Compiling Intensional Sets in CLP

Paola Bruscoli
Università di Ancona, Ist. di Ingegneria Informatica
ANCONA (Italy)
e-mail: paola@di.unipi.it

Agostino Dovier
Università di Pisa, Dip. di Informatica
PISA (Italy)
e-mail: dovier@di.unipi.it

Enrico Pontelli
New Mexico State University, Dept. of Computer Science
LAS CRUCES (USA)
e-mail: epontell@cs.nmsu.edu

Gianfranco Rossi
Università di Parma, Dip. di Matematica
PARMA (Italy)
e-mail: gianfr@prmat.math.unipr.it

Abstract

Constructive negation has been proved to be a valid alternative to negation as failure, especially when negation is required to have, in a sense, an 'active' role. In this paper we analyze an extension of the original *constructive negation* in order to gracefully integrate with the management of set-constraints in the context of a Constraint Logic Programming Language dealing with finite sets. We show that the marriage between *CLP* with sets and constructive negation gives us the possibility of representing a general class of *intensionally defined sets* without any further extension to the operational semantics of the language. The presence of intensional sets allows a definite increase in the expressive power and abstraction level offered by the host logic language.

1 Introduction

In [7] we have shown that an increase in expressivity and abstraction capability can be obtained by embedding the basic notion of *set* in a logic programming language. By adding simple set constructors ({} and `with`) and a limited collection of predicates ($\in$, $=$, $\neq$, and $\notin$) we get a language (called {log}–read 'setlog') able to express rather complex set expressions, allowing to sensibly narrow the gap between problem specification and program development. While the construction of a declarative semantics for a

logic programming language extended with these new features is quite natural (thanks also to the many works devoted to set theory axiomatization) the design of a sound and complete operational semantics presents challenging problems. In previous works [5, 6, 7] we have developed a suitable operational framework based on an extended unification procedure (able to deal with unification between sets) and a constraint manager (used to deal with the negative distinguished predicates $\neq$- and $\notin$). These results have been successively refined and integrated in the context of Constraint Logic Programming [9], where all the new four set-predicates are uniformly manipulated as constraints. Unfortunately the expressive power of {log} is still not satisfactory, especially when applied to many real-life problems. This is due to the lack of a real *set grouping capability*, i.e. the capability of defining *intensional* set expressions of the form $\{X : p(X)\}$, where p is an arbitrary property. Simply put, {log} lacks a `setof` facility, like the one used in Prolog.

The purpose of this work is to show how intensional sets can be added to a *CLP* language dealing with sets, maintaining soundness and completeness (which are lost in the `setof` of Prolog), without imposing too severe restrictions on the admissible programs/queries (like in LDL [1], for instance). The basic idea of our approach is the reduction of the set grouping problem to the problem of dealing with *normal logic programs*, i.e. programs containing negation in the body of the clauses. This creates an interesting line of contact between negation and intensional sets (which was, by the way, already implicitely exploited by the various works on circumscription and similar forms of non-monotonic reasoning techniques).

The work is organized in three parts. The first part is dedicated to a review of the general ideas about logic programming with sets with an emphasis on the definitions related to the {log} language. The second part analyzes the core relationship between intensional sets and negation, showing a detailed algorithm which allows to convert programs containing set grouping operations into equivalent programs without set grouping (containing negative literals). The third part analyzes an extended *CLP*-like operational semantics endowing the management of negative literals. This extension has been inspired by the various works on *constructive negation* [3, 4, 12, 14], which appears to be the most suitable form of negation to be integrated in the {log} framework.

2 The {log} language

We will first recall the basic *CLP* concepts as defined in [10]. The *CLP* framework is defined using a many-sorted first order language, where $SORT = \bigcup SORT_i$ denotes a finite set of sorts. One sort is sufficient for our purposes.

By Σ and Π we denote possibly denumerable collections of function symbols and predicate symbols with their signatures. We assume there is a denumerable set of variables V. Moreover, $\Pi = \Pi_C \cup \Pi_B$ and $\Pi_C \cap \Pi_B = \emptyset$,

where Π_C and Π_B are the sets of constraint predicate symbols and programer defined predicate symbols, respectively. $\tau(\Sigma \cup V)$ and $\tau(\Sigma)$ denote the set of terms and ground terms built on $\Sigma \cup V$ and Σ (ground terms), respectively. A (Π, Σ)-*atom* is an element $p(t_1, \ldots, t_n)$ where $p \in \Pi$ is n-ary and $t_i \in \tau(\Sigma \cup V)$, $i = 1, \ldots, n$. A (Π, Σ)-*literal* is a (Π, Σ)-atom or its negation. An *atomic constraint* is a (Π_C, Σ)-atom. A (Π_C, Σ)-*constraint* is a first order formula of atomic constraints (for a more detailed description of the form of constraints used see section 5). The empty constraint will be denoted by *true*. A (Π, Σ)-*normal program* is a finite set of clauses of the form $H \leftarrow c \Box B_1, \cdots, B_n$ where c is a finite (Π_C, Σ)-constraint, H (the head) is a (Π_B, Σ)-atom and $B_1, \ldots, B_n$ (the body) are (Π_B, Σ)-literals (n $\geq$ 0). A *normal goal* is a program clause with no head and with a non-empty body. In the following (Π, Σ)-normal programs and (Π_C, Σ)-constraints will be called normal programs and constraints, respectively.

As a further notation, the symbol $\bar{}$ will denote a finite sequence of symbols. If t is a syntactic object, $FV(t)$ is the set of variables which are not explicitly quantified in t and by $t[x]$ we mean a term in which x occurs, except x itself. A sentence is a well formed formula with no free variables.

We can now define the basic {log} syntactic entities. The set of constraint predicates Π_C is fixed in {log} to be equal to $\{\in, \notin, =, \neq\}$. As shown in [9] this set of primitive set-theoretic operations suffices to define other usual set operations (such as union, intersection, ...).

Definition 2.1 *A* {log}-term *is either an* extensional *or an* intensional *term.*

- *An* extensional term *is an element of* $\tau(\Sigma \cup V)$*, i.e.*
  1. X*, for each* $X \in V$*;*
  2. $f(t_1, \ldots, t_n)$ *s.t.* $\forall i \in \{1, \ldots, n\}$, t_i *is an extensional term and* $f \in \Sigma$*;*
- *an* intensional term *is*
  1. $\{X : c \Box B_1, \ldots, B_n\}$ *s.t.* $X \in FV(c \Box B_1, \ldots, B_n)$*, where* c *is a conjunction of* (Π_C, Σ)*-atoms and* $B_1, \ldots, B_n$ *are* (Π_B, Σ)*-literals;*
  2. $f(t_1, \ldots, t_n)$ *s.t.* $\exists i, 1 \leq i \leq n$, t_i *is an intensional term,* $f \in \Sigma$*.*

Definition 2.2 *A* {log}-extended literal *is either*

- *a* {log}-literal $p(t_1, \ldots, t_n)$ *or* $\neg p(t_1, \ldots, t_n)$ *(resp. positive, negative), where* $t_1, \ldots, t_n$ *are* {log}*-terms;*
- *a RUQ-literal*[1] $(\forall X_1 \in t_1) \ldots (\forall X_n \in t_n)(c \Box \bar{B})$ *where* c *is a conjunction of* (Π_C, Σ)*-atoms,* $\bar{B}$ *is a finite sequence of* (Π_B, Σ)*-literals,* $X_j s$ *are pairwise distinct variables, and* $t_i s$ *are* extensional terms *s.t.* $X_j \cap FV(t_i) = \emptyset$ *for* $i \leq j$*.*

A few words about *RUQ*-literals are in order. First, recall that $(\forall x \in t)\varphi$, φ any first order formula, is a shorthand for $\forall x\, (x \in t \rightarrow \varphi)$. Moreover, observe that the condition $X_j \cap FV(t_i) = \emptyset$ prevents us from writing formulas such as $(\forall v \in x)(\forall x \in y)\varphi$, where the two occurrences of x would refer to

[1] *RUQ* stands for Restricted Universal Quantifier.

two distinguished variables. In [6] we have proved the equivalence between {log} programs containing restricted universal quantifications and {log} programs which are *RUQ*-free. Each occurrence of a *RUQ* may be removed by performing a simple syntactic translation. Thanks to this we can assume from now on that the program on which we are working does not contain any *RUQ*.

Definition 2.3 *A* {log}-clause *is a normal clause* $A \leftarrow c\Box B_1, \ldots, B_n$ *where* A *is a positive literal,* c *is a conjunction of* (Π_C, Σ)*-atoms and* $B_1, \ldots, B_n$ *are* {*log*}*-extended literals. A* {log}-goal *is a* {*log*}*-clause with empty head. A* {log}-program *is a finite set of* {*log*}*-clauses.*

As a notational convenience we will write $\{X : B_1, \ldots, B_n\}$ and $A \leftarrow B_1, \ldots, B_n$ whenever c is *'true'* (the empty constraint).

In order to be able to deal with extensional sets, as well as standard Herbrand terms, the following two functional symbols are assumed to be always present in Σ [5]:

- $\emptyset$, nullary, to be interpreted as the empty set;
- a binary function symbol, `with` (used as an infix left associative operator), to be interpreted as follows: s `with` t stands for the set that results from adding t as a new element to the set s.

In view of the intended interpretation, an extensional term of any of the forms, $\emptyset$ or X `with` t_n `with` $\cdots$ `with` t_1 or k `with` t_n `with` $\cdots$ `with` t_0, $n \leq 0$, X variable, and k a non variable extensional term with main functor different from `with`/2, is called a *set term*. The term k is the *kernel* of the set and a set term where k is not $\emptyset$ is intended to designate a *colored set* based on the kernel k[2]. For the sake of simplicity special syntactic forms are introduced to designate set terms: $\{t_1, \ldots, t_n | s\}$ stands for s `with` t_n `with` $\cdots$ `with` t_1 and $\{t_1, \ldots, t_n\}$ stands for $\emptyset$ `with` t_n `with` $\cdots$ `with` t_1 where $n \geq 1$ and $s, t_1, \ldots, t_n$ are terms. $\{\}$ is a syntactic sugar for $\emptyset$. For example:

- {}, {1,X,Y,2}, {1,1,{2,{}},f(a,{b})}, and any term $\{t_1, \ldots, t_n | R\}$ with a 'tail' variable R, are set terms;
- f(a,{5}), i.e. f(a,{} `with` 5), is an extensional term, but not a set term;
- {a | f({b})} is a colored set term based on the kernel f({b});
- {X : X ≠ 1 □ p(X)}, f(Y,{Z : Z ∈ Y □ p(Z,W)}) are intensional set terms.

Here are a few sample {log} programs (the precise meaning of these programs will be clarified in the next section).

- Checking membership of an element to the set Set1 \ Set2:
  in_difference(X, Set1, Set2) ← X ∈ Set1 ∧ X ∉ Set2 □.
- Sorting a set into an ordered list:
  quicksort({ }, []).
  quicksort(S, L) ← X ∈ S □

[2]Colored set terms do not designate sets of any conventional kind. Nevertheless, we deem it convenient to always regard such terms as legal set terms when $t_1, \ldots, t_n, k$ are legal, to make the language structure absolutely uniform and the inference mechanisms (e.g. unification) more straightforward.

$\mathsf{quicksort}(\{\mathsf{Y} \ : \ \mathsf{Y} \in \mathsf{S} \ \square \ \mathsf{less}(\mathsf{X},\mathsf{Y})\},\mathsf{L1}),$
$\mathsf{quicksort}(\{\mathsf{Y} \ : \ \mathsf{Y} \in \mathsf{S} \ \square \ \mathsf{less}(\mathsf{Y},\mathsf{X})\},\mathsf{L2}),$
$\mathsf{append}(\mathsf{L1},[\mathsf{X}|\mathsf{L2}],\mathsf{L}).$

- Computing the set of prime numbers less than a given limit N:
  $\mathsf{primes}(\mathsf{N},\mathsf{S}) \leftarrow \mathsf{S} = \{\mathsf{X} : \mathsf{between}(1,\mathsf{N},\mathsf{X}),\mathsf{prime}(\mathsf{X})\}.$
  $\mathsf{prime}(\mathsf{X}) \leftarrow \mathsf{S} = \{\mathsf{Y} : \mathsf{between}(1,\mathsf{X},\mathsf{Y})\}\square(\forall \mathsf{Z} \in \mathsf{S})(\mathsf{non\_div}(\mathsf{Z},\mathsf{X})).$
  $\mathsf{between}(\mathsf{A},\mathsf{B},\mathsf{C}) \leftarrow \mathsf{less}(\mathsf{A},\mathsf{C}) \ , \ \mathsf{less}(\mathsf{C},\mathsf{B}).$

3 Compiling intensional sets

In [5] we argued that intensional sets can be programmed in a logic language with sets like {log}, provided the language supplies either a set grouping mechanism or some form of negation in goals and clause bodies. This allows us, on one hand, to consider intensional sets as a syntactic extension to be dealt with a simple preprocessing phase, and, on the other hand, not to be concerned with intensional sets when defining the semantics of our language.

Let us try, first of all, to understand why the negative information representable in {log} by the use of $\neq$ and $\notin$ is not sufficient for a satisfactory definition of a set grouping mechanism, and full negation is required instead.

An intensional set $\mathcal{S}$ can be defined in the following equivalent ways:

$$\{X : p(X)\} \ = \ \mathcal{S} \leftrightarrow \forall X(X \in \mathcal{S} \leftrightarrow p(X))$$
$$\{X : p(X)\} \ = \ \mathcal{S} \leftrightarrow \forall X(X \in \mathcal{S} \rightarrow p(X)) \wedge \forall X(p(X) \rightarrow X \in \mathcal{S}).$$

As we can see, a set grouping feature requires the ability to perform *restricted universal quantification* as well as universal quantification of the solutions of an arbitrary predicate.

Though {log} supports restricted universal quantification [6], it is unable to express the other form of quantification. However, one can observe that:

$$\forall X(p(X) \rightarrow X \in \mathcal{S}) \leftrightarrow \forall X(\neg p(X) \vee X \in \mathcal{S}) \leftrightarrow \neg\exists X(X \notin \mathcal{S} \wedge p(X)).$$

The outcome shows that what we need is just a form of negation (notice that the negated formula can be easily expressed by using a new clause with a local variable).

The correlation between set grouping and negation can be further shown by the following example. Suppose that given a natural number N we want to define a predicate returning the greatest prime number X in its decomposition in prime factors. We use an intensional construct to collect the prime divisors of N as follows:

$$\mathsf{max}_{\mathsf{pdiv}}(\mathsf{N},\mathsf{X}) \ \leftarrow \ \mathsf{max}(\{\mathsf{Y} \ : \ \mathsf{pdiv}(\mathsf{N},\mathsf{Y})\},\mathsf{X}).$$
$$\mathsf{max}(\mathsf{S},\mathsf{M}) \ \leftarrow \ \mathsf{M} \in \mathsf{S} \ \square \ (\forall \mathsf{Z} \ \in \ \mathsf{S})(\mathsf{geq}(\mathsf{M},\mathsf{Z})).$$

where pdiv and geq define the divisibility relation and the greatest or equal relation, respectively.

In order to compute $\mathsf{max}_{\mathsf{pdiv}}$ we should be able to collect the set of prime divisors computed by the predicate pdiv and, at the same time, to reject any partial solution, namely any element in the powerset of the set of all possible solutions. This could be implemented as follows:

$\mathsf{setof_{pdiv}(S,N)} \leftarrow (\forall \mathsf{Y} \in \mathsf{S})(\mathsf{pdiv(N,Y)}), \neg \mathsf{partial_{pdiv}(S,N)}.$
$\mathsf{partial_{pdiv}(S,N)} \leftarrow \mathsf{Z} \notin \mathsf{S} \square \mathsf{pdiv(N,Z)}.$

with the call to max in the clause defining $\mathsf{max_{pdiv}}$ replaced by $\mathsf{setof_{pdiv}(S,N)}$, $\mathsf{max(S,X)}$.

Replacement of intensional set terms by the setof predicates which allow the corresponding extensional sets to be constructed is performed by a two steps program transformation. This process will transform a given $(\Pi_C \cup \Pi_B, \Sigma)$-program into the equivalent $(\Pi_C \cup \Pi'_B, \Sigma)$-program where Π'_B contains Π_B and all the new predicate symbols which are required to express both the *discriminant* part of intensional sets and set grouping (along with the new predicate symbols generated by *RUQ*'s translation).

The first step leads the source code to a *normal form* where all variable instantiations in clauses and goals are expressed as constraints and each *discriminant* $(c\square\bar{B})$ of intensional terms is expressed by a unique predicate symbol. Such a predicate symbol has arity equal to $|FV(c\square\bar{B})|$, and it is defined by a unique clause having the corresponding discriminant as its body.

Step 1 - Program normalization

Let C be the {log}-clause

$$p(s_1, \ldots, s_m) \leftarrow c\square A_1(t^1_1, \ldots, t^1_{n_1}), \ldots, A_r(t^r_1, \ldots, t^r_{n_r})$$

where s_i's and t^i_j's are terms, and $A_i(t^i_1, \ldots, t^i_{n_i})$ are {log}-literals (as it ensues from the discussion following def. 2.2, there is no need here to consider *RUQ*-literals).

Repeatedly perform the following actions until none applies.

- Replace C by the equivalent clause

$$p(X_1, \ldots, X_m) \leftarrow c \wedge \bigwedge_{i=1,\ldots,m} (X_i = s_i) \wedge \bigwedge_{\substack{i=1,\ldots,r \\ j=1,\ldots,n_i}} (X^i_j = t^i_j)\square$$
$$A_1(X^1_1, \ldots, X^1_{n_1}), \ldots, A_r(X^r_1, \ldots, X^r_{n_r})$$

  where X_i's, X^i_j's and Y^i_j's are new distinct variables.
- Replace each atomic constraint $s\,\pi\,t$, where $\pi \in \Pi_C$ and s and/or t are intensional terms, by the constraint

$$s'\,\pi\,t' \wedge \bigwedge_{i=1}^{m} (S_i = s_i) \wedge \bigwedge_{j=1}^{n} (T_j = t_j),$$

  where s_i's and t_j's are all the basic intensional terms occurring in s and t respectively, s' and t' are the extensional terms obtained by replacing the intensional terms s_i's and t_j's in s and t with the new variables S_i's and T_j's respectively.
- Replace each atomic constraint of the form $X = \{Y : c\square\bar{B}\}$ by the constraint

$$X = \{Y : \delta(Y, Z_1, \ldots, Z_m)\},$$

  where $\{Y, Z_1, \ldots, Z_m\} = FV(c\square\bar{B})$, and δ is a newly generated predicate symbol, and add to the program the new clause

$$\delta(Y, Z_1, \ldots, Z_m) \leftarrow c\square\bar{B}.$$

Step 2 - Eliminating intensional set terms

The second step is intended to remove intensional set terms from a normalized program according to the general idea for implementing set grouping sketched at the beginning of this section. For each predicate symbol δ generated by the normalization step to represent discriminants in intensional set terms, two new predicate symbols $setof_\delta$ and $partial_\delta$ are introduced, and their corresponding {log} definitions added to the generated program, according to the following transformation rule:

- Replace each normalized clause of the form

$$h(\bar{Y}) \leftarrow c \wedge X_1 = \{X : \delta(X, \bar{Z})\} \square \bar{B}$$

by the set of clauses

$$\begin{array}{lcl} h(\bar{Y}) & \leftarrow & c \square setof_\delta(X_1, \bar{Z}), \bar{B}. \\ setof_\delta(X_1, \bar{Z}) & \leftarrow & (\forall X \in X_1)\delta(X, \bar{Z}), \neg partial_\delta(X_1, \bar{Z}). \\ partial_\delta(X_1, \bar{Z}) & \leftarrow & V \notin X_1 \square \delta(V, \bar{Z}). \end{array}$$

For example, the definition of the predicate $\mathsf{max_{pdiv}}$ shown above is first replaced by the following clauses:

$$\begin{array}{lcl} \mathsf{max_{pdiv}(N, X)} & \leftarrow & \mathsf{Z = \{Y : \delta(Y, N)\} \square max(Z, X)}. \\ \delta\mathsf{(Y, N)} & \leftarrow & \mathsf{pdiv(N, Y)}. \end{array}$$

Then (second step), the normalized definition of the predicate $\mathsf{max_{pdiv}}$ is replaced by:

$$\mathsf{max_{pdiv}(N, X)} \leftarrow \mathsf{setof_\delta(Z, N), max(Z, X)}.$$

adding the clauses defining $\mathsf{setof_\delta(Z, N)}$ to the transformed program.

4 Negation

Different forms of negation can be introduced in logic programming, most of them based on the notion of *Completed Program* [13]. In particular, the well-known *negation as failure* technique could be used to handle negation in {log} programs. Negation as failure has various advantages, related in particular to its simplicity: it is quite easy to come up with a reasonable and fairly efficient implementation. On the other hand, negation as failure has various drawbacks, mostly related to the strict requirements necessary in order to maintain soundness and completeness results. Just to point out one of such restrictions: soundness of the *SLD* + negation as failure resolution rule is guaranteed only if the program and the goal are *allowed*. Allowedness requires that every variable occurring in a clause occurs in a positive literal in the body of the clause. While this restriction may be acceptable in many contexts (e.g. deductive databases), in our framework it may create some serious complications. Just to mention one, the algorithm which translates Restricted Universal Quantifiers to pure {log} programs [5, 6] generates clauses which do not satisfy the allowedness restriction.

Various proposals have been made in the last few years to get around the inability of providing computed answers to non-ground negative literals in negation as failure. The approach that we are following here is the one called

constructive negation [3, 4]. As we will see later on, this approach gracefully integrates with {log}. The basic idea behind constructive negation is the following. Given a program P the set of all the solutions to a goal ($\leftarrow G$), $\sigma_1, \ldots, \sigma_n$, is such that $Comp(P) \models G \leftrightarrow \sigma_1 \vee \cdots \vee \sigma_n$ where $Comp(P)$ is the completed version of the program P. Taking the negation of the formula, $Comp(P) \models \neg G \leftrightarrow \neg(\sigma_1 \vee \cdots \vee \sigma_n)$, gives an idea of how to obtain a solution to a negative literal. The key point is the development of an effective procedure to extract actual solutions from the negation $\neg(\sigma_1 \vee \cdots \vee \sigma_n)$. The description given by Chan [3, 4] is specialized for the case of pure logic programming (each σ_i is a substitution). The relations between constructive negation and *CLP* have been studied in [14].

5 Constraints

In [9] we have shown that {log} can be conveniently viewed as an instance of the general *CLP* scheme [10]. To this purpose first we have fixed Σ and Π_C to be equal to $\{\emptyset, \texttt{with}, \ldots\}$ and $\{=, \neq, \in, \notin\}$, respectively, and then we have defined a suitable algebraic structure $\mathcal{S}$, whose domain S is defined as the quotient set of the Herbrand universe w.r.t. a suitable congruence over $\tau(\Sigma)$ to abstract from the ordering of the elements of `with`-based terms [9].

Also we have characterized the kind of sets to be handled via axioms of a suitable set theory *Set* [6, 9], from which it is easy to derive, among others, the two fundamental properties of the set construct `with`, namely *Right permutativity* (i.e. $(X \,\texttt{with}\, Y)\,\texttt{with}\, Z = (X \,\texttt{with}\, Z)\,\texttt{with}\, Y$) and *Right absorption* (i.e. $(X \,\texttt{with}\, Y)\,\texttt{with}\, Y = X \,\texttt{with}\, Y$).

It has been proved that the structure $\mathcal{S}$ is *solution compact*. Moreover, it has been proved that the satisfaction complete theory *Set* and the structure $\mathcal{S}$ correspond. Thus, having developed also a suitable constraint satisfaction procedure (cf. [9]), we have been in the position of using the ordinary machinery of the general *CLP* scheme to implement both the algebraic and the logical derivation (actually, the implementation of {log} described in [8] is a specialized version of the *CLP* logical derivation for {log} programs).

In this section we show how the {log} constraint satisfiability procedure and the general *CLP* operational semantics need to be modified in order to accommodate for constructive negation. More precisely, the resolution procedure needs to deal with positive atoms as well as with the negative ones (i.e. literals of the form $\neg p(\bar{t})$, where p is a user-defined predicate), whereas the constraint solver needs to deal with positive atomic constraints of the form $t_1 \in t_2$ or $t_1 = t_2$, as well as with negative constraints of the form $\forall \bar{X}\ (t_1 \notin t_2)$ or $\forall \bar{X}\ (t_1 \neq t_2)$ where $\bar{X}$ represents some (eventually none) of the variables in t_1, t_2.

Indeed the latter kind of constraints, though not present in the program generated by the transformation process described in the previous section, may be generated during the computation due to the presence of negation (i.e. dealing with negation leads to explicit universal quantifications).

We first examine the constraint satisfaction procedure and then the extended resolution procedure, devoting special care to the way constructive negation is dealt with.

The key notion of the constraint satisfiability procedure developed for {log} [9] is represented by the concept of *normal form* (or, following the nomenclature used in [6, 7, 9], *canonical form*) for a constraint.

Definition 5.1 *Given a constraint C, an atomic constraint c in C is in* normal form *whenever it satisfies one of the following conditions:*
a. $c \equiv X = t$ *and X is a variable which does not occur elsewhere in C;*
b. $c \equiv t \notin X$ *and X is a variable which does not occur in t;*
c. $c \equiv X \neq t$ *and X is a variable which does not occur in t.*[3]

A constraint C *is in* normal form *if either it is 'false' or all the atomic constraints in it are in normal form.*

It can be proved that if C is in normal form and other than *'false'* then C is satisfiable in the theory *Set* (or, equivalently, solvable in the structure $\mathcal{S}$).

The approach used in {log} to detect satisfiability of a generic constraint C, therefore, is based on the use of a procedure, called $\mathcal{SAT}$, which tries to transform C into an equisatisfiable disjunction of constraints in normal form (whose satisfiability is guaranteed). The transformation of C to a normal form is performed by using the following non-deterministic function

$$\begin{array}{rcl} \mathsf{step}(C) & = & \textbf{if } \textit{'false'} \text{ in } C \textbf{ then } \textit{'false'} \\ & & \textbf{else } \mathsf{notequal}(\mathsf{notmember}(\mathsf{unify}(\mathsf{member}(C)))). \end{array}$$

Each of the functions unify, notmember, and notequal (see [9]) reduces $=$-constraints, $\notin$-constraints, and $\neq$-constraints to their normal forms, respectively, whereas $\in$-constraints are completely eliminated by member by replacing them with suitable $=$-constraints. Since each of these functions may produce constraints of a different form (for example *notequal* may produce $\in$-constraints), then step needs to be iterated as long as a fixpoint is reached:

$$\begin{array}{rcl} \mathcal{SAT}(C) & = & \textbf{while } \mathsf{step}(C) \neq C \textbf{ do} \\ & & \quad C = \mathsf{step}(C); \\ & & \textbf{return } C. \end{array}$$

It has been proved [9] that this fixpoint is always reached in a finite number of steps and that each constraint that $\mathcal{SAT}$ non-deterministically computes is in normal form.

The key result proved in [9] is that given a constraint C, then

$$\mathit{Set} \vdash C \leftrightarrow \exists(C_1 \vee \cdots \vee C_n),$$

where $C_1, \cdots, C_n$ is the collection of constraints in normal form computed by $\mathcal{SAT}$. Therefore, C is satisfiable if and only if there exists a non-deterministic choice such that $\mathcal{SAT}(C) \neq$ *'false'*.

[3] For the sake of simplicity, hereafter, we will not consider set terms based on a kernel other than $\emptyset$. Actually, the results proved here are still valid when considering colored sets, too, provided the constraint satisfiability procedure is suitably extended to accommodate for this more general case as shown in [9].

When dealing with negation we need to update these definitions, due to the possibility, as announced before, of generating explicit universal quantification over negative constraints.

First of all, a constraint C may contain not only (Π_C, Σ)-atoms but also universally quantified formulae of the of the form $\forall \bar{Z}(X \neq t)$ or $\forall \bar{Z}(t \notin X)$. Thus the previous definition of normal form (def. 5.1) must be updated by replacing cases *(b)* and *(c)* with the following new ones:

b'. $c \equiv \forall \bar{Z}(t \notin X)$, X *does not occur in* t *nor in* $\bar{Z}$ *and, if* $t \equiv Y$, Y *variable, then* Y *should not occur in* $\bar{Z}$*;*

c'. $c \equiv \forall \bar{Z}(X \neq t)$, X *and* t *as above.*

Dealing with constraints of the form $\forall \bar{Z}(X \neq t)$ requires the ability to manage disjunctions of constraints as well as conjunctions. Indeed, to solve a constraint of the form $\forall \bar{Z}\ (f(t_0, \ldots, t_n) \neq f(s_0, \ldots, s_n))$, where f is different from `with` and $\{\bar{Z}\} \subseteq FV(t_0, \ldots, t_n, s_0, \ldots, s_n)$, one needs to solve the disjunction $\forall \bar{Z}(t_0 \neq s_0 \vee \cdots \vee t_n \neq s_n)$.

Unfortunately, the following general result, used for instance by Chan [3], does not hold in our theory *Set*.

> *Let* t *be a term such that* $FV(t) = \{x_1, \ldots, x_n\}(= \{\bar{x}\})$*, and* D *be a f.o.f. such that* $\{y_1, \ldots, y_m\} = FV(D) \setminus \{x_1, \ldots, x_n, u\}$*, then* $\vdash_{EQ} \forall u\ (\forall \bar{x}\bar{y}\ (u \neq t \vee D) \leftrightarrow \forall \bar{x}\ (u \neq t) \vee \exists \bar{x}\ (u = t \wedge \forall \bar{y}\ D))$. (1)

In particular, the $\leftarrow$ part of (1) is not true in our theory[4]. As an example, the formula $\forall xyvw\ (\{\emptyset, \{\emptyset\}\} \neq \{x, y\} \vee x \neq \{w|v\})$ (equivalent to $\forall xy\ (\{\emptyset, \{\emptyset\}\} \neq \{x, y\} \vee \forall vw(x \neq \{w|v\}))$) is not true in *Set* (e.g., by taking $x = \{\emptyset\}, y = \emptyset$), while $\forall xy\ (\{\emptyset, \{\emptyset\}\} \neq \{x, y\}) \vee \exists xy\ (\{\emptyset, \{\emptyset\}\} \neq \{x, y\} \wedge \forall vw(x \neq \{w|v\}))$ is true in *Set* (e.g., by taking $x = \emptyset, y = \{\emptyset\}$).

The problem here originates from the fact that the *uniqueness* property of the mgu which holds in the standard theory *EQ* and is exploited to prove (1), does not hold in our theory *Set*. The following lemma that can be proved to hold in *Set* will provide us an alternative to (1).

Lemma 5.2 *Let* t *be a term such that* $FV(t) = \{x_1, \ldots, x_n\}(= \{\bar{x}\})$*, and* D *be a f.o.f. such that* $\{y_1, \ldots, y_m\} = FV(D) \setminus \{x_1, \ldots, x_n, u\}$*, then* $\vdash_{Set} \forall u \left(\forall \bar{x}\bar{y}(u \neq t \vee D) \leftrightarrow \forall \bar{x}(u \neq t) \vee \exists \theta_1 \cdots \theta_k \bigwedge_{1 \leq i \leq k}(u = t^{\theta_i} \wedge \forall \bar{y}\ D^{\theta_i})\right)$ *where* $\theta_1 \cdots \theta_k$ *are independent substitutions and* $\{\bar{x}\} \subseteq \mathrm{dom}(\theta_{\mathrm{i}})$.

As an effective application of this lemma, one can prove, for instance, that the following equivalence holds:

$$\forall u \left(\begin{array}{l} \forall xy(u \neq \{x, y\} \vee \varphi) \leftrightarrow \\ \forall xy(u \neq \{x, y\}) \vee \exists xy \left(u = \{x, y\} \wedge \varphi \wedge \varphi^{\{x \mapsto y, y \mapsto x\}}\right) \end{array} \right)$$

This result will be exploited in the constraint satisfiability procedure, in particular in that part of the procedure aimed at simplifying $\neq$-constraints (function notequal), and in the extended resolution procedure. The function

[4]For the other direction, notice that $\forall x, y\ (\varphi(x) \vee \psi(x, y)) \rightarrow \forall x\ \varphi(x) \vee \exists x\ (\neg\varphi(x) \wedge \forall y\ \psi(x, y))$ is a theorem of predicate calculus.

notequal is shown in detail in the following (notice that $C_{\neq}$ is used in the function to denote the part of the given constraint C containing $\neq$-constraints only). $\notin$-constraints can be dealt with in a similar, though simpler way, so the pertaining function notmember is not shown here. Nothing needs to be changed w.r.t. [9] in the treatment of $=$ and $\in$ constraints.

function notequal(C);
if $C_{\neq}$ is in canonical form **then return** C
else choose any c not in canonical form in $C_{\neq}$; let $C = C' \wedge c$;
case c **of**

1. $\forall Z_1 \ldots Z_n\ (s \neq t)$, and $Z_1, \ldots, Z_k$ do not belong to $FV(s \neq t)$, $k > 0$:
   return notequal($C' \wedge \forall Z_{k+1} \ldots Z_n (s \neq t)$);
2. $\forall \bar{Z}\ (f(t_1, \ldots, t_n) \neq g(s_1, \ldots, s_m))$, f and g are different function symbols: **return** notequal(C');
3. $\forall \bar{Z}\ (f(t_0, \ldots, t_n) \neq f(s_0, \ldots, s_n))$, f is different from `with`, and $\{\bar{Z}\} \subseteq FV(t_0, \ldots, t_n, s_0, \ldots, s_n)$:
   return notequal($C' \wedge \forall \bar{Z}(t_0 \neq s_0 \vee \cdots \vee t_n \neq s_n)$);
4. $\forall \bar{Z}\ (s \neq t \vee D)$, and $\{\bar{Z}\} \subseteq FV(s \neq t \vee D)$, $\{\bar{Z}\} = \{\bar{Z}_D\} \cup \{\bar{Z}_C\}$, $\{\bar{Z}_D\} = \{\bar{Z}\} \cap FV(D)$, $\{\bar{Z}_C\} \cap \{\bar{Z}_D\} = \emptyset$:
   if unify(s, t) fails **then return** C'
   else let θ_i, $i = 1, \ldots, k$ be the mgu's of s and t such that
   $\{\bar{Z}_C\} \subseteq dom(\theta_i)$ and $Z_i = FV(s^{\theta_i} \neq t^{\theta_i}) \setminus \{\bar{Z}_C\}$:
   return notequal($C' \wedge \bigwedge_{i=1}^{k} \forall \bar{Z}_i \bar{Z}_D D^{\theta_i}$);
5. $f \neq f$, f is a constant: **return** *false*;
6. $X \neq X$ or $\forall X (X \neq X)$, X is a variable: **return** *false*;
7. $\forall \bar{Z}\ (t \neq X)$ and t is not a variable: **return** notequal($C' \wedge \forall \bar{Z}(X \neq t)$);
8. $\forall \bar{Z}_1 X \bar{Z}_2 (X \neq t)$ and t is not a variable, **or** $\forall XY\,(X \neq Y)$ **or** $\forall XY\,(Y \neq X)$: **return** *false*;
9. $\forall \bar{Z}(X \neq f(t_1, \ldots, t_n))$, f is different from `with` and $X \in FV(f(t_1, \ldots, t_n))$, **or** $\forall \bar{Z}\ (X \neq h\ \texttt{with}\ s_m \ldots\ \texttt{with}\ s_0)$, h is a variable or $\emptyset$ and $X \in FV(s_0) \cup \ldots \cup FV(s_m)$: **return** notequal($C'$);
10. $\forall \bar{Z}\ (X \neq X\ \texttt{with}\ t_n \ldots\ \texttt{with}\ t_0)$:
    return notequal($C' \wedge \forall \bar{Z}\ (t_0 \notin X \vee \cdots \vee t_n \notin X)$);
11. $\forall \bar{Z}\ (r \neq s)$, where $s \equiv h\ \texttt{with}\ t_n \ldots\ \texttt{with}\ t_0$ and $s \equiv k\ \texttt{with}\ t'_m\ \texttt{with} \cdots\ \texttt{with}\ t'_0$, h, k terms with main functor different from `with`:
    select non-deterministically one of the following actions (let X and N denote new variables):
    i. take a solution θ for the constraint $X \in r$; **case** θ **of**
       a. $\{X \mapsto t_i\}$, $i = 0, \ldots, n$: **return** notequal($C' \wedge \forall \bar{Z}\ (X^{\theta} \notin s)$)
       b. $\{h \mapsto N\ \texttt{with}\ X\}$, h variable not in $\bar{Z}$:
          return notequal($C' \wedge\ (h =\ N\ \texttt{with}\ X) \wedge \forall \bar{Z}\ (X \notin s)$)
       c. $\{h \mapsto N\ \texttt{with}\ X\}$, h variable in $\bar{Z}$:
          return notequal($C' \wedge \forall \bar{Z}' N\ (X \notin s^{\{h \mapsto N\ \texttt{with}\ X\}})$)
          where $\bar{Z}'$ is the list of variables obtained by eliminating h from $\bar{Z}$.
    ii. take a solution θ for $X \in s$; symmetrical to the previous case.

A remark on action 4 of the function notequal. If the unification between s and t fails then $\forall \bar{Z}_C(s \neq t)$ is always true and the selected constraint c can be deleted. If, on the contrary, the unification between s and t terminates successfully, yielding the complete set of unifiers $\{\theta_1, \ldots, \theta_k\}$, then lemma 5.2 is applied to simplify the selected constraint c. Actually, notice that a weaker form of this lemma is used here where the variable u is instantiated to the specific term s and the equation $(s = t)^{\theta_i}$ has been deleted being necessarily true (the full power of lemma 5.2 will be exploited, instead, in the extended resolution procedure to be discussed in the next section).

6 Resolution Procedure

The resolution procedure adopted represents an extension of the classical *CLP* operational semantics. The main difference is related to the explicit management of negative atoms. We express the resolution procedure following the rewriting model proposed for AKL [11]. A resolvent at each step is represented by a *goal*, defined as follows.

$$\begin{aligned} < goal > &::= < and-box > | < or-box > \\ < and-box > &::= \mathbf{and}(< literals > \Box < constraint >) \\ < or-box > &::= \mathbf{or}(< sequence\ of\ goals >). \end{aligned}$$

The resolution procedure assumes that the constraints to be dealt with are always transformed into a *simplified normal form* which is obtained by removing from a constraint C in normal form all the redundant variables and equalities and all the irrelevant negative constraints which can possibly occur in it, as described by the following procedure.

function nored$(C, vars)$;
select a constraint c in C enabling one of the following actions;
if no such c exists **then return** C; let $C = C' \wedge c$;
1. (remove redundant variables and equalities)
   if $c \equiv X = Z$ and $X \in vars$ and $Z \notin vars$ **then return** nored$(C'^{\{Z \mapsto X\}}, vars)$;
   if $c \equiv Z = t$ and $Z \notin vars$ **then return** nored$(C', vars)$;
2. (remove irrelevant inequalities)
   if $c \equiv \forall Y_1 \cdots Y_h(s \neq t)$ and $FV(\forall Y_1 \cdots Y_h(s \neq t)) \setminus (vars \cup FV(C_=)) \neq \emptyset$
   then return nored$(C', vars)$;
3. (remove irrelevant non-memberships)
   if $c \equiv \forall Y_1 \cdots Y_h(s \notin t)$ and $FV(\forall Y_1 \cdots Y_h(s \notin t)) \setminus (vars \cup FV(C_=)) \neq \emptyset$
   then return nored$(C', vars)$.

Lemma 6.1 *If D_i = nored$(C_i, FV(C))$, and C_i is one of the constraints returned by $\mathcal{SAT}(C)$, then C_i and D_i are equi–satisfiable.*

Function nored is used in the definition of the *normalization procedure* $\mathcal{N}$, which takes a constraint C and performs the following two actions:
- call $\mathcal{SAT}(C)$ to obtain a disjunction of normal form constraints $C_1, \ldots, C_k$.
- if $k > 0$ (i.e. C is satisfiable), then for each $i = 1, \ldots, k$ call the procedure nored$(C_i, FV(C))$ to obtain the constraint C'_i in simplified normal form.

Thus $\mathcal{N}(\mathcal{C})$ non-deterministically returns the constraints $C'_1, \ldots, C'_k$.

The resolution procedure is essentially based on two *rewriting* rules, called **Fork** and **Negate** rule, used to deal respectively with positive and negative atoms.

Rule 1. (**Fork**) if $\mathsf{p}(s^i_1, \ldots, s^i_n) \leftarrow C_i \Box \bar{B}_i$, for $i = 1, \ldots, m$, are the clauses defining a given predicate p, then

$$\begin{array}{l}\mathbf{and}(A_1, \ldots, A_{i-1}, \mathsf{p}(t_1, \ldots, t_n), A_{i+1}, \ldots, A_n \Box C) \mapsto \\ \quad \mathbf{or}(\mathbf{and}(A_1, \ldots, A_{i-1}, \bar{B}_1, A_{i+1}, \ldots, A_n \Box \mathcal{N}(C \wedge C_1 \wedge \bigwedge_{j=1}^n (t_j = s^1_j))), \cdots, \\ \qquad \mathbf{and}(A_1, \ldots, A_{i-1}, \bar{B}_m, A_{i+1}, \ldots, A_n \Box \mathcal{N}(C \wedge C_m \wedge \bigwedge_{j=1}^n (t_j = s^m_j))))\end{array}$$

For the sake of simplicity we have indicated a unique and-box for each of the possible resolvents. In the actual system the procedure $\mathcal{N}$ is non-deterministic and may lead to multiplication of the relative and-boxes. Analogously, we have not indicated the cases in which the procedure $\mathcal{N}$ fails to report a normalized form (due to unsatisfiability of the original constraint). In this case the corresponding and-box is removed.

Rule 2. (**Negate**) if a negative literal $\neg A$ occurs in an and-box $\mathbf{and}(A_1, \ldots, A_{i-1}, \neg A, A_{i+1}, \ldots, A_n \Box C)$, a subcomputation is started, by applying the derivation rules to the goal $\mathbf{and}(A \Box C)$ and producing a resulting or-box $\mathbf{or}(C_1, \ldots, C_m)$. If $\{C'_1, \ldots, C'_r\}$ is the solution returned by the procedure $\mathsf{NegateSolution}(C_1 \vee \cdots \vee C_m)$ (defined below), then the Negate rule acts in the following way:

$$\begin{array}{l}\mathbf{and}(A_1, \ldots, A_{i-1}, \neg A, A_{i+1}, \ldots, A_n \Box C) \mapsto \\ \quad \mathbf{or}(\mathbf{and}(A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \Box C \wedge C'_1), \cdots, \\ \qquad \mathbf{and}(A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \Box C \wedge C'_r)).\end{array}$$

The rewriting system is composed by these two basic rules together with some auxiliary rules used to simplify goals such as[5],

$$\begin{array}{lrcl}\mathbf{Alternatives - Promotion}: & \mathbf{or}(\bar{A}_1, \mathbf{or}(\bar{B}), \bar{A}_2) & \mapsto & \mathbf{or}(\bar{A}_1, \bar{B}, \bar{A}_2) \\ \mathbf{Fail - Propagation} & \mathbf{and}(\bar{A}_1, \mathbf{fail}, \bar{A}_2 \Box C) & \mapsto & \mathbf{fail} \\ \mathbf{Choice - Elimination}: & \mathbf{or}(\bar{A}_1, \mathbf{fail}, \bar{A}_2) & \mapsto & \mathbf{or}(\bar{A}_1, \bar{A}_2).\end{array}$$

The $\mathsf{NegateSolution}$ procedure represents the key of the implementation of constructive negation. Let us see, briefly, how it works.

By the assumption that *the resolution tree for the goal* $\mathbf{and}(G)$ *in* P *is finite* the resolution procedure will return a number k of computed answers $C_1, \ldots, C_k$, such that $Comp(P) \vdash_{Set} \forall(G \leftrightarrow \exists C_1 \vee \cdots \vee \exists C_k)$. Now let us consider each of the $C_i s$. Firstly we may simplify it by using the procedure nored, i.e. by calling $\mathsf{nored}(C_i, FV(G))$. As corollary of lemma 6.1, we have that $Comp(P) \vdash_{Set} \forall(G \leftrightarrow \exists \bar{w}_1 D_1 \vee \cdots \vee \exists \bar{w}_k D_k)$, hence $Comp(P) \vdash_{Set} \forall(\neg G \leftrightarrow \forall \bar{w}_1 \neg D_1 \wedge \cdots \wedge \forall \bar{w}_k \neg D_k)$.

$\mathsf{NegateSolution}$ then proceeds as follows:

1. simplify each $\forall \bar{w}_i \neg D_i$ using the transformation defined in lemma 5.2 so as to obtain a disjunction of normal form constraints E_i;

[5] the symbol **fail** is used to denote an empty or-box.

2. perform all the boolean operations over the $E_i s$ so as to obtain a constraint in disjunctive normal form $F_1 \vee \cdots \vee F_p$;
3. apply the satisfiability algorithm $\mathcal{SAT}$ to each F_i, obtaining the disjunction $F_1^i \vee \cdots \vee F_{k_i}^i$;
4. finally, let $C'_1, \ldots, C'_r$ (output of the function) be the *non-'false'* constraints in $F_1^1, \ldots, F_{k_1}^1, \ldots, F_1^p, \ldots, F_{k_p}^p$.

Consider the following example (read w for *wife*, h for *husband*):

$\mathsf{proper\_pair(U)} \leftarrow \mathsf{U} = \{\mathsf{X}, \mathsf{Y}\} \wedge \mathsf{X} = \mathsf{w(Y)}\Box.$ $\quad$ $\mathsf{proper\_pair(U)} \leftarrow \mathsf{U} = \{\mathsf{X}, \mathsf{Y}\} \wedge \mathsf{X} = \mathsf{h(Y)}\Box.$

Suppose to call the goal $\mathbf{and}(\neg\mathsf{proper\_pair(U)}, \varepsilon)$: from *Comp(P)* we get:

$$\forall\, \mathsf{U} \quad \left(\begin{array}{lll} \neg\mathsf{proper\_pair(U)} & \leftrightarrow & \forall \mathsf{X_1Y_1}(\mathsf{U} \neq \{\mathsf{X_1}, \mathsf{Y_1}\} \vee \mathsf{X_1} \neq \mathsf{w(Y_1)}) \wedge \\ & & \forall \mathsf{X_2Y_2}(\mathsf{U} \neq \{\mathsf{X_2}, \mathsf{Y_2}\} \vee \mathsf{X_2} \neq \mathsf{h(Y_2)})) \end{array} \right)$$

By applying step 1 the r.h.s. is equivalent to:
$\forall \mathsf{X_1Y_1}(\mathsf{U} \neq \{\mathsf{X_1}, \mathsf{Y_1}\}) \vee \exists \mathsf{X_1Y_1}(\mathsf{U} = \{\mathsf{X_1}, \mathsf{Y_1}\} \wedge \mathsf{X_1} \neq \mathsf{w(Y_1)} \wedge \mathsf{Y_1} \neq \mathsf{w(X_1)}) \wedge$
$\forall \mathsf{X_2Y_2}(\mathsf{U} \neq \{\mathsf{X_2}, \mathsf{Y_2}\}) \vee \exists \mathsf{X_2Y_2}(\mathsf{U} = \{\mathsf{X_2}, \mathsf{Y_2}\} \wedge \mathsf{X_2} \neq \mathsf{h(Y_2)} \wedge \mathsf{Y_2} \neq \mathsf{h(X_2)})$
After performing steps 2–4, we have:
$C'_1 \equiv \forall \mathsf{X_1Y_1}(\mathsf{U} \neq \{\mathsf{X_1}, \mathsf{Y_1}\}) \wedge \forall \mathsf{X_2Y_2}(\mathsf{U} \neq \{\mathsf{X_2}, \mathsf{Y_2}\})$, and
$C'_2 \equiv \mathsf{U} = \{\mathsf{X_1}, \mathsf{Y_1}\} \wedge \mathsf{X_1} \neq \mathsf{h(Y_1)} \wedge \mathsf{Y_1} \neq \mathsf{h(X_1)} \wedge \mathsf{X_1} \neq \mathsf{w(Y_1)} \wedge \mathsf{Y_1} \neq \mathsf{w(X_1)}$.

7 Conclusions and Future Works

This work represents the natural continuation of our previous studies on embedding sets in logic programming. In fact it supplies a sound and complete technique to deal with both extensional and intensional sets. This has been obtained by slightly modifying the standard *CLP* operational semantics (through the use of a negation rule) and by supplying a new constraint manager capable of dealing with constraints containing explicit universal quantifications. As a side-effect of this study, we have shown how the problem of performing set grouping operations can be reduced to the problem of dealing with negation.

Various issues are still open. First of all, the relations between set grouping and negation need to be studied in more depth. We are currently investigating also the inverse reduction, i.e. reducing the management of negative literals to set grouping operations. Moreover, we are planning to consider forms of negation different from the constructive one (like the *intensional negation* proposed in [2]), comparing the kind of requirements that they impose on the admissible programs in order to obtain soundness and completeness results. Finally, the marriage of negation and *CLP* seems to provide a very promising framework to support some more general forms of sets (like hypersets or other forms of infinite sets).

Acknowledgements

The research presented in this paper has benefited from discussions with D. Aliffi, M. Carro, G. Gupta, G. Levi, E. G. Omodeo, and A. Policriti all

of whom we would like to thank. E. Pontelli is partially supported by NSF Grant CCR92-11732 and by a fellowship from Phillips Petroleum.

References

[1] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set Constructors in a Logic Database Language. *Journal of Logic Programming*, 10:181–232, 1991.

[2] P. Bruscoli, F. Levi, G. Levi, and M. C. Meo. Compilative Constructive Negation in Constraint Logic Programs. In *Proc. 1994 Coll. on Trees in Algebra and Programming*. To appear in *LNCS*, Springer-Verlag, Berlin, 1994.

[3] D. Chan. Constructive Negation Based on the Completed Database. In R. A. Kowalski and K. A. Bowen, eds., *Proc. Fifth Int'l Conf. on Logic Programming*, pp. 111–125. MIT Press, 1988.

[4] D. Chan. An Extension of Constructive Negation and its Application in Coroutining. In E. Lusk and R. Overbeek, eds., *Proc. North American Conf. on Logic Programming'89*, pp. 477–493. MIT Press, 1989.

[5] A. Dovier, E. G. Omodeo, E. Pontelli and G. Rossi. {log}: A Logic Programming Language with Finite Sets. In K. Furukawa, ed., *Proc. of the Eighth Int'l Conf. on Logic Programming*. MIT Press, 1991.

[6] A. Dovier, E. G. Omodeo, E. Pontelli and G. Rossi. Embedding Finite Sets in a Logic Programming Language. In E. Lamma and P. Mello, eds., *ELP92*, volume 660 of *LNAI*. Springer-Verlag, Berlin, 1993.

[7] A. Dovier, E. G. Omodeo, E. Pontelli and G. Rossi. Embedding Finite Sets in a Logic Programming Language. RAP.04.93, Università di Roma: "La Sapienza", May 1993.

[8] A. Dovier, E. Pontelli. A WAM-based Implementation of a Logic Language with Sets. In M. Bruynooghe and J. Penjam, eds., *PLILP'93*, volume 714 of *LNCS*, Springer-Verlag, Berlin, 1993.

[9] A. Dovier, G. Rossi. Embedding extensional finite sets in *CLP*. In *Proceedings of 1993 Int. Logic Programming Symp.*, (D. Miller ed.), MIT Press, 1993.

[10] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Dept. of Computer Science, Monash University, June 1986.

[11] S. Janson, S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proc. 1991 Int. Logic Programming Symp.*, MIT Press, 1991.

[12] T. Przymusinski. On Constructive Negation in Logic Programming. In *Proc. North American Conf. on Logic Programming'89*, Addendum to the volume, MIT Press, 1989.

[13] J. C. Shepherdson. Language and equality theory in logic programming. Technical Report PM-91-02, School of Mathematics, University of Bristol, 1991.

[14] P. J. Stuckey. Constructive Negation for Constraint Logic Programming. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pp. 328–339. IEEE Computer Society Press, 1991.