On the Representation and Management of Finite Sets in CLP languages

Agostino Dovier
Università di Verona, Italy
dovier@sci.univr.it

Carla Piazza and **Gianfranco Rossi**
Università di Parma, Dipartimento di Matematica
Via M. D'Azeglio 85/A, 43100 Parma, Italy
gianfr@prmat.math.unipr.it

Enrico Pontelli
New Mexico State University, USA
epontell@cs.nmsu.edu

Abstract

We review and compare the main techniques considered to represent finite sets in logic languages. We present a technique that combines the benefits of the previous techniques, avoiding their drawbacks. We show how to verify satisfiability of any conjunction of (positive and negative) literals based on $=, \subseteq, \in$, and $\cup, \cap, \setminus$, and $||$, viewed as predicate symbols, in a (hybrid) universe of finite sets. We also show that $\cup$ and $||$ (i.e., disjointness of two sets) are sufficient to represent all the above mentioned operations.

1 Introduction

We consider the problem of representing and manipulating *hereditarily finite well-founded sets* in the framework of CLP languages. *Hereditarily finite* means containing a *finite* number of elements that can be either memberless objects or *hereditarily finite* sets. *Hybrid* means that they can involve free Herbrand functors, i.e., distinct from the set constructor and the empty-set symbol $\emptyset$.

In the literature two main approaches have been adopted for the (extensional) representation of sets: *ACI*-terms [2, 5, 21], and list-like terms [4, 15, 7, 13]. Each has advantages and disadvantages over the other.

Regarding operations on sets, they include primarily the ability to *compare* sets. This, when considering the most general case of non-ground and partially specified sets, amounts to solving a (non trivial) *set unification problem* [2, 7, 1, 26]. Besides equality, a number of other basic set operations are usually required for dealing with sets, like membership ($\in$), inclusion and strict inclusion ($\subseteq$, $\subset$), union ($\cup$), intersection ($\cap$), and difference ($\setminus$). These operations are conveniently viewed as *constraints.* According to this view, systems of (set) constraints are tested as a whole for satisfiability by suitable *constraint solvers* [14]. The need to adopt a constraint-based management of set operations derives from the impossibility or impracticality of producing extensional representations of the solutions of such operations. Which operations are provided as constraints and which are programmed in the language is also an important language design decision.

In this paper we present a technique for representing hereditarily well-founded sets which combines the benefits of the two above mentioned techniques, avoiding their drawbacks. This technique is based on the use of the function symbol $\{\cdot\,|\,\cdot\}$, interpreted as the element insertion operation, used as set constructor, along with two predicate symbols, $\cup_3$ and $||$, respectively interpreted as union and disjointness, as the only primitive constraints directly supported by the language. We show that this choice allows to represent nested sets of any depth (not allowed when sets are represented as ACI-terms) as well as to represent complex unification problems involving more than one set variable, such as $X_1 \cup X_2 \cup X_3 = \{a, b\}$ (not allowed when using the list-like representation). Moreover, it allows all other basic set operation (such as $\cap$, $\subseteq$) to be expressed in a quite straightforward way.

The paper is organized as follows. In Sect. 2 we review the main techniques employed to represent finite sets in logic languages. In Sect. 3 we describe the syntax and semantics of the language we have chosen for our proposal. In Sect. 4 we show how to implement in the language itself the main set-theoretical operations. In Sect. 5 we describe in detail the constraint solver for our language, first dealing only with non-colored sets, and then extending the algorithms to the more general case of colored sets.

2 Set representations

We assume all the definitions and notations usually employed in unification theory (e.g., [18, 22]). $\Sigma = \Pi \cup \mathcal{F}$ will be used to denote a signature with *arity* function $ar : \Sigma \longrightarrow \mathbb{N}$. $\mathcal{V}$ is a denumerable set of logical variables disjoint from Σ. If t is a term, then $vars(t)$ denote the set of variables occurring in t. Capital letters X, Y, Z, etc. are used to represent variables; f, g, etc. stand for function symbols (i.e., elements of $\mathcal{F}$). We assume the usual notions (e.g., [11]) of T-unifier, complete set of T-unifiers, and related notions involved in the general theory of unification w.r.t. an equational theory T.

There are three main ways of representing sets as terms:[1]

- using the binary union symbol $\cup$ as the set constructor, as done, for instance, in [2, 5, 21];
- using the binary element insertion operator $\{\cdot\,|\,\cdot\}$ as the set constructor, as done, for instance, in [4, 15, 7, 13];

[1]A few other proposals for logic programming with sets can be hardly classified as following one of these approaches. In CLPS [19] all three approaches appear viable, as no choice is explicitly made. In [12], instead, sets are intended as subsets of a finite domain D of objects. At the language level, each ground set is represented as an individual constant (where all constants are partially ordered to reflect the $\subseteq$ lattice).

- using an infinite collection of function symbols of different arity [17, 25]—the set $\{a_1, \ldots, a_n\}$ is encoded by the term $\{_n(a_1, \ldots, a_n)$, using the n-ary functor $\{_n$.

We will concentrate on the first two methods only. The third one, in fact, does not suit to our purposes. In particular, stating equality in axiomatic form would require a non-trivial axiom schema like the following one: *for each pair of natural numbers m and n,*

$$\{_m(X_1, \ldots, X_m) = \{_n(Y_1, \ldots, Y_n) \quad \leftrightarrow \quad \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n} X_i = Y_j \wedge \bigwedge_{j=1}^{n} \bigvee_{i=1}^{m} X_i = Y_j .$$

The two considered methods are compared as concerns the key problem of (set) unification: given any two $\mathcal{F} \cup \mathcal{V}$ terms ℓ and r, find a complete set of T-unifiers of $\ell = r$, where T is an equational theory describing the relevant properties of the set constructor symbols.

2.1 The ∪-based representation

$\mathcal{F}$ contains the binary function symbol $\cup$ and the constant symbol $\emptyset$. $\cup$ fulfills the equational axioms:

$$\begin{array}{lrcl} (A) & (X \cup Y) \cup Z & = & X \cup (Y \cup Z) \\ (C) & X \cup Y & = & Y \cup X \\ (I) & X \cup X & = & X \end{array}$$

while $\emptyset$ is the *identity* of the operation $\cup$:

$$(1) \quad X \cup \emptyset \;=\; X\,.$$

Intuitively, $\cup$ and $\emptyset$ have the meaning of the set union operator and emptyset, respectively.

2.1.1 Set unification algorithm: Flat sets

When $\mathcal{F}$ is composed by $\cup$, $\emptyset$, and by an arbitrary number of constant symbols, the unification problem falls into *ACI*1*-unification with constants.* Various solutions to this problem have been studied in the literature [2, 5, 21]. A viable solution is, for example, the one proposed by Baader and Büttner [2], based on the use of *ACI-matrices.* An ACI-matrix is a boolean matrix which describes a most general unifier (mgu) for a given ACI problem. The columns of the matrix are associated with the variables present in the original problem, while the rows are associated with the constants present in the unification problem as well as to a set of new variables. Mechanical procedures can be devised [2] to generate a finite collection of ACI-matrices representing a minimal and complete set of unifiers.

Example 2.1 *The matching problem $X_1 \cup X_2 \cup X_3 = a \cup b$ can be associated with $(2^3 - 1)^2 = 49$ ACI-matrices (in each row there must be at least one 1). Two of them, for example, are:*

$$\begin{array}{ccc|c} X_1 & X_2 & X_3 & \\ \hline 0 & 1 & 1 & a \\ 1 & 0 & 1 & b \end{array} \qquad \begin{array}{ccc|c} X_1 & X_2 & X_3 & \\ \hline 0 & 0 & 1 & a \\ 0 & 1 & 0 & b \end{array}$$

yielding, resp., the unifiers: $[X_1/b, X_2/a, X_3/a \cup b]$, $[X_1/\emptyset, X_2/b, X_3/a]$.

In [2] the number of mgu's needed to cover the solution space (actually, exactly the unifiers generated by the algorithm) is made explicit.

2.1.2 Set unification algorithm: Nested sets

$ACI1$-unification with constants does not distinguish explicitly between sets and elements of sets. This makes it difficult to handle set unification when sets are defined by enumerating their elements, especially when elements are allowed to be variables. For example, the problem

$$\{X_1, X_2, X_3\} = \{a, b\} \quad (1)$$

(which admits 6 distinct solutions) is difficult to handle using $ACI1$-unification. In fact, one could map this into the $ACI1$-unification problem of Example 2.1, by interpreting the constants a and b as the singleton sets $\{a\}$ and $\{b\}$ (both in the formulation and in the unifiers), and then "filtering" the 49 distinct $ACI1$-unifiers, removing the solutions in which (at least) one of the X_i's is mapped to $\emptyset$ or to $a \cup b$ (in general to the union of more than one distinct constant). This is an impractical way of solving this problem in the general case.[2] Furthermore, this technique does not allow nested sets to be taken into account at all.

A viable solutions to these problems, still using the $\cup$-based representation, is obtained by assuming that a unary (free) operator $\{\cdot\}$ is present in Σ. The set $\{s_1, \dots, s_m\}$ can be described as $\{s_1\} \cup \dots \cup \{s_m\}$. To our knowledge, the only proposals that have addressed this problem so far are [20, 16, 3]. In particular, [3] shows how to combine unification algorithms for equational theories with disjoint signatures and theories. A *general* $ACI1$ unification algorithm (namely, $ACI1$ dealing also with free symbols, such as $\{\cdot\}$) can be obtained by combining $ACI1$-unification for $\emptyset$, $\cup$ and constants (as in Sect. 2.1.1), with unification in the free theory for all other symbols.

The generality of the combination procedure of [3], however, leads to the generation of a large number of non-deterministic choices.[3] In [3] some optimizations are proposed that allow to take advantage of certain features of the equational theories involved (e.g., certain non-deterministic choices can be avoided if one of the theory is a free theory), but it is unclear whether these optimizations can make the approach more practical.

2.2 The $\{\cdot \mid \cdot\}$-based representation

The $\{\cdot \mid \cdot\}$-based representation (or list-like representation) is the approach used in most papers dealing with sets in logic programming languages, such as [4, 7, 13, 15, 26].

[2]E.g., the problem $X_1 \cup \dots \cup X_7 = a \cup b$ admits 16129 unifiers instead of the 126 of $\{X_1, \dots, X_7\} = \{a, b\}$ [1].

[3] The non-deterministic selection of a partition of the variables in the problem generates $O((\frac{n}{2})^n)$ number of choices, while the selection of a given indexing ordering of the variables is of the order $O(n! \times 2^n)$.

$\mathcal{F}$ contains the binary function symbol $\{\cdot\,|\,\cdot\}$, the empty-set constant symbol $\emptyset$, and an arbitrary number of constant and function symbols. The function symbol $\{\cdot\,|\,\cdot\}$ fulfills the equational axioms [6, 7]:

$$\begin{array}{lrcl} (Ab) & \{X\,|\,\{X\,|\,Z\}\} & = & \{X\,|\,Z\} \\ (C\ell) & \{X\,|\,\{Y\,|\,Z\}\} & = & \{Y\,|\,\{X\,|\,Z\}\}. \end{array}$$

Intuitively, $\{\cdot\,|\,\cdot\}$ is interpreted as the *element insertion* operation and $\emptyset$ as the empty set. For the sake of simplicity, we denote by $\{t_0,\ldots,t_n\,|\,t\}$ the term $\{t_0\,|\,\ldots\{t_n\,|\,t\}\ldots\}$; in particular, if $t=\emptyset$, then we write it as $\{t_0,\ldots,t_n\}$.

2.2.1 Set unification algorithms

General (i.e., dealing also with free functors) $(Ab)(C\ell)$*-unification algorithms* have been proposed in [15, 7, 26, 1].

The algorithm in [15] is weaker than the others since its aim is that of solving matching problems instead of unification problems. [7] basically extends it by covering (nested) set unification problems of the form:

$$\{s_1,\ldots,s_m\,|\,s\} = \{t_1,\ldots,t_n\,|\,t\}$$

where $m,n\geq 0$, s_i and t_j can be generic terms and s,t can be either variables (possibly identical) or non-set terms (i.e., terms with an outermost symbol different from $\{\cdot\,|\,\cdot\}$, including $\emptyset$). [26] solves exactly the same problem using an elegant algorithm based on (temporarily generated) membership constraints. Moreover, the algorithm reduces the generation of redundant (namely, repetitions, instances) solutions of [7]. Finally, the algorithm in [1], though less elegant than that of [26], further reduces the number of redundancies. In particular, it is proved to be minimal for a number of sample set unification problems that are proposed as ‘benchmarks’.

Differently from the $\cup$-based representation, the $\{\cdot\,|\,\cdot\}$-based representation naturally accommodates for *nested* sets. Thus, for instance, problem (1) can be rendered directly as $\{X_1,X_2,X_3\}=\{a,b\}$ (that is, using the internal notation, $\{X_1\,|\,\{X_2\,|\,\{X_3\,|\,\emptyset\}\}\}=\{a\,|\,\{b\,|\,\emptyset\}\}$), and all the proposed algorithms [15, 7, 26, 1] return exactly the 6 mgu’s:

$$\begin{array}{ccc} [X_1/a,X_2/a,X_3/b] & [X_1/b,X_2/b,X_3/a] & [X_1/a,X_2/b,X_3/b] \\ [X_1/b,X_2/a,X_3/a] & [X_1/a,X_2/b,X_3/a] & [X_1/b,X_2/a,X_3/b] \end{array}$$

without the need of any filtering of solutions.

Therefore, the $\{\cdot\,|\,\cdot\}$-based representation allows to solve set unification problems which cannot be expressed using $ACI1$-unification with constants. On the other hand, the $\cup$-based representation allows to write set terms that cannot be directly expressed using a list-like representation, as for instance the term $X\cup a\cup Y$ (as well as the terms involved in Example 2.1). The $\{\cdot\,|\,\cdot\}$-based representation, in fact, can only represent the union of a sequence of singletons with, eventually, a single variable.

Thus, the two different representations offer opposite advantages and disadvantages. One of the main contribution of this work is to define a representation of sets which allows to capture the benefits of both the approaches.

3 A (new) proposal for set representation

In this section we propose a new language for representing sets which combines the two approaches of Sect. 2.1 and 2.2, as it uses a $\{\cdot \mid \cdot\}$-based representation and it provides $\cup_3$ as a primitive constraint.

3.1 The language

The language is based on a signature $\Sigma = \Pi \cup \mathcal{F}$, which includes

- the predicate symbols $\cup_3$ (ternary) and $||$ (binary);
- the binary function symbol $\{\cdot \mid \cdot\}$, and the constant symbol $\emptyset$.

In view of the intended interpretation, variables and any terms whose outermost symbol is $\{\cdot \mid \cdot\}$ are called *set terms*, whereas all other terms, including $\emptyset$, are called *non-set terms.* As shown in Sect. 2.2, terms denoting sets can be *nested* at any depth—e.g., $\{a, \{X, \{\emptyset, f(\emptyset)\}\} \mid Y\}$. Observe that sets can be constructed by adding elements to any arbitrary (non-set) term: we call this term the *color* of the set. Sets constructed starting from a non-variable term different from $\emptyset$ are called *colored sets* [7].[4]

The *primitive constraints* are the (positive and negative) literals based on the predicate symbols $\cup_3$ and $||$. A *constraint* is any conjunction of primitive constraints. We will use $\not\pi(t_1, \ldots, t_n)$ to denote $\neg\pi(t_1, \ldots, t_n)$, for $\pi \in \Pi$.

3.2 The Semantics

Different proposals for an axiomatic semantics of terms denoting sets, suitable for CLP languages, have been recently presented (e.g., [7, 10, 6, 9]). In this paper we follow the approach of [9] which has been shown to be well suited for a parametric approach to the design of CLP languages which integrates different kinds of aggregate data objects (namely, sets, multi-sets, lists, and compact-lists).

The set theory, named *Set*, includes, besides the standard equality axioms, the axioms for the existence of the empty set (K), the axioms for the element insertion operation (W), the Clark Equality Theory axioms for the non-set terms (F_1', F_2), along with a weak form of the foundation axiom (F_3^s), and a suitable version of the extensionality axiom (E_k^s). *Set* is now augmented with the axioms for $\cup_3$ and $||$ (see the table below), which pro-

[4] Colored sets are motivated primarily by "technical" reasons. Actually, we could avoid considering colored sets by introducing two sorts, "terms" and "set terms", where the latter is a sub-sort of the former. This would allow us to restrict admissible solutions to well-sorted substitutions only. However, we prefer to take the opposite approach, which does not distinguish between different sorts, allowing colored sets to come into play. Thus the user can take advantage of the additional expressive power of colors (whose potentialities, however, still need to be further investigated).

vides the following intuitive meaning for the two new predicates: $\cup_3(r, s, t)$ means $t = r \cup s$, whereas $s||t$ stands for $s \cap t = \emptyset$.[5]

for all m, n, k and for all $x_0 \cdots x_m y_0 \cdots y_n z_0 \cdots z_k$:	
$(\cup)$	$\left(\begin{array}{l}\cup_3(\{x_1,\dots,x_m \,\vert\, x_0\},\{y_1,\dots,y_n \,\vert\, y_0\},\{z_1,\dots,z_k \,\vert\, z_0\})\wedge \\ \forall w\,(w \notin x_0 \wedge w \notin y_0 \wedge w \notin z_0)\end{array}\right) \leftrightarrow$ $(\{z_1,\dots,z_k\} = \{x_1,\dots,x_m,y_1,\dots,y_n\} \wedge x_0 = y_0 \wedge x_0 = z_0)$
$(\vert\vert)$	$\left(\begin{array}{l}\{x_1,\dots,x_m \,\vert\, x_0\} \vert\vert \{y_1,\dots,y_n \,\vert\, y_0\}\wedge \\ \forall w\,(w \notin x_0 \wedge w \notin y_0)\end{array}\right) \leftrightarrow$ $(\ \forall z\,(z \in \{x_1,\dots,x_m\} \to z \notin \{y_1,\dots,y_n\}) \wedge x_0 = y_0\ )$

The interpretation domain is the ordinary Herbrand Universe H_Σ (or, equivalently, the set of finite trees over Σ), modulo the smallest congruence relation $\cong$ induced by the (Ab) and $(C\ell)$ axioms (denoted $H_\Sigma/\cong$).

The interpretation of $\cup_3$ and $||$ on the domain $H_\Sigma/\cong$ is strictly forced by their axioms. The axiom allows to unite (resp., verify disjointness of) two sets only if they are based on the same color (note that the colors are those terms that are forced by axiom (K) to not contain any elements.)

The interpretation $H_\Sigma/\cong$ can be proved to be a model of the simple theory of sets considered above. Therefore:

Proposition 3.1 *The axiomatic theory $KWE_k^sF_1'F_2F_3^s \cup ||$ and the interpretation $H_\Sigma/\cong$ correspond [14] on the class of formulae constituted by conjunctions of literals based on $=, \in, \cup_3, ||$.*

Finally, it must be observed that the possibility to deal with non-set terms allows one to build (nested) sets with the same elements but different *colors*. This fact naturally induces the introduction of a predicate dealing with colors. Although the theory has been defined without making use of this predicate, the binary predicate symbol $\mathsf{c^{of}}$ (read 'color of') is useful to implement the constraint manager. As a matter of fact, if the color of a term is not a variable, then it is easy to make it explicit. On the contrary, a constraint of the form $\cup_3(X, Y, Z)$ is satisfiable only if X, Y, and Z have the same color. We could make this fact explicit by stating $\mathsf{c^{of}}(X, K) \wedge \mathsf{c^{of}}(Y, K) \wedge \mathsf{c^{of}}(Z, K)$. For this reason, we will make use of the predicate symbol $\mathsf{c^{of}}$, to be handled as a constraint. As our aim is to make the axioms work properly for finite sets only, we do not need to insert directly $\mathsf{c^{of}}$ into the theory, but we expand the language with the definition axioms ($f, g \not\equiv \{\cdot \,|\, \cdot\}$):

$$\begin{array}{rcl}\mathsf{c^{of}}(f(s_1,\dots,s_m), g(t_1,\dots,t_n)) & \text{iff} & f(s_1,\dots,s_m) = g(t_1,\dots,t_n) \\ \mathsf{c^{of}}(\{t \,|\, s\}, k) & \text{iff} & \mathsf{c^{of}}(s, k).\end{array}$$

The interpretation above can be easily completed to model $\mathsf{c^{of}}$ as well.

[5]The theory deals also with equality ($=$) and membership ($\in$). Actually, it would be possible to restate the theory using only $\cup_3$ and $||$ (see Prop. 4.1 and 4.2): for the sake of readability, however, we prefer to use this enlarged signature.

4 Set operations

In this section we show that the $\cup_3$ constraint, along with the other primitive constraint, $||$, if properly managed, allows to express most of the other usual set operations in a quite straightforward way (in particular, without having to resort to any universal quantification).

Proposition 4.1 *Literals based on the predicate symbols $\in$, $=$, $\subseteq$, can be equivalently replaced by literals based only on $\cup_3$.*

Proof. (sketch) $s \in t$ iff $\cup_3(t, t, \{s \mid t\})$; $s \subseteq t$ iff $\cup_3(s, t, t)$; $s = t$ iff $\cup_3(s, s, t)$. For the corresponding negative literals, it is enough to replace $\cup_3$ with $\not\cup_3$. $\Box$

The operation $\cap$ (viewed as ternary predicate symbol) could be easily programmed using (recursive) clauses in the language itself. However, this approach may cause problems when using $\cap_3$ with insufficiently instantiated arguments. For this reason, we have introduced the operation $||$ as a basic constraint symbol; in fact, $||$ allows to express $\cap_3$ literals (and other interesting operations) as simple combination of basic constraints, as shown by the following proposition.[6]

Proposition 4.2 *Literals based on predicate symbols: $\cap$, $\setminus$, and $\triangle$ can be equivalently replaced by conjunctions of literals based on $\cup_3$ and $||$.*

Proof. (sketch) $\setminus(x, y, z)$ denotes $z = x \setminus y$.

$$\begin{array}{rcl}
\cap_3(r,s,t) & \text{iff} & \exists R, S(\cup_3(R,t,r) \wedge \cup_3(S,t,s) \wedge R||S) \\
\not\cap_3(r,s,t) & \text{iff} & \exists T(\cap_3(r,s,T) \wedge T \neq t) \\
\setminus(r,s,t) & \text{iff} & \exists W\,(\cup_3(t,r,r) \wedge \cup_3(s,t,W) \wedge \cup_3(r,W,W) \wedge s||t) \\
\not\setminus(r,s,t) & \text{iff} & \exists T(\setminus(r,s,T) \wedge T \neq t)
\end{array}$$

Similarly for $\triangle$ ($s \triangle t = s \setminus t \cup t \setminus s$). $\Box$

5 Constraint solver

Like in $CLP(Set)$ [7], in order to check satisfiability of a given constraint C, we transform C to an equivalent disjunction of constraints in *solved form*, guaranteed to be satisfiable.

The problem we tackle here extends the satisfiability problem for set unification, shown to be NP-complete in [16]. Thus, it is an NP-hard problem. NP-completeness ensues from [23]. The algorithm we propose here clearly does not belong to NP since it is non-deterministic and it applies syntactic substitutions. However, one could devise implementations of the algorithm adopting standard techniques (e.g., those of [22]) to avoid problems originated by substitutions, in order to achieve better complexity results.

[6] It can be proved that having only $\cup_3$ in the language one cannot define $\cap_3$ in the same "direct" way as done for $=$, $\subseteq$, and $\in$ in Prop. 4.1. Moreover, it can be proved that to express $\cup$ (as well as $\subseteq, \cap$) in the language of $\{\emptyset, \{\cdot \mid \cdot\}, =, \in\}$ universal quantifiers are *needed*. This implies that constraints based on $\cup_3$ are more expressive than those of $CLP(Set)$.

In the algorithms (see Fig. 1) s, s_i, t, t_i denote generic terms, f, g function symbols different from $\{\cdot \mid \cdot\}$, and $X, X_i, Y, Y_i, W, W_i, Z, Z_i$ variables. $\{\_ \mid \_\}$ denotes a generic term which has $\{\cdot \mid \cdot\}$ as its outermost function symbol. The symbol $\equiv$ denotes syntactic equality. If a variable occurs only in the r.h.s. of a rewriting rule, then it must be intended as a 'newly generated' variable, distinct from all the others.

Although Prop. 4.1 allows one to forget '$=$' and '$\in$' literals, we prefer, for the sake of simplicity, to assume that the constraint solver is capable of managing also this kind of constraints. Thanks to Prop. 4.1, however, one can think of them as special cases of the $\cup_3$ constraint handler. Rewriting procedures for them can be obtained immediately from those of [10, 7] and are omitted due to lack of space.

Moreover, in order to make the presentation more readable we find it convenient to split it into two parts. In the first part, we deal only with non-colored sets, that is, we assume that in $\{t \mid s\}$, s is necessarily either $\emptyset$ or a set term, and, that in $\cup_3(r, s, t)$ and in $s||t$, r, s and t are necessarily either $\emptyset$ or set terms. In the second part, this assumption is relaxed and notions and algorithms presented in the first part are extended to the more general case of colored sets. This extension allows us to prove correctness and completeness of the proposed satisfiability checking procedure, without the restriction on the kind of sets considered.

5.1 Non-colored sets

5.1.1 Solved form

A constraint C, different from false, can be conveniently seen as the conjunction of the constraints $C_= \wedge C_{\neq} \wedge C_{\in} \wedge C_{\notin} \wedge C_{\cup_3} \wedge C_{\not\cup_3} \wedge C_{||} \wedge C_{\not||}$, where each C_π ($C_{\not\pi}$) is composed by the positive (negative) literals based on the predicate symbol π, π belonging to $\{=, \in, \cup_3, ||\}$. A primitive constraint c of C is in *solved form* if it is in one of the following forms:

(i) $X = t$, and X does not occur neither in t nor in the rest of C;

(ii) $X \neq t$, and X does not occur in t;

(iii) $t \notin X$, and X does not occur in t;

(iv) $\cup_3(X_1, X_2, X_3)$, $X_1 \not\equiv X_2$, and $X_1 \neq t_1$, $X_2 \neq t_2$, $X_3 \neq t_3$, t_1, t_2, t_3 any terms, do not occur in C;

(v) $X_1||X_2$, $X_1 \not\equiv X_2$.

A constraint C (C_π) is in solved form if it is empty or all its components are simultaneously in solved form (observe that the primitive constraints based on $\in$, $\not\cup_3$, and $\not||$ are completely eliminated; hence, $C_\in$, $C_{\not\cup_3}$, and $C_{\not||}$ are empty whenever C is in solved form).

For $C_=, C_{\neq}, C_\in$, and $C_{\notin}$ the conditions for the solved forms are the same as in [7, 10], as they capture the notion of a constraint which cannot be further simplified. The conditions for $C_{\cup_3}$ are aimed at avoiding situations like $\cup_3(X, Y, Z) \wedge \cup_3(X, Y, W) \wedge Z \neq W$, which are not satisfiable. The condition for $C_{||}$ are aimed at avoiding a situation like $X||X \wedge X \neq \emptyset$, which is also clearly not satisfiable in the context of non-colored sets.

The notion of solved form plays a fundamental role in the definition of the constraint satisfiability procedure, as emerges from the following theorem.

Theorem 5.1 *Let C be a constraint in solved form. Then C is satisfiable in $H_\Sigma / \cong$.*

Proof. (sketch) The proof is basically the construction of a mapping for the variables of C into H_Σ. The construction is divided into two parts. In the first part, $C_=$ is not considered. A solution for the other constraints is computed by looking for mappings of the form

$$X_i \mapsto \underbrace{\{\cdots\{}_{n_i}\emptyset\}\cdots\}$$

fulfilling all $\neq, ||, \notin$, and $\cup_3$ constraints. In particular, the variables appearing in $\cup_3$ are mapped into $\emptyset$ ($n_i = 0$) and the numbers n_i for the other variables are computed choosing one possible solution of a trivial integer system of inequations obtained by analyzing the "depth" of the occurrences of the variables in terms (to this aim, $||$ and $\neq$ constraints are treated in the same way). Then, all the variables occurring only in r.h.s. of equations of $C_=$ are bound to $\emptyset$ and the mappings for the variables of the l.h.s. are bound to the uniquely induced (ground) substitution. Complete proof can be found in [24]. A proof for the simpler cases, $C_=, C_{\neq}, C_{\in}, C_{\notin}$, can be found in [6]. □

5.1.2 Rewriting rules

The constraint satisfiability test is performed by a procedure, $SAT_{\mathcal{SET}}$, which uses a distinct (non-deterministic) rewriting algorithm for each distinct C_π. Algorithms for $C_=$, $C_{\neq}$, $C_{\in}$, $C_{\notin}$ can be found in [10, 6]. In this paper we present the algorithms union for $\cup_3$, disj for $||$, not_union for $\not\cup_3$, and not_disj for $\not||$ (see Figure 1).

5.1.3 Constraint satisfiability

For any given constraint C, the combination of the rewriting procedures of the previous section allows to compute, through non-determinism, either false or a finite collection of constraints in solved form, whose disjunction is equi-satisfiable to C in the theory presented (provided we restrict to non-colored sets). The constraint satisfiability procedure $SAT_{\mathcal{SET}}$ uses the following function STEP to call, in the proper order, the rewriting procedures:

$$\begin{array}{ll}\mathsf{STEP}(C): & \mathsf{not\_union}(C); \mathsf{not\_disj}(C); \mathsf{member}(C);\\ & \mathsf{union}(C); \mathsf{disj}(C); \mathsf{equal}(C); \mathsf{not\_member}(C); \mathsf{not\_equal}(C).\end{array}$$

$SAT_{\mathcal{SET}}$ repeatedly calls STEP until a fix-point is reached (i.e., the resulting constraint is in solved form and no further rewriting applies to it):

$$SAT_{\mathcal{SET}}(C): \quad \mathsf{while}\ (\mathsf{STEP}(C) \neq C)\ \mathsf{do}\ C := \mathsf{STEP}(C).$$

Correctness and completeness of $SAT_{\mathcal{SET}}$ is stated by the following theorem:

```
union($C$) :
    while $C_{\cup_3}$ is not in solved form and $C \neq$ false do
        apply any of the following rules to any
        (conjunction of) primitive constraint(s) of $C$:
```

(1)	$\cup_3(s,s,t)$	$\mapsto$	$s = t$
(2)	$\cup_3(s,t,\emptyset)$	$\mapsto$	$s = \emptyset \wedge t = \emptyset$
(3)	$\cup_3(s_1,s_2,\{t_1 \mid t_2\})$	$\mapsto$	$\{t_1 \mid t_2\} = \{t_1 \mid X\} \wedge t_1 \notin X \wedge$ *any of* (i) $s_1 = \{t_1 \mid Z\} \wedge \cup_3(Z,s_2,X)$ (ii) $s_2 = \{t_1 \mid Z\} \wedge \cup_3(s_1,Z,X)$ (iii) $s_1 = \{t_1 \mid Z\} \wedge s_2 = \{t_1 \mid W\} \wedge \cup_3(Z,W,X)$
(4)	$\cup_3(\emptyset,t,X)$ or $\cup_3(t,\emptyset,X)$	$\mapsto$	$X = t$
(5)	$\cup_3(\{t_1 \mid t_2\},t,X)$ or $\cup_3(t,\{t_1 \mid t_2\},X)$	$\mapsto$	$\{t_1 \mid t_2\} = \{t_1 \mid Z\} \wedge t_1 \notin Z \wedge X = \{t_1 \mid Y\} \wedge t_1 \notin Y \wedge$ *any of* (i) $\cup_3(Z,t,Y)$ (ii) $t = \{t_1 \mid W\} \wedge t_1 \notin W \wedge \cup_3(Z,W,Y)$
(6)	$\cup_3(X,Y,Z) \wedge Z \neq t$	$\mapsto$	$\cup_3(X,Y,Z) \wedge$ *any of* (i) $Z = \{Z_1 \mid Z_2\} \wedge Z_1 \notin t$ (ii) $t = \{Z_1 \mid Z_2\} \wedge Z_1 \notin Z$ (iii) $Z = \emptyset \wedge t \neq \emptyset$
(7)	$\cup_3(X,Y,Z) \wedge X \neq t$ or $\cup_3(Y,X,Z) \wedge X \neq t$	$\mapsto$	$\cup_3(X,Y,Z) \wedge$ *any of* (i) $X = \{Z_1 \mid Z_2\} \wedge Z_1 \notin t$ (ii) $t = \{Z_1 \mid Z_2\} \wedge Z_1 \notin X$ (iii) $X = \emptyset \wedge t \neq \emptyset$

```
disj($C$) :
    while $C_{||}$ is not in solved form and $C \neq$ false do
    apply any of the following rules to any primitive constraint of $C$:
```

(1)	$\emptyset \,||\, t$ or $t \,||\, \emptyset$	$\mapsto$	*true*
(2)	$\{t_1 \mid t_2\} \,||\, X$ or $X || \{t_1 \mid t_2\}$	$\mapsto$	$t_1 \notin X \wedge X \,||\, t_2$
(3)	$\{t_1 \mid s_1\} \,||\, \{t_2 \mid s_2\}$	$\mapsto$	$t_1 \neq t_2 \wedge t_1 \notin s_2 \wedge t_2 \notin s_1 \wedge s_1 || s_2$
(4)	$X \,||\, X$	$\mapsto$	$X = \emptyset$

```
not_union($C$) :
    while $C_{\not\cup_3}$ is not in solved form and $C \neq$ false do
    apply the following rule to any primitive constraint of $C$:
```

(1)	$\not\cup_3(s_1,s_2,s_3)$	$\mapsto$	*any of* (i) $X \in s_3 \wedge X \notin s_1 \wedge X \notin s_2$ (ii) $X \in s_1 \wedge X \notin s_3$ (iii) $X \in s_2 \wedge X \notin s_3$

```
not_disj($C$) :
    while $C_{\not||}$ is not in solved form and $C \neq$ false do
    apply the following rule to any primitive constraint of $C$:
```

(1)	$s \not|| t$	$\mapsto$	$X \in s \wedge X \in t$

Figure 1: Rewriting Procedures

Theorem 5.2 *Let C be a constraint not involving any colored sets and C'_i be a constraint in solved form returned by $SAT_{\mathcal{SET}}(C)$ other than* false. *Then $Set \models C \leftrightarrow \vec{\exists} \bigvee_i C'_i$.*

Proof. (sketch) The proof is by case analysis of all rules of all the algorithms presented. For the procedures equal, not_equal, member, and not_member, the result follows from the corresponding proofs in [10].

Let us analyze briefly the other procedures. Correctness (and completeness) of rule (1) of union follows directly from Prop. 4.1. Rules (2) and (4) are the cases in which at least one of the arguments is $\emptyset$. Rules (3) and (5) are the cases in which at least one of the arguments is surely non-empty. Rules (5), (6), and (7) are needed to reach solved form. Their correctness and completeness derive from axioms $(\cup), (K), (E^s_k), (W)$. The same holds for the single rule of not_union. Correctness and completeness of rules (2) and (3) of disj follow immediately from axiom $(||)$. The same consideration holds for the unique rule of not_disj. Correctness and completeness of rules (1) and (4) of disj follow from axiom $(||)$ and from (K), instantiated to the constant symbol $\emptyset$. □

The termination of $SAT_{\mathcal{SET}}$ and the finiteness of the number of non-deterministic choices generated during its computation ensure the finiteness of the number of constraints non-deterministically returned by $SAT_{\mathcal{SET}}$.

Theorem 5.3 *The $SAT_{\mathcal{SET}}$ procedure terminates for every constraint C.*

Proof. In [7] it is shown that the iteration of the procedures member, not_member, equal, and not_equal on a system of $\{=, \neq, \in, \notin\}$-constraints always terminates. We can use this result to provide an argument for the termination of the $SAT_{\mathcal{SET}}$ procedure proposed here. The proof is divided into two parts:
(i) show that each individual rewriting procedure terminates;
(ii) show that $SAT_{\mathcal{SET}}$ globally terminates.
Step (i) is straightforward. In step (ii), we can safely ignore not_union and not_disj, as they are going to be applied only once. Let us assume that an infinite computation can occur; for the time being let us ignore rules (6), (7) of the procedure union. The body of $SAT_{\mathcal{SET}}$ is organized as union(disj(equal(not_member(not_equal(C))))).

From [7] we know that the sequence equal(not_member(not_equal(C))) cannot lead to non-terminating executions. Thus, a non-terminating execution must contain infinite activations of at least one of union and disj. The disj by itself cannot cause infinite executions. This case can be easily ruled out observing that:

- since no new $||$ are ever created, an infinite computation must originate from infinite reductions of one $||$-constraint;
- the two rules which can be applied repeatedly (without removing the constraint from the system) are the rule (2) and (3). They both create only negative knowledge which cannot further feed the production of bindings.

Thus an infinite computation will require infinite activations of union. This implies that there is a $\cup_3$-constraint which is reduced infinite times, using either rule (3) or (5). But, if this is the case, then it will be possible to produce (by fixing one argument) a constraint system not involving any $\cup_3$-constraints which will involve the same infinite computation, and which would contradict the results in [7].

In the previous discussion we have ignored the rules (6) and (7) of the union procedure. It can be shown [24] that their presence does not endanger the global termination. □

5.2 Colored sets

In order to deal with colored sets, we first need to update the notion of solved form, in order to properly reflect the presence of colored sets. This is obtained by adding, in cases *(iv)* and *(v)*, the further condition that all the involved sets must have the *same color t*. This means that C must contain the primitive constraints $\mathsf{c^{of}}(X_i, t)$, $i = 1, 2, (3)$, t term not of the form $\{\_ \,|\, \_\}$.

Moreover, a new solved form for $\mathsf{c^{of}}$ constraints must be introduced. This solved form requires the verification of two special conditions, the *sufficient wideness of $\mathcal{F}$* condition and the *acyclicity* condition. The first condition, formally introduced in [24], is used to guarantee the ability of the language to generate a sufficient number of distinct colors to satisfy all the $\mathsf{c^{of}}$ atoms and $\neq$ literals present in the constraint. For example, assume $\mathcal{F} = \{\{\cdot \,|\, \cdot\}, \emptyset, c_1, \dots, c_{n-1}\}$ and C has the form:

$$\bigwedge_{i=0}^{n} \mathsf{c^{of}}(X_i, Y_i) \ \wedge \ \bigwedge_{j,k=0,j\neq k}^{n} Y_j \neq Y_k \ .$$

This constraint is not satisfiable, as $\mathcal{F}$ allows the creation of at most n different colors. A single function symbol of arity greater or equal to 1 guarantees the sufficient wideness of $\mathcal{F}$.

The *acyclicity condition* represents a generalization of the concept of occur-check. Testing the acyclicity condition means checking that C does not contain any chain of the form $\mathsf{c^{of}}(X_1, f_1(X_2)) \wedge \mathsf{c^{of}}(X_2, f_2(X_3)) \wedge \dots \wedge \mathsf{c^{of}}(X_n, f_n(X_1))$ which is clearly unsatisfiable. Therefore, the following case is added to the definition of a constraint C in solved form:

(vi) $\mathsf{c^{of}}(X, t)$, $X \notin \mathit{vars}(t)$, $t \not\equiv \{\_ \,|\, \_\}$, and $\mathsf{c^{of}}(X, s)$, $s \not\equiv t$, does not occur in C, and the sufficient wideness and the acyclicity conditions hold.

A constraint in solved form involving $\mathsf{c^{of}}$ primitive constraints is still satisfiable. To prove this fact one can extend proof of Theorem 5.1 to deal with this kind of constraints. In the second part of the proof, the solution computed in the first part is refined by mapping the variables occurring in $C_{\mathsf{c^{of}}}, C_{\not\mathsf{c}^{\mathsf{of}}}$ in different colors fulfilling the constraints and modifying the colors of the images of the mappings computed in the first step.

A new constraint rewriting algorithm, called colors, is developed for dealing with the $C_{\mathsf{c^{of}}}$ and the $C_{\not\mathsf{c}^{\mathsf{of}}}$ parts of a given constraint C. Accordingly, the STEP function is extended by including a call to colors after the call to not_equal. The colors procedure reduces $\mathsf{c^{of}}$ and $\not\mathsf{c}^{\mathsf{of}}$ constraints, possibly generating some $=$ and $\neq$ constraints, and possibly returning false if $\mathcal{F}$ is not wide enough.

Algorithms of Sect. 5 must be revised in order to correctly manipulate the colors of sets. For instance, rule (4) of union is modified as follows:

(4.1)	$\cup_3(f(s_1,\dots,s_m), g(t_1,\dots,t_n), X)$	$\mapsto$	$f(s_1,\dots,s_m) = g(t_1,\dots,t_n) \wedge X = f(s_1,\dots,s_m)$
(4.2)	$\cup_3(f(s_1,\dots,s_m), t, X)$ or $\cup_3(t, f(s_1,\dots,s_m), X)$ $t \equiv \{\_ \,|\, \_\}$ or $t \in \mathcal{V}$	$\mapsto$	$X = t \wedge \mathsf{c^{of}}(t, f(s_1,\dots,s_m))$

Similar technical variations are applied to rules (2), (6), and (7) of the union, and for rules (1) and (4) of disj. A new rule (rule (8)) is added to union for imposing that, given the constraint $\cup_3(X, Y, Z)$, X, Y, and Z have the same color. Finally, a few non-deterministic choices dealing with c^{of} are added to the single rule of not_union and not_disj. (Complete constraint handling algorithms, as well as the colors procedure, can be found in [8].)

These extensions allow us finally to obtain the same results of Theorem 5.2 (correctness and completeness of $SAT_{\mathcal{SET}}$) and Theorem 5.3 (termination of $SAT_{\mathcal{SET}}$), but relaxing the restriction to non-colored sets.

To conclude this section, we point out that the $ACI1$ unification problem of Example 2.1, $X_1 \cup X_2 \cup X_3 = \{a, b\}$, as well as other similar unification problems, can be written as a conjunction of primitive constraints:

$$\cup_3(X_1, X_2, X) \wedge \cup_3(X, X_3, \{a, b\}).$$

The execution of $SAT_{\mathcal{SET}}$ on this constraint will return the desired answers.

6 Conclusions

In this work we have compared existing proposals for handling finite sets in CLP languages, and proposed a novel technique, that captures the benefits of the existing ones. The new representation scheme uses $\{\cdot \mid \cdot\}$ as set-constructor and $\cup_3$ as basic constraint predicate. We have described the syntactic and semantics components of the language and presented sound and complete operational semantics and constraint handlers.

Acknowledgements

We wish to thank Klaus U. Schulz for his kindness in suggesting interesting references and results, Alberto Policriti and Desh Ranjan for the comments in Footnotes 6 and 3, respectively. A. Dovier is partially supported by MURST project: *Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software*. E. Pontelli is partially supported by NSF grants HRD 93-53271, INT 95-15256, and CCR 96-25358. The work is partially supported by CNR Grant 97.02426.CT12.

References

[1] P. Arenas-Sánchez and A. Dovier. A Minimality Study for Set Unification. *J. of Functional and Logic Programming*, No. 7, Volume 1997.

[2] F. Baader and W. Büttner. Unification in commutative and idempotent monoids. *Theoretical Computer Science*, 56:345–352, 1988.

[3] F. Baader and K. U. Schulz. Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures. *J. of Symbolic Computation*, 21(2):211–243, 1996.

[4] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set Constructors in a Logic Database Language. *J. of Logic Programming*, 10(3):181–232, 1991.

[5] W. Büttner. Unification in the Data Structure Sets. In J. K. Siekmann, ed., *Proc. of CADE'86*, vol. 230, 470–488. Springer-Verlag, 1986.

[6] A. Dovier. *Computable Set Theory and Logic Programming.* PhD thesis, Università degli Studi di Pisa, March 1996. TD–1/96.

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

[8] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. On the Representation and Management of Finite Sets in CLP-Languages. NMSU-CSTR-97-18, Department of Computer Science, Las Cruces, New Mexico, USA, December 1997.

[9] A. Dovier, A. Policriti, and G. Rossi. Integrating lists, multisets, and sets in a logic programming framework. In F. Baader and K. U. Schulz, eds., *Proc. of FROCOS'96*, vol. 3 of *Applied Logic*, 213–229. Kluwer A. P., 1996.

[10] A. Dovier and G. Rossi. Embedding Extensional Finite Sets in CLP. In *Proc. of ILPS'93*, 540–556. The MIT Press, 1993.

[11] F. Fages and G. Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43:189–200, 1986.

[12] C. Gervet. Conjunto: Constraint Logic Programming with Finite Set Domains. In *Proc. of ILPS'94*, 339–358. The MIT Press, 1994.

[13] P. Hill and J. Lloyd, *The Gödel Programming Language.* The MIT Press, 1994.

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

[15] B. Jayaraman and D. A. Plaisted. Programming with Equations, Subsets and Relations. In *Proc. of NACLP'89*, 1051–1068. The MIT Press, 1989.

[16] D. Kapur and P. Narendran. Complexity of Unification Problems with Associative-Commutative Operators. *J. of Automated Reasoning*, 9:261–288, 1992.

[17] G. M. Kuper. Logic Programming with Sets. *J. of Computer and System Science*, 41(1):44–64, 1990.

[18] J. L. Lassez, M. J. Maher, and K. Marriot. Unification revisited. In *LNCS*, vol. 306, 1986.

[19] B. Legeard and E. Legros. Short overview of the CLPS system. In J. Maluszynsky and M. Wirsing, eds., *Proc. of PLILP'91*, vol. 528 of *LNCS*, 431–433. Springer-Verlag, 1991.

[20] P. Lincoln and T. Christian. Adventures in associative-commutative unification. *J. of Symbolic Computation*, 8(1,2):217–240, 1989.

[21] M. Livesey and J. Siekmann. *Unification of Sets and Multisets.* Technical report, Institut für Informatik I, Universität Karlsruhe, 1976.

[22] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[23] E. G. Omodeo and A. Policriti. Solvable set/hyperset contexts: I. Some decision procedures for the pure, finite case. *Communication on Pure and Applied Mathematics.* 9–10:1123–1155, 1995.

[24] C. Piazza. Analisi e Definizione di Linguaggi di "Set Constraint". *Master Thesis*, Università di Parma, Dip. di Matematica, `http://prmat.math.unipr.it/~gianfr/tesi/union.ps.gz`.

[25] O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of Set Terms in the Logic Data Language (LDL). *J. of Logic Programming*, 12(1):89–120, 1992.

[26] F. Stolzenburg. Membership-Constraint and Complexity in Logic Programming with Sets. In *Proc. of FROCOS'96*, vol. 3 of *Applied Logic*, 285–302. Kluwer A. P., 1996.