved lower bound for the elementary theories of trees. In *Automated Deduction CADE-13 LNAI 110*, pages 275–287. Springer Verlag, 1996.