A general tabulation procedure for extended constraint logic programs

Carlos Viegas Damásio[1,2] and Luís Moniz Pereira[2]

[1] Universidade Aberta
Rua da Escola Politécnica 141-147
1250 Lisboa, Portugal
cd@univ-ab.pt
[2] CENTRIA, Departamento de Informática, U. Nova de Lisboa,
2825 Monte da Caparica, Portugal
lmp@di.fct.unl.pt

1 Introduction

The applications of logic programming are wide-ranging, namely for knowledge representation. Nevertheless, a mechanism for explicitly declaring the falsity of propositions is not available in the restricted language of normal logic programs. The importance of extending logic programming with an explicit form of non-classical negation, "¬", alongside the usual default or implicit one, has been stressed for use in deductive databases, knowledge representation, and non-monotonic reasoning. This has been pointed out by several authors, and an enhanced language and semantics have been proposed. For a recent study of this explicit form of negation consult [2].

However, the introduction of explicit negation permits and requires being able to reason with, or at least detect, contradictory information. Indeed, information is not only normally incomplete but contradictory as well. The idea of introducing paraconsistent reasoning in logic programming is fairly recent.

Constraint logic programming adds to knowledge espressivity by merging two declarative programming paradigms: constraint solving and logic programming. A constraint logic program is a logic program parametrized with a constraint system defining the meaning of several primitive predicates (the basic constraints). The advantage of such frameworks is the smooth integration of a domain associated with constraints, often an arithmetic domain, into a logic programming language with clear semantical properties, namely the equivalence of operational, model theoretic, and fixpoint semantics.

The starting point of this paper is our paraconsistent well-founded semantics with explicit negation ($WFSX_{\mathrm{p}}$) [1, 8], to which we add general constraints. In order to have an usable working system several issues must be addressed: soundness, completeness, termination and efficiency. We achieve this by relying on a proof procedure which integrates tabulation and constraint handling features and, in particular, constructive negation.

The tabulation mechanisms provide a solution to the problems of termination of the proof procedure, namely negative and positive recursions. The integration of constraint technology in tabulation procedures is not new, see for instance [6]. However, the systems present in the literature only take care of positive recursions, i.e. recursions which do not involve default negated literals, or are incomplete [15]. The novelty of our procedure is the simultaneous treatment of positive and negative recursions, together with general constraint handling features used to implement constructive negation, enabling us to address in this way the floundering problem in its full generality.

For lack of space we only provide a high-level description of our procedure focusing on its major features.

2 Language and Semantics

The use of variables in the language of (extended) logic programs adds expressive power to it and also eases the problem of knowledge representation, but can create problems when used under default negation.

The semantics and procedures for definite logic programs are very well understood and characterized: SLD-resolution was proven sound and complete with respect to the minimal model semantics of definite logic programs. However, the introduction of default negation in the bodies of logic program rules raises new issues and undermines the nice and desirable theoretical equivalences between the semantics and the underlying computational mechanisms of logic programming languages. Without a proper understanding of the semantics of negation, the languages cannot be said to be logical. The issue par excellence is the well-known floundering problem [19, 18].

A method which deals properly with the floundering problem must be able to construct solutions to negative goals, which justifies the terminology "constructive negation methods" introduced by [4]. There is a growing tendency to define constructive negation inside a *CLP* framework [12, 13]. This is easily recognizable in the state-of-the-art methods. This is due to the following reasons:

- There is a need for representing complex answers to goals;
- The notion of type (or sort) is important for practical applications;
- Unification is reducible to equality in the algebra of finite trees (or the Herbrand Domain) and complete axiomatizations are known [16, 7];
- CLP enhances the power of the declarative logic programming paradigm.

We assume the reader is acquainted with the terminology and notions of constraint logic program as presented, for instance, in [12, 14, 10]. A constraint domain is defined with respect to a given many-sorted language $\langle \Sigma, \Pi \rangle$, where Σ is the set of functional symbols and Π the set of predicate symbols. Additionally, we assume that the set of predicate symbols is partitioned into two disjoint sets Π_C and Π_B. Set Π_C contains the primitive constraints and set Π_B the user defined predicates. We assume that the equality predicate symbol "=" is always a primitive constraint. A constraint is a formula of the language $\langle \Sigma, \Pi_C \rangle$. Using constraints and the other predicate symbols we can define extended constraint logic programs:

Definition 1 *XCLP* **program.** Let $\langle \Sigma, \Pi \rangle$ be a many-sorted language. An extended constraint logic program, *XCLP* for short, is a finite set of rules of the form:

$$L_0 \leftarrow c \Box L_1, \ldots, L_m, not\, L_{m+1}, \ldots, not\, L_n \qquad (m, n \geq 0)$$

where c is a finite $\langle \Sigma, \Pi \rangle$-constraint. Each objective literal L_i $(0 \leq i \leq n)$ is an atomic formula A_i or the explicit complement of an atomic formula $\neg A_i$ of the language $\langle \Sigma, \Pi_B \rangle$. Default literals are of the form $not\, A$ and $not\, \neg A$, where A is an atomic formula in the same language $\langle \Sigma, \Pi_B \rangle$.

A constraint domain $\mathcal{D}$ is obtained by assigning some meaning to the primitive constraints and constraint formulae by interpreting the function symbols in Σ and the predicate symbols in Π_C, defining a satisfiability relation, "$\models$". For the sake of simplicity, we have imposed some restricting properties on the constraint domain: enumerability of the constraint domain and finite denotation of every element in that domain, entailing solution compactness properties [12, 13].

The semantics for extended constraint logic programs is obtained by the usual technique for defining new semantics for CLP programs, given a new semantics for propositional programs [17]. The model of a *XCLP* is given by the paraconsistent wellfounded model of a transformed program. This transformed program is obtained by instantiating all variables in every program rule. Only the rules with satisfiable constraint part are kept, but with that constraint removed. The program thus obtained is an ordinary ground extended logic program and its model can be determined by the propositional version of the semantics.

Definition 2. Let $\mathcal{D}$ be a constraint domain and P a *XCLP* program over the language of $\mathcal{D}$. The propositional extended logic program $G_P^{\mathcal{D}}$ is obtained as follows:

$$\begin{aligned} G_P^{\mathcal{D}} = \{\ & L_0(\tilde{d}_0) \leftarrow L_1(\tilde{d}_1), \ldots, L_m(\tilde{d}_m), not\, L_{m+1}(\tilde{d}_{m+1}), \ldots, not\, L_n(\tilde{d}_n) \mid \\ & L_0(\tilde{t}_0) \leftarrow c \square L_1(\tilde{t}_1), \ldots, L_m(\tilde{t}_m), not\, L_{m+1}(\tilde{t}_{m+1}), \ldots, not\, L_n(\tilde{t}_n) \in P \\ & \text{and } \exists \text{ a valuation } \vartheta \text{ such that } \mathcal{D} \models c\vartheta \text{ and } \forall_{0 \leq i \leq n} \mathcal{D} \models (\tilde{t}_i \vartheta = \tilde{d}_i) \qquad \} \end{aligned}$$

The paraconsistent well-founded semantics with explicit negation of program P is $WFM_p\left(G_P^{\mathcal{D}}\right)$, where WFM_p is the semantics as defined in [1, 8] for (propositional) extended logic programs.

We recall here that the paraconsistent well-founded model of a propositional extended logic program Q is the least fixpoint of $\Gamma\Gamma_s$, where Γ is the Gelfond-Lishitz operator applied to program Q [11] and Γ_s is the same operator applied to program Q_s. Program Q_s is the semi-normal version of Q, obtained from Q by adding to the (possibly empty) *Body* of each rule $L \leftarrow Body$ the default literal $not\, \neg L$, where $\neg L$ is the complement of L with respect to explicit negation. This links the semantics for extended constraint logic programs with $WFSX_{\mathrm{p}}$, where the latter is subsumed by the former. Furthermore, for normal logic programs, it coincides with *WFS*.

3 The $SLX(\mathcal{D})$ tabulation proof procedure

Allowing variables and constraints in programs introduces additional complexity in proof procedures for our semantics. One has to cope with two situations: infinite positive recursion and infinite negative recursion and how to ensure termination. In the propositional case, we have solved these problems by means of a loop-checking mechanism [1, 8]. For extended constraint logic programs we have to resort to tabulation techniques. The reader can find below an outline of the solution presented in [8].

Tabulation is nowadays well-understood for the case of definite logic programming. Recently, some extensions of tabulation methods for the case of normal logic programs have appeared in the literature [5, 3, 15]. Both [5, 3] provide a sound and complete query-evaluation procedure for non-floundering normal logic programs under wellfounded semantics. The procedures in [15] include an incremental constructive negation schema but it is only defined for finite terminating normal logic programs, and is not complete. It should be also understood that the "classical" tabulation technique and its enhancements can be adapted to *CLP* frameworks: subsumption is replaced by constraint entailment and the answers are constrained atoms. The description of one framework based on OLDT-resolution can be found in [6].

We introduce, for the first time, a general goal-oriented "algorithm", $SLX(\mathcal{D})$, for extended constraint logic programs with tabulation, constructive negation techniques, and arbitrary constraint theories. Irrespective of the novel paraconsistent semantics, none of these three features are simultaneously present in any system reported in the literature. In particular, for the more restricted class of normal logic programs under wellfounded semantics, the single proposal in the literature [15], SLGCN, is incomplete.

Our procedure is general and has a simple description in terms of three basic intuitive operations: *extension by clause*, *extension by answer*, and *extension by negative answer*. These notions were introduce by Bol and Degerstedt for the propositional case [3, 5, 9], and we are consistent with their nomenclature.

Basically, $SLX(\mathcal{D})$ corresponds to a goal-oriented tabulated evaluation of objective literals, intermixed with a goal-oriented bottom-up evaluation of default negated literals, producing a (possible) transfinite sequence of forests. A forest is an abstraction of a tabulated evaluation of a query with respect to a given set of negative answers. Each forest is made of search trees of two types: T-search trees and TU-search trees. The former is used to prove that goals are true and the latter that they are true or undefined. Each tree abstracts the notion of table used by tabulation mechanisms. One moves from one forest to the subsequent one by doing a one-step propagation of negative answers. It should be pointed out that *extension by clause* and *extension by answer* are enough to define a sound and complete procedure for definite extended constraint logic programs, i.e. without occurrences of default negation both in the query and in the program. We will resort to the program of Example 1 to illustrate our method.

Example 1. Consider the following XCLP program with respect to the domain containing three objects a, b, and c:

$$\begin{array}{llll} p(X) \leftarrow q(X). & q(X) \leftarrow not\, r(X). & r(X) \leftarrow not\, t(X), not\, s(X). & \\ s(X) \leftarrow not\, r(X). & q(a). & \neg r(X) \leftarrow X \neq a \wedge X \neq c. & t(c). \end{array}$$

In the model of this program we have $q(a)$, $q(b)$ and $q(c)$ true. Equivalently, $p(X)$ is true for all X. Regarding predicate $r/1$ one has $r(a)$ undefined and the rest of its instances false, since $t(c)$ and $\neg r(b)$ are true; $s(b)$ and $s(c)$ are true.

Suppose that we want to evaluate the query $p(X)$ in the above program. We now detail how $SLX(\mathcal{D})$ determines all the answers to the latter query. First, the query must be encoded in some form. We start the construction of the first forest in the sequence with the tree having nodes 0 and 1 (see Figure 1).

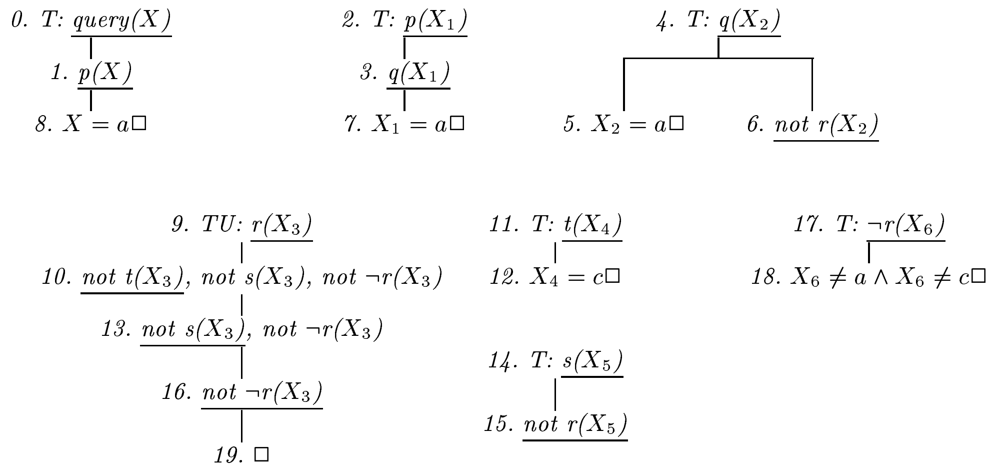

Fig. 1. Initial $SLX(\mathcal{D})$ forest for the motivative example.

Predicate *query* must not occur in the language of the program and has as arguments all free variables that appear in the query. This tree is nothing more than a table for obtaining the answers to goal $p(X)$. Mark that the root of the tree is labeled with T, meaning that the solutions should be obtained in a T context. The goals are underlined because we enforce that in each node of these trees there is one and only one selected literal, i.e. an abstraction of the usual computation rule in logic programming query evaluation procedures. We number all nodes according to a (possible) order of creation.

To obtain the solutions to the query one has to know the answers to the goal $p(X)$. To achieve that we introduce a new tree with root $p(X_1)$ in the forest. We rename all variables to avoid name clashes. This new tree is said to be obtained by *extension by clause* on node 1. Since the selected literal is an objective one and the tree to which it belongs is of type T, the new tree will also be of type T. The children of the root node of the tree for $p(X_1)$ will be the resolvents of the root literal with the program rules for $p/1$ in the program. The forest is extended with two trees, corresponding to nodes 2 to 6.

Node 5 contains an answer to goal $q(X_2)$. Obviously, this answer has to be propagated to node 3. This operation is dubbed *extension by answer*. Basically, a child of node 3 is obtained by adding the constraint $X_1 = X_2 \wedge X_2 = a$ and removing the selected goal. The first conjunct expresses parameter passing among the search trees, while the second conjunct is the answer obtained. Since X_2 does not occur free in the root node, the answer $\exists_{X_2} (X_1 = X_2 \wedge X_2 = a)$ is equivalent to $X_1 = a$, as it should be expected. By the same process the answer to $p(X_1)$ is propagated to the query tree. Thus, the trees rooted at nodes 0 and 2 are updated with nodes 7 and 8.

However there is still a node to be expanded, namely node 6. In this node the selected literal is a default one. We apply *extension by clause* and a new tree for $r(X_3)$ is introduced in the forest. But the context of the new tree should be TU instead of T, since the invoking goal is negative. In TU contexts we perform in the *extension by clause* operation the semi-normalization transformation to implement the coherence principle, i.e. we add to each body the default negation of the explicit complement of the invoked goal. The child of the node labeled with $TU : r(X_3)$ is $not\, t(X_3), not\, s(X_3), not\, \neg r(X_3)$ since there is only one rule for $r/1$ in the program. We get the tree with nodes 9 and 10.

Node 11 is obtained by *extension by clause* on node 10. Since a default literal was selected then one has to swap contexts. Now a problem arises: do we propagate the negation of the answer of node 12 to node 10 ? No. We wait for the next forest to do that if necessary. The reason is simply a matter of additional complexity necessary to handle such situations. Such "expedients" can be done at the implementational level; in such cases we call the negation process incremental. What we do is to use the answers determined in previous steps to generate the answers to default negated goals. For the base case, i.e. the initial forest, we assume all default literals undefined, corresponding to a situation of minimal knowledge. Thus, all default negated literals appearing in T contexts are failed and the ones appearing in TU contexts are successful. The operation of determining the answers for a default negated literal is called *extension by negative answer*.

The operation *extension by negative answer* is defined with respect to the set of previously obtained answers in T and TU contexts in previous forests. Thereby each forest will be parametrized by a set of constraints, the $TAns$ and $TUAns$, which will collect these old answers for the purpose of defining *extension by negative answer*. To expand a negative goal in a T context we will negate the corresponding $TUAns$ and, dually, to expand a negative goal in a TU context we will negate the corresponding $TAns$. Intuitively, in the initial forest the constraint $TAns$ for a given goal is $false$ and the constraint $TUAns$ is $true$[3]. Performing the three transformations described before, till no longer applicable, we reach the situation depicted in Figure 1. It is important to notice that node 15 does not create a new tree for $r(X_5)$ since there is already node 9 in the forest with root equivalent to $r(X_5)$ modulo variable renaming.

The construction of the second forest is deployed starting with the initial tree with root node 0. The difference to the previous forest is that the solutions to some goals are now known, and thus can be used to obtain new answers to default negated goals. Goal $r(X_3)$ has the single solution *true* and thus its negation is $false$. Therefore node 6 still has no answer and the trees rooted at nodes 0, 2 and 4 do not change in the second forest. But for $t(X_4)$ we have already obtained the answer $X_4 = c$ and it must be "consumed" by node 10. *Extension by negative answer* uses this answer to produce a new child of node 10, with label 13 (see Figure 2).

[3] In fact we use a more restricted constraint for $TUAns$, but for the purpose of explanation that is irrelevant.

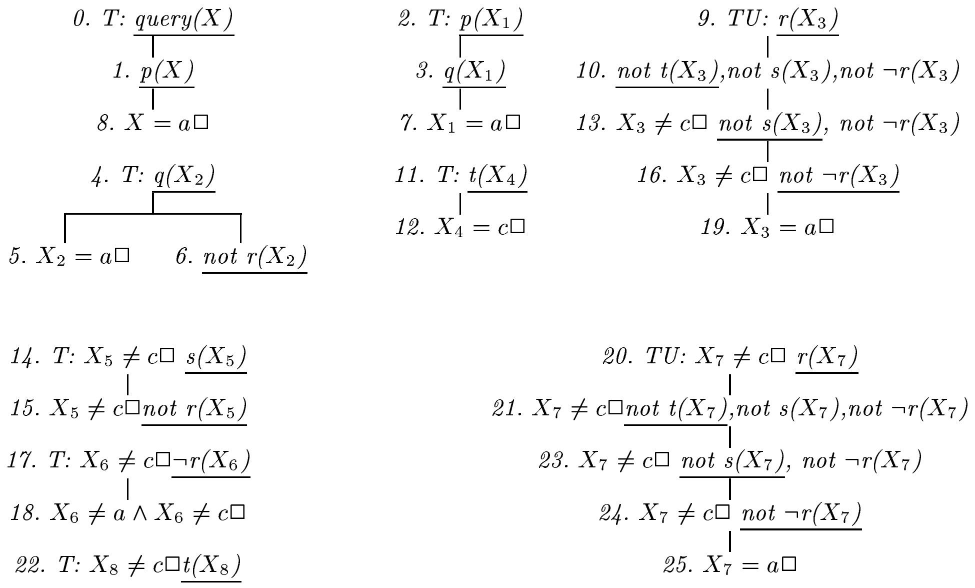

Fig. 2. Second $SLX(\mathcal{D})$ forest for motivative example.

Extension by clause on the selected literal in 13 creates the tree rooted at node 14, and having node 15 as its single child. Two problems now occur. First, there is no tree in a TU context with root equivalent to $X_5 \neq c \Box r(X_5)$ and therefore we need to create such a new tree by *extension by clause* at node 15. However, the root node is an instance of an already existing tree, namely the tree rooted at node 9. Nevertheless, in order to keep with the idea of goal-orientedness the new tree must be retained. Implementational mechanisms which can share the common parts of a tree and of a more general tree might be helpful to reduce search space.

Second, for the first time there appears in the forest the goal $X_5 \neq c \Box \underline{s(X_5)}$. Thus, which answers should be used by *extension by negative answer* to expand node 13 ? The ones of $T : s(X_5)$ in the initial forest or the ones for $T : X_5 \neq c \Box s(X_5)$? Since there are no answers for the former it really does not matter in this case. However, the situation is different for node 16, the next node in the tree. The tree introduced by *extension by clause* on node 16 is now $T : X_6 \neq c \Box \neg r(X_6)$. In the previous forest at node 17 we obtained the answer $X_6 \neq a \wedge X_6 \neq c$ for goal $\neg r(X_6)$, i.e. $TAns(\neg r(X)) \equiv X \neq a \wedge X \neq c$. Since there is no tree in that forest for goal $X_6 \neq c \Box \neg r(X_6)$ we have $TAns(X \neq c \Box \neg r(X)) \equiv false$. If we insist on using the answers to the latter goal all the work done in the initial forest is lost, i.e. we have to start the computation of $X_6 \neq c \Box \neg r(X_6)$ although being subsumed by the computation for $\neg r(X_6)$ in the first forest. To solve the problem we let *extension by negative answer* use the answers of all trees with root more general than the goal being negated. Since $\neg r(X_6)$ is more general than $X_6 \neq c \Box \neg r(X_6)$ we can use the answers of the former to constrain the latter. The second forest generated is shown in Figure 2.

In the third forest, the answers $X_3 = a$ and $X_7 = a$ of the trees rooted at nodes 9 and 20, respectively, are used to expand nodes 6 and 15. A new solution for predicate $q(X_2)$ exists and is propagated by *extension by answer* to the tree for $p(X_1)$ and to the query tree. The other trees remain the same. The updated trees are represented in Figure 3.

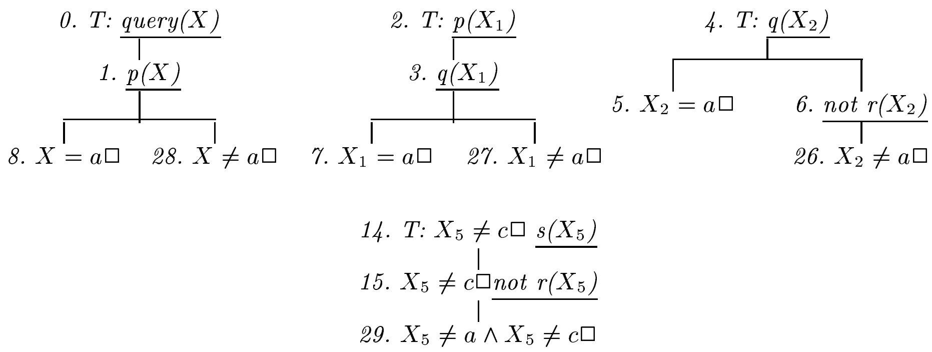

Fig. 3. Trees that changed in the third forest of the motivative example.

Mark that for the top-goal query we have finished evaluation since $X = a \vee X \neq a$ is equivalent to *true*, therefore no new answer is possible. However, this test is not performed by our algorithm. Since an answer appeared in the tree for goal $X_5 \neq c \Box s(X_5)$ it must "consumed" by the default literals at nodes 13 and 23. A fourth forest is created identical to the third forest, with the exception of constraints $X_3 \neq c$ and $X_7 \neq c$ in nodes 16 and 24 which are substituted by the more specific ones $X_3 = a$ and $X_7 = a$, respectively. The reader can check that this is in fact the fixpoint. Furthermore, the answers for each tree are according to the ground paraconsistent wellfounded model of Example 1.

In order to prove the correctness of $SLX(\mathcal{D})$ we require that the T-TU search forest sequence to increase monotonically in the following sense. Assume that the ground atom $L(\tilde{s})$ is an instance of a solution for the goal $C_1 \Box L(\tilde{s}_1)$ in a T-search tree in some T-TU search forest. In a subsequent forest the goal $C_2 \Box L(\tilde{s}_2)$ is invoked and is a restriction of $C_1 \Box L(\tilde{s}_1)$ but still $L(\tilde{s})$ is an instance of $C_2 \Box L(\tilde{s}_2)$. We expect in this case that $L(\tilde{s})$ to be an instance of solution for the goal $C_2 \Box L(\tilde{s}_2)$. Dually, for the case of TU-search trees, we expect that if some ground atom $L(\tilde{s})$ was shown false in $C_1 \Box L(\tilde{s}_1)$ it will also be false later in $C_2 \Box L(\tilde{s}_2)$.

To guarantee monotonicity of our search forest sequence we found a rather general sufficient condition on the *expansion by negative answer* operation. Consider two goals $C_1 \Box L(\tilde{s}_1)$ and $C_2 \Box L(\tilde{s}_2)$ such that $C_2 \Box L(\tilde{s}_2)$ is subsumed by $C_1 \Box L(\tilde{s}_1)$. First we require that all answers for $C_2 \Box not\, L(\tilde{s}_2)$ which were found in a T-search tree for $C_1 \Box not\, L(\tilde{s}_1)$ in forest $\mathcal{F}_{j_1}$ and are compatible with C_2 will also appear in $\mathcal{F}_{j_2}$ as children of $C_2 \Box not\, L(\tilde{s}_2)$, with $j_1 \leq j_2$. Thus we do not loose any previously found answer. This also means that the TU-search trees introduced for the children of goal $C_1 \Box not\, L(\tilde{s}_1)$ in $\mathcal{F}_{j_1}$ can be maximally reused by the children of $C_2 \Box not\, L(\tilde{s}_2)$ in $\mathcal{F}_{j_2}$.

In fact, this is what the whole problem is about: how to reuse previous trees when negating literals. Secondly, for the case of TU-search trees we impose that the answers for $C_2 \Box not\, L(\tilde{s}_2)$ are restrictions of the answers for this goal which can be extracted from the more general one $C_1 \Box not\, L(\tilde{s}_1)$. Thus, the reuse of trees is guaranteed once more. Consequently, answers in TU-search trees must decrease.

We have shown that for every constraint domain there is at least one way of performing the negation operation such that the above two conditions are both satisfied, therefore showing the correctness of the $SLX(\mathcal{D})$ proof procedure.

4 Main Contributions

In this paper we sketched the first goal-oriented sound and complete tabulation proof-procedure, designated $SLX(\mathcal{D})$, for extended constraint logic programs under *WFS* and $WFSX_{\mathrm{p}}$. The procedure's main features are the following:

- The procedures are defined for the full first-order language with constraints;
- Termination and efficiency are fostered by incorporated tabulation mechanisms;
- The floundering problem is properly handled for arbitrary constraint domains by means of constructive negation;
- For terminating non-floundering programs its data complexity is polynomial.

As stressed before, in order to obtain a practical procedure, several important implementation problems of $SLX(\mathcal{D})$ must be addressed first:

- Avoiding recomputation in the construction of the sequence of forests;
- Answers to negative literals are always one step behind, i.e. they are propagated to their consumer nodes only in the successor forest of the one where they first appeared;
- When negating a goal, the answers in all the trees in all the previous forests which are relevant for the goal must be consulted before the operation takes place;
- Search trees in T and TU contexts are very similar, but no sharing of common computations is provided by ordinary $SLX(\mathcal{D})$;
- The forest may be cluttered with several instances of some more general tree.

We pinpoint in the full version of this work [8] an optimized incremental plus sharing version of the tabulation proof procedure $SLX(\mathcal{D})$, in addition to a lot of original contributions, relations and results. It corresponds to a high-level description of existing algorithms, namely SLG [5] and SLGCN [15].

The turning point from a theoretical algorithm to a tangible one is the definition of the three incremental expansion operations, in particular *extension by negative answer*. This solves at once the recomputation problem, the delayed propagation of answers to default negated literals and the global consultation problem of *extension by negative answer*.

A sharing schema for T and TU search trees is provided in the referred work. We first tackle the problem for non-floundering programs. The net result is a procedure which generalizes SLG [5] for extended logic programs. Sharing for arbitrary programs is more complex, but proven possible. A new type of search tree is required to keep separate the computations which appear solely in T or TU contexts from the ones in common. This procedure generalizes the sharing algorithm for non-floundering programs. We also give specialized inference rules for the classes of consistent extended logic programs and of normal logic programs.

The full version of this work is available from the authors, and can be found at the following URL: $\underline{http://www.univ\text{-}ab.pt/\sim cd/investigacao/tese}$.

References

1. J. J. Alferes, C. V. Damásio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 14(1):93–147, 1995.
2. J. J. Alferes, L. M. Pereira, and T. Przymusinski. Strong and explicit negation in non-monotonic reasoning and logic programs. In *JELIA'96, European Workshop on Logic in Artificial Intelligence*. Springer-Verlag, 1996.
3. R. Bol and L. Degerstedt. Tabulated resolution for well founded semantics. In *Proc. ILPS'93*, pages 199–219. MIT Press, 1993.
4. D. Chan. Constructive negation based on the completed database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth International Conference and Symposium on Logic Programming*, pages 111–125. MIT Press, 1988.
5. W. Chen and D. S. Warren. Query evaluation under the well founded semantics. In *PODS'93*, 1993.
6. P. Codognet. A tabulation method for constraint Logic Programs. Technical report, INRIA-Rocquencourt, 1995.
7. H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.
8. C. V. Damásio. *Paraconsistent Extended Logic Programming with Constraints*. PhD thesis, Universidade Nova de Lisboa, October 1996. 350 pages.
9. L. Degerstedt. *Tabulation-based Logic Programming*. PhD thesis, Linkoping University, 1996.
10. M. Gabrielli, G. Dore, and G. Levi. Observable semantics for constraint logic programs. *Journal of Logic and Computation*, 5(2):133–171, 1995.
11. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *ICLP'88*, pages 1070–1080. MIT Press, 1988.
12. J. Jaffar and J.-L. Lassez. Constraint logic programming. Technical Report 86/73, Department of Computer Science, Monash University, 1986.
13. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th ACM Symposium in Principles of Programming Languages*, pages 111–119, 1987.
14. J. Jaffar and M. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19,20:503–581, 1994.
15. J. Y. Liu, L. Adams, and W. Chen. Constructive negation under the well-founded semantics. Technical report, Computer Science and Engineering, Southern Methodist University, 1997. Accepted for publication.
16. M. Maher. Complete axiomatization of the algebra of finite, rational and infinite trees. In *Proc. Third Symposium on Logics in Computer Science*, pages 348–357, 1988.
17. M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.
18. J. Shepherdson. Negation in logic programming for general logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, 1988.
19. J. C. Shepherdson. Negation as failure: a comparison of Clark's completed data base and Reiter's Closed World Assumption. *Journal of Logic Programming*, 1(1):51–79, 1984.