

Embedding extensional finite sets in CLP

Agostino Dovier

Università di Pisa, Dip. di Informatica
Corso Italia 40, 56100-PISA (Italy)
e-mail: dovier@di.unipi.it

Gianfranco Rossi

Università di Bologna, Dip. di Matematica
Piazza di Porta S.Donato 5, 40127-BOLOGNA (Italy)
e-mail: gianfr@dm.unibo.it

Abstract

In this paper we review the definition of $\{\text{log}\}^1$, a logic language with sets, from the viewpoint of CLP. We show that starting with a CLP-scheme allows a more uniform treatment of the built-in set operations (namely, $=$, \in and their negative counterparts), and allows all the theoretical results of CLP to be immediately exploitable. We prove this by precisely defining the privileged interpretation domain and the axioms of the selected set theory. Then we define a non-deterministic procedure for checking constraint satisfiability based on the reduction of a given constraint to a collection of constraint in a suitable canonical form, which is provable to be sound and complete w.r.t. the given theory. Algorithms for transforming each one of the set constraints the language provides ($=$, \neq , \in and \notin) into their corresponding canonical forms are described in details. It is also shown that the resulting language is powerful enough to allow all the usual operations on sets (such as \subseteq , \cup , etc.) to be effectively programmed in the language itself.

1 Introduction

The problem of enriching logic programming with set constructs has deserved an increasing attention in recent years. Indeed the availability of set abstractions is widely recognized as a valuable feature of high-level programming languages. And logic programming languages seem to be the right candidates for hosting such a feature due to their potentially high declarative nature.

Attention to this problem has come first from the field of *deductive databases* [1, 4, 14, 22]. However many other fields, including rapid software prototyping and knowledge based systems, may benefit from the availability of sets constructs. And, recently, a number of papers have addressed the problem also in a wider setting. General-purpose set constructs and basic

¹Read 'setlog'.

operations on sets are inserted into some general logic-based framework: an equational-logic language in [15, 16], a pure logic programming language in [7, 9] and a CLP language in [17].

The development of such kind of extensions raises several interesting theoretical as well as practical problems (some of them are discussed also in [23]):

- which kind of objects one should aim at dealing with: extensional, intensional, multi-, hyper- sets;
- what is the term representation of sets;
- what operations on sets should the language provide as primitives;
- what is the role of negation and set grouping in intensional set definition;
- which is the host language: pure Horn clause, CLP, equational logic language;
- what kind of applications may benefit more from such an extension;
- what are the main issues in developing efficient implementations.

In previous papers [7, 9] we have addressed most of these problems assuming a pure logic programming language as the base language. Here we argue that starting with a CLP-scheme provides us with a more convenient solution. We assume the signature Σ is endowed with two functional symbols, \emptyset for the *emptyset* and **with** as the set constructor, and that the set of constraint predicate symbols contains $=$, \neq , \in and \notin , for equality, not equality, membership and not membership, respectively. It will turn out that a more uniform treatment between $=$ and \in and their negative counterparts than that provided by the previous solution is now feasible. Also (and even more importantly) such a CLP-based approach turns out to be well suited to accommodate for *intensional set formers*.²

Many different answers are applicable to each one of the issues listed above. In Section 2 we will briefly analyze a number of them and we will try to motivate our choices. Section 3 provides the theoretical framework for our work: that is, the language interpretation domain and the logical axioms of the selected *set theory*. The constraint simplification algorithms are described in detail in Section 4, and then used in Section 5 to define the constraint satisfiability procedure which is proved to be correct and complete w.r.t. the given set theory.

2 Designing a logic language with sets

Which kind of sets?

We restrict our attention to (finite) extensional sets, such as $\{t_0, \dots, t_n\}$. This is mainly motivated by the desire to keep the presentation uncluttered, allowing an in depth analysis of that basic case. Problems implied by the

²Preliminary versions of these ideas also appeared in [11] and [5].

introduction of *intensional sets*, such as $\{x \in S : \varphi\}$, were already addressed in [7], but no precise solution has emerged there. It is also the central problem discussed in [3]. It is well accepted that this problem is strongly connected with that of introducing *negation* in Horn clause logic. A proposal for embedding intensional sets in the CLP-based language of this paper augmented with intensional negation is under development at present [5].

By allowing membership to form cycles, *hypersets* could come on the scene. Hypersets may be defined to be rooted labelled graphs of some special canonical form, and rendered concretely as systems of equations in some canonical form. Dealing with hypersets thus requires the notion of syntactically equal ground terms of the standard case to be replaced by the notion of ground graphs having the same canonical representative. Axioms of the set theory are modified so as to include some form of the anti-foundation axiom typical of hyperset theory. All that definitively results in a non trivial increasing in the complexity of the problem at hand. A formal characterization of hypersets and the definition of a suitable unification algorithm dealing with them are given in [2]. However a precise embedding of hypersets in the language presented in this paper, and more convincing motivations for such an extension, still need further investigation.

Finally notice that we are talking here about sets, not about *multisets* where members occur with a multiplicity factor. Work on introducing multisets is still in progress at present. An analysis of the problems that the introduction of multisets - as well as of sets and hypersets - is reported in [19]. However, our set theory and our set representation technique (see below) seem to accommodate quite well for multisets too.

Which representation of sets?

At least two alternatives are viable:

- i.* $\{t_0, \dots, t_n\}$ is represented as *union of singletons*, i.e. $\{t_0\} \cup \dots \cup \{t_n\}$;
- ii.* $\{t_0, \dots, t_n\}$ is represented as a *list*, i.e. $(\dots (\emptyset \mathbf{with} t_n) \dots) \mathbf{with} t_0$.

Solution *i* requires that three functional symbols are introduced: \emptyset , of arity 0, $\{\cdot\}$, of arity 1, and \cup , of arity 2. In a non trivial set theory (such as *ZF*), \cup must be *Associative* (i.e. $A \cup (B \cup C) = (A \cup B) \cup C$), *Commutative* (i.e. $A \cup B = B \cup A$) and *Idempotent* (i.e. $A \cup A = A$). Moreover \emptyset is the *identity* element w.r.t. union (i.e. $A \cup \emptyset = \emptyset \cup A = A$).

Solution *ii* requires that two functional symbols are introduced: \emptyset , of arity 0, and \mathbf{with} , of arity 2. Again in a significant set theory, \mathbf{with} must exhibit a *Right permutativity property* (i.e. $(X \mathbf{with} Y) \mathbf{with} Z = (X \mathbf{with} Z) \mathbf{with} Y$) and a *Right absorption property* (i.e. $(X \mathbf{with} Y) \mathbf{with} Y = X \mathbf{with} Y$).

Representation *ii* is quite usual when dealing with sets in logic programming. It is used for instance in [14], in [4] (where \mathbf{with} is called \mathbf{scons}) and also in the Gödel language [12]. [15] uses the \cup operator but actually its

behaviour is that of the **with** operator of approach *ii*. Representation *ii* is also adopted, for instance, in [20].

Representation *i*, on the contrary, is often used when dealing with the problem of set unification on its own, e.g. [6] and [18], where set unification is dealt with as a problem of *ACI-unification*.

A set term of approach *ii* can always be translated to a corresponding set term of approach *i*. The converse is not always true. For instance, the term $X \cup \{Y\} \cup Z$ has no correspondent representation in approach *ii*. Actually, by taking representation *ii* we are considering just a particular case of the general ACI-unification problem. The full power of ACI-unification is not necessary nor useful in that particular case. Indeed we would like to rule out problems such as the one represented by the equation $X \cup Y \cup Z = \{a\} \cup \{b\} \cup \{c\}$ which admits 343 independent solutions.³

In this paper, as well as in [7, 9], we have chosen approach *ii*. This is further motivated by the desire to have a flexible system, which is able to deal not only with sets, but also, with few changes, with multisets or other particular kinds of lists.

However, notice that a unification algorithm dealing with *set terms* would be NP-complete in both approaches [9].

Which primitive operations on sets?

We focus on basic operations on sets such as equality ($=$), membership (\in), inclusion (\subseteq), strict inclusion (\subset), union (\cup), intersection (\cap) and difference (\setminus).

Let us assume the language at hand is an Horn clause language augmented with set terms, represented as in approach *ii*. Unification is extended accordingly so to allow unification between set terms to be performed taking into account the properties of **with** described above.

We are interested in establishing which of the operations on sets listed above are to be provided as primitive operations of this language and which on the contrary can be conveniently programmed in the language itself. The selection should be performed on the basis of a number of features of the resulting language, such as expressive power, effectiveness and efficiency.

The following basic predicates can be programmed in the considered language (set predicates will be written in infix notation).

Equality :	Membership :	Inclusion :
$X = Y \leftarrow .$	$X \in Y \text{ with } X \leftarrow .$	$X \subseteq Y \leftarrow$ $(\forall Z \text{ in } X) (Z \in Y).$

³Also in [17] sets are represented as in approach *i*. However, since set operations are evaluated only when applied to *ground sets*, problems arising when solving an equation such as $X \cup Y \cup Z = \{a\} \cup \{b\} \cup \{c\}$ are avoided ‘a priori’. Instead, such an equation is considered as a constraint and possibly returned as part of the computed answer.

where the construct $(\forall x \text{ in } y)\varphi$, φ conjunction of atoms, is intended to denote the formula $\forall x(x \in y \rightarrow \varphi)$. In [9] it is shown that an extended Horn clause containing such kind of *restricted universal quantifiers* can be translated via simple pre-processing to a set of pure Horn clauses enriched with set terms and set unification. Thus, for instance, the definition of the predicate \subseteq given above can be re-written as follows:

$$\begin{aligned} \emptyset \subseteq Y &\leftarrow . \\ Z \text{ with } X \subseteq Y &\leftarrow \\ X \in Y, Z \subseteq Y. & \end{aligned}$$

Union :

$$\begin{aligned} \cup(A, B, C) &\leftarrow & \text{in_one}(X, B \text{ with } X, C) &\leftarrow . \\ (\forall X \text{ in } A) (X \in C), & & \text{in_one}(X, B, C \text{ with } X) &\leftarrow . \\ (\forall Y \text{ in } B) (Y \in C), & & & \\ (\forall Z \text{ in } C) (\text{in_one}(Z, A, B)). & & & \end{aligned}$$

The considered language (i.e. HCL + extensional set terms + set unification) is powerful though simple. Nevertheless, the following two issues (at least) cannot be faced adequately:

- *effectiveness*: if, for instance, the resolution algorithm is applied to the goal $\leftarrow \subseteq(A, \emptyset \text{ with } a)$ then an infinite SLD-tree is generated trying to compute the (sound) answers $A \mapsto \emptyset, A \mapsto \emptyset \text{ with } a, A \mapsto \emptyset \text{ with } a \text{ with } a, \dots$. To solve the problem one could add the literal $X \notin Z$ to the body of the second clause defining \subseteq ;
- *expressive power*: other basic set-operations, such as $\notin, \neq, \cap, \subset, \setminus$ cannot be programmed in the present language unless some form of negation is introduced in it.

It turns out that having either \notin or \neq as primitive operations would suffice to solve all these problems without requiring full negation to be introduced in the language.

First, notice that \notin and \neq can be easily defined each one in terms of the other:

$$\begin{aligned} A \neq B &\leftarrow & A \notin B &\leftarrow \\ A \notin \emptyset \text{ with } B. & & B \text{ with } A \neq B. & \end{aligned}$$

Then, all the remaining basic operations on sets listed above can be easily programmed in the extended language with \notin and \neq .

Intersection :

$$\begin{aligned} \cap(A, B, C) &\leftarrow & \text{in_both}(X, A, B) &\leftarrow \\ (\forall X \text{ in } C) (X \in A \wedge X \in B), & & X \in A, X \in B. & \\ (\forall Y \text{ in } A) \text{in_both}(Y, B, C), & & \text{in_both}(X, A, B) &\leftarrow \\ (\forall Z \text{ in } B) \text{in_both}(Z, A, C). & & X \notin A, X \notin B. & \end{aligned}$$

Difference :	Strict inclusion:
$\setminus(A, B, C) \leftarrow$	$\subset(X, Y) \leftarrow$
$(\forall X \text{ in } A) \text{ in_one}(X, B, C),$	$X \subseteq Y,$
$(\forall Y \text{ in } C) (Y \in A \wedge Y \notin B).$	$X \neq Y.$

While it is feasible in principle to have either \neq or \notin as the only primitive set operations of the language, efficiency considerations lead us to assume our language provides also few other basic operations for set manipulation as primitive operations, namely $=$, \in , \neq , \notin .

Which host language?

The solution adopted in $\{\text{log}\}$ [7, 9] assumes as its starting point a *pure logic programming language* which is then augmented with set terms, set unification and few distinguished predicates representing set membership, equality and their negative counterparts, as described above.

The way the distinguished predicates are dealt with in $\{\text{log}\}$ is in a sense an hybrid one. Indeed, \in and $=$ are embedded directly in the resolution algorithm. On the contrary, in order to avoid the well-known drawbacks of negation in Horn clause logic, \notin and \neq are treated as *constraints*. The resolution procedure is extended accordingly.

The main drawback of this solution is the non-uniformity of treatment between \in and $=$ and their negative counterparts, and the rather ad-hoc extensions to the resolution procedure needed to handle them (which, as a consequence, complicate correctness and completeness proofs).

An alternative approach to the one adopted in $\{\text{log}\}$ is starting with a real *CLP-scheme* [13], where all the predefined predicates dealing with sets are viewed as *constraints*.

This allows a more uniform treatment of the set operations and, more importantly, allows all the theoretical general results of CLP to be immediately available, provided the CLP-scheme is instantiated w.r.t. the actual domain of interest.

In this paper we review the $\{\text{log}\}$ language definition from the viewpoint of CLP. A CLP-based solution to logic programming with sets is also advocated in [25] though the kind of sets considered there substantially differs from the sets we are interested in here. A solution based on a CLP-scheme which, on the contrary, shares much with our work is the one described in [17]. The main difference with our proposal is that no precise definition of the set theory nor of the interpretation domain is given in [17]. This prevents that proposal from providing formal correctness and completeness results. An efficient implementation of the language in [17], however, is available.

3 Our language definition

Standard CLP notations and results ([13]) are assumed hereafter. As noticed in previous sections, we would like to have a CLP language which is able to deal with extensional set terms as well as with standard Herbrand terms. Sets are represented using `with` as the set constructor (approach *ii* of the previous section). Therefore we require the signature Σ contains at least the two functional symbols `with`, binary, and \emptyset , nullary. `with` will be used infix, left associative. For example the term $\emptyset \text{ with } c \text{ with } (\emptyset \text{ with } b \text{ with } a) \text{ with } a$ will denote the set $\{a, \{a, b\}, c\}$ (provided a , b , and c belong to Σ).

If other functional symbols (e.g. a and b) are in Σ , we would like to write terms of the form $(a \text{ with } \emptyset) \text{ with } b$. Such a term will be interpreted as a ‘coloured’ set, i.e. a set based on an object different from \emptyset (in this case a). Two sets will be considered equal if (and only if) they have the same elements and they are based on the same ‘kernel’.

Finally we fix the set Π_C of constraint predicative symbols of the CLP-scheme to be $\{\in, \notin, =, \neq\}$.

3.1 Interpretation

First of all we must define the *interpretation domain* \mathcal{A} (a single sort is sufficient for our purposes).

Let U_H be the ordinary Herbrand universe on $\Sigma = \{\emptyset, \text{with}, \dots\}$. Assuming a suitable ordering on U_H , one then takes the finest equivalence relation \equiv over U_H that fulfils the right absorption and right permutativity properties of `with` mentioned in the previous section. Then, one chooses a *canonical representative* term from each one of the equivalence classes forming $U_H \equiv$ (according to specific criteria to be hinted at below), and finally one puts $\mathcal{A} = \{t : t \in U_H \wedge t \text{ is canonical}\}$.

Constructively, let $<_{\Sigma}$ be an order relation over Σ . By exploiting this ordering, it is easy to define an order relation $<$ over terms based on reverse lexicographic order. This means, in particular, that $r \text{ with } t < s \text{ with } u$ holds if $(t < u)$ or $(t = s \text{ and } r < s)$.

A ground term t is said to be *canonical* if

- t is a constant, or
- each subterm of t is canonical and, furthermore, for each subterm of t of the form $s \text{ with } u \text{ with } v$, $u < v$ holds.

We define the function τ mapping each term to its canonical representative, in the following way:

- $\tau(f(t_1, \dots, t_n)) = f(\tau(t_1), \dots, \tau(t_n))$ if f is different from `with`;
- $\tau(k \text{ with } t_1 \text{ with } \dots \text{ with } t_n) = \tau(k) \text{ with } s_1 \text{ with } \dots \text{ with } s_m$ where s_1, \dots, s_m ($m \leq n$) are the *distinct* canonical representatives of t_1, \dots, t_n such that $s_1 < \dots < s_m$ (i.e. $\{s_1, \dots, s_m\}$ and $\{\tau(t_1), \dots, \tau(t_n)\}$ *coincides*).

Now we are ready to define the *interpretation functions*.

I_f^A is defined as $I_f^A = \lambda(x_1, \dots, x_n). \tau(f(x_1, \dots, x_n))$ for each n -ary f occurring in Σ .

$I_{=}^A$ is the identity function on \mathcal{A} . $I_{\neq}^A(s, t) = True$ if and only if $I_{=}^A(s, t) = False$. $I_{\in}^A(s, t) = False$ if t is of the form $f(t_1, \dots, t_n)$, f different from **with**; $I_{\in}^A(r, s \text{ with } t) = True$ if and only if $I_{=}^A(r, t)$ or $I_{\in}^A(r, s) = True$. $I_{\notin}^A(r, s) = True$ if and only if $I_{\in}^A(r, s) = False$.

$I^A(t \pi s) = I_{\pi}^A(\tau(t), \tau(s))$, for π in $\{=, \in, \neq, \notin\}$. Then I^A can be inductively extended to first order formulas in the usual way.

Such an interpretation is clearly *solution compact*. In fact each element $a \in \mathcal{A}$ is uniquely definable by the finite constraint $C = \{X = a\}$. Furthermore, there is no limit element in \mathcal{A} since \mathcal{A} is a subset of U_H .

3.2 The theory

We are looking for a set theory \mathcal{T} such that \mathcal{A} and \mathcal{T} correspond [13]. In what follows, free variables are intended to be universally quantified.

- (U) (Scheme) $x \notin f(x_1, \dots, x_n)$, for each f different from **with**;
- (W) $x \in z \text{ with } y \leftrightarrow (x \in z \vee x = y)$;
- (L) $y \in x \rightarrow \exists z (y \notin z \wedge x = z \text{ with } y)$;
- (E) $v \text{ with } x = w \text{ with } y \leftrightarrow$
 $(x = y \wedge v = w) \vee (x = y \wedge v \text{ with } x = w) \vee$
 $(x = y \wedge v = w \text{ with } y) \vee \exists z (v = z \text{ with } y \wedge w = z \text{ with } x)$;
- (R) $\exists z \forall y (y \in x \rightarrow (z \in x \wedge y \notin z))$.

Remarks. Since $\emptyset \in \Sigma$, (U) states, in particular, the existence of an object which does not contain any element: the *emptyset*. (W) describes the behavior of the functional symbol **with**, the set constructor. The *less* axiom (L) states the existence of the set $x \setminus \{y\}$. The *extensionality* axiom (E) shows how to decide if two sets can be considered equal. Finally the *regularity* axiom (R) assures that membership form no cycles.

Such a theory departs from the ‘standard one’ (e.g. finite ZF) in two aspects:

- Presence of *urelements*. By (U), each term with main functional symbol different from **with** is a set lacking in elements (i.e. an urelement). A predicate $ur(x)$, which holds when x is an urelement, can be easily defined in the theory: $ur(x) \leftrightarrow \forall y (y \notin x)$.
- Each term t has a *kernel* associated with it. If $ur(t)$ then t is also its kernel, otherwise the kernel is the ‘seed’ on which the set term is based:

$$ker(x) = y \leftrightarrow (ur(x) \wedge y = x) \vee (\exists v (v \text{ with } x = v \wedge v \notin v \wedge y = ker(v))).$$

Notice that if Σ contains at least one functional symbol other than \emptyset and **with**, the symbol ker will also be included in Σ in order to define proper constraints on set kernels (see sect 4.4).

Using axioms of the theory listed above, it is easy to derive the following properties:

- $(x \text{ with } y) \text{ with } z = (x \text{ with } z) \text{ with } y$ (*Permutativity*);
- $(x \text{ with } y) \text{ with } y = x \text{ with } y$ (*Absorption*) (by using **(W)** and **(E)**);
- Let **(E*)** be the formula $v = w \leftrightarrow (\ker(v) = \ker(w) \wedge \forall z (z \in v \leftrightarrow z \in w))$. Then **(E*)** is equivalent to **(E)** in \mathcal{T} , i.e. **(W)**, **(L)** \vdash **(E)** \leftrightarrow **(E*)**.

In addition to the set axioms, standard *equality axioms* are assumed, along with the following *freeness* axiom schemes:

- (1) $f(x_1, \dots, x_n) \neq g(y_1, \dots, y_m)$, if f is different from g ;
- (2) $f(x_1, \dots, x_n) = f(y_1, \dots, y_m) \rightarrow (x_1 = y_1 \wedge \dots \wedge x_n = y_n)$, if f is different from **with**;
- (3) $t[x] \neq x$ ($t[x]$ stands for a term containing x , except $x \text{ with } t_1 \dots \text{ with } t_n$ whereas x does not occur in t_1, \dots, t_n).

In Section 5 we will show that \mathcal{T} is satisfaction complete. Furthermore it can be proved that the following lemma holds:

Lemma 3.1 $\mathcal{T}(\Pi_C, \Sigma)$ and $\mathcal{A}(\Pi_C, \Sigma)$ correspond.

4 Constraint simplification algorithms

An *atomic constraint* is any (Π_C, Σ) -atom; a (set) constraint C is a collection of atomic constraints. $C_=$, C_{\neq} , C_{\in} , C_{\notin} , and C^F are disjoint subsets of C such that:

- $C = C_= \cup C_{\neq} \cup C_{\in} \cup C_{\notin} \cup C^F$,
- the elements of C_{π} are all of the form $t_1 \pi t_2$, t_1 and t_2 terms, and π a symbol in $\{=, \in, \neq, \notin\}$,
- C^F may be \emptyset or $\{False\}$.

For each constraint predicate symbol π in Π_C we supply a simplification algorithm which non-deterministically computes for each given constraint C a finite collection of constraints $\{C_1, C_2, \dots, C_n\}$ that is provable to be *equisatisfiable* to C (in the sense specified by Theorem 5.3), and such that each $C_{i_{\pi}}$ in the collection is in some simplified canonical form.⁴

4.1 Constraint =

Given a constraint C , an atomic constraint $X = t$ of C is said to be in *canonical form* if X is a variable not occurring neither in t nor in $C \setminus \{X = t\}$. A constraint $C_=$ is in *canonical form* if every its element is canonical, that is if $C_= = \{X_1 = t_1, \dots, X_n = t_n\}$ and each variable X_i occurs in C exactly once.

⁴Hereafter we will use the term *urelement* to denote also non ground terms of the form $f(t_1, \dots, t_n)$, $f \neq \text{with}$, which whatever instantiated come to be urelements in the sense given above.

The $C_=_$ constraint simplification algorithm can serve as well as a (set) unification algorithm. Actually a $C_=_$ in canonical form is an (Herbrand) system of equations in *solved form*. A solution σ for such a system can be immediately obtained from it by taking $\sigma = \{X_1 \mapsto t_1, \dots, X_n \mapsto t_n\}$. In [7, 9] the same algorithm is used as the unification algorithm of the extended resolution procedure. It has been proved to terminate in [8]. This algorithm generalizes in a sense the set unification algorithm proposed in [15] which intentionally does not take into account duplicates in a set and assumes that one of the terms in a set-set equation is ground.⁵

```

function unify( $C$ )
  if  $C_=_$  is in canonical form
    then return  $C$ 
    else choose any  $c$  in  $C_=_$  not in canonical form; let  $C' = C \setminus \{c\}$ ;
    case  $c$  of
      1.  $X = X$ : return unify( $C'$ );
      2.  $t = X$ ,  $t \notin \mathcal{V}$ : return unify( $\{X = t\} \cup C'$ );
      3.  $X = t$ ,  $t$  is an urelement and  $X$  occurs in  $t$ : return  $\{False\}$ ;
      4.  $X = t$  with  $t_n$  with... with  $t_0$  and  $X$  occurs in  $t_0$  or ... or in  $t_n$ ,
         or  $t$  is an urelement and  $X$  occurs in  $t$ : return  $\{False\}$ ;
      5.  $X = X$  with  $t_n$  with... with  $t_0$  and  $X$  does not occur in  $t_0 \dots t_n$ :
         return unify( $\{X = N$  with  $t_n$  with... with  $t_0\} \cup C'$ ),  $N$  new variable;
      6.  $X = t$ ,  $X$  does not occur in  $t$ : return unify( $C' \sigma \cup \{X = t\}$ ,
         where  $\sigma = \{X \mapsto t\}$ );
      7.  $f(t_1, \dots, t_n) = g(s_1, \dots, s_m)$ ,  $f$  different from  $g$ : return  $\{False\}$ ;
      8.  $f(t_1, \dots, t_n) = f(s_1, \dots, s_n)$ ,  $f$  is not with:
         return unify( $\{t_1 = s_1, \dots, t_n = s_n\} \cup C'$ );
      9.  $h$  with  $\{t_n, \dots, t_0\} = k$  with  $\{s_m, \dots, s_0\}$   $h$  and  $k$  urelements
         or variables:
         a. if  $h$  and  $k$  are not the same variable, then select
            non-deterministically one of the following actions:
            i. return unify( $\{t_0 = s_0,$ 
                 $h$  with  $\{t_n, \dots, t_1\} = k$  with  $\{s_m, \dots, s_1\}\} \cup C'$ )
            ii. return unify( $\{t_0 = s_0,$ 
                 $h$  with  $\{t_n, \dots, t_0\} = k$  with  $\{s_m, \dots, s_1\}\} \cup C'$ )
            iii. return unify( $\{t_0 = s_0,$ 
                 $h$  with  $\{t_n, \dots, t_1\} = k$  with  $\{s_m, \dots, s_0\}\} \cup C'$ )
            iv. return unify( $\{h$  with  $\{t_n, \dots, t_1\} = N$  with  $s_0$ ,  $N$  with  $t_0 =$ 
                 $k$  with  $\{s_m, \dots, s_1\}\} \cup C'$ ),  $N$  new variable;
         b. otherwise select non-deterministically a number  $i$  in  $\{0, \dots, m\}$ 
            and one of the following actions:
            i. return unify( $\{t_0 = s_i,$ 
                 $h$  with  $\{t_n, \dots, t_1\} = k$  with  $\{s_m, \dots, s_{i-1}, s_{i+1}, \dots, s_0\}\} \cup C'$ )

```

⁵For the sake of readability we will use the notation h with $\{t_0, \dots, t_n\}$ to denote the term h with t_0 with... with t_n

- ii. return $\underline{\text{unify}}(\{t_0 = s_i,$
 $h \text{ with } \{t_n, \dots, t_0\} = k \text{ with } \{s_m, \dots, s_{i-1}, s_{i+1}, \dots, s_0\}\} \cup C')$
- iii. return $\underline{\text{unify}}(\{t_0 = s_i,$
 $h \text{ with } \{t_n, \dots, t_1\} = k \text{ with } \{s_m, \dots, s_0\}\} \cup C')$
- iv. return $\underline{\text{unify}}(\{X = N \text{ with } t_0, N \text{ with } \{t_n, \dots, t_1\} =$
 $N \text{ with } \{s_m, \dots, s_0\}\} \cup C')$, N new variable.

Some remarks are mandatory here to understand this algorithm and particularly actions 5 and 9.

Action 5 deals with the case of an equation such that $X = X \text{ with } a$ which does not admit any solution in the standard unification case whereas it has the solution $\{X \mapsto N \text{ with } a\}$, N new variable (any set), in our case.

Aim of action 9 of the algorithm is the reduction of set-set equations by exploiting axiom **(E)**. In particular, cases ii and iii take care of duplicates in the left-hand side term and in the right-hand side term, respectively. Case iv, instead, takes care of permutativity of the set constructor **with**.

Equations of the form $X \text{ with } \{t_n, \dots, t_0\} = X \text{ with } \{s_m, \dots, s_0\}$, where the two sides are set terms with the same variable tail element, are dealt with as a special case by action 9-b. The problem here is how permutativity of the **with** operator can be guaranteed. In fact, it is easy to check that by applying action 9-a, and in particular case iv, to an equation of this form (e.g. $X \text{ with } a = X \text{ with } b$) the algorithm might go into an infinite loop. The solution we have adopted is to avoid action 9-a-iv of the general case by requiring the algorithm non-deterministically considers each element of one of the two sets involved in the given set-set equation, thus trying all possible combinations.

4.2 Constraint \in

Membership atomic constraints can be completely eliminated by replacing them with suitable equality atomic constraints.

```
function member(C)
  if  $C_\in = \emptyset$ 
    then return C
    else choose any c in  $C_\in$ ; let  $C' = C \setminus \{c\}$ ;
    case c of
  1.  $t \in s$  and s is an urelement: return {False};
  2.  $t \in X$  and X is a variable:
    return  $\{X = N \text{ with } t\} \cup \underline{\text{member}}(C')$ , N new variable;
  3.  $t \in v \text{ with } w$ : select non-deterministically from:
    a. return  $\underline{\text{member}}(\{t \in v\} \cup C')$ 
    b. return  $\{t = w\} \cup \underline{\text{member}}(C')$ .
```

4.3 Constraint \notin

An \notin -atomic constraint is said to be in *canonical form* if it has the form $t \notin X$ and X is a variable not occurring in t . A constraint C_{\notin} is in *canonical form* if every its element is canonical.

function notmember(C)
 if C_{\notin} is in canonical form
 then return C
 else choose any c in C_{\notin} not in canonical form; let $C' = C \setminus \{c\}$;
 case c of
 1. $t \notin r$ with s : return $\{t \neq s\} \cup \text{notmember}(\{t \notin r\})$
 2. $t \notin f(t_1, \dots, t_n)$, f different from with: return notmember(C')
 3. $t \notin X$, X variable occurring in t : return notmember(C').

4.4 Constraint \neq

For the sake of simplicity, C_{\neq} is further partitioned in two parts, C'_{\neq} and C''_{\neq} , putting a \neq -atomic constraint into C'_{\neq} if and only if it contains at least one occurrence of the functional symbol *ker*.

A \neq -atomic constraint in C'_{\neq} is said to be in *canonical form* if it is of the form $X \neq t$ and X is a variable not occurring in t . A constraint C'_{\neq} is in *canonical form* if all the atomic constraints in it are canonical. Function notequal shown below deals with the ‘canonization’ of the C'_{\neq} part of a given constraint C .

function notequal(C)
 if C'_{\neq} is in canonical form
 then return C
 else choose c (not in canonical form) in C'_{\neq} ; let $C' = C \setminus \{c\}$
 case c of
 1. $f(t_1, \dots, t_n) \neq g(s_1, \dots, s_m)$, f different from g : return notequal(C');
 2. $f(t_0, \dots, t_n) \neq f(s_0, \dots, s_n)$, f different from with: select non-deterministically i in $\{0, \dots, n\}$: return notequal($\{t_i \neq s_i\} \cup C'$);
 3. $f \neq f$, f constant: return $\{False\}$;
 4. $X \neq X$, X variable: return $\{False\}$;
 5. $t \neq X$ and t is not a variable: return notequal($\{X \neq t\} \cup C'$);
 6. $X \neq t$, t is an *urelement* and X occur in t , or t is h with $t_n \dots$ with t_1 , h *urelement* or variable, and X occurs in t_1 or \dots or in t_n :
 return notequal(C')
 7. $X \neq X$ with $t_n \dots$ with t_0 select non-deterministically i in $\{0, \dots, n\}$:
 return $\{t_i \notin X\} \cup \text{notequal}(C')$;
 8. r with $s \neq t$ with u : select non-deterministically one of the three following actions (X denotes a new variable):
 a. return $\{X \in r$ with s , $X \notin t$ with $u\} \cup \text{notequal}(C')$
 b. return $\{X \in t$ with u , $X \notin r$ with $s\} \cup \text{notequal}(C')$
 c. if $\Sigma \supset \{\emptyset, \text{with}, \text{ker}\}$ then return notequal($C' \cup \{\text{ker}(r) \neq \text{ker}(t)\}$).

A \neq -atomic constraint in C''_{\neq} is said to be in canonical form if it has one of the forms:

- $ker(X) \neq f(t_1, \dots, t_n)$, X variable, f different from **with** and ker , or
- $ker(X) \neq ker(Y)$, X, Y distinct variables.

If all the atomic constraints in C''_{\neq} are in canonical form, then C''_{\neq} is in canonical form. Function ker_analyzer shown below deals with the simplification of disequations involving the functional symbol ker . Notice that we explicitly require that such kind of constraints cannot be introduced by the user but only by the constraint simplification procedures (namely by action 8-c of function notequal).

```
function ker_analyzer(C)
  if  $C''_{\neq}$  is in canonical form
    then return C
    else choose any  $c$  (not in canonical form) in  $C''_{\neq}$ ; let  $C' = C \setminus \{c\}$ 
    case  $c$  of
      1.  $ker(s) \neq t_1$  with  $t_2$ : return ker_analyzer( $C'$ );
      2.  $ker(r$  with  $s) \neq t$ ,  $t$  has not the form  $t_1$  with  $t_2$ :
         return ker_analyzer( $\{ker(r) \neq t\} \cup C'$ );
      3.  $ker(f(t_1, \dots, t_n)) \neq t$ ,  $f$  different from with and  $t$  has not the form
          $s_1$  with  $s_2$ : return ker_analyzer( $\{f(t_1, \dots, t_n) \neq t\} \cup C'$ );
      4.  $f(t_1, \dots, t_n) \neq ker(t)$ ,  $f$  different from  $ker$ :
         return ker_analyzer( $\{ker(t) \neq f(t_1, \dots, t_n)\} \cup C'$ );
      5.  $ker(X) \neq ker(X)$ ,  $X$  is a variable: return  $\{False\}$ .
```

5 Constraint satisfiability

A constraint C is said to be in *canonical form* if

- $C^F = \{False\}$, or
- $C_{\in} = C^F = \emptyset$; $C_{=}$, C_{\neq} , C'_{\neq} and C''_{\neq} are in canonical form.

Lemma 5.1 *Let C be a constraint in canonical form, $C^F = \emptyset$. Then C is \mathcal{A} -solvable and \mathcal{T} -satisfiable, provided C''_{\neq} is satisfiable.*

The condition on C''_{\neq} originates from the fact that the normal form of a constraint C''_{\neq} does not necessarily imply its satisfiability. For instance, assuming $\Sigma = \{\emptyset, \mathbf{with}, ker, a\}$, the following constraint in canonical form

$$\{ker(X) \neq ker(Y), ker(X) \neq ker(Z), ker(Z) \neq ker(Y)\}$$

is clearly unsatisfiable. More precisely, if the signature Σ contains either an infinite number of constant symbols or, at least, a functional symbol of arity greater than 0, distinct from **with** and ker , then a C''_{\neq} in canonical form is satisfiable (in fact, in these cases it is possible to construct an infinite number of different *kernels*). If, on the contrary, Σ contains a *finite* number

of constant symbols other than \emptyset , then satisfiability of C''_{\neq} must be checked explicitly by finding a suitable assignment from the variables of the constraint to the constants in Σ . Such a satisfiability check is performed by the function `ker_sat` (not shown here; actually a simple graph construction procedure), which returns the given constraint C unchanged if C''_{\neq} is satisfiable, and $\{False\}$ otherwise.

The following (non-deterministic) function is used to compute the canonical form of a constraint C :

$$\text{step}(C) = \text{ker_sat}(\text{ker_analyzer}(\text{notequal}(\text{notmember}(\text{unify}(\text{member}(C)))))).$$

Lemma 5.2 *Whatever are the non-deterministic choices performed by `step` there exists n such that $\text{step}^{n+1}(C) = \text{step}^n(C)$ is a constraint in canonical form (step^n denoting n nested applications of `step`, i.e. $\text{step}(\text{step}(\dots(\text{step}(C)\dots))$).*

Let `sat` be the function which computes the minimum fixpoint of `step` on the input C , defined by:

$$\begin{aligned} \text{sat}(C) &= \text{while } \text{step}(C) \neq C \text{ do} \\ &\quad C := \text{step}(C); \\ &\text{return } C. \end{aligned}$$

The termination of `step` stated by Lemma 5.2, and the finiteness of the number of non-deterministic choices generated by each call of `step` in `sat`, assure the finiteness of the number of constraints non-deterministically returned by `sat`. We may then state the following:

Theorem 5.3 $\mathcal{T} \vdash C \leftrightarrow \exists \bigvee_{\tilde{C}=\text{sat}(C)} \tilde{C}$.

The following statements follow trivially:

Corollary 5.4 C is \mathcal{T} -satisfiable if and only if there exists a non-deterministic choice such that $\text{sat}(C) \neq \{False\}$.

Corollary 5.5 \mathcal{T} is satisfaction complete.

The *algebraic derivation* is then algorithmically implementable. In order to implement the *logic derivation* it is therefore sufficient choosing one of the \tilde{C} constraints different from $\{False\}$ returned by $\text{sat}(C)$, and considering the substitution induced by $\tilde{C}_=$.

As an example of how `sat` works, we show part of the (non-deterministic) computation of `sat` on a simple constraint (the sequence of actions performed is written in square brackets):

1. $\{X = X \text{ with } a, \emptyset \text{ with } a \text{ with } b \neq X\}$ [unify (5)];
2. $\{X = N \text{ with } a, \emptyset \text{ with } a \text{ with } b \neq X\}$ [unify (6)];
3. $\{X = N \text{ with } a, \emptyset \text{ with } a \text{ with } b \neq N \text{ with } a\}$ [notequal (8)];
by choosing alternative (b) we get
4. $\{X = N \text{ with } a, Z \in N \text{ with } a, Z \notin \emptyset \text{ with } a \text{ with } b\}$:
[member (3-b) \rightarrow 5.1], [member (3-a) (2) \rightarrow 5.2];
- 5.1 $\{X = N \text{ with } a, Z = a, Z \notin \emptyset \text{ with } a \text{ with } b\}$ [unify (6)]
 - $\{X = N \text{ with } a, Z = a, a \notin \emptyset \text{ with } a \text{ with } b\}$ [notmember (1) (1) (2)]
 - $\{X = N \text{ with } a, Z = a, a \neq a, a \neq b\}$ [notequal (1)]
 - $\{False\}$ (a canonical constraint)
- 5.2 $\{X = N \text{ with } a, N = M \text{ with } Z, Z \notin \emptyset \text{ with } a \text{ with } b\}$ [unify (6)]
 - $\{X = M \text{ with } Z \text{ with } a, N = M \text{ with } Z, Z \notin \emptyset \text{ with } a \text{ with } b\}$
[notmember (1) (1) (2)]
 - $\{X = M \text{ with } Z \text{ with } a, N = M \text{ with } Z, Z \neq a, Z \neq b\}$
(a canonical constraint).

6 Conclusions and future work

We have analyzed problems and solutions related to the introduction of finite set formers and basic operations on sets in a logic programming language. It has been shown that a good choice is starting with a CLP-scheme whose signature Σ is endowed with two functional symbols, \emptyset for the *emptyset* and **with** for the *set construction symbol*, using the symbols \in , \notin , $=$, \neq as *constraint predicate symbols*. Other usual set operators (such as \subseteq , \cup , etc.) have been shown to be definable in this language.

As mentioned in Section 2, a number of possible extensions to this basic language could be explored in the future. In particular, as concerns the kind of sets one wants to deal with, we are studying the problem of defining and incorporating *hypersets* and *multisets* in the CLP-based framework defined in this paper.

Also, such a framework turns out to be well suited to accommodate for *intensional set formers*, provided the language is endowed with some form of negation. *Intensional negation* seems to be adequate to this purpose [5].

A general framework for the design of languages manipulating decidable theories based on a modular extension of the resolution algorithm is presented in [21]. A comparison between it and the CLP-scheme seems to be promising.

Finally, a WAM-based implementation of the $\{\log\}$ language has been developed [10] based on the definition of the language given in [7]. A new implementation of the language exploiting the CLP technology is planned as a future work.

Acknowledgments

We have enjoyed useful discussions with D. Aliffi, P. Bruscoli, G. Levi, A. Policriti, E. Pontelli, E. Omodeo, and A. Roncato.

References

- [1] S. Abiteboul, and S.Grumbach. A Rule-Based Language with Functions and Sets. *ACM Trans. on Database Systems*, vol.16, n.1, 1991.
- [2] D. Aliffi, A. Dovier, E.G. Omodeo, and G. Rossi. Unification of hypersets terms. Presented at *ICLP'93 Workshop on Logic Programming with Sets*, Budapest, June 1993.
- [3] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Sets and Negation in a Logic Database Language. *Proc. 6th ACM SIGMOD Symp.*, 1987.
- [4] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set Constructors in a Logic Database Language. *J. of Logic Programming*, vol.10, n.3, 1991.
- [5] P. Bruscoli, A. Dovier, E.Pontelli, and G. Rossi. Extensional and Intensional Sets in CLP with Intensional Negation. Presented at *2nd COMPULOG-Network Subgroup Meeting on Programming Languages*, Pisa, May 1993.
- [6] W.Büttner. Unification in the Data Structure Sets. *Proc. 8th Int. Conf. on Automated Deduction*, Oxford, July 1986, 470-488.
- [7] A. Dovier, E.G. Omodeo, E. Pontelli, and G. Rossi. $\{\log\}$: A Logic Programming Language with Finite Sets. In *Logic Programming: Proc. of the 8th Int. Conf. (K.Furukawa, ed.)*, The MIT Press, 1991.
- [8] A. Dovier, E.G. Omodeo, E. Pontelli and G. Rossi. Embedding Finite Sets in a Logic Programming Language. *Research Report*, University of Rome, *La Sapienza*, June 1993.
- [9] A. Dovier, E.G. Omodeo, E. Pontelli, and G. Rossi. Embedding finite sets in a logic programming language. In *Extension of Logic Programming (E.Lamma, P.Mello, eds)*, *Lecture Notes in Artificial Intelligence*, No. 660, Springer Verlag, 1992.
- [10] A. Dovier and E.Pontelli. A WAM-based implementation of a Logic Language with Sets. *PLILP'93*, Tallinn, August 1993.
- [11] A. Dovier. A Language with Finite Sets embedded in the CLP-scheme. Presented at *WELP'93*, St.Andrews, March 1993.

- [12] P.M.Hill and J.W.Lloyd. The Gödel Programming Language. *Technical Report, CSTR-92-27*, University of Bristol, Department of Computer Science, 1992.
- [13] J. Jaffar and J.L. Lassez. Constraint Logic Programming. *Research Report*, June 1986.
- [14] G.M. Kuper. Logic programming with sets. *J. of Computer and System Sciences*, vol.41, n.1, 1990.
- [15] B. Jayaraman and D.A. Plaisted. Programming with Equations, Subsets and Relations. *Proc. of NACLP89*, Cleveland, 1989.
- [16] B. Jayaraman. Implementation of Subset-Equational Programs. *J. of Logic Programming*, vol.12, n.4, 1992.
- [17] B. Legeard and E. Legros. CLPS: A Set Constraint Logic Programming Language. *Research Report*, Laboratoire d'automatique de Besançon, Institut de Productique, Besançon, France, 1991.
- [18] M.Livesey and J.Siekmann. Unification of Sets and Multisets. Int. Report. MEMO SEKI-76-II, Institut für Informatik I, Univ. Karlsruhe, 1976.
- [19] E.G. Omodeo, A. Policriti and G. Rossi. Che genere di insiemi/multiinsiemi/iperinsiemi incorporare nella programmazione logica? (in Italian). In *Proc. GULP'93 - 8th Italian Conf. on Logic Programming*, June 1993.
- [20] F.Parlamento and A.Policriti. Decision procedures for elementary sublanguages of set theory. IX. Unsolvability of the decision problem for a restricted subclass of Δ_0 -formulas in set theory. *Communications of Pure and Applied Mathematics*, 41, 1988.
- [21] A.Policriti and J.T.Schwartz. T-Theorem Proving. *Research Report*, Univ. of Udine and Courant Inst. of Math. Sciences, New York, 1992.
- [22] O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of set terms in the logic data language (LDL). *J. of Logic Programming*, vol.12, n.1, 1992.
- [23] R.Sigal. Desiderata for Logic Programming with sets. *GULP89: 4th Italian Conf. on Logic Programming*, Bologna, 1989.
- [24] P.J. Stuckey. Constructive Negation for Constraint Logic Programming. *Proc. 6th IEEE Symp. on Logic In Computer Science*, IEEE Computer Society Press, 1991.
- [25] C.Walinsky. CLP(S*): Constraint Logic Programming with Regular Sets. *Proc. 6th Int. Conf. on Logic Programming*, Lisboa, 1989.