A Necessary Condition for Constructive Negation in Constraint Logic Programming

Agostino Dovier* Enrico Pontelli† Gianfranco Rossi‡

Abstract

In [15] a sound and complete procedure for Constructive Negation in Constraint Logic Programming has been presented, together with a sufficient condition, called *admissible closure*, which guarantees an effective implementation. In this paper we analyze this condition and relate it to the decidability of the underlying constraint structure. We prove that the admissible closure condition is also necessary to guarantee the existence of an effective implementation of Constructive Negation.
Keywords: Constraint Logic Programming, Constructive Negation.

1 Introduction

Constructive Negation (CN) [3, 2, 6, 8, 15] is a technique for handling negation in (Constraint) Logic Programming. The technique relies on the explicit construction of *answers* to a goal which may possibly involve negation. This is commonly achieved by explicitly computing the negation of the solutions to the corresponding positive goal. This scheme allows one to overcome the restrictions imposed by *Negation as Failure (NAF)* [14, 1]; in particular, negative literals in a goal are not restricted to be ground, and answers are not limited to true/false.

The term Constructive Negation was first introduced in [3], as a technique for extending the NAF rule in the context of Logic Programming. The semantic properties of CN in logic programming have been studied by various authors (e.g., [6, 13]). CN has been extended to the more general context of *Constraint Logic Programming (CLP)* by Stuckey [15], who gave the first completeness result for this method. In [8] Fages introduced a *pruning* technique to implement Constructive Negation in CLP and Concurrent Constraint languages.

In [15] Stuckey develops a general framework for handling CN in CLP, and proves its soundness and completeness (in 3-valued logic). The development of an actual implementation of such operational model requires the ability to verify satisfiability of universally quantified constraints—or the ability to eliminate the universal quantification via constraint simplification. Stuckey introduces in [15] a sufficient condition on the constraint structure which allows to achieve such results. Such condition, called *admissible closure*, is proved to be naturally satisfied by the most commonly used structures for CLP, such as $\mathcal{R}$ (the real numbers), Finite Domains, and Rational Trees [15]. In this paper we show, by using an undecidability result, that the above condition is also *necessary*. The result also characterizes a meaningful class of structures for CLP which are *not* suitable to effectively support CN. We also provide results connecting the notions of quantifier elimination, admissible closure, and decidability of the CLP structure.

The paper is organized as follows. Section 2 provides a basic introduction to Constraint Logic Programming and Constructive Negation in CLP. In Section 3 we recall some of the results proved in [15] and we point out some new propositions relating admissible closure and decidability. In Section 4 we show that the existence of an effective implementation of CN implies the decidability and admissible closure of the constraint structure. Some conclusions are drawn in Section 5.

*Università di Verona, Dip. Scientifico-Tecnologico, Strada Le Grazie. 37134 Verona (IT). dovier@sci.univr.it
†New Mexico State University, Dept. Computer Science, Las Cruces, NM 88003 (USA). epontell@cs.nmsu.edu
‡Università di Parma, Dip. di Matematica. Via M. D'Azeglio, 85/A. 43100 Parma (IT). gianfr@prmat.math.unipr.it

2 Preliminary Definitions

Let Σ be a first-order signature which defines a set of predicate symbols Π, a set of function and constant symbols $\mathcal{F}$, and a function $ar : \Pi \cup \mathcal{F} \longrightarrow \mathbb{N}$ which associates an arity to each symbol in Σ. Moreover, let $\mathcal{V}$ be a denumerable collection of variables. We use $\bar{X}$ and $\bar{t}$ to denote a list of variables and a list of terms, respectively. Additionally, we use $\{\bar{X}\}$ to denote the *set* of variables occurring in the list $\bar{X}$.

Π is assumed to be composed of two disjoint sets of predicate symbols: Π_C is the set of constraint predicate symbols and Π_P is the set of program predicate symbols. A *primitive constraint* is an atom $p(t_1, \ldots, t_n)$ where $p \in \Pi_C$, $ar(p) = n$, and $t_1, \ldots, t_n$ are terms built using symbols in $\mathcal{F}$ and $\mathcal{V}$. A *constraint* is any first-order formula over $\langle \Pi_C, \mathcal{F}, \mathcal{V}\rangle$. Let $\mathcal{C}$ denote the set of all constraints.

We will use the notation $\varphi[\bar{X}]$ to emphasize the fact that $\bar{X}$ are all the free variables occurring in the formula φ. In this case we also say that $vars(\varphi) = \{\bar{X}\}$. $\vec{\exists}\varphi$ and $\vec{\forall}\varphi$ will be used to denote $\exists \bar{X}\varphi$ and $\forall \bar{X}\varphi$, respectively. Generally, capital letters X, Y, Z, etc. will be used to denote variables, f, g, etc. will denote function symbols, and p, q, etc. will denote program predicate symbols.

A CLP program P with negation is a set of clauses of the form $A \leftarrow c \wedge \bar{B}$, where A (the *head*) is a $\langle \Pi_P, \mathcal{F}, \mathcal{V}\rangle$ atom, c is a (possibly empty) conjunction of constraints and $\bar{B}$ is a (possibly empty) conjunction of $\langle \Pi_P, \mathcal{F}, \mathcal{V}\rangle$ literals. A *goal* is a clause with empty head. We can also assume that the CLP programs are in *canonical form*, i.e., the arguments in the head of each clause are distinct variables.

Example 2.1 *Let Σ contain the constant and function symbols a, b, c, f, with $ar(a) = ar(b) = ar(c) = 0$, $ar(f) = 2$, the program predicate symbols p and q, with $ar(p) = ar(q) = 2$, and the binary constraint predicate symbol $=$. The primitive constraints are equations between terms.*[1] *A sample CLP program is*

$$\begin{array}{l} \mathtt{p(X,Z)} \leftarrow \mathtt{X} = \mathtt{f(a,Y)} \wedge \mathtt{X} = \mathtt{f(b,Y)} \\ \mathtt{q(X,Y)} \leftarrow \mathtt{X} = \mathtt{c} \wedge \mathtt{Y} = \mathtt{c} \\ \mathtt{q(X,Y)} \leftarrow \mathtt{X} \neq \mathtt{c} \wedge \mathtt{p(X,Y)} \end{array}$$

A sample goal is $\leftarrow \neg\mathtt{p(X_1,X_2)}$.

Given a CLP program P we use P^* to denote the theory obtained by the following transformation of P [14]: assume that $p(\bar{X}) \leftarrow \bar{A}_1, \ldots, p(\bar{X}) \leftarrow \bar{A}_n$ are all the clauses in P defining the predicate symbol p, and let $vars(\bar{A}_i) \setminus \{\bar{X}\} = \{\bar{Y}_i\}$, then such clauses are rewritten as: $\forall \bar{X}\left(p(\bar{X}) \leftrightarrow \bigvee_{i=1}^{n} \exists \bar{Y}_i \bar{A}_i\right)$. The formula $\forall \bar{X} \neg p(\bar{X})$ is introduced in P^* if there are no clauses in P having p as head predicate.

Constraints and programs in the CLP language based on Σ are interpreted with respect to a selected Σ-structure. A Σ-*structure* (or, simply, a *structure*) $\mathcal{A}$ is a pair $\langle A, (\cdot)^{\mathcal{A}}\rangle$, where A is a non-empty set—the *domain* of $\mathcal{A}$—and $(\cdot)^{\mathcal{A}}$ is a function which maps *(i)* each constant symbol of $\mathcal{F}$ to an element of A; *(ii)* each $f \in \mathcal{F}$, $ar(f) = n > 0$ to a function $(f)^{\mathcal{A}} : A^n \longrightarrow A$, and *(iii)* each n-ary predicate symbol $p \in \Pi$, $ar(p) = n$, to a n-ary relation $(p)^{\mathcal{A}} \subseteq A^n$. Structures for CLP are assumed to be models of equality [9].

$CLP(\mathcal{A})$ denotes a particular instance of a CLP language based on the structure $\mathcal{A}$. This instance is further characterized by the considered signature Σ, the class of constraints $\mathcal{C}$, a constraint theory $\mathcal{T}$ which describes the logical semantics of the constraints in $\mathcal{C}$, and a *constraint solver*, that is a procedure which is used to check the satisfiability of constraints from $\mathcal{C}$ with respect to $\mathcal{A}$.[2]

Given a structure $\mathcal{A}$, a particular class of constraints from $\mathcal{C}$ is identified and called *admissible constraints* (AC) of $\mathcal{A}$. Admissible constraints are a subset of those constraints that the constraint solver is capable of handling [15]—in particular it is decidable to verify whether $\mathcal{A} \models \vec{\exists}\varphi$ for $\varphi \in AC$. In other words, the constraint solver is guaranteed to be *complete* with respect to AC [10].

A structure $\mathcal{A}$ is said to be *decidable* if for any first-order formula φ it is decidable if $\mathcal{A} \models \vec{\exists}\varphi$. For a decidable structure $\mathcal{A}$ we could choose AC to be equal to the set $\mathcal{C}$. Even in presence of decidable

[1] Since negation is allowed, negated equations can also occur in a constraint. In this case, we will use the symbol $\neq$, and we usually refer to them as disequations.

[2] The class of CLP languages is parametric w.r.t. the collection of all these components (the *constraint domain* [10]). For simplicity, we use the simpler notation in which an instance of the general CLP scheme is simply identified by its structure.

structures (as explained in [15]), the class AC is commonly restricted to guarantee efficient decision procedures—e.g., by keeping AC as close as possible to the set of primitive constraints. Throughout the paper we assume, as common, that AC is closed with respect to conjunction—conjunctions of constraints are generated during resolution and need to be tested for satisfiability.

Example 2.2 *Let Σ contain a collection $\mathcal{F}$ of constant and function symbols and the binary constraint predicate symbol $=$. Let A be the set of finite trees built in the usual way from symbols in $\mathcal{F}$ and let $(\cdot)^{\mathcal{A}}$ allow terms to be mapped to trees (as usual in Prolog). $\langle A, (\cdot)^{\mathcal{A}} \rangle$ is the Herbrand structure $\mathcal{H}$ over Σ. Clark's Equality Theory is a complete axiomatization for this structure [12]. According to [3] one can choose the admissible constraints to be $s = t$ and $\forall \bar{Z}(s \neq t)$, where s and t are terms and $\bar{Z}$ a (possibly empty) set of variables. We denote this CLP instance as usual as CLP($\mathcal{FT}$).*

If $(\cdot)^{\mathcal{A}}$ associates to each predicate of Π a function over $\{\mathsf{true}, \mathsf{false}, \mathsf{undefined}\}$, then the structure (model) is 3-valued. According to [15], we adopt the notation $\models$ for the two-valued (standard) logical consequence and $\models_3$ for the 3-valued consequence. If $\mathcal{A}$ is a 2-valued structure $\langle \Pi_C, \mathcal{F}, \mathcal{V} \rangle$ and T is a first-order theory, then $\mathcal{A}, T \models_3 \varphi$ means that in every 3-valued model $\mathcal{B}$ of T and of the theory associated with the structure $\mathcal{A}$ the formula φ is true. Observe that every two-valued model is a 3-valued model in which nothing is undefined. Thus, the above condition implies $\mathcal{A}, T \models \varphi$.

A $CLP(\mathcal{A})$ interpreter is a rewriting system operating on *states* [10]. Each state has the form $\langle c \,|\, G \rangle$ where c is an $\mathcal{A}$-satisfiable constraint and G is a conjunction of Π_P atoms. c is called the *constraint store* while G is the *goal*. One derivation step consists of selecting an element $p(\bar{t})$ from G and a clause $p(\bar{X}) \leftarrow c' \wedge B$ from the program and producing the new state $\langle c \wedge c' \wedge \bar{X} = \bar{t} \,|\, G \setminus \{p(\bar{t})\} \cup B \rangle$, as long as the constraint $c \wedge c' \wedge \bar{X} = \bar{t}$ is satisfiable with respect to $\mathcal{A}$. A state $\langle c \,|\, B \rangle$ is failed if B is not empty and no derivation steps are possible. A state is successful if B is empty and c is satisfiable in $\mathcal{A}$. As in SLD-resolution, the derivations starting from a given goal can be represented using a *derivation tree*.

This notion of derivation has been extended in [15] to support CN. In presence of negation, the notion of state has to be modified allowing *complex subgoals* to appear in the second part of the state, where a complex subgoal is of the form $\neg \exists \bar{X}(c \wedge B)$—$c$ is a constraint and B is a conjunction of (simple or complex) subgoals. Let us assume a state $\langle \tilde{c} \,|\, G \rangle$. Selection of a complex subgoal $g \equiv \neg \exists \bar{X}(c \wedge B)$ in G during derivation leads to the following steps:

1. A collection of states $\langle c_1 \,|\, B_1 \rangle, \ldots, \langle c_k \,|\, B_k \rangle$ is determined from the derivation tree of $\langle c' \,|\, B \rangle$, such that
$$\mathcal{A}, P^* \models_3 \vec{\forall}(\exists \bar{X} c' \wedge B \leftrightarrow \exists \bar{Y}_1 c_1 \wedge B_1 \vee \cdots \vee \exists \bar{Y}_k c_k \wedge B_k)$$
where Y_i are the variables of c_i, B_i not in c, B, and c' is a constraint such that $\mathcal{A} \models c \rightarrow c'$. In [15] the indicated states are obtained by developing a *finite* part of the derivation tree and collecting its *frontier*.

2. The collection of states from the sub-derivation is negated to obtain the following equivalence:
$$\mathcal{A} \models (\neg \exists Y_1(c_1 \wedge B_1) \wedge \cdots \wedge \neg \exists Y_k(c_k \wedge B_k)) \leftrightarrow (d_1 \wedge G_1) \vee \cdots \vee (d_h \wedge G_h)$$
where each $d_i \wedge G_i$ is a (simple or complex) subgoal.

3. The resulting states obtained from this derivation step are those of the form $\langle \tilde{c} \wedge d_i \,|\, G \setminus \{g\} \cup G_i \rangle$ as long as $\tilde{c} \wedge d_i$ is (two-valued) satisfiable with respect to $\mathcal{A}$.

The soundness and completeness of this procedure w.r.t. 3-valued consequences of $\mathcal{A}, P^*$ has been proved in [15].

Example 2.3 *Consider the CLP($\mathcal{FT}$) program*

$$\mathtt{p(X)} \leftarrow \mathtt{X} = 0 \qquad \mathtt{p(X)} \leftarrow \mathtt{X} = 1 \qquad \mathtt{q(X)} \leftarrow \neg \mathtt{p(X)}$$

and the goal ($\leftarrow$ `q(X)`). *The output of a CLP($\mathcal{FT}$) interpreter with CN for this program and goal will be*

$$\mathtt{X} \neq \mathtt{0} \wedge \mathtt{X} \neq \mathtt{1}$$

The core components of the procedure are step (2) and the satisfiability test for $\tilde{c} \wedge d_i$. The design of step (2) will determine the structure of the constraints d_i, which will be later tested for satisfiability in conjunction with $\tilde{c}$. For other standard definitions (e.g., $CLP(\mathcal{A})$ derivation, refutation tree) in the context of CLP with Constructive Negation, the reader is referred to [15, 10].

Example 2.4 *Consider the CLP($\mathcal{FT}$) program of Example 2.1 and the goal* ($\leftarrow \neg$`p`$(\mathtt{X}_1, \mathtt{X}_2)$). *According to [3], to solve this goal the CLP($\mathcal{FT}$) interpreter with CN first computes the answer for the corresponding positive goal* $\leftarrow p(X_1, X_2)$, *that is* $\exists Y(X_1 = f(a,Y) \wedge X_2 = f(b,Y))$. *This answer is then negated, leading to* $\forall Y(X_1 \neq f(a,Y) \vee X_2 \neq f(b,Y))$. *Finally, the satisfiability of this constraint is verified by suitably rewriting it to a simpler form (see Example 3.2 in the next section).*

3 Sufficient Conditions for CN

The CN scheme as presented in [15]—and briefly summarized in the previous section—has been proved to be sound and complete. These properties do not depend on the ability to simplify the negated states obtained from sub-derivations (step (2)), nor the ability to *effectively decide* the satisfiability of newly generated constraint stores (step (3)). On the other hand, any *practical* implementation of CLP with CN according to the presented scheme will require at least the ability to perform step (3): without such ability the computation will run the risk of carrying around inconsistent constraint stores, thus defeating one of the most relevant practical aspects of CLP.

Stuckey [15] identified a property of structures which ensures feasibility of Constructive Negation in CLP—guaranteeing effectiveness of its implementation. Let us consider a given structure $\mathcal{A}$ and let us assume that the collection AC of admissible constraints has been determined.

Definition 3.1 *Given a structure $\mathcal{A}$ and a collection of formulae AC (admissible constraints), the structure is* admissible closed *with respect to AC if, for each conjunction $\varphi[\bar{X}, \bar{Y}]$ of formulae from AC, there exists a formula $\psi[\bar{X}, \bar{Z}]$ such that*

- *ψ is a disjunction of conjunctions of formulae from AC, and*
- $\mathcal{A} \models \forall \bar{X}(\neg \exists \bar{Y} \varphi[\bar{X}, \bar{Y}] \leftrightarrow \exists \bar{Z} \psi[\bar{X}, \bar{Z}])$

If ψ can be effectively computed, then $\mathcal{A}$ is said to be constructively admissible closed *w.r.t. AC.*

Example 3.2 *Let $\mathcal{H}$ and AC be the Herbrand structure and the set of admissible constraints as defined in Example 2.2. $\mathcal{H}$ is admissible closed with respect to AC [15]. For instance, if $\mathcal{F} = \{a, b, c, f\}$ like in Example 2.1, then the constraint $\neg \exists Y(X_1 = f(a,Y) \wedge X_2 = f(b,Y))$ can be rewritten to $\forall W(X_1 \neq f(a,W)) \vee (X_1 = f(a,Z) \wedge \forall W(X_2 \neq f(b,W)))$.*

It is important to contrast this notion of admissible closure with the similar notion of *quantifier elimination*. We recall here the standard definition of quantifier elimination:

Definition 3.3 *A structure $\mathcal{A}$ admits* quantifier elimination *[7] if for each well-formed first-order formula φ there exists an open (i.e., quantifier free) formula ψ such that $\mathcal{A} \models \vec{\forall}(\varphi \leftrightarrow \psi)$. If the formula ψ can be effectively computed we say that $\mathcal{A}$ admits* constructive quantifier elimination.

If there exists at least a constant symbol c in $\mathcal{F}$ and the equality symbol $=$ belongs to Π_C, then we can require, without loss of generality, that $vars(\varphi) = vars(\psi)$.[3]

[3] Assume there exists ψ equivalent to φ such that $vars(\psi) \subseteq vars(\varphi)$. Then $\psi \wedge \bigwedge_{X \in vars(\varphi)} X = X$ fulfils the requirement. On the other hand, if $X \in vars(\varphi)$ and $Y \in vars(\psi) \setminus vars(\varphi)$ then $\psi[Y/X]$ is still equivalent to φ. If $vars(\varphi) = \emptyset$ then it must either be equivalent to $c = c$ or $c \neq c$.

Example 3.4 *Let* $\Pi_C = \{=\}$ *and* $\mathcal{F} = \{0, s\}$, $ar(0) = 0$ *and* $ar(s) = 1$. *Let* $\mathcal{H}$ *be the Herbrand structure associated to this language. Then* $\mathcal{H}$ *admits constructive quantifier elimination. For instance,* $\varphi \equiv \forall Y(X \neq 0 \wedge X \neq s(s(Y)))$ *can be replaced by* $\psi \equiv (X = s(0))$.

Observe that, differently from quantifier elimination, in the admissible closure condition new variables $\bar{Z}$ can be used in the new formula.

Constructive admissible closure allows one to simplify formulae of the type $\neg\exists\bar{Z}c$ (c conjunction of admissible constraints) into a formula of the type $\exists\bar{Y}c'$, where c' is a disjunction of conjunctions of admissible constraints. Since we are assuming that the satisfiability of each conjunction of formulae from AC can be decided in $\mathcal{A}$, then the admissible closure condition is sufficient to develop an effective implementation of $CLP(\mathcal{A})$ with Constructive Negation. Admissible closure differs from quantifier elimination:

- On one hand, admissible closure is weaker than quantifier elimination, since the simplification process of the former stops at the level of admissible constraints. Admissible constraints may be formulae more complex than quantifier-free formulae, since they may still contain quantifiers. For instance, if formulae of the form $\forall\bar{X}(s \neq t)$ are allowed in AC then these may occur in the formula ψ (e.g., [3]).
- On the other hand, if $\mathcal{A}$ admits quantifier elimination, we may occasionally be unable to rewrite a formula into an equivalent disjunctions of conjunctions of formulae from AC. In fact, there are cases in which some constraints (e.g., negative literals) do not belong to AC but they can occur in the formula ψ obtained by applying quantifier elimination.

The relationship between quantifier elimination and admissible closure can be better understood from the following observations.

Lemma 3.5 *Let* AC *contain all primitive constraints over* Σ *and let* $\mathcal{A}$ *be admissible closed with respect to* AC. *Then for each first-order formula* $\varphi[\bar{X}]$ *there exists a disjunction of conjunctions of formulae from* AC $\psi[\bar{X}, \bar{Z}]$ *such that* $\mathcal{A} \models \forall\bar{X}(\varphi[\bar{X}] \leftrightarrow \exists\bar{Z}\psi[\bar{X}, \bar{Z}])$. *If* $\mathcal{A}$ *is constructively admissible closed w.r.t.* AC *then* ψ *can be effectively constructed.*

Proof. [Sketch] The result can be proved by induction on the number of occurrences of logical connectives ($\wedge$, $\forall$, $\neg$) in the formula φ. Let us outline here the interesting case—i.e., dealing with $\forall$. Let us assume that φ is a formula $\forall A\varphi_1$. By inductive hypothesis we have that

$$\mathcal{A} \models \forall\bar{X}A(\varphi_1[\bar{X}, A] \leftrightarrow \exists\bar{Z}\eta[\bar{X}, \bar{Z}, A])$$

where $\eta[\bar{X}, \bar{Z}, A] = \eta_1[\bar{X}_1, \bar{Z}_1] \vee \cdots \vee \eta_k[\bar{X}_k, \bar{Z}_k]$ is a disjunction of conjunctions of AC formulae, and $\bar{X}_i, \bar{Z}_i$ are the variables from $\bar{X}$, $\bar{Z}$, A occurring in η_i. Simple considerations lead to

$$\mathcal{A} \models \forall\bar{X}(\forall A\varphi_1[\bar{X}, A] \leftrightarrow \forall A\exists\bar{Z}\eta[\bar{X}, \bar{Z}, A])$$

This can be rewritten as

$$\mathcal{A} \models \forall\bar{X}(\varphi[\bar{X}] \leftrightarrow \neg\exists A\neg\exists\bar{Z}(\eta_1[\bar{X}_1, \bar{Z}_1] \vee \cdots \vee \eta_k[\bar{X}_k, \bar{Z}_k]))$$

Each η_i is a conjunction of elements from AC, thus we can take advantage of the admissible closure of $\mathcal{A}$ and obtain

$$\mathcal{A} \models \forall\bar{X}A(\neg\exists\bar{Z}\eta_i \leftrightarrow \exists\bar{W}_i\xi_i)$$

and, without loss of generality, we can assume that $\{\bar{W}_i\} \cap \{\bar{W}_j\} \neq \emptyset$ for $i \neq j$. Thus:

$$\mathcal{A} \models \forall\bar{X}(\varphi \leftrightarrow \neg\exists A\bar{W}_1 \ldots \bar{W}_k(\xi_1 \wedge \cdots \wedge \xi_k))$$

The formula $\xi_1 \wedge \cdots \wedge \xi_k$ can be transformed into an equivalent disjunctive formula $\mu_1 \vee \cdots \vee \mu_h$, with each μ_i a conjunction of formulae from AC. The admissible closure property leads to

$$\mathcal{A} \models \forall\bar{X}(\neg\exists A\bar{W}_1 \ldots \bar{W}_k\mu_i \leftrightarrow \exists\bar{V}_i\psi_i)$$

Without loss of generality, we can assume that $\{\bar{V}_i\} \cap \{\bar{V}_j\} = \emptyset$. This leads to

$$\mathcal{A} \models \forall \bar{X}(\varphi \leftrightarrow \exists \bar{V}_1 \ldots \bar{V}_h(\psi_1 \wedge \cdots \wedge \psi_h))$$

The formula $\psi_1 \wedge \cdots \wedge \psi_h$ can be converted into the disjunctive form, leading to ψ. $\square$

Observe that the condition of Lemma 3.5 requires the presence of *at least* all positive primitive constraints in AC. Similarly, it is also possible to prove the following modified version of the lemma.

Lemma 3.6 *Let AC consist exactly of all the literals (primitive constraints over Σ and their negation) and let $\mathcal{A}$ be admissible closed with respect to AC. Then for each first-order formula $\varphi[\bar{X}]$ there exists a quantifier-free formula in disjunctive normal form $\psi[\bar{X}, \bar{Z}]$ such that $\mathcal{A} \models \forall \bar{X}(\varphi[\bar{X}] \leftrightarrow \exists \bar{Z}\psi[\bar{X}, \bar{Z}])$. If $\mathcal{A}$ is constructively admissible closed w.r.t. AC then ψ can be effectively constructed.*

The requirement that AC is contained in the set of all the literals guarantees the ability to reach a quantifier-free formula ψ. As a matter of fact, if a non-literal formula (e.g., $\forall Y(X \neq f(Y))$) was in AC, then we might not be able to simplify it any further. On the other hand, if some literals were absent from AC, then we would not be able to rewrite the formula $p(\bar{t})$ $(\neg p(\bar{t}))$ when that atom (literal) is not in AC. Lemma 3.6 allows us to see more clearly the differences between admissible closure and quantifier elimination. Let ψ_{qe} and ψ_{ac} be the formulae obtained from φ using quantifier elimination and the admissible closure procedure, respectively. Quantifier elimination leads to:

$$\begin{array}{lcll}(1) & \mathcal{A} & \models & \forall \bar{X}(\psi_{qe}[\bar{X}] \rightarrow \varphi[\bar{X}]) \wedge \\ (2) & \mathcal{A} & \models & \forall \bar{X}(\varphi[\bar{X}] \rightarrow \psi_{qe}[\bar{X}])\end{array}$$

while admissible closure leads to:

$$\begin{array}{lcll}(1) & \mathcal{A} & \models & \forall \bar{X}\bar{Z}(\psi_{ac}[\bar{X}, \bar{Z}] \rightarrow \varphi[\bar{X}]) \wedge \\ (3) & \mathcal{A} & \models & \forall \bar{X}(\varphi[\bar{X}] \rightarrow \exists \bar{Z}\psi_{ac}[\bar{X}, \bar{Z}])\end{array}$$

In both the (1) cases, $\mathcal{A} \models \vec{\exists}\psi$ implies that $\mathcal{A} \models \exists \bar{X}\varphi$. This ensures a partial satisfiability test for φ with respect to $\mathcal{A}$ whenever the test $\mathcal{A} \models \vec{\exists}\psi$ is decidable. In the case (3) of admissible closure, if $\mathcal{A} \models \exists \bar{X}\varphi$ then we also have that $\mathcal{A} \models \exists \bar{X}\exists \bar{Z}\psi_{ac}[\bar{X}, \bar{Z}]$. Thus, it holds that

$$\mathcal{A} \models \exists \bar{X}\varphi \text{ if and only if } \mathcal{A} \models \exists \bar{X}\exists \bar{Z}\psi_{ac}$$

This implies the *decidability* of the structure $\mathcal{A}$. In fact ψ_{ac} is a disjunction of conjunctions of formulae of AC—and thus it is also a disjunction of formulae of AC, since AC is closed with respect to conjunction—and by definition the satisfiability test is feasible on this kind of formulae. The situation is similar for quantifier elimination (2), but only when satisfiability of each conjunction of literals can be decided. This allows us to state the following:

Proposition 3.7 *Let AC contain all the primitive constraints over Σ. If $\mathcal{A}$ is constructively admissible closed, then $\mathcal{A}$ is decidable.*

Note that the constructive nature of the admissible closed structure is essential to the definition of a decision procedure for $\mathcal{A}$ in the above result.

Proposition 3.8 *If $\mathcal{A}$ admits constructive quantifier elimination and it is decidable whether a conjunction of literals is satisfiable in $\mathcal{A}$, then $\mathcal{A}$ is decidable.*

We conclude this analysis with a proposition from [15]:

Proposition 3.9 (Stuckey, 95) *If $\mathcal{A}$ admits quantifier elimination and AC contains all the primitive constraints over Σ and their negation, then $\mathcal{A}$ is admissible closed with respect to AC.*

An example of a structure which satisfies the requirements of Proposition 3.7 is the structure of finite trees over an infinite set of function symbols [12]. This structure has been shown to be (constructively) admissible closed with respect to the collection AC containing all the primitive constraints $s = t$ and $\forall \bar{Z}(s \neq t)$, where s, t are finite Herbrand trees. Effective procedures for transforming formulae into disjunctions of admissible constraints have been presented in [3, 12]. Thus, these procedures can be used to provide a decision procedure for such structure.

4 A Necessary Condition for CN

In this section we argue that the ability of developing an effective implementation of $CLP(\mathcal{A})$ with CN implies, under some simple assumptions on AC, that $\mathcal{A}$ is a constructively admissible closed structure. Let us consider a CLP($\mathcal{A}$) language, having a predefined class AC of admissible constraints.

Definition 4.1 *A* non-recursively complete *implementation of $CLP(\mathcal{A})$ with CN is a procedure which*

- *given as input a* non-recursive *$CLP(\mathcal{A})$ program P, and*
- *given as input an initial state $\langle c \,|\, B\rangle$, where B may contain complex subgoals and c is a satisfiable (with respect to $\mathcal{A}$) conjunction of admissible constraints,*
- *it produces as output a finite collection of satisfiable admissible constraints $\nu_1, \dots, \nu_h$ such that*

$$\mathcal{A}, P^* \models_3 \forall \bar{X}((c \wedge B) \leftrightarrow \exists Y_1 \nu_1 \vee \cdots \vee \exists Y_h \nu_h)$$

where Y_i are all the variables in ν_i not in $\langle c \,|\, B\rangle$. If $h = 0$ then the disjunction reduces to false.

The procedure developed by Stuckey for CN, with a *decidable* test for satisfiability of the constraint store (as needed in step (3) of the procedure in Section 2), fits the requirement of being a non-recursively complete implementation for $CLP(\mathcal{A})$ with CN.

Definition 4.2 *We say that AC is* literal-saturated *if for every $p \in \Pi_C$, $ar(p) = n$, and for all terms $t_1, \dots, t_n$ we have $p(t_1, \dots, t_n) \in AC$ or $\neg p(t_1, \dots, t_n) \in AC$.*

Literal-saturation does not require all literals (primitive constraints and their negation) to belong to AC. Nevertheless, it is common to have all the positive primitive constraints present in AC—which implies that AC is *literal-saturated*.

In Figure 1 we present the procedure MakeProg, used to translate any first-order formula into a $CLP(\mathcal{A})$ program P with negation and goal. This procedure can be seen as a simple extension of the procedure described in [11]. For any first-order formula Ψ written in the language of the structure $\mathcal{A}$ let us consider, without loss of generality, its equivalent form that can be easily obtained from its prenex normal form (see, e.g., [7]):

$$\Psi = (\neg)\exists \bar{X}_1 \neg\exists \bar{X}_2 \neg\exists \bar{X}_3 \dots \neg\exists \bar{X}_n (\psi_1 \vee \cdots \vee \psi_k),$$

where $vars(\Psi) = \{\bar{Y}\}$, $vars(\psi_1, \dots, \psi_k) = \{\bar{X}_1\} \cup \cdots \cup \{\bar{X}_n\} \cup \{\bar{Y}\}$, and the various ψ_i are conjunctions of literals. Calling MakeProg with initial parameters 0 and the given formula Ψ (and with P initially equal to $\emptyset$), MakeProg returns a set of CLP clauses for the program P: it generates clauses for the predicates $p_0, \dots, p_m$ where $m \leq 2(n + \sum_{i=1}^{k} h_i)$, h_i being the number of literals in ψ_i. p_0, with arguments $\bar{Y}$, where $vars(\Psi) = \{\bar{Y}\}$, is the first predicate produced. newvalue is a function that returns an unused integer number.

Example 4.3 *Consider*

$$\exists X_1 \neg\exists X_2 \neg\exists X_3 ((r_1\ (X_1, X_2) \wedge r_2\ (X_1, X_3)) \vee (r_3\ (X_1, X_3)))$$

```
MakeProg(i,Psi):
    let {X̄} be vars(Psi) (i.e., free variables in Psi);
    case Psi of
        p(t_1,...,t_n) or ¬ p(t_1,...,t_n), p ∈ Π_C and Psi ∈ AC        %% AC literal
                P := P ∪ {p_i(X̄) ← Psi.};
        p(t_1,...,t_n) or ¬ p(t_1,...,t_n), p ∈ Π_C and Psi ∉ AC        %% non-AC literal
                j := newvalue();
                P := P ∪ {p_i(X̄) ← ¬p_j(X̄).};
                MakeProg(j, ¬ Psi)
         psi_1 ∧ ··· ∧ psi_k, psi_i literals:                %% quantifier free conj.
                for i := 1 to k do
                        j_i := newvalue();
                        let {X̄_i} be vars(psi_i);
                        MakeProg(j_i, psi_i);
                P := P ∪ {p_i(X̄) ← p_{j_1}(X̄_1) ∧ ··· ∧ p_{j_k}(X̄_k).};
         psi_1 ∨ ··· ∨ psi_k, psi_i conj. of literals:        %% quantifier free disj.
                for i := 1 to k do
                        j_i := newvalue();
                        let {X̄_i} be vars(psi_i);
                        MakeProg(j_i, psi_i);
                        P := P ∪ {p_i(X̄) ← p_{j_i}(X̄_i).};
        ¬Phi, Phi not a literal:        %% negated formula
                j := newvalue();
                P := P ∪ {p_i(X̄) ← ¬p_j(X̄).};
               MakeProg(j,Phi);
        ∃Ȳ Phi, Phi not a literal:        %% existentially quantified formula
                j := newvalue();
                let {Z̄} be vars(Phi);
               P := P ∪ {p_i(X̄) ← p_j(Z̄).};
               MakeProg(j,Phi);
    end case;
```

Figure 1: Translation of a formula to $CLP(\mathcal{A})$

where $r_1, r_2, r_3 \in \Pi_C$, *and* $r_2\ (X_1, X_3) \notin AC$. *This formula is translated by* MakeProg *into the following* $CLP(\mathcal{A})$ *program with negation:*

$$
\begin{array}{ll}
\mathtt{p}_0 \leftarrow \mathtt{p}_1(\mathtt{X}_1). & \mathtt{p}_5(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3) \leftarrow \mathtt{p}_6(\mathtt{X}_1,\mathtt{X}_2)\ \wedge\ \mathtt{p}_7(\mathtt{X}_1,\mathtt{X}_3). \\
\mathtt{p}_1(\mathtt{X}_1) \leftarrow \neg\mathtt{p}_2(\mathtt{X}_1). & \mathtt{p}_5(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3) \leftarrow \mathtt{p}_8(\mathtt{X}_1,\mathtt{X}_3). \\
\mathtt{p}_2(\mathtt{X}_1) \leftarrow \mathtt{p}_3(\mathtt{X}_1,\mathtt{X}_2). & \mathtt{p}_6(\mathtt{X}_1,\mathtt{X}_2) \leftarrow \mathtt{r}_1\ (\mathtt{X}_1,\mathtt{X}_2). \\
\mathtt{p}_3(\mathtt{X}_1,\mathtt{X}_2) \leftarrow \neg\mathtt{p}_4(\mathtt{X}_1,\mathtt{X}_2). & \mathtt{p}_7(\mathtt{X}_1,\mathtt{X}_3) \leftarrow \neg\mathtt{p}_9(\mathtt{X}_1,\mathtt{X}_3). \\
\mathtt{p}_4(\mathtt{X}_1,\mathtt{X}_2) \leftarrow \mathtt{p}_5(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3). & \mathtt{p}_8(\mathtt{X}_1,\mathtt{X}_3) \leftarrow \mathtt{r}_3\ (\mathtt{X}_1,\mathtt{X}_3). \\
 & \mathtt{p}_9(\mathtt{X}_1,\mathtt{X}_3) \leftarrow \not\mathtt{r}_2\ (\mathtt{X}_1,\mathtt{X}_3).
\end{array}
$$

where $\mathtt{p}_0, \ldots, \mathtt{p}_9$ *are new predicate symbols.*

Note that only admissible constraints occur in the program generated by MakeProg. In Example 4.3, for instance, the literal $r_2\ (X_1, X_3)$ of the input formula is replaced by $\neg \not r_2\ (X_1, X_3)$ in the CLP program (the body of the predicate $\mathtt{p}_7$).

Lemma 4.4 *Let* P *be the* $CLP(\mathcal{A})$ *program generated by* MakeProg$(0,\Psi)$. *Then*

$$\mathcal{A}, P^* \models_3 \forall \bar{Y} (\Psi \leftrightarrow p_0(\bar{Y}))$$

Proof. Let us start observing that MakeProg$(0,\Psi)$ generates always $CLP(\mathcal{A})$ programs in canonical form (i.e. without non-variable terms in the heads). The result is immediate by induction on the structure of Ψ, from the definition of P^*, and from the fact that $\mathcal{A}$ is a model of equality. □

Since the program P produced by `MakeProg` is always a non-recursive program, there is no need to switch to 3-valued semantics. By using MakeProg we can prove the following:

Proposition 4.5 *If there is a non-recursively complete implementation of $CLP(\mathcal{A})$ with CN and AC is literal-saturated, then $\mathcal{A}$ is decidable.*

Proof. First, observe that the $CLP(\mathcal{A})$ programs obtained by MakeProg are always recursion-free. This guarantees that their refutation trees are always finite. Let Ψ be any first-order formula in prenex normal form written in the language of the structure $\mathcal{A}$, $vars(\Psi) = \{\bar{Y}\}$, and execute the procedure MakeProg with initial parameters 0 and the given formula Ψ. According to Definition 4.1, we know that the interpreter will be capable of producing $\nu_1, \ldots, \nu_h$ for the goal $\leftarrow p_0(\bar{Y})$ such that

$$\mathcal{A}, P^* \models_3 \forall \bar{X}(p_0(\bar{Y}) \leftrightarrow \exists Y_1 \nu_1 \vee \cdots \vee \exists Y_h \nu_h)$$

By Lemma 4.4 the disjunction of the admissible constraints $\nu_1, \ldots, \nu_h$ is a formula which is equivalent to Ψ. In particular observe that the equivalence

$$\mathcal{A}, P^* \models_3 \forall \bar{X}(\Psi \leftrightarrow \exists \bar{Y}_1 \nu_1 \vee \cdots \vee \exists \bar{Y}_h \nu_h)$$

holds also if we consider $\models$ with respect to 2-valued consequences. Thus:

- if Ψ is $\mathcal{A}$-satisfiable, then the interpreter returns an equivalent disjunction of admissible constraints;
- if Ψ is not $\mathcal{A}$-satisfiable, then the interpreter returns false, which is itself equivalent to Ψ.

Thus, the problem of verifying satisfiability of an arbitrary formula w.r.t. $\mathcal{A}$ is decidable. □

Proposition 4.6 *If there is a non-recursively complete implementation of $CLP(\mathcal{A})$ with CN and AC is literal-saturated, then $\mathcal{A}$ is constructively admissible closed with respect to AC.*

Proof. It is sufficient to call MakeProg with an initial formula Ψ of the form $\neg\exists \bar{X}_1(c_1 \wedge \cdots \wedge c_n)$ with c_i in AC. The resulting disjunction of constraints is the equivalent formula. □

The existence of a non-recursively complete implementation of $CLP(\mathcal{A})$ with CN guarantees the existence of a class AC such that $\mathcal{A}$ is admissible closed with respect to AC. As from the discussion in the previous sections, the AC class will have to meet the necessary requirements for being a class of admissible constraints—i.e., being closed under conjunction and allowing to decide $\mathcal{A} \models \exists \bar{X}\varphi$ for each φ in AC.

On the other hand, observe that the requirement imposed on the CLP interpreter—i.e., to provide a complete behavior on non-recursive programs—is very weak and, we believe, very natural—especially considering that non-recursive programs naturally generate finite derivation trees.

These results indicate that undecidable structures are not suitable to Constructive Negation in CLP. For example, the structure of *Hereditarily Finite Sets* with constraints $=$ and $\in$ (cf., e.g., [4]) does not allow full CLP with CN, since this structure is undecidable. Undecidability follows from [16], where it is proved the essential undecidability of a first-order theory based on the two axioms N (existence of a *nullset*) and W (given two sets x and y, it states the existence of a set w whose elements are x plus those of y). This implies, as from the axiomatization in [5], the undecidability of all structures of sets modeling $=$ and $\in$ according to NW, as well as structures akin to sets, such as multisets, compact-lists, and lists.

5 Conclusions

In this paper we have analyzed the problem of handling Constructive Negation in the context of CLP. Existing results from Stuckey [15] prove that the (constructive) *admissible closed* condition on the structure of the considered CLP language is sufficient to guarantee effective implementations of the CN scheme. In this paper we have proved that the admissible closed condition is actually sufficient to guarantee (under simple assumptions on the class of admissible constraints) that the underlying structure is decidable. Furthermore, we have proved that the admissible closed condition is also a *necessary* condition to realize an effective implementation of CN in CLP. These important results demonstrate the existence of meaningful classes of CLP structures (e.g., hereditarily finite sets) for which CN will not work properly.

Acknowledgments

The research presented in this paper has benefited from discussions with A. Marcone, E. Omodeo, F. Parlamento, C. Piazza, and A. Policriti, and from many useful comments from the anonymous referees, all of whom we would like to thank. E. Pontelli is partially supported by NSF Grants EIA 98-10732, CDA 97-29848, and CCR-9875279. A. Dovier and G. Rossi are partially supported by the MURST project *Certificazione automatica di programmi mediante interpretazione astratta.*

References

[1] K. R. Apt and R. Bol. Logic Programming and Negation: a Survey. *Journal of Logic Programming*, 19–20:9–71, 1994.

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

[3] D. Chan. Constructive Negation Based on the Completed Database. In R. A. Kowalski and K. A. Bowen eds., *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pages 111–125. The MIT Press, 1988.

[4] A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A Language for Programming in Logic with Finite Sets. *Journal of Logic Programming*, 28(1):1–44, 1996.

[5] A. Dovier, A. Policriti, and G. Rossi. A uniform axiomatic view of lists, multisets and sets, and the relevant unification algorithms. *Fundamenta Informaticae*, 36(2/3):201–234, 1998.

[6] W. Drabent. What is Failure? An Approach to Constructive Negation. *Acta Informatica*, 32(1):27–29, 1995.

[7] H. B. Enderton. *A mathematical introduction to logic.* Academic Press, 1973. 2nd printing.

[8] F. Fages. Constructive Negation by Pruning. *Journal of Logic Programming*, 32(2):85–118, 1997.

[9] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19–20:503–581, 1994.

[10] J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37:1–46, 1998.

[11] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[12] M. J. Maher. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In *Proceedings of 3rd Symposium Logic in Computer Science*, pages 349–357, Edinburgh, 1988.

[13] T. C. Przymusinski. On Constructive Negation in Logic Programming. In E. L. Lusk, R. A. Overbeek ed., *Logic Programming, Proceedings of the North American Conference 1989.* The MIT Press, 1989.

[14] J. C. Shepherdson. Negation in Logic Programming. In J. Minker, ed., *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, 1988.

[15] P. J. Stuckey. Negation and Constraint Logic Programming. *Information and Computation*, 1:12–33, 1995.

[16] R. L. Vaught. On a Theorem of Cobham Concerning Undecidable Theories. In *Proceedings of the 1960 International Congress*, pages 14–25. Stanford University Press, Stanford, 1962.