J. LOGIC PROGRAMMING 28(1):1–44 (1996)

{log}: A LANGUAGE FOR PROGRAMMING IN LOGIC WITH FINITE SETS

AGOSTINO DOVIER, EUGENIO G. OMODEO,
ENRICO PONTELLI AND GIANFRANCO ROSSI

▷ An extended logic programming language is presented, that embodies the fundamental form of set designation based on the (nesting) element insertion operator. The kind of sets to be handled is characterized both by adaptation of a suitable Herbrand universe and via axioms. Predicates $\in$ and $=$ designating set membership and equality are included in the base language, along with their negative counterparts $\notin$ and $\neq$. A unification algorithm that can cope with set terms is developed and proved correct and terminating. It is proved that by incorporating this new algorithm into SLD resolution and providing suitable treatment of $\in$, $\neq$ and $\notin$ as constraints, one obtains a correct management of the distinguished set predicates. Restricted universal quantifiers are shown to be programmable directly in the extended language, and thus are added to the language as a convenient syntactic extension. A similar solution is shown to be applicable to intensional set-formers, provided either a built-in set collection mechanism or some form of negation in goals and clause bodies is made available. ◁

1. INTRODUCTION

The availability of set data abstractions is widely recognized as a valuable feature of high-level programming languages. In particular, sets can be used conveniently in *rapid software prototyping*, where efficiency is not a primary requirement whereas

Address correspondence to Professor Gianfranco Rossi, Dipartimento di Matematica, Università di Parma, Via M. D'Azeglio 85/A, 43100 Parma, Italy. Email: gianfr@prmat2.math.unipr.it.
Received November 1993; accepted September 1995.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010

0743-1066/96/$15.00
SSDI 0743–1066(95)00147-6

the availability of high-level data and operation abstractions are key features of the implementation language.

Only relatively few programming languages provide sets as primitive objects. Among them, the (nonexecutable) specification language Z [50], the procedural language SETL [45], and the functional languages MIRANDA [53] and ME TOO [41]. More recently, increasing attention has been devoted to embedding sets into *logic programming languages*. Indeed such languages seem to be good candidates for hosting set data abstractions, thanks to their highly declarative nature.

Attention to incorporating sets into logic-based languages has come first from the field of *deductive databases* [1, 6, 7, 33, 42]. Recently, however, a number of papers have addressed the problem also in a wider setting. General-purpose set constructs and basic operations on sets have been added to general logic-based frameworks: pure logic programming languages [16, 17], equational logic languages [28, 29], and Constraint Logic Programming languages [22, 35, 36]. The importance of sets as a high-level data structure in a programming language has been also recognized in `Gödel` [26], which supplies the user with a few basic facilities to define (both extensionally and intensionally) finite sets, and to manipulate them.

Actual Prolog implementations already provide some facilities to support sets in the form of the built-in predicates `setof` and `bagof`. However, it is widely recognized that the definition of these facilities is quite unsatisfactory. Sets are simply represented as *lists* and dealt with as ordinary terms, ignoring all the characteristic properties of sets (e.g., lack of ordering, immateriality of duplicates). In brief, `setof` lacks a precise declarative semantics.

A logic language embodying sets, on the contrary, ought to deal with sets as genuine first-class objects, whose semantics adheres to that specified by a suitable set theory. More precisely, the following issues should be faced adequately in the design of a logic language with sets (some of them are addressed also in [49]):

- Which kind of aggregates one should deal with: multisets, ordinary (well-founded) sets, non-well-founded sets.
- How to conveniently represent such aggregates and by which designations (extensional or intensional) to refer to them.
- 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 clauses, CLP, equational logic language, etc.
- Which kind of applications the language is oriented to.
- How to achieve efficient implementations.

In this paper we define an extended logic language, called {log} (read "set-log") which tries to provide a reasonable and coherent set of answers to the foregoing questions.

Unlike some of the cited proposals, we do not restrict ourselves to any specific application domain, as our aim is the development of a *general-purpose* programming language. Our starting point is a *pure logic programming language*, that is, definite Horn clauses with no extra-logical constructs. This basic language is enriched with a new category of terms, the *set terms*, employed for the *extensional* representation of finite sets. A set term has the form $s\ \mathtt{with}\ t$, denoting the set $\{t\} \cup s$ obtained by adding t as an element to the set s. Very few distinguished predicates, namely *set membership* and *equality*, are added to the base language, along with their negative counterparts ($\notin$ and $\neq$). These have been shown to supply sufficient computational expressivity to cover a wide range of applications. The interpretation domain adopted is an extension of the Herbrand universe, developed according to a proper *axiomatic characterization* of the kind of sets embodied in our language.

Usual SLD resolution is modified in order to incorporate a new unification algorithm, capable of managing set terms in a way complying with the semantic properties of the set constructor `with` (namely, *permutativity* and *absorption*; see below), and to implement the distinguished predicates for set operations.

It is shown that the usual set-theoretic operations, such as union, intersection, etc., can be effectively programmed in this extended language. Also *restricted universal quantifiers* are shown to be expressible in the language itself; they can be added to it as a simple syntactic extension (managed by a suitable preprocessor). Finally, simple *abstract set-formers* are added to the language, providing the user with an intensional means to express sets. It is argued, however, that also such an extension can be programmed directly in the language, provided the latter is endowed with either a built-in set collection mechanism or some form of *negation* in goals and clause bodies.

Throughout the paper we will consistently restrain our investigation only to *finite sets*. The adaptation of the proposed set theory and set handling mechanisms to *multisets*, where members occur with a multiplicity factor, is currently under investigation [24] and is not considered here. Also, we do not consider the treatment of non-conventional sets, akin to the *non-well-founded sets* of [2], among which membership is allowed to form cycles. Sets of this novel kind, sometimes called *hypersets* [5], are studied in [18], where a new unification algorithm capable of handling hypersets is presented.

Finally, we have assumed here a pure logic programming language as the base language. Alternatively, we could have relied on a *CLP scheme* [27], by considering the predefined set predicates as *constraints*. A review of {log} as an instance of the CLP scheme is, in fact, the aim of [22]. The reference to a CLP framework offers the advantage of giving a more uniform treatment of the various set-handling operations. Furthermore, most of the theoretical results developed for CLP can automatically be transferred into the {log} framework. Nevertheless, since our intention here is to give a self-contained presentation of the language, we avoid the intricacies of the full CLP scheme, developing our own operational semantics with the appertaining soundness and completeness results. As an advantage, this approach enables straightforward implementations of our language to be devised by adapting well-known techniques currently applied to conventional Prolog implementations [4].

The paper is organized as follows. Section 2 briefly describes the minimal syntactic extensions to a pure logic programming language required to support set notation. The declarative semantics of the resulting language is presented in Section 3. Section 4 addresses the problem of extending unification in order to respect the intrinsic declarative properties of set terms: a new unification algorithm which can manipulate set terms is described in detail here. Section 5 presents the extended SLD procedure incorporating the new unification algorithm and the management of $\in$, $\notin$ and $\neq$, based on the so-called *constraint canonization algorithm*. Section 6 shows how restricted universal quantifiers can be defined in our language. Various {log} programs illustrating the use of the set manipulation facilities are shown in

Section 7. Section 8 analyzes the issues related to the introduction of intensional set formers, focusing on the strong link existing between intensional definitions and negation. Sections 9 and 10 terminate the work, by drawing a comparison between {log} and other logic languages dealing with sets proposed by different researchers and by sketching the lines for future extensions.

2. {log} SYNTAX

The main purpose of this section is to illustrate the syntactic representation adopted in {log} to express *extensionally* defined sets. All that is presupposed is the availability of:

- An interpreted constant {} used to represent the empty set, $\emptyset$.
- A binary function symbol `with` (used as an infix, left-associative operator), to be interpreted as follows: $s\,\texttt{with}\,t$ stands for the set that results from adding t as an element to the set s.[1]

Apart from these two symbols, {log} contains the usual equipment of clausal Horn logic [38] along with distinguished predicates for set membership and equality ($\in$ and $=$) and their negative counterparts ($\notin$ and $\neq$).

The extensional representation for sets mentioned above is provided by a collection $\mathcal{G}$ of ground terms, defined as the smallest set such that

- the constant {} belongs to $\mathcal{G}$;
- $s\,\texttt{with}\,t$ belongs to $\mathcal{G}$ when$s$ and t are ground terms (not necessarily in $\mathcal{G}$).

In view of the intended interpretation, any s in $\mathcal{G}$ will be called a *set term*. A nonground term t is called a set term if there exists an instantiation σ of the variables in t such that t^σ belongs to $\mathcal{G}$. In particular a variable is a set term. Set terms of the form $t\,\texttt{with}\,t_n\,\texttt{with}\cdots\texttt{with}\,t_1$, $n \geq 1$, where t is *not* a set term, are intended to designate sets based on a *kernel* t other than {}. We call such entities *colored sets*.

For the sake of simplicity, we introduce special syntactic forms to designate set terms: $\{t_1,\ldots,t_n|s\}$ stands for $s\,\texttt{with}\,t_n\,\texttt{with}\cdots\,\texttt{with}\,t_1$, and $\{t_1,\ldots,t_n\}$ stands for $\{\}\,\texttt{with}\,t_n\,\texttt{with}\cdots\texttt{with}\,t_1$, where $s, t_1, \ldots, t_n$ are terms. For example:

- f(a,{}), is a term, but not a set term;
- {2,g(3),a}, is a ground set term;
- {}, {1,X,Y,2}, {1,1,{2,{}}, f(a,{b})} and any term $\{t_1,\ldots,t_n \mid R\}$ with a "tail" variable R, are set terms;
- {a | f({b})} is a colored set term based on the kernel f({b}).

For the rest of this paper, we will always represent set terms using the $\{\cdots\}$ notation to simplify program reading. Moreover, we will freely exploit the usual syntactic features of Prolog in addition to the constructs discussed above.

Three sample {log} clauses are:

- q(X) :– X $\in$ {a,b,c,d}, p(X).
- singleton(X) :– X = {Y}.
- in_difference(X,Set1,Set2) :– X $\in$ Set1, X $\notin$ Set2.

[1] Should t already belong to s, the latter will coincide with $s\,\texttt{with}\,t$.

Colored set terms (i.e. terms of the form $\{t_1,\ldots,t_n|t\}$ where t is not a set term) do not designate sets of any conventional kind. Nevertheless, we deem it convenient to always regard $\{t_1,\ldots,t_n|t\}$ as an admissible set term when $t_1,\ldots,t_n,t$ are legal, to make the language structure absolutely uniform and the inference mechanisms (e.g., unification) more straightforward. Furthermore, in certain frameworks (e.g., deductive databases) the availability of different "kinds" of sets may be useful: coloring can be exploited to differentiate sets with the same members representing relations with distinct attributes.

3. {log} DECLARATIVE SEMANTICS

To conveniently interpret the language introduced so far, a richer semantic structure than the one usually associated with definite Horn clauses is needed. The focal point of this extension—which will now be discussed—is the interpretation of the function symbol `with`, which should be interpreted as a set constructor.

We begin by providing the *axioms* of a suitable first-order set theory with equality. The legal interpretations of {log} will be the models of these axioms (cf. [23]). Next we will focus on a privileged interpretation domain formed by terms U_H resulting from a suitable transformation of the classical Herbrand universe [38]. Moreover, we will designate a fixed binary relation over U_H as the privileged interpretation of $\in$ (the predicate $=$ will be interpreted as simple syntactic equality).

3.1. Set Axioms—The Elementary Theory Set

The following list contains the basic axioms composing our theory *Set* (here v,w,x,y,z, and the x_is stand for distinct variables):

(**Z**)	$\forall v\,(v \notin \{\,\})$
(**W**$_)$)	$\forall x\,\forall y\,\forall v\,(v \neq y \rightarrow (v \in \{y \mid x\} \leftrightarrow v \in x))$
(**W**$_2$)	$\forall x\,\forall y\,(y \in \{y \mid x\})$
(**L**)	$\forall x\,\forall y\,(y \in x \rightarrow \exists z(y \notin z \;\&\; x = \{y \mid z\}))$
(**K**$_0$)	$\forall x\,((\forall z\, z \notin x) \rightarrow (x = \mathtt{ker}(x)))$
(**K**$_1$)	$\forall x\,\forall v\,(v \notin \mathtt{ker}(x))$
(**K**$_2$)	$\forall y\,\forall v\,(\mathtt{ker}(\{y \mid v\}) = \mathtt{ker}(v))$
(**E**)	$\forall x\,\forall y\,((\mathtt{ker}(x) = \mathtt{ker}(y) \;\&\; \forall z(z \in x \leftrightarrow z \in y)) \rightarrow x = y)$
(**R**)	$\forall x\,\exists z\,\forall w\,(w \in x \rightarrow (z \in x \;\&\; w \notin z))$
(**U**)	$\forall x_1 \cdots \forall x_n\,\forall v\,(f(x_1,\ldots,x_n) \neq \{\,\} \;\&\; v \notin f(x_1,\ldots,x_n))$ where n is the arity of f and $f/n \notin \{\{\,\}/0, \mathtt{ker}/1, \mathtt{with}/2\}$.

Our theory slightly deviates from the classical ones (i.e., Zermelo-Fraenkel, von Neumann-Bernays-Gödel) in two respects:

- Presence of memberless entities other than $\{\,\}$ (called *urelements* in [30]). It follows from (**U**) and the freeness axioms (see below) that a single function symbol, say $f/1$, can be used to generate infinitely many such entities, namely $f(\{\,\}), f(\{\{\,\}\}), f(\{\{\{\,\}\}\}),\ldots$. Hereafter, we will refer to any nonvariable term whose main functor differs from `with`/2 generically as a *memberless term* (as opposed to a set term; in fact, $\{\,\}$ is the only set term which is memberless).
- Each term t in the interpretation domain has an associated *kernel*, $\mathtt{ker}\,(t)$, denoted by a memberless term. Intuitively speaking, we think of t as resulting

from repeated insertions (possibly none) of members into this initial kernel, insertions being achieved by the operation `with`.

Let us briefly comment upon the extensionality and regularity axioms, (**E**) and (**R**). The first of these states that, in order to be equal, x and y must have the same kernel and the same members. It follows, in view of $(\mathbf{W}_{1,2})$, $(\mathbf{K}_2)$ and (**E**) that `with` has the following properties:

$$\begin{array}{rcll} \{\, Y_1\,,\, Y_2 \,|\, X\,\} & = & \{\, Y_2\,,\, Y_1 \,|\, X\,\} & \text{(permutativity)},\\ \{\, Y\,,\, Y \,|\, X\,\} & = & \{\, Y \,|\, X\,\} & \text{(absorption)}. \end{array}$$

Informally speaking, these identities express the fact that the order of elements in a set is immaterial (permutativity property) and that duplicates are not relevant (absorption property).

Another very useful consequence of $(\mathbf{W}_{1,2})$, $(\mathbf{K}_2)$ and $(\mathbf{E})$ is that

$$Y \in X \leftrightarrow \exists z (X = \{\, Y \,|\, z\,\})\,,$$

whereby one can express membership in terms of equality.

By exploiting the element removal axiom (**L**) as well, one obtains

$$(\star) \qquad (\{\, Y \,|\, X\,\} \;=\; \{\, S \,|\, V\} \,\&\, Y \neq S) \rightarrow \exists\, w\, (\{\, S \,|\, w\,\} \;=\; X \,\&\, \{\, Y \,|\, w\,\} \;=\; V\,)\,,$$

which will prove crucial in our unification algorithm.

The regularity axiom (**R**) states that from each nonempty set x one can choose a member z which belongs to x and does not intersect x. This is a well-known expedient to state that membership does not form cycles.

The theory *Set* is completed by the standard equality axioms (not reported here; cf., e.g., [23]), and by the following *freeness axioms* (cf. [46]):

$$\begin{array}{l} \forall\, x_1 \cdots \forall\, x_n\, \forall\, y_1 \cdots \forall\, y_n\, (f(x_1,\ldots,x_n) \neq g(y_1,\ldots,y_m))\,,\\ \forall\, x_1 \cdots \forall\, x_n\, \forall\, y_1 \cdots \forall\, y_n\, (f(x_1,\ldots,x_n) \;=\; f(y_1,\ldots,y_n) \rightarrow \\ \qquad\qquad\qquad\qquad (x_1 = y_1 \,\&\, \cdots \,\&\, x_n \;=\; y_n))\,,\\ \forall\, x\, (\, t[x] \;\neq\; x\,)\,, \end{array}$$

where f/n and g/m are distinct function symbols which differ also from `ker`/1 and from `with`/2. In addition to requiring—as usual—that $t[x]$ be a term involving the variable x and distinct from it, we must require that $t[x]$ be not a set term.

Technically, one can regard (**U**) as a novel freeness axiom, and (**R**) as an analogue for sets of the *occur* axiom $\forall\, x\, (\, t[x] \neq x\,)$. The regularity axiom (**R**) needs to be strengthened for our purposes, by adding to it

$$(\mathbf{R}') \qquad \forall\, x\, (\, t[x] \notin x \,\&\, \mathtt{ker}\,(x) \neq t[x]\,)\,.$$

3.2. An Adapted Herbrand Universe

We aim in this section at defining the *privileged interpretation domain* U_H, to be used as the basis for the model-theoretic semantics of our language.

Let $\mathcal{F}$ be the collection of function symbols available in the language (in particular $\{\,\}/0\,, \mathtt{with}/2 \in \mathcal{F}$ and $\mathtt{ker}/1 \notin \mathcal{F}$), and let H represent the usual Herbrand universe. To reflect the two properties of `with` previously described (absorption and permutativity), we define an equivalence relation $\sim$ over H: this is the smallest equivalence relation satisfying the analogues of those properties with $\sim$ in place of $=$. U_H is then defined as the set obtained by taking one representative out of each of the equivalence classes forming $H/{\sim}$. Below we will sketch a criterion to constructively select a specific representative from each equivalence class.

Establishing an order $<$ over H can help in filling up the details of this plan. For the sake of simplicity, it is reasonable to assume a total ordering over $\mathcal{F}$ to be given. By exploiting this ordering, $<$ can be defined antilexicographically. This means, among others, that $\{t|r\} < \{u|s\}$ holds when either $t < u$ or t coincides with u and $r < s$.

A ground term g is said to be *canonical* if either g is a constant or every one of its subterms is canonical and, moreover, $t < u$ holds for every subterm of the form $\{u, t|s\}$ of g. U_H will be formed by all canonical terms (it is straightforward to prove that there is one and only one canonical term in each equivalence class modulo $\sim$). Let τ be the function that maps a ground term t to its canonical representative. If t has the form $f(t_1, \dots, t_n)$ with $f/n \neq \mathtt{with}/2$, $\tau(t)$ will be $f(t'_1, \dots, t'_n)$ where $t'_i = \tau(t_i)$ for each i, $1 \leq i \leq n$. If t has the form $\{t_1, \dots, t_n \mid k\}$, $n \geq 1$, where k is a memberless term, then $\tau(t)$ is the term $\{t'_{\pi_1}, \dots, t'_{\pi_m} \mid k'\}$ where k', $t'_{\pi_1}, \dots, t'_{\pi_m}$ are the distinct canonical representatives of k, $t_1, \dots, t_n$ arranged so that $t'_{\pi_m} < \cdots < t'_{\pi_1}$ (by the last requirement, any element repetitions in a set are avoided). The canonical representative of each term will be its value in the privileged interpretation.

To complete the picture of the privileged interpretation, it suffices to add, for any $t, u \in U_H$:

- The kernel $k \equiv \mathtt{ker}(t)$ is obtained by decomposing t in the form

  $$k \;\mathtt{with}\; t_n \;\mathtt{with} \cdots \mathtt{with}\; t_1,$$

  $n \geq 0$, where k is a memberless term (note that if t is itself a memberless term then $\mathtt{ker}(t) = t$).
- Whether the relation $u \in t$ holds or not can be established by decomposing t in the same manner and by checking whether u occurs as one of the t_is.

In order to access the kernel $\mathtt{ker}(t)$ of a term t, the language will provide a predicate $\mathtt{ker\_of}/2$—to be dealt with as a constraint (Section 5)—defined as

$$t_1 \;\mathtt{ker\_of}\; t_2 \;\leftrightarrow\; t_1 = \mathtt{ker}(t_2)$$

Note that the same construction just described was carried out in [22] and [8] to define the single-sorted interpretation domain $\mathcal{A}$ of the algebraic structure $\mathcal{S}$ underlying an instance of the CLP scheme tailoring {log}. As shown there, within the theoretical framework of CLP (cf. [27]) one can prove that $\mathcal{S}$ is solution-compact and that it corresponds to the satisfaction-complete theory *Set* (hence, $\mathcal{S}$ is a model of *Set*).

Here we have chosen to stay within the consolidated theoretical framework of logic programming (cf., e.g., [38]). Accordingly, we have to extend the definition of the term canonization function τ described above to encompass atoms and clauses. We do this as

- For each ground atom $A \equiv p(t_1, \dots, t_n)$,

  $$\tau(A) = p(\tau(t_1), \dots, \tau(t_n)).$$

- For each ground clause $C \equiv (A :- B_1, \cdots, B_n)$,

$$\tau(C) = \tau(A) :- \tau(B_1), \cdots, \tau(B_n).$$

- For each set I composed by ground atoms or ground clauses,

$$\tau(I) = \bigcup_{A \in I} \{ \tau(A) \}.$$

As anticipated at the beginning of the section, we are taking $U_H = \tau(H_P)$ (where H_P is the ordinary Herbrand universe of the program P) as the interpretation domain for any given program P. Furthermore, functors are interpreted in such a way that each ground term t is mapped to $\tau(t)$. We will regard the collection $\tau(B_P)$ as the Herbrand base of P, where B_P is defined as usual (without atoms involving $=$ or $\in$ though, since these have a rigid meaning). A (*set*) *interpretation* I of P can be characterized simply as a subset of $\tau(B_P)$; it will be a *model* of P if it satisfies the whole collection $ground(P)$ of ground instances of P.[2]

With the semantics thus restricted to set interpretations only, one can prove the usual model-theoretic and fixpoint semantics results (cf. [38]). In particular, considering

$$\begin{array}{rcl} T_P(I) & = & \{ a \mid a :- b_1, \cdots, b_n \in \tau(ground(P)) \ and, \\ & & \text{for each } i \in \{1, \ldots, n\}, \text{either } b_i \in \tau(I) \text{ , or } b_i \ is \ s \, \pi \, t \text{ where} \\ & & \pi \in \{=, \in, \neq, \notin, \mathtt{ker\_of}\} \text{ and } b_i \text{ is true w.r.t. the axioms} \} \end{array}$$

and

$$M_P = \bigcap_{M \ model \ of \ P} M \, ,$$

one can prove that

$$M_P = T_P \uparrow \omega \, .$$

4. SET UNIFICATION

The development of a procedural semantics for {log} requires an enhancement of the unification algorithm in order to deal with sets, and a modification of the SLD procedure, in order to include set unification and a proper management of the equality and membership relations. In what follows we cope with the first of these two points, postponing the second one to the next section.

4.1. The Set Unification Problem

Regarding the unification problem, we assume all the definitions (e.g., Herbrand system, substitution, solution, $vars(t)$ notation, etc.) given in [34] and [39].

Standard unification is not adequate to deal with set terms. The first reason is that the inherent lack of order inside a set causes the decay of the uniqueness property of the most general unifier of standard unification. This is clear from the following example: Consider the singleton Herbrand system $\mathcal{E} = \{\{X, Y\} = \{1, 2\}\}$.

[2]The notion of satisfaction is the usual one; cf. [38].

There are only two solutions, namely, $\sigma_1 = \{X \leftarrow 1, Y \leftarrow 2\}$ and $\sigma_2 = \{X \leftarrow 2, Y \leftarrow 1\}$, neither of which is more general than the other. A second reason for the inadequacy of standard unification is that duplicate elements in a set are not relevant as far as unification is concerned. Thus, for instance, the two set terms $\{1\}$ and $\{1, 1\}$ should unify, although standard unification treats them as nonunifiable. Furthermore, an equation such as $X = \{1|X\}$, which does not admit any solution in the standard case (unless infinite terms are taken into account), has the solution $\sigma = \{X \leftarrow \{1|N\}\}$ in the extended framework we are considering.

What is needed is some form of *generalized unification*, i.e., unification w.r.t. a theory T which describes the properties of a set of function symbols (`with` and $\{\,\}$ in our case) by means of a set of equational axioms. In this framework, two terms s and t are said to be *T-unifiable* iff there exists a substitution σ such that $s^\sigma =_T t^\sigma$ (i.e., $T \models s^\sigma = t^\sigma$); such a σ is called a *T-unifier*. The set of all T-unifiers of two terms s and t is denoted by $\bigcup_T(s,t)$ (cf. [48]). A set U of unifiers of s and t is said to be *complete* if

$$\forall \sigma \in \bigcup_T(s,t)\ \exists \theta \in U\ \exists \gamma\ \ (\sigma = \theta \circ \gamma)\,.$$

A solution which is often adopted for special theories is that of developing an extended unification algorithm embodying the axioms of the theory itself. Unfortunately, known extended unification algorithms described in the literature fail to capture the properties of `with` described so far. In particular, ACI-unification (i.e., unification under associativity, commutativity, and idempotency [3, 10, 37, 51]) cannot be directly applied in our case. In fact, the identity $\{Z|\{Y|X\}\} = \{\{Z|Y\}|X\}$, representing associativity, does not hold, for instance, under the substitution $\{X \leftarrow \{c\}, Y \leftarrow \{b\}, Z \leftarrow a\}$, because the two sets $\{a, \{b\}, c\}$ and $\{\{a, b\}, c\}$ are distinct in our interpretation. Similarly, the idempotency property $\{X|X\} = X$ does not hold under the substitution $\{X \leftarrow \{a\}\}$ since $\{\{a\}, a\} \neq \{a\}$.

Therefore, we have developed a new unification algorithm that extends standard unification, embodying the set axioms presented in Section 3. For any given Herbrand system $\mathcal{E}$ involving set terms, the algorithm is able to compute through non-determinism each element of a complete set of unifiers of $\mathcal{E}$.

4.2. Complexity

Before presenting our set unification algorithm we want to point out that the problem of deciding whether two set terms are unifiable is *NP-complete*. NP-hardness ensues from a reduction of 3-SAT to the problem at hand. Given the formula

$$\Phi = (l_1^{(1)} \vee l_2^{(1)} \vee l_3^{(1)})\ \&\ \cdots\ \&\ (l_1^{(m)} \vee l_2^{(m)} \vee l_3^{(m)})$$

with

$$l_i^{(j)} = a_i^{(j)} \text{ or } l_i^{(j)} = \neg a_i^{(j)}\,,$$

where $a_i^{(j)} \in \{d_1, \ldots, d_k\}, \forall i \in \{1,2,3\}, \forall j \in \{1, \ldots, m\}, k \leq 3 \cdot m,$

3-SAT tries to find a truth-value assignment ν to the propositional variables $d_1, \ldots, d_k$ such that $\Phi^\nu \leftrightarrow$ `true`.

We define a transformation function f as follows:

$$f(l) = \begin{cases} X_i & \text{if } l \equiv d_i, \\ Y_i & \text{if } l \equiv \neg d_i, \\ \{\, \mathtt{false},\ f(l_1), f(l_2), f(l_3) \,\} & \text{if } l \equiv l_1 \vee l_2 \vee l_3 \end{cases}$$

and translate Φ to the equation

$$\{\{X_1, Y_1\}, \ldots, \{X_k, Y_k\}, f(l_1^{(1)} \vee l_2^{(1)} \vee l_3^{(1)}), \ldots, f(l_1^{(m)} \vee l_2^{(m)} \vee l_3^{(m)})\} \;=\; \{\, \{\, \mathtt{false}, \mathtt{true} \} \,\},$$

which requires each one of the sets $\{X_j, Y_j\}$ to become $\{\,\mathtt{false}, \mathtt{true}\,\}$, and at least one of

$$f(l_1^{(i)}), f(l_2^{(i)}), f(l_3^{(i)})$$

to become `true` for every i. (To stay strictly inside the realm of pure sets, `false` could be replaced by $\{\}$ and `true` by $\{\,\{\}\,\}$ in this translation).

Indeed, an algorithm for unifying the two sides of such an equation in polynomial time could also be exploited for solving 3-SAT in polynomial time.

The remaining part of the NP-completeness proof (verifying the existence of a polynomial time verification algorithm) is quite mechanical and we refer the interested reader to [31] for a similar and easily adaptable proof.

4.3. The Unification Algorithm

Let $\mathcal{E}$ be a *Herbrand system*, i.e., a finite set $\{\, l_1 = r_1, \ldots, l_n = r_n \}$ of equations where each l_i and each r_i is a term. When l_i is a variable and moreover it occurs in $\mathcal{E}$ exactly once, the equation $l_i = r_i$ is said to be in *solved form* w.r.t. $\mathcal{E}$. $\mathcal{E}$ itself is said to be in solved form if $l_i = r_i$ is in solved form w.r.t. $\mathcal{E}$ for $i = 1, \ldots, n$. Note that in the latter case $\mathcal{E}$ has the obvious solution $\{\, l_1 \leftarrow r_1, \ldots, l_n \leftarrow r_n \,\}$.

Aim of the following algorithm is to bring any given Herbrand system $\mathcal{E}$ to a solved form, or to report failure if $\mathcal{E}$ has no solutions.

Set Unification Algorithm. Let $\mathcal{F}$ be a set of function symbols comprising $\{\}/0$ and `with`$/2$. Let $\mathcal{V}$ be a denumerable set of variables and $\mathcal{T}$ be the set of first-order terms over $\mathcal{F} \cup \mathcal{V}$. In the following X will stand for a generic variable (i.e., $X \in \mathcal{V}$), and t, t_i, t'_i will stand for terms in $\mathcal{T}$.

function unify($\mathcal{E}$: Herbrand_system): Herbrand_system;
begin
  if $\mathcal{E}$ is in solved form
  then return $\mathcal{E}$
  else select arbitrarily from $\mathcal{E}$ an equation e <u>not</u> in solved form w.r.t. $\mathcal{E}$:
    case e **of**

1. $X = X$: **return** unify($\mathcal{E} \setminus \{e\}$);
2. $t = X$, $t \notin \mathcal{V}$: **return** unify($(\mathcal{E} \setminus \{e\}) \cup \{X = t\}$);
3. $X = t$, t is a memberless term and X occurs in t: **fail**;
4. $X = \{t_0, \ldots, t_n \,|\, t\}$, t is a memberless term and X occurs in t or $X \in vars(t_0) \cup \cdots \cup vars(t_n)$: **fail**;
5. $X = \{t_0, \ldots, t_n \,|\, X\}$ and X does not occur in $t_0, \ldots, t_n$:
   return unify($(\mathcal{E} \setminus \{e\}) \cup \{X = \{t_0, \ldots, t_n \,|\, N\}\}$), N new variable;

6. $X = t$, X does not occur in t:
   return unify$((\mathcal{E} \setminus \{e\})^\sigma \cup \{e\})$, where σ is the substitution $\{X \leftarrow t\}$;
7. $f(t_1, \ldots, t_n) = g(t'_1, \ldots, t'_m)$, $f/n \neq g/m$: **fail**;
8. $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$, $f/n \neq$ `with`/2:
   return unify$((\mathcal{E} \setminus \{e\}) \cup \{t_1 = t'_1, \ldots, t_n = t'_n\})$;
9. $\{t_0, \ldots, t_n \,|\, h\} = \{t'_0, \ldots, t'_m \,|\, k\}$, h, k memberless terms or variables:
   if h, k are not the same variable
   then

   (a) choose one of the following actions:
      i. **return** unify$((\mathcal{E} \setminus \{e\}) \cup \{t_0 = t'_0, \{t_1, \ldots, t_n \,|\, h\} = \{t'_1, \ldots, t'_m \,|\, k\}\})$
      ii. **return** unify$((\mathcal{E} \setminus \{e\}) \cup \{t_0 = t'_0, \{t_0, \ldots, t_n \,|\, h\} = \{t'_1, \ldots, t'_m \,|\, k\}\})$
      iii. **return** unify$((\mathcal{E} \setminus \{e\}) \cup \{t_0 = t'_0, \{t_1, \ldots, t_n \,|\, h\} = \{t'_0, \ldots, t'_m \,|\, k\}\})$
      iv. **return** unify$((\mathcal{E} \setminus \{e\}) \cup \{\{t_1, \ldots, t_n \,|\, h\} = \{t'_0 \,|\, N\}$,
      $\{t_0 \,|\, N\} = \{t'_1, \ldots, t'_m \,|\, k\}\})$, N new variable;

   else $h, k \in \mathcal{V}$, $h \equiv k \equiv X$

   (b) select arbitrarily i in $\{0, \ldots, m\}$; choose one of the following actions:
      i. **return** unify$((\mathcal{E} \setminus \{e\}) \cup \{\, t_0 = t'_i$,
      $\{t_1, \ldots, t_n \,|\, h\} = \{t'_0, \ldots, t'_{i-1}, t'_{i+1}, \ldots, t'_m \,|\, k\}\,\})$
      ii. **return** unify$((\mathcal{E} \setminus \{e\}) \cup \{\, t_0 = t'_i$,
      $\{t_0, \ldots, t_n \,|\, h\} = \{t'_0, \ldots, t'_{i-1}, t'_{i+1}, \ldots, t'_m \,|\, k\}\,\})$
      iii. **return** unify$((\mathcal{E} \setminus \{e\}) \cup \{\, t_0 = t'_i$,
      $\{t_1, \ldots, t_n \,|\, h\} = \{t'_0, \ldots, t'_m \,|\, k\}\,\})$
      iv. **return** unify$((\mathcal{E} \setminus \{e\}) \cup \{\, X = \{t_0 \,|\, N\}$,
      $\{t_1, \ldots, t_n \,|\, N\} = \{t'_0, \ldots, t'_m \,|\, N\}\,\})$, N new variable

end.

Let us briefly comment upon action 9 of the algorithm. Its aim is the reduction of set-set equations. 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` by exploiting the property $(\star)$ of `with` pointed out in Section 3.1.

As an example, let us consider the following system

$$\{\{a|X\} = \{b, a|Y\}\}.$$

The algorithm applies action 9(a) to it, requiring one of the following systems to be solved

i. $\{a = b, X = \{a|Y\}\}$,
ii. $\{a = b, \{a|X\} = \{a|Y\}\}$,
iii. $\{a = b, X = \{b, a|Y\}\}$,
iv. $\{X = \{b|N\}, \{a|Y\} = \{a|N\}\}$.

The first three clearly have no solution, whereas system iv can be further transformed by applying again action 9(a) to its second equation, which leads to the following new systems:

i. $\{X = \{b|N\}, Y = N\}$, from which, by variable substitution, we get
   $\{X \leftarrow \{b|Y\}\}$.

ii. $\{X = \{b|N\}, \{a|Y\} = N\}$, from which, by variable substitution, we get $\{X \leftarrow \{b, a|Y\}\}$.
iii. $\{X = \{b|N\}, Y = \{a|N\}\}$, from which we get $\{X \leftarrow \{b|N\}, Y \leftarrow \{a|N\}\}$.
iv. $\{X = \{b|N\}, Y = \{a|N'\}, N = \{a|N'\}\}$ from which, by variable substitution, we get $\{X \leftarrow \{b, a|N'\}, Y \leftarrow \{a|N'\}\}$.

The substitutions we have got constitute the set of unifiers for the initial system we were looking for. Note that this set is not minimal even though sound and complete. For instance, the fourth solution $\{X \leftarrow \{b, a|N'\}, Y \leftarrow \{a|N'\}\}$ can be obtained, apart from duplicates, from the second solution $\{X \leftarrow \{b, a|Y\}\}$ by applying to it the substitution $\{Y \leftarrow \{a|N'\}\}$. In general, the set of substitutions computed by our unification algorithm can contain substitutions which are less general and/or equivalent (w.r.t. the given theory) to other substitutions in the set. However, the number of these "redundancies" is in any case finite and could be reduced by adding suitable checks to the algorithm.

Equations of the form $\{t_0, \ldots, t_n \,|\, X\} = \{t'_0, \ldots, t'_m \,|\, X\}$, where the two sides are set terms with the same variable tail element, are handled as a special case by action 9(b). The problem here is to properly reflect the permutativity of `with`: our algorithm might go into an infinite loop if we simply applied action 9(a), and in particular its case iv, to an equation of this form, e.g., to $\{a|X\} = \{b|X\}$.

The solution we have adopted is to avoid action 9(a)iv of the general case, by resorting to action 9(b) in its stead. The latter forces our algorithm to consider nondeterministically each element of one of the two sets involved in the set-set equation. This way, all possible combinations are explored.

Of course, this solution opens a big (but, anyhow, finite) number of alternatives, possibly leading to redundant solutions. A perhaps preferable formulation of action 9(b), less uniform with the rest of the algorithm but more efficient as for the number of generated solutions, can be found in [20, 44].

To end, let us remark that our unification algorithm is akin to the one sketched by Jayaraman and Plaisted in [28], but it solves a larger number of cases. In particular, the algorithm in [28] intentionally does not take into account the *idempotency* property of sets (i.e., our absorption property). While this restriction enables a simplification of the unification algorithm, it leads, on the other hand, to a loss in expressivity and flexibility. Furthermore, the algorithm in [28] does not properly take into account the situation dealt with by action 9(b) of our algorithm (two set terms with the same variable tail element).

4.4. Soundness, Completeness and Termination

The following theorems state soundness, completeness and termination of *unify*$(\mathcal{E})$ for any given system $\mathcal{E}$ of equations.

Lemma 4.1. Given a system $\mathcal{E}$ and an equation e belonging to it, suppose $\mathcal{E}_1, \ldots, \mathcal{E}_n$ ($n \geq 0$) are all the systems such that unify$(\mathcal{E})$ *can directly invoke* unify$(\mathcal{E}_i)$ *upon selecting e at the beginning (i.e., $\mathcal{E}_i$ is obtained after one unification step by selecting e). Then $\mathcal{E}$ is logically equivalent (under the set axioms) to*

$$\exists x_1 \cdots \exists x_m \bigvee_{i=1}^{n} \bigwedge_{e' \in \mathcal{E}_i} e'$$

where $x_1, \ldots, x_m$ are the variables occurring in $\mathcal{E}_1 \cup \cdots \cup \mathcal{E}_n$ but not in $\mathcal{E}$.

PROOF. Actions 1–4 and 6–8 of the algorithm trivially yield the desired equivalence, in view of the basic properties (reflexivity, congruency, etc.) of equality and of the freeness assumptions (including the regularity axiom, needed to justify action 4) made at the end of Section 3.1. Justifying actions 5, 9(a) and 9(b)i–iii is also straightforward, in view of the permutativity and absorption properties of `with`. Action 9(b)iv is justified as follows: in one direction, by the obvious observation that $\exists w(\{S|w\} = X \,\&\, \{Y|w\} = V)$ implies $\{Y|X\} = \{S|V\}$. The opposite direction can be proved using the fact, anticipated as $(\star)$ in Section 3.1, that $\{Y|X\} = \{S|V\}$ together with $Y \neq S$ implies $\exists w(\{S|w\} = X \& \{Y|w\} = V)$. This fact comes directly from the observation that the hypotheses imply $S \in X$ and $Y \in V$, by analyzing the following four cases:

1. $Y \in X$ and $S \in V$. Then $X = \{Y|X\} = \{S|V\} = V$. We instantiate w as X, so that $\{S|w\} = X = V = \{Y|w\}$.
2. $Y \notin X$ and $S \notin V$. We take $w = X\,\texttt{less}\,S$, so that $w = V\,\texttt{less}\,Y$ (here we are indicating by `less` the single-element removal operation, whose existence is guaranteed by **(L)** and **(E)**).
3. $Y \in X$ and $S \notin V$. We take $w = X\,\texttt{less}\,S$, so that $\{Y|w\} = V$.
4. $Y \notin X$ and $S \in V$. Symmetrical to the preceding one. □

By iterating the process and applying the Lemma 4.1, it is possible to prove that the set of unifiers produced by *unify*, developing completely every branch of the computation tree, covers all the solutions of the original system of equations.

*Theorem 4.1 (*Soundness and completeness*). Given a system $\mathcal{E}$, let $\mathcal{E}_1, \ldots, \mathcal{E}_n$ be all the systems in solved form produced by the unification algorithm. Then* $\mathrm{Soln}(\mathcal{E}) = \mathrm{Soln}(\mathcal{E}_1)|_{vars(\mathcal{E})} \cup \cdots \cup \mathrm{Soln}(\mathcal{E}_n)|_{vars(\mathcal{E})}$*, where* $\mathrm{Soln}(X)$ *is the set of all ground set-unifiers of X and* $\mathrm{Soln}(\mathcal{E}_i)|_{vars(\mathcal{E})}$ *is* $\mathrm{Soln}(\mathcal{E}_i)$ *restricted to the variables of $\mathcal{E}$.*

For the preceding theorem to make full sense, we need the following:

*Theorem 4.2 (*Termination*). For any Herbrand system $\mathcal{E}$, unify($\mathcal{E}$) terminates, provided a suitable strategy is adopted for selecting e across recursive levels.*

PROOF. We will assume the following execution strategy is adopted for sequencing the *unify* actions:

- Action 5 is immediately followed by action 6, performed on the newly generated equation e', unless e' is already in solved form.
- Action 9 generates two new equations, e_1 and e_2, the second of which typically enables action 9 again. In any case, e_2 is immediately processed. When this sequence of consecutive actions reaches an end, a similar "9-exhaustive" treatment of e_1 is triggered in the case of actions 9(a)iv and 9(b)iv.

The collection of all terms appearing as sides of equations in $\mathcal{E}$ can be represented as a forest structure with superimposed links, akin to the one in [43].[3]
Initially, a link connects two leaves in $\mathcal{G}_{\mathcal{E}}$ if and only if the two are labeled by the

[3]As usual, there will be an ordered m-tuple of edges issuing from a node labeled f/m and there will be no edge issuing from a variable node.

same variable. An execution of action 6 does not add new nodes or edges to $\mathcal{G}_{\mathcal{E}}$: rather, it creates a link between two nodes, representing X and t respectively. When new set terms are generated by action 5 or 9, new nodes are added to $\mathcal{G}_{\mathcal{E}}$ to represent them.

A *path* through $\mathcal{G}_{\mathcal{E}}$ is defined to be a list $n_0, \ldots, n_l$ of nodes such that each $\langle n_i, n_{i+1} \rangle$ is either a link or an edge (unlike edges, links can be exploited in both directions). By the very construction of $\mathcal{G}_{\mathcal{E}}$, all paths issuing from a variable will hit the same nonvariable term, if any.

Let us indicate how to measure the *cost* of a path. Links, as well as leftmost edges issuing from `with`-nodes, have cost 0; the cost of the remaining edges is 1; the cost of a path is the sum of the costs of all edges and links constituting it, increased by 1 if the path ends in a constant node. A rough explanation of these conventions is that we intend to combine the notion of height of a term with the (equally widespread) notion of rank of a set.

To conclude this preamble, let us define the *pseudorank* of a node n to be the maximum cost of a path issuing from n. Relevant to our analysis of *unify* is that the cost of a path will never exceed the initial number p of nonvariable nodes in $\mathcal{G}_{\mathcal{E}}$. On the one hand, this results from the occur checks made in actions 3 and 4: these in fact forestall the formation of cyclic paths. On the other hand, it follows from inspection of the *unify* algorithm (cf. actions 5, 9(a)iv and 9(b)i,ii,iv) that no addition of edges can disrupt this bound on the path length. In fact, each new edge of cost 1 shares the target node with a pre-existing edge of cost 1: this implies that no path will ever contain both such edges at once.

Let $E^{(0)}, E^{(1)}, E^{(2)}, \ldots$ be the successive values of $\mathcal{E}$ along a branch of the *unify* execution not ending with a failure.

Indicating by prk_i the pseudorank function defined on the nodes of $\mathcal{G}_{\mathcal{E}}$ at the time when $\mathcal{E} = E^{(i)}$, it makes sense to define $prk_\infty(n)$ to be the ultimate value of the sequence $prk_i(n), prk_{i+1}(n), \ldots$, for any node n eventually introduced in $\mathcal{G}_{\mathcal{E}}$. This sequence, being bound by p as we have already noticed, can in fact increase at most p times.

Plainly, every prk_i (even the one with $i = \infty$) satisfies the identities

$$\begin{array}{rcl} prk_i(f(t_1, \ldots, t_m)) & = & 1 + \max_{j=1}^{m} prk_i(t_j)\,, \\ prk_i(\{t \,|\, s\}) & = & \max\{1 + prk_i(t), prk_i(s)\}\,. \end{array}$$

We can exploit the "limit pseudorank" $prk \equiv prk_\infty$, whose domain is enlarged by putting $prk(l = r) = prk(l) + prk(r)$, to associate with $\mathcal{E}$ the following $(2 \cdot p + 1)$-tuple of non-negative integers

$$Size(\mathcal{E}) = \langle\, |\, \{e \text{ in } \mathcal{E} \,:\, prk(e) = 2 \cdot p\} \,|, \ldots, |\, \{e \text{ in } \mathcal{E} \,:\, prk(e) = 0\} \,|\, \rangle\,.$$

Indicating by $\lhd$ the lexicographic ordering, we will prove that $Size(E^{(i+1)}) \lhd Size(E^{(i)})$ unless the action leading from $E^{(i)}$ to $E^{(i+1)}$ is one of the actions 2, 5, 9. Even then, a decrease will turn out to be the outcome of a "phase" consisting of consecutive actions. In particular, in the case of actions 5 and 9, the phase is the series of actions imposed by the execution strategy described at the beginning. It will follow, thanks to the well-foundedness of $\lhd$, that the $E^{(i)}$ sequence eventually terminates, which proves our thesis.

It is helpful to view $\mathcal{E}$ as composed of two disjoint collections $\mathcal{E}_1$, $\mathcal{E}_2$ of equations: $\mathcal{E}_2$ is the part of the system which has already been brought to solved form; the remaining equalities form the "working system" $\mathcal{E}_1$.[4]

We can now proceed to show in detail that every successful phase of *unify* reduces $Size(\mathcal{E}_1)$. In the following, each number on the left indicates the first action of a phase.

1. One equation with pseudorank $2 \cdot prk(X)$ is removed: *Size* decreases.
2. This action occasionally decreases *Size*, but may also leave it unchanged. Anyway, it is unproblematic, because it cannot be performed an indefinite number of consecutive times. It suffices for our purposes to regard a series of such actions as preamble of the subsequent phase.
6. One equation $X = t$ with pseudorank $prk(X) + prk(t) = 2 \cdot prk(X)$ is moved from $\mathcal{E}_1$ to $\mathcal{E}_2$. This lowers *Size*.
5. The presence in the new system of the equation $X = \{t_0, \ldots, t_n \mid N\}$ forces $prk(N) \leq prk(X)$ to hold, which implies that *Size* does not increase in consequence of the replacement of $X = \{t_0, \ldots, t_n \mid X\}$ by $X = \{t_0, \ldots, t_n \mid N\}$ in $\mathcal{E}_1$. *Size* will decrease thanks to action 6, which is required to be performed immediately.
8. One equation of pseudorank $2 + \max\{prk(t_1), \ldots, prk(t_n)\} + \max\{prk(t'_1), \ldots, prk(t'_n)\}$ is replaced by equations of lower pseudorank in $\mathcal{E}_1$: *Size* decreases.
9. One can view the global effect of the phase starting with action 9, as that of replacing the selected equation $e \equiv \{t_0, \ldots, t_n \mid h\} = \{t'_0, \ldots, t'_m \mid k\}$ by a collection $t_{i_1} = t'_{j_1}, \ldots, t_{i_p} = t'_{j_p}$ of equations relating elements of the two sets, one or two equations regarding h and k being also added, as explained in detail below.

   The pseudorank $\bar{l} = \max\{1 + prk(t_0), \ldots, 1 + prk(t_n), prk(h)\} + \max\{1 + prk(t'_0), \ldots, 1 + prk(t'_m), prk(k)\}$ of e, exceeds that of any $t_i = t'_j$ equation. Hence we only need to focus on the equations concerning h and k. Various cases need to be considered.

   - If neither h nor k is a variable, there will be only one equation e' regarding h, k. If the form of e' differs from $h \equiv f(r_1, \ldots, r_q) = f(r'_1, \ldots, r'_q) \equiv k$, an immediate failure will ensue due to action 7. Else, since the pseudorank of e' cannot exceed $\bar{l}$, action 8, immediately performed on e', will cause *Size* to decrease.
   - If h is a variable and k is not, the only case that does not immediately lead to failure or to a *Size* decrease, is when the new equation e' has the form $h = \{t'_{i_1}, \ldots, t'_{i_q} \mid k\}$, with h not occurring in the right-hand side, and e' gets added to the working system $\mathcal{E}_1$. Action 6, immediately performed on e', will cause *Size* to decrease. The case when k is a variable and h is not is entirely analogous.
   - If h and k are both variables, with $h \not\equiv k$, the equation(s) dealing with them added to the working system can be of the form:
     - $h = \{t'_{i_1}, \ldots, t'_{i_q} \mid k\}$, or

[4] Conceptually, all equations which are in solved form are moved from $\mathcal{E}_1$ to $\mathcal{E}_2$ before each action; then, after e has been chosen, a (possibly empty) collection of newly generated equations replace it in $\mathcal{E}_1$. As for e, sometimes it gets moved to $\mathcal{E}_2$, sometimes it simply gets discarded.

– $k = \{t_{i_1}, \ldots, t_{i_q} \mid h\}$, or
– $h = \{t'_{i_1}, \ldots, t'_{i_q} \mid N\}$, and $k = \{t_{j_1}, \ldots, t_{j_r} \mid N\}$, with N new variable.

After action 6 is performed (possibly twice), *Size* decreases.[5]

- If h and k are the same variable X (action 9(b) of *unify*), an equation of the form $X = \{t_{i_1}, \ldots, t_{i_q}, t'_{j_1}, \ldots, t'_{j_{q'}} \mid X\}$ (possibly of pseudorank $\bar{l}$) is added to the working system. After actions 5 and 6 are performed on this equation, *Size* decreases. □

5. {log} RESOLUTION PROCEDURE

The resolution procedure developed for {log} is an extension of the usual SLD-resolution procedure, where standard unification is replaced by the set unification algorithm presented above. In addition, some changes are required in order to properly manage equality and membership and their negative counterparts as predicates with a pre-assigned meaning.

The main idea behind the management of such predicates is the use of a simple *constraint logic programming* scheme [27]. In this context, an *atomic constraint* is any atom of the form $t_1 \in t_2$ or $t_1 \neq t_2$ or $t_1 \notin t_2$ or t_1 `ker_of` t_2, where t_1, t_2 are terms. t_1 `ker_of` t_2 is intended to hold iff t_1 is the kernel of the term t_2.

A *constraint* is a conjunction of atomic constraints. A constraint is in *canonical form* if all of the atomic constraints in it (if any) have the form:

- $X \neq t$ and X does not occur in t, or
- $t \notin X$ and X does not occur in t, or
- t `ker_of` X and t is either a variable or a memberless term not involving X,

where X is a variable and t is a term.

The extended resolution step is based on a constraint satisfiability analyzer called *Can*. Given a constraint C, the application of *Can* to $\langle C, \varepsilon\rangle$ nondeterministically generates each element of an equivalent finite set

$$C_{Can} = \{\langle \Gamma_1, \theta_1\rangle, \ldots, \langle \Gamma_d, \theta_d\rangle\},$$

where each Γ_i is a constraint in canonical form, and each θ_i is a substitution which keeps track of the bindings for the auxiliary variables created by the canonization process. If such a transformation process terminates successfully, then the given constraint is guaranteed to be satisfiable, and the set of constraints in canonical form can be used in place of the given initial constraint to continue the resolution process. We will describe more precisely our resolution procedure and the canonization algorithm in the next two subsections.

5.1. Specialized SLD-Resolution Procedure

Let P be a {log} program and G be a goal

$$:\!-\; C_1 , \cdots, \; C_n \;, B_1, \cdots, \; B_k$$

[5]Note how important is the assumption that h differs from k: without it—in the third case—, after the substitution $\{h \leftarrow \{t'_{i_1}, \ldots, t'_{i_q} \mid N\}\}$ is performed, the second equation would become a set-set equation, possibly of pseudorank $\bar{l}$.

(where the C_is, are atomic constraints and the B_js are {log} atoms). The goal G' is said to be *derived from* G with substitution σ if the following conditions hold:

- If $k = 0$ then
  μ is the empty substitution ε, h and i are 0, and C' is $\emptyset$.
- If $k > 0$ then
  - $B_i \quad (i \in \{1, \ldots, k\})$ is the selected atom;
  - μ and C' are computed as follows:
    case B_i **of**
    a. $p(t_1, \ldots, t_l)$, with p an ordinary predicate, and there exists a clause
    $p(t'_1, \ldots, t'_l) :- C'_1, \cdots, C'_g, B'_1, \cdots, B'_h$ in P:
    μ is an mgu of the system $\{t_1 = t'_1, \ldots, t_l = t'_l\}$,
    C' is $\{C'_1, \ldots, C'_g\}$;
    b. $t = t'$: μ is an mgu of the system $\{t = t'\}$, C' is $\emptyset$;
- $\langle \{D_1, \ldots, D_d\}, \theta \rangle$ is one of the pairs generated by applying *Can* to $\langle (\{C_1, \ldots, C_n\} \cup C')^\mu, \varepsilon \rangle$.
- G' is $:- D_1, \cdots, D_d, (B_1, \cdots, B_{i-1}, B'_1, \cdots, B'_h, B_{i+1}, \cdots, B_k)^\sigma$, where σ is $\mu \circ \theta$.

A *derivation* of $P \cup \{G\}$ is a (finite or infinite) sequence $G_0 = G, G_1, G_2, \ldots$ of goals such that G_{i+1} is derived from G_i. A *refutation* of $P \cup \{G\}$ is a finite derivation of $P \cup \{G\}$ whose last derived goal, G_m, contains only canonical atomic constraints.

Finally, a *computed answer* for a refutation $G, G_1, \ldots, G_m$ of $P \cup \{G\}$ is a pair $\langle C, \sigma \rangle$ where C is the constraint (in canonical form) contained in G_m and σ is the substitution $\sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_m$ restricted to the variables of G, where $\sigma_1, \sigma_2, \ldots, \sigma_m$ are the substitutions generated during consecutive steps of the refutation.

It is interesting to notice that such a resolution algorithm involves four kinds of *nondeterministic* choices (all but the first potentially leading to backtracking):

- Which atom in the goal to select (don't care nondeterminism).
- Which clause in the program to select (don't know nondeterminism).
- Which mgu in the set of unifiers computed by the unification algorithm to select (don't know nondeterminism).
- Which one of the pairs generated by *Can* should be used (don't know nondeterminism).

5.2. The Constraint Analyzer and Its Soundness

In this section we describe a nondeterministic algorithm which is able to compute, for any given constraint C, the corresponding set C_{Can} of constraints in canonical form. The algorithm starts with the pair $\langle C, \varepsilon \rangle$ and generates, through nondeterminism, each element of C_{Can}. Note that a constraint is represented here as a *set* of atomic constraints; in particular, $\emptyset$ is a constraint in canonical form.

Constraint Canonization Algorithm. Below, t, t', t_i, t'_i, r, and s, denote first-order terms, and X denotes a generic variable.

function Can($\langle$ C : constraint, σ : substitution $\rangle$): $\langle$ constraint, substitution $\rangle$;
begin
  if C is in canonical form
    then return $\langle C, \sigma \rangle$

else select arbitrarily an atomic constraint c in C <u>not</u> in canonical form:

case c **of**

1. $t \notin s\,\mathtt{with}\,t'$: **return** $\mathrm{Can}(\langle (C \setminus \{c\}) \cup \{t \neq t', t \notin s\}, \sigma\rangle)$;
2. $t \notin s$, s is a memberless term: **return** $\mathrm{Can}(\langle C \setminus \{c\}, \sigma\rangle)$;
3. $t \notin X$, X is a variable, and X occurs in t: **return** $\mathrm{Can}(\langle C \setminus \{c\}, \sigma\rangle)$;
4. $f(t_1, \ldots, t_n) \neq g(t'_1, \ldots, t'_m)$, $f/n \neq g/m$: **return** $\mathrm{Can}(\langle C \setminus \{c\}, \sigma\rangle)$;
5. $f(t_0, \ldots, t_n) \neq f(t'_0, \ldots, t'_n)$, $f/(n+1) \neq \mathtt{with}/2$:
   choose i from $\{0, \ldots, n\}$; **return** $\mathrm{Can}(\langle (C \setminus \{c\}) \cup \{t_i \neq t'_i\}, \sigma\rangle)$;
6. $f \neq f$ or $X \neq X$, X variable: **fail**;
7. $t \neq X$ and t is not a variable: **return** $\mathrm{Can}(\langle (C \setminus \{c\}) \cup \{X \neq t\}, \sigma\rangle)$;
8. $X \neq t$, t is a memberless term and X occurs in t, or t is $\{t_1, \ldots, t_n \,|\, h\}$, h memberless term or variable, and either X occurs in h and $X \not\equiv h$, or X occurs in at least one of $t_1, \ldots, t_n$: **return** $\mathrm{Can}(\langle C \setminus \{c\}, \sigma\rangle)$;
9. $X \neq \{t_0, \ldots, t_n \,|\, X\}$ and X does not occur in any of $t_0, \ldots, t_n$:
   choose i from $\{0, \ldots, n\}$; **return** $\mathrm{Can}(\langle (C \setminus \{c\}) \cup \{t_i \notin X\}, \sigma\rangle)$;
10. $\{t \,|\, r\} \neq \{t' \,|\, s\}$: choose one of the following actions:
    (a) **return** $\mathrm{Can}(\langle ((C \setminus \{c\}) \cup \{Z \in \{t \,|\, r\}, Z \notin \{t' \,|\, s\}\}), \sigma\rangle)$, Z new variable;
    (b) **return** $\mathrm{Can}(\langle ((C \setminus \{c\}) \cup \{Z \in \{t' \,|\, s\}, Z \notin \{t \,|\, r\}\}), \sigma\rangle)$, Z new variable;
    (c) **return** $\mathrm{Can}(\langle ((C \setminus \{c\}) \cup \{Z\,\mathtt{ker\_of}\,r, W\,\mathtt{ker\_of}\,s, Z \neq W\}), \sigma\rangle)$, Z and W new variables;
11. $t \in s$, s is a memberless term: **fail**;
12. $t \in X$, X is a variable: **return** $\mathrm{Can}(\langle (C \setminus \{c\})^{\delta}, \delta \circ \sigma\rangle)$, where δ is the substitution $\{X \leftarrow \{t \,|\, N\}\}$, and N is a new variable;
13. $t \in \{t' \,|\, s\}$: choose one of the following actions:
    (a) **return** $\mathrm{Can}(\langle ((C \setminus \{c\}) \cup \{t \in s\}), \sigma\rangle)$;
    (b) **return** $\mathrm{Can}(\langle (C \setminus \{c\})^{\delta}, \delta \circ \sigma\rangle)$, where δ is an mgu of the system $\{t = t'\}$;
14. $t\,\mathtt{ker\_of}\,\{t' \,|\, s\}$, t is a memberless term:
    return $\mathrm{Can}(\langle ((C \setminus \{c\}) \cup \{t\,\mathtt{ker\_of}\,s\}), \sigma\rangle)$;
15. $t\,\mathtt{ker\_of}\,t'$, t' memberless term and t memberless term or variable:
    if there exists an mgu δ of the system $\{t = t'\}$
    then return $\mathrm{Can}(\langle (C \setminus \{c\})^{\delta}, \delta \circ \sigma\rangle)$
    else fail;
16. $\{t \,|\, s\}\,\mathtt{ker\_of}\,t$: **fail**;
17. $t\,\mathtt{ker\_of}\,X$, t is not X and X occurs in t: **fail**;

end.

Note that membership constraints can be completely eliminated by reducing their solution to the solution of the corresponding unification problems.

Let us see now how the {log} resolution procedure works on a pair of simple examples involving also negative answers. Given the program

```
in_difference(X,Set1,Set2) :-
        X ∈ Set1,X ∉ Set2.
```

and the goal

```
:- in_difference(X,{1,2},{1,3}),
```

the only clause of the program is selected as a possible resolvent of the goal. Solving the set membership goal in the body of the selected clause generates (see the definition of *derived goal* in Section 5.1) the two alternative substitutions {X ← 1}, {X← 2}. *Can* is applied nondeterministically to either the constraint {1 ∉ {1,3}} or {2 ∉ {1,3}}. The first application generates (action 1 of *Can*) the new constraint {1 ≠ 1, 1 ∉ {3}} that clearly fails. The second application generates the constraint {2 ≠ 1, 2 ∉ {3}} from which, after a few iterations of *Can*, the pair ⟨∅, ε⟩ is obtained. So the final computed answer is

⟨ ∅ , { X ← 2 } ⟩ .

If the following goal is given, instead:

:− in_difference(X,Set1,{1,3}),

solving the set membership atom in the body of the selected clause generates the substitution { Set1 ← {X|N}}, whereas applying *Can* to the constraint {X ∉ {1,3}} generates the pair ⟨ {X ≠ 1, X ≠ 3},ε ⟩. So the final computed answer is

⟨ { X ≠ 1, X ≠ 3}, { Set1 ← {X|N} } ⟩.

Note that two sets may differ because of their kernels being different (see action 10.(c)). For instance, given the goal

:− {a | X} ≠ {a | Y},

the *Can* function computes, along with others, the solution

⟨ {Z `ker_of` X, W `ker_of` Y, Z ≠ W}, ε ⟩.

Thus, a perfectly legitimate answer such as

⟨∅,{X ← a, Y ← { }}⟩

can be subsequently obtained as an instance of the computed solution.

Whenever one needs to guarantee that two (colored) sets have the same kernel, one can make explicit use of the `ker_of` constraint in a program. For instance, the definition of a predicate subset(A,B) intended to hold when A is included in B and A, B have the same kernel, goes as follows:

```
subset(A,B) :-
        A ker_of B.
subset({X|A},B) :-
        X ∉ A, X ∈ B, subset(A,B).
```

With this definition the goal :− subset({ 1 | a }, { 1 , 2 }) clearly fails, whereas the goal :− subset({ 1 | a }, { 1 , 2 | a }) succeeds.

Finally, note that by using the `ker_of` constraint it is easy to define a predicate is_ker that tests whether or not a term is a kernel (i.e., a memberless term).

```
is_ker(X) :- X ker_of X.
```

The following theorems state termination, soundness and completeness of the constraint canonization algorithm. Complete proofs of the theorems can be found in the Appendix.

*Theorem 5.1 (*Termination). *Can*(⟨C, ε⟩) *terminates for any constraint* C.

Let C be a constraint such that

$$C_{Can} = \{\, \langle C_1, \theta_1 \rangle, \ldots, \langle C_n, \theta_n \rangle \,\}.$$

Then:

A) *if $n = 1$, $C_1 = C$, and $\theta_1 = \varepsilon$ (i.e., C is in canonical form), then C admits a solution;*

B) *if σ is a solution of C_i ($i \in \{1, \ldots, n\}$), then σ is a solution of C^{θ_i}.*

*Theorem 5.3 (*Completeness*). Let C be a constraint such that*

$$C_{Can} = \{\, \langle C_1, \theta_1 \rangle, \ldots, \langle C_n, \theta_n \rangle \,\}.$$

If θ is a solution of C, then there exist an i, $1 \leq i \leq n$, and a solution σ of C_i, such that $\theta = \theta_i \circ \sigma$.

Actually, a specific ground substitution γ such that C^γ is provable from the axioms can be exhibited in case A) of Theorem 5.2. This fact has the following consequence:

Theorem 5.4. The axiomatic set theory specified in Section 3.1 makes it possible to prove either $C^\exists$ or $\neg(C^\exists)$ for any constraint C.

5.3. Soundness and Completeness of the Resolution Procedure

Having developed a new specialized resolution procedure, we are now to prove its soundness and completeness. Note that theorems and proofs concerning the termination, soundness and completeness of the constraint canonization algorithms *unify* and *Can* remain almost unchanged when {log} is viewed in the context of CLP [22]. Soundness and completeness of the resolution procedure, on the contrary, are more influenced by the fact we have adopted an approach based on Horn clause logic in this paper.

*Theorem 5.5 (*Soundness*). Let P be a {log} program and G be a goal. If G has a refutation in P with computed answer $\langle C, \theta \rangle$ and σ is a solution of C, then we have $P \models_{Set} G^{\theta \circ \sigma}$.*

Lemma 5.1. Let C be a constraint and σ be a substitution such that $dom(\sigma) \subseteq vars(C)$. If $\langle C_1, \theta_1 \rangle$ is generated by the first step of Can applied to C^σ (i.e., to $\langle C^\sigma, \varepsilon \rangle$), then there exists a pair $\langle C_2, \theta_2 \rangle$ generated by the first step of Can applied to C such that:

(i) *if μ is a solution of C_1, then there exists γ such that $\gamma \circ \mu$ is a solution of C_2;*

(ii) *there exists an η such that $\theta_2 \circ \eta|_{vars(C)} = \sigma \circ \theta_1|_{vars(C)}$;*

(iii) *with η as in (ii), either $C_1 = C_2^\eta$ holds or $\langle C_1, \varepsilon \rangle$ is returned by $Can(\langle C_2^\eta, \varepsilon \rangle)$.*

Lemma 5.2. Let C be a constraint, σ be a substitution such that $dom(\sigma) \subseteq vars(C)$. If $\langle C_1, \theta_1 \rangle$ is returned by $Can(\langle C^\sigma, \varepsilon \rangle)$, then there exists $\langle C_2, \theta_2 \rangle$ returned by $Can(\langle C, \varepsilon \rangle)$ such that (i), (ii) and (iii) as in Lemma 5.1.

PROOF. The statement follows immediately from Lemma 5.1. □

Lemma 5.3 (Lifting lemma for {log}). *Let P be a {log} program, R a nonempty conjunction of {log} atoms, C a constraint, and σ a substitution for variables in (C, R). If $:\!-\,(C\,,\,R)^\sigma$ has a refutation in P with computed answer $\langle C_1, \theta_1\rangle$, then $:\!-\,(C\,,\,R)$ has a refutation in P with computed answer $\langle C_2, \theta_2\rangle$ such that*

(i) there exists a substitution γ such that $\theta_2 \circ \gamma|_{vars(C)} = \sigma \circ \theta_1|_{vars(C)}$, and

(ii) for each solution α of C_1, there exists a β such that $\beta \circ \alpha$ is a solution of C_2.

Lemma 5.4. Let P be a {log} program and $Succ(P)$ be the set

$$\{\,\tau(a) \mid \ a \text{ is atomic, its leading predicate is none of } \in, \notin, \neq, = \text{ and there exists a refutation for } P \cup \{\,:\!-\,a\,\}\,\}\,.$$

Then $Succ(P) = M_P$.

In the two propositions that follow, the program P and the goal G are assumed to have no variables in common:

Lemma 5.5. Let P be a {log} program, $G \equiv (\,:\!-\,C\,,\,B\,,\,L_1\,,\cdots,\,L_h\,)$ be a goal such that:

C is a constraint,
B is a conjunction of atoms $t = s$,
the L_is are ordinary atoms, and
$P \cup \{\neg C \vee \neg B \vee \neg L_1 \vee \cdots \vee \neg L_h\}$ is unsatisfiable.

Then there is a refutation for G in P.

Theorem 5.6 (Completeness). *Let P be a {log} program and $G \equiv (\,:\!-\,Q\,)$ be a goal. Assume moreover that $P \models_{Set} \neg Q^\sigma$. Then G has a refutation in P.*

6. RESTRICTED UNIVERSAL QUANTIFIERS

Restricted Universal Quantifiers (RUQs) are formulas of the form

$$(\forall X \in s)\, F,$$

with F an arbitrary formula. This form stands for the quantified implication

$$\forall X((X \in s) \rightarrow F).$$

The usefulness of providing RUQs as part of the representation language has been demonstrated by several authors (e.g., [11, 33]). In fact, RUQs allow basic set-theoretic operations (such as subset, union, intersection and so on) to be expressed in a clear and concise way. In what follows we will show how the language presented so far can be extended in order to encompass RUQs. To wit, RUQs will be introduced in {log} only at the syntactic level, as a convenient notation, without any extension at the semantic level.

An *extended Horn clause* is a formula: $p(t_1,\ldots,t_n)\,:\!-\,B_1\,,\cdots,\,B_m$ where each B_i can be either an atom or a RUQ formula of the form $(\forall X_1 \in s_1)\cdots(\forall X_k \in s_k)G$, G conjunction of atoms, satisfying the following properties:

- The variables $X_1,\ldots,X_k$ can occur only in G (in particular, they are not allowed to occur in $s_1,\ldots,s_k$).

- If $i \neq j$ then $X_i \neq X_j$.

These two restrictions ensure that a B_i of the form $(\forall X_1 \in t_1) \cdots (\forall X_k \in t_k)G$ is logically equivalent to $\forall X_1 \cdots \forall X_k(X_1 \in t_1, \cdots, X_k \in t_k \to G)$. [Note that the first restriction, motivated (cf. [11]) by the set finiteness requirement, is implicitly present in [33] since nesting of sets is not allowed there.]

For example, by using RUQs, it is easy to define the following set-theoretic operations:

(a)
```
subset(S1,S2) :-
        (∀ X ∈ S1)(X ∈ S2).
```

(b)
```
disj(S1,S2) :-
        (∀ X ∈ S1) (∀ Y ∈ S2) (X ≠ Y).
```

where predicate subset tests whether S1 is a subset of S2 and predicate disj tests whether S1 and S2 are disjoint sets.

One might proceed as in [33, 9], by enhancing resolution to deal directly with RUQs. However, both for conceptual simplicity and for soundness concerns we prefer to transform extended Horn clauses into equivalent {log} clauses without RUQs (hints about a similar idea can be found in [32]). We have proved that such a transformation is always possible, and have developed an algorithm to perform it.

RUQs Elimination Algorithm. Let $C = H :- B_1 , \cdots , B_k , B_{k+1} , \cdots , B_n$ be an extended Horn clause, where $B_1, \ldots, B_k$ $(k \leq n)$ are {log} formulas and $B_{k+1}, \ldots, B_n$ are formulas containing RUQs.

1. Replace C by the set of clauses

$$
\begin{array}{rcl}
I & = & \{ H :- B_1 , \cdots , B_k, D_1 , \cdots , D_{n-k} \\
 & & D_1 :- B_{k+1} \\
 & & \ldots , \\
 & & D_{n-k} :- B_n \}
\end{array}
$$

   where each D_j is an atom $q_j^{new}(X_1^j, \ldots, X_{k_j}^j)$ with q_j^{new} brand new predicate symbol and $\{X_1^j, \ldots, X_{k_j}^j\}$ are all the variables in B_{k+j} which are not quantified by any RUQ of B_{k+j}.
2. Replace each element in I of the form

$$p(t_1, \ldots, t_n) :- (\forall X_1 \in s_1)(\forall X_2 \in s_2)G$$

   by the two clauses:

$$
\begin{array}{l}
p(t_1, \ldots, t_n) :- (\forall X_1 \in s_1) r(Y_1, \ldots, Y_k) \\
r(Y_1, \ldots, Y_k) :- (\forall X_2 \in s_2)G
\end{array}
$$

   where $Y_1, \ldots, Y_k$ are all the variables (different from X_2) occurring in s_2 or free in G, and r is a new predicate symbol. This step is repeated as long as there are objects with nested quantifiers in I.

3. Replace each extended Horn clause of the form

$$p(t_1, \ldots, t_n) :- (\forall X \in \{t'_1, \ldots, t'_m | h\})G$$

by

$$p(t_1, \ldots, t_n) :- G^{\{X \leftarrow t'_1\}}, \cdots, G^{\{X \leftarrow t'_m\}}$$

if h is a memberless term, or by

$$p(t_1, \ldots, t_n) :- G^{\{X \leftarrow t'_1\}}, \cdots, G^{\{X \leftarrow t'_m\}}, D$$
$$D :- (\forall X \in h)G$$

if h is a variable, where D is built as described in step 1.

4. Replace each simple extended Horn clause

$$p(t_1, \ldots, t_n) :- (\forall X \in Y)G[X, Z_1, \ldots, Z_m],$$

where Y is a variable and $X, Z_1, \ldots, Z_m$ ($m \geq 0$) are all the variables occurring in G, by the following three {log} clauses:

$$p(t_1, \ldots, t_n) :- r(Y, Z_1, \ldots, Z_m)$$
$$r(K, Z_1, \ldots, Z_m) :- \mathsf{is\_ker}(K)$$
$$r(\{A|R\}, Z_1, \ldots, Z_m) :- (A \notin R), G^{\{X \leftarrow A\}}, r(R, Z_1, \ldots, Z_m)$$

where r is a new predicate symbol and is_ker is defined using `ker_of` as shown in Section 5.2.

For example, the extended Horn clause (a) for the subset operation given at the beginning of this section will be transformed into the equivalent three {log} clauses:

```
subset(S1,S2) :- r(S1,S2).
r(S1,S2) :- is_ker(S1).
r({A|R},S2) :- (A ∉ R), (A ∈ S2), r(R,S2).
```

Note that a subset predicate which works correctly also on colored sets can be obtained by modifying the original clause so as to enforce the additional check that the two given sets have the same kernel:

```
subset(S1,S2) :- (∀ X ∈ S1)(X ∈ S2), K ker_of S1, K ker_of S2.
```

To make another example, by applying the RUQs elimination algorithm to the clause defining the predicate `disj` shown above, we get:

```
disj(S1,S2) :- r_1(S1,S2).
r_1(K,S2) :- is_ker(K).
r_1({A|R},S2) :- (A ∉ R), r_2(A,S2), r_1(R,S2).
r_2(X,S2) :- r_3(S2,X).
r_3(K,X) :- is_ker(K).
r_3({A|R},X) :- (A ∉ R), X ≠ A, r_3(R,X).
```

It is worth noticing the use of the $\notin$ constraint in the second clause of the definition of the predicate r generated by the RUQs elimination algorithm (action 4).

Without such a constraint, a goal like :− subset(X, { 1}) would generate an infinite number of equivalent solutions, namely X ← { 1}, X ← {1 ,1}, X ← { 1 ,1 ,1}, and so on. Similarly, the goal :− subset({ 1}, { 2 ,X}) would generate infinite many answers X ← { 1}. The problem originates from both the recursive structure of the definition of r in the generated code and the use of set unification (specifically, the way set unification deals with duplicates). In particular, given the goal $:- r(\{B|S\}, Z_1, \ldots, Z_m)$, set unification will generate, among others, the solution $\{A \leftarrow B, R \leftarrow \{B|S\}\}$, so that the recursive call to r is exactly the same as the given goal. The problem was already noted in [28] where the absorption property is disallowed during matching "to avoid a potential infinite loop in recursive definitions". In {log}, in contrast, one can circumvent the problem by a proper use of the $\neq$ and $\notin$ constraints.

7. PROGRAMMING WITH SETS

Various standard set operations, e.g. union, intersection, difference, etc., can be straightforwardly programmed in {log} using the set manipulation facilities introduced so far, as shown in [19, 22, 44].

The {log} definition of these operations can be quite similar to the usual Prolog definition obtainable by representing sets as lists (cf., e.g., [40]). For instance, the following three {log} clauses can be used to define the intersection of two sets:

```
∩ ({}, _ , {}).
∩ ({X | A}, B,{X | C}) :−
    X ∉ A, X ∈ B, ∩ (A, B, C).
∩ ({X | A}, B, C) :−
    X ∉ A, X ∉ B, ∩ (A, B, C).
```

There are, however, a number of new facilities in {log}, especially devoted to set manipulation, which can make the behaviour of {log} programs significantly better than that of the corresponding Prolog programs and which can be exploited to drastically simplify the program development effort.

A first notable difference w.r.t. the Prolog solution comes out when dealing with *nonground sets*. Due to the ability to treat simple *set constraints*, {log} can provide answers to goals which are hardly managed in standard Prolog. For example, given the {log} goal

```
:− ∩ ({X} , {Y} , Z).
```

we get from the above definition of the predicate intersection the following two answers (the second of which containing a negative constraint):

```
X = Y , Z = { Y },
X ≠ Y , Z = {}.
```

Another feature of {log} which, as noted in Sect. 6, strongly enhances the expressive power of the language, is the availability of *Restricted Universal Quantifiers*. Some definitions using such a facility—namely, the subset and disjoint predicates—have been shown in the previous section. Some further usages of RUQs will be shown in next examples.

A third important feature of {log} is *set-unification.* There are several problems, e.g., resource allocation problems and combinatorial problems in general, where the nondeterminism embedded in the set-unification mechanism can be advantageously exploited to make programs simpler to write and more declarative to understand than those obtainable by using conventional Prolog programming techniques.

We prove this by showing the {log} solution to two well-known combinatorial problems, namely the SEND + MORE = MONEY puzzle and the coloring of a map.

Cryptarithmetic puzzle. This is the well-known problem of solving the equation SEND + MORE = MONEY by assigning a distinct digit between 0 and 9 to each letter appearing in it. A classical solution to this problem can be developed by adopting a *generate and test* approach, i.e., by successively generating assignments of digits to the letters of the puzzle and then testing, for each assignment, whether the generated pattern satisfies the required constraints.

By using sets and the extended resolution procedure (in particular, set unification), one can avoid the explicit use of a generator and of mechanisms for backtracking in search for new solutions.

```
solve_puzzle(S,E,N,D,M,O,R,Y) :-
    {S,E,N,D,M,O,R,Y,_,_} = {0,1,2,3,4,5,6,7,8,9},
     M * 10000   +   O * 1000  +  N * 100  +  E * 10  +  Y
                =:=  S * 1000  +  E * 100  +  N * 10  +  D
                 +   M * 1000  +  O * 100  +  R * 10  +  E.
```

Of course it would be nice to replace the extra-logical literal involving =:= by an integer equation (or by a system of integer equations and inequalities), assuming that a suitable integer constraint solver interacts with the set unifier. As usual, a drastic reduction of the search space can in fact ensue from the interplay between two solvers.

Coloring. Given a set $\{c_1, \ldots, c_m\}$ of Colors, a set $\{R_1, \ldots, R_n\}$ of Regions, $m \leq n$, and a set Map $= \{\{R_{i_1}, R_{j_1}\}, \ldots, \{R_{i_k}, R_{j_k}\}\}$, $i_p \neq j_p$ for all $p = 1, \ldots, k$, of pairs of neighboring regions, the predicate coloring returns an assignment of colors to the regions so that no two neighboring regions have the same color.

```
coloring( Regions, Map, Colors ) :-
    Regions = Colors,
    (∀ R ∈ Regions) ({R} ∉ Map).
```

A sample goal:

```
:- coloring( {R1,R2,R3},{{R1,R2},{R2,R3}}, {c1,c2} ).
     R1 ← c1, R2 ← c2, R3 ← c1
     R1 ← c2, R2 ← c1, R3 ← c2.
```

As mentioned at the beginning of the paper, {log} has been developed as a general-purpose programming language. Nevertheless, there seem to be certain application areas in which the use of sets fits more naturally, allowing some definite improvements both in the quality and in the development time of the final software product. These "interesting" areas include database applications (see for instance [33, 42]), combinatorial problems, graph-related applications and operational research in general (e.g., resource allocation problems), as pointed out for instance in [25, 35].

To conclude, we show also a simple example in a quite unusual application area, namely music writing, where sets turn out to be a very natural notation to represent collections of notes.

Music. A combination of three or more tones sounded together in harmony is said to be a *chord*. There is an unambiguous symbolic notation, mainly used in rock and jazz scores, for denoting chords.

Writing a program that, given the symbolic denotation, returns the corresponding chord, is a simple task in any programming language. Solving the reverse problem is more difficult, since chords are genuine sets and each string of symbols usable to denote a chord is based on only one of the notes in it (the *tonic* note). Both problems are solved by the following {log} program, working for 3-notes chords. It is easy to extend it so as to treat n-notes chords (in jazz music 7-notes chords are common).

Note representation: for the sake of simplicity we assume a unique representative for each altered note (e.g. b♭ stands also for a♯).

```
note(a, 0).    note(b♭, 1).   note(b, 2).    note(c, 3).
note(d♭, 4).   note(d, 5).    note(e♭,6).    note(e, 7).
note(f,8).     note(g♭, 9).   note(g,10).    note(a♭, 11).
```

The {log} program:

```
chord({T | R},[T | S]) :-      %%% T for "tonic"
    rest(R,S,T).
rest({X3,X5},[min],T) :-       %%% min: minor chord
    interval(X3,T,3), interval(X5,T,7).
rest({X3,X5},[],T) :-          %%% major chord
    interval(X3,T,4), interval(X5,T,7).
rest({X3,X5},[maj5],T) :-      %%% maj5: augmented chord
    interval(X3,T,4), interval(X5,T,8).
rest({X3,X5},[dim],T) :-       %%% dim: diminished chord
    interval(X3,T,3), interval(X5,T,6).
interval(Y,X,Offset) :-        %%% offset determination
    note(X,N1), N2 is (N1 + Offset) mod 12, note(Y,N2).
```

Sample goals:

```
:- chord({ g, e♭, g, b♭}, X ).          :- chord(X,[f,min]).
   X ← [e♭]                                 X ← { f, a♭, c }

:- chord(X,Y).
   X ← {a,c,e}, Y ← [a, min]
       ⋮
   X ← { a♭, b, d }, Y ← [a♭, dim].
```

8. INTENSIONAL SETS

8.1. Abstract Set-Formers

In the practice of mathematics, only seldom a set S is denoted *extensionally*, that is by enumeration of its elements. Much more often, one provides a condition $\varphi \equiv \varphi[x]$ that is necessary and sufficient for an element x to belong to S. This *intensional* denotation of a set is achieved by use of an *abstract set-former*, whose typical syntactic form is

$$\{ x : \varphi[x] \}.$$

A notation of this kind is very useful in programming languages that embody sets (cf., e.g., [45]). The rest of this section is dedicated to the development of this feature in {log}.

A major theoretical difficulty related to the use of the notation $\{ x : \varphi[x] \}$ is that the latter does not make much sense unless one can show that

$$\exists S \, \forall x \, (x \in S \leftrightarrow \varphi)$$

follows as a theorem from the axioms of the set theory at hand. No theory of sets can make all formulas of this kind provable, without being inconsistent; this is why convenient syntactic restrictions are usually placed on φ, to the effect that whenever a set-former $\{ x : \varphi[x] \}$ is used, the corresponding existential statement is provable, i.e., there is truly a designated entity S.

Quite often, the only restriction placed on φ is *separation*; this is to say, $\varphi[x]$ is required to be of the form $x \in y \,\&\, \psi[x]$, where y is a variable that (unlike x) cannot occur in ψ. Reassuringly, this restriction suffices to prevent one from using meaningless set-formers in the most common theories of sets. In practice, however, even this small restriction may be an obstacle to the naturalness of expression: we prefer not to enforce it in our language.

Hence, an *intensional set term* in {log} will have the unrestrained form

$$\{X : G[X]\},$$

with X a variable and G any {log} goal (generally involving X). Note that the scope of X, in the clause containing the intensional set term, is the term itself. An intensional set term is allowed to occur in any position where an extensional set term (or, in fact, any ordinary term) can occur: for instance, it may occur inside the head of a clause.

The goal G in an intensional set term can be as complex as necessary (in particular G can contain any of the predefined predicates $=$, $\neq$, $\in$, $\notin$, as well as RUQs). For example

$$\{\mathsf{Z} : \mathsf{Z} = [\mathsf{Obj},\mathsf{People}],\ (\forall\ \mathsf{X} \in \mathsf{People})(\mathsf{likes}(\mathsf{X},\mathsf{Obj}))\}$$

defines a set of pairs of the form [Obj,People], where Obj and People are meant to represent an object and a set of people who like that object.

We have already pointed out our willingness to restrict our attention to *finite sets* in this paper.[6] Insuring finiteness in the case of intensional sets requires that

[6] Notice, however, that this restriction is not stated by our axioms. To properly reflect it, e.g. an induction scheme axiom could be added.

predicates in abstract set-formers represent finite relations. However, the problem of ascertaining the finiteness of a predicate is known to be undecidable in the general case. In this context we rely on the programmer's attention to avoid generation of infinite sets (as done also in [33], for instance). In [19], instead, we propose some syntactic properties of predicate definitions which conservatively ensure finiteness of the defined relations. Those properties are obtained by generalization of the Datalog approach to this problem [54].

Finally, note that, for the sake of simplicity, we assume that intensional sets are always based on an empty set kernel (i.e., we are not concerned with intentionally defined colored sets).

8.2. Programming with Intensional Sets

Let us see a few simple examples aimed at showing the *declarative programming style* supported by intensional sets.

8.2.1. Intersection. As a first example, we show a very straightforward definition of the intersection predicate (see Section 7) using intensional sets:

```
∩ (A, B, { X : X ∈ A, X ∈ B }).
```

Other basic set-theoretic operations can be redefined using intensional sets in a very similar way.

8.2.2. Prime numbers. Our second example is the definition of a predicate, primes, that computes all the prime numbers smaller than a given N:

```
primes(N,{X : between(1,N,X), is_prime(X)}).
is_prime(X) :-
    (∀ Z ∈ {Y : between(1,X,Y)}) non_div(Z,X).
between(A,B,C) :- A < C, C < B.
non_div(A,B) :- 0 =\= (B mod A).
```

8.2.3. Connected components of a graph. The next example regards undirected graphs. Assuming that every node belongs to at least one arc, a set Arcs of doubletons suffices to represent a graph.

The set of all nodes of a given graph can be determined as follows:

```
nodes(Arcs, {X : { X, _ } ∈ Arcs }).
```

To determine the set of all connected components of a given graph Arcs of cardinality N, we can exploit the following predicate:

```
components(1, Arcs, { Z : Z = { X }, nodes(Arcs, Nodes), Z ∈ Nodes }).
components(N, Arcs, Comp) :-
    N > 1,
     N1 is N - 1, components(N1, Arcs, Comp1),
    Comp = { Z : Z = { X | C }, Y ∈ C, C ∈ Comp1, X ∉ C, { X, Y } ∈ Arcs }.
```

8.2.4. Finite State Automata. Let us consider an *NFSA* $M = \langle \Sigma, Q, q_0, F, \delta \rangle$, whose transition function $\delta : Q \times \Sigma \to Pow(Q)$ we assume to have been modeled by the relation

$$\mathsf{A} = \{[q, s, \delta(q, s)] \,:\, q \in Q, s \in \Sigma\}\,.$$

We define the predicate delta_star that performs the transitive closure of a given transition function δ on a given input string S.

```
delta_star(SetQ, A, [], SetQ).
delta_star(SetQ_in, A, [S | R ], SetQ_out) :-
    delta_star( { Q : step(SetQ_in, A , S , Q) }, A , R, SetQ_out).
step(SetQ, A , S, Q) :-
    Q_p ∈ SetQ, [Q_p, S, SetQ_p] ∈ A, Q ∈ SetQ_p.
```

A predicate that checks whether or not a string S is accepted by an automaton defined by its transition function A, its initial state q_0, and its set Final of final states can be defined as follows:

```
accepted(S,A,Final) :-
    delta_star({q0},A,S,States), Z ∈ States, Z ∈ Final.
```

It is now straightforward to define procedures capable of testing common properties of FSA. For example, the following fragment of {log} program is intended to verify the equivalence between two deterministic FSA (it is just a direct application of the classical pumping lemma for DFSA, and we are not concerned here with the efficiency of the resulting code):

```
equivalent(A1,Final1,A2,Final2,Sigma) :-
    states(A1,N1), states(A2,N2), maximum( N1,N2,N),
    S = {X : string(Sigma,N,X) },
    {X : X ∈ S, accepted(X,A1,Final1)} = {X : X ∈ S, accepted(X,A2,Final2)}.
```

where

- states(A ,N) holds if N is the number of states of the automaton A;
- maximum(N_1,N_2,N) states that N is the maximum between N_1 and N_2;
- string(Sigma,N,X) holds if X is a string on the alphabet Sigma of length less than or equal to N.

8.3. Compiling Intensional Sets

Since we are assuming sets to be finite, we can replace abstract set formers by the corresponding extensional set terms. To allow such a replacement to be effective, however, the language should provide a *set-collection* mechanism capable of constructing the extensional set term corresponding to the abstract set former $\{x : \varphi[x]\}$ (i.e., collecting in the extensional term all the instances of x satisfying φ).

Given such a set-collection capability, abstract set-formers can be translated at compile time (via simple preprocessing) into the {log} clauses which construct the corresponding extensional set terms. Similarly to RUQs, therefore, abstract set-formers in {log} exist at the syntactic level only, without affecting the semantic structure of the language. Unification between abstract set-formers is not required at all. The extensional representations corresponding to set-formers are always built first, and then the set unification algorithm of Section 4 can be applied to the extensional set terms. For example, the goal

```
:- {X : p(X)} = {X : q(X)}
```

is first translated to the equivalent goal

$$:-\ \mathsf{setof\_p(S1),\ setof\_q(S2),\ S1 = S2}$$

where setof_q provide the required set-collection facility and S1 and S2 will be instantiated to set terms.

The set-collection mechanism could be a *built-in* feature of the language as in [6, 42]. The programmer could then write clauses (called *grouping* clauses) of the form

$$s(\langle X\rangle, Y_1, \ldots, Y_n) :- p(X, Y_1, \ldots, Y_n)\ .$$

The informal interpretation of such a clause is that, for a given instantiation of $Y_1, \ldots, Y_n$, all the values of X for which $p(X, Y_1, \ldots, Y_n)$ is true are collected into a set $(\langle X\rangle)$. X is called the *reference variable* and $p(X, Y_1, \ldots, Y_n)$ is the *indexing literal*. A bottom-up procedural semantics for the resulting language is presented in [6, 42], whereas a top-down approach is analyzed in [19] in the context of the {log} language extended with a built-in set-collection mechanism similar to that of [6, 42].

The set manipulation facilities supplied by {log}, however, give us the possibility of defining the set-collection mechanism directly within the language itself, provided thelanguage is endowed with some form of *negation*, allowed to occur in goals and clause bodies. Indeed such an enhanced version of {log} would allow a predicate that collects in a set S all values of X for which $p(X)$ is true to be easily defined as follows:

$$\begin{array}{l} setof\_p(S) :- \\ \qquad ((\forall X \in S)\, p(X)), \texttt{not}\ partial\_p(S), \{\}\ \texttt{ker\_of}\ S. \\ partial\_p(S) :- \\ \qquad X \notin S, p(X). \end{array}$$

Note that predicate *setof_p* is intended to collect all the possible values of X and, at the same time, to reject any partial set of computed answers for p.

More precisely, the definition of predicate *setof_p* comes out from the following observations. By exploiting the extensionality axiom (actually the standard one, i.e., disregarding set kernels, since we have assumed that in this context kernels are always equal to the empty set), one can prove that:

$$\begin{array}{rcl} \{X : p(X)\} = S & \longleftrightarrow & \forall X (X \in S \leftrightarrow p(X)) \\ & \longleftrightarrow & \forall X (X \in S \rightarrow p(X))\ \&\ \forall X (p(X) \rightarrow X \in S). \end{array}$$

From the last formula, we can see that set grouping calls for the ability to perform restricted universal quantification as well as universal quantification over the solutions of an arbitrary predicate. Even though {log} supports restricted universal quantification, it is unable to express the other form of quantification. However, one can further observe that:

$$\forall X (p(X) \rightarrow X \in S) \leftrightarrow \forall X (\neg p(X) \vee X \in S) \leftrightarrow \neg \exists X (X \notin S\ \&\ p(X)),$$

where the existentially quantified formula can be rendered in {log} as in the definition of *partial_p* shown above. Thus, what we really need is just some form of negation in clause bodies.

The use of different forms of negation in {log} has been investigated in previous works. In particular:

Negation As Failure: the use of negation as failure [12, 14, 46] in {log} has been considered in [16, 19]. The extension of SLDNF-resolution to include the new features of our language is quite straightforward and appears to be easily implementable. On the other hand the syntactic restrictions imposed by negation as failure on the source program, in order to obtain soundness and avoid floundering, impose some limitations on the range of representable intensional sets. In particular only *ground* elements may appear in these sets.

Nevertheless, the use of negation as failure is sufficient to give to {log} the same power (in terms of intensional definitions) offered by similar proposals described in the literature (e.g., [6, 7]).

Constructive Negation: the use of constructive negation [13, 52] in {log} has been studied in detail in [8, 21]. Constructive Negation fits quite elegantly in the constraint-based framework of {log} and allows to overcome some of the limitations encountered in the case of negation as failure. In particular some large families of *nonground* sets can be defined.

The relationships between intensional set definitions and negation, and the intrinsic features and limitations of the various forms of negation that can be embodied into the {log} framework, are currently under investigation.

9. RELATED WORK

Among the proposals that have been put forward aiming at integrating sets into a logic paradigm, we briefly recall $\mathcal{LDL}$ [6, 7, 42], LPS [32, 33], SEL [28, 29], CLPS [35, 36], and Conjunto [25].

9.1. LDL

$\mathcal{LDL}$ (Logical Data Language) [6, 7, 42] is a language originally developed at MCC and oriented towards the development of deductive databases. Its basic language is quite similar to the one used in {log}, using a set constructor `scons` with the same meaning of `with`. On the other hand, the different objectives underlying the development of the two languages lead to a number of relevant differences between them.

- $\mathcal{LDL}$ relies on a *bottom-up* operational semantics, instead of using a top-down semantics like {log}. This requires, among others, a number of conditions to be satisfied to guarantee termination (e.g. range restriction).
- Since $\mathcal{LDL}$ is intended for data intensive applications, it is assumed that only ground answers are of interest.
- Set unification is not required in $\mathcal{LDL}$. The bottom-up semantics allows matching to be used in place of full unification, by guaranteeing that one of the terms to be compared is ground. Furthermore, in [47], it is shown that a program can be rewritten at compile-time into an equivalent one in which each rule containing set terms is replaced by a *set of rules*, each containing first-order, i.e., nonset, terms only.
- $\mathcal{LDL}$ offers a built-in set-collection mechanism. The class of acceptable programs is restricted using syntactic properties (e.g. stratification across intensional definitions) in order to avoid badly defined intensional sets (like

those allowing cyclic memberships). However, most of the syntactic restrictions imposed are equivalent to those typically found in works regarding the completeness of SLDNF resolution, thus confirming the tight link between intensional definitions and negation.

9.2. LPS

LPS (Logic Programming with Sets) [32, 33] is another language developed with the intent of supporting deductive databases. LPS adopts a slightly different approach to set representation, since an apparently infinite signature of function symbols is introduced, one for each possible cardinality of the described sets. Furthermore set terms are restricted to be *flat*, i.e., no nesting of sets is allowed. Differently from $\mathcal{LDL}$, LPS adopts a top-down operational semantics, consisting of a modified SLD-resolution procedure, capable of handling explicit Restricted Universal Quantifications. No set-collection capability is provided in LPS.

9.3. Subset-Logic Programming

The Subset-Logic Programming paradigm and its related languages (e.g. SEL) [28, 29] offers an interesting combination between functional and logic programming, based on the use of subset assertions

$$expression \subseteq f(terms)$$

and subset-relational clauses

$$set \subseteq f(terms) :- B\,.$$

Sets are represented as in {log}, using a `with`-like constructor. The execution model of programs containing subset assertions and equational assertions is based on innermost reduction and *restricted A-C matching* (restrictions are concerned with the use of the `with`-like constructor in place of the more general $\cup$ constructor). Arguments to functions defined by subset assertions are required to be ground terms. Program are "completed" by incorporating the Collect-all Assumption and the Emptiness-as-failure Assumption, which basically allow equality assertions to be derived from subset assertions. Differently from {log}, here, the absorption property of the `with`-like set constructor is intentionally disallowed. Thus, by writing $s\,\mathtt{with}\,t$ one implicitly assumes that $t \notin s$.

In [28] it is shown that this paradigm can be further extended in the direction of logic programming with sets—in the sense of {log}—by allowing also subset-relational clauses and definite clauses to occur in a program. The need for *restricted A-C unification* is briefly discussed as well as the need for stratification of programs. Also it is shown that subset-relational programs can be used to simulate Prolog's `setof` feature. Unfortunately a more comprehensive description of the proposed solutions and related problems is not provided.

9.4. CLPS

CLPS (constraint logic programming on sets) [35, 36] is a general-purpose language like {log}. Set terms in CLPS can be expressed extensionally using either a `with`-like operator or using more generic set constructors (e.g., $\cup$, $\cap$, etc.). The operational semantics of the language is based on a CLP scheme [27], where atomic formulas built using $=$, $\neq$, $\in$, $\notin$, $\cup$, $\cap$, and $\#$ (cardinality) are treated as constraints. Some powerful algorithms for managing these constraints have been developed (by adapting the traditional *forward-checking*, and *partial lookahead* algorithms), enabling one to get very good results in terms of execution time on various benchmarks.

Compared with {log}, CLPS has been developed using a different perspective: while CLPS always delays constraint execution waiting for a sufficient level of instantiation of the terms involved, {log} always tries to solve as much as possible immediately. This results in a possible slower execution in {log}, with the advantage of

- Having more explicit and meaningful final answers for problems where only partial input data are supplied.
- Eventually, the ability to perform early pruning of certain branches of the computation tree.

9.5. Conjunto

Conjunto [25] is a constraint logic programming language in which constraints are applied to finite set domain variables. The syntax of the language is considerably simpler than {log}, since it allows only finite and extensionally defined sets, composed exclusively of ground Herbrand terms (nested sets are not allowed). The language allows to build fairly complex set expressions, using the operators $\cup$, $\cap$, $\setminus$, and $\#$ (cardinality). These expressions are used as part of set constraints, which are based on the binary predicates $\subseteq$, $\in$, $\notin$, $\neq^0$, and :: (where $\neq^0$ indicates *disjointness* and :: is used to associate a *domain* to a variable).

The management of set constraints is based on a generalization of the finite domain techniques as used in CHIP [15]. The generalization lies in the fact that in Conjunto a *set domain* is constituted by a finite sets of known sets, which is expressed by supplying the greatest lower bound of the domain (i.e., intersection of all the sets in the domain) and the least upper bound (i.e., the union of all the sets in the domain). For example, a domain composed by the sets $\{a\}, \{a,b\}, \{a,c\}, \{a,b,c\}$ is described by the glb $\{a\}$ and by the lub $\{a,b,c\}$ (i.e., the domain is $\{\{a\}..\{a,b,c\}\}$ using Conjunto's notation).

This language has a narrower scope of application than {log}, being mainly targeted to express and solve combinatorial problems (as happens for most of the finite-domain based programming languages). On these particular problems Conjunto has been proven to be extremely effective, guaranteeing a good level of expressiveness and, more important, a considerably speed-up in execution time over traditional logic programming solutions.

10. CONCLUSION AND FUTURE WORK

In this paper we have addressed the problem of introducing sets in a logic programming language. The approach we have adopted is that of a deep integration between simple set designations and operations and the usual logic programming machinery. This has required primarily the development of a suitable semantical extension of Horn clause logic.

The first implementation of the {log} language was achieved by well-known metaprogramming techniques: a simple, yet complete, {log} interpreter was developed as a new layer on top of a standard Prolog machine (namely, Arity Prolog). A more efficient implementation of {log} has been developed too, based on an extended Warren Abstract Machine [4], called {WAM}, expressly designed to support the features of a logic language with sets like {log}. A detailed description of the {WAM} and of the whole compiler based on it can be found in [44] and in [20].

We have already mentioned various open problems and possible extensions to the present version of our language. Among them:

- Inserting new set abstractions into the language, namely multisets and hypersets.
- Precisely defining connections between intensional sets and negation.
- Reducing the number of redundant answers generated by the unification algorithm.

Work is in progress at present to tackle most of these problems. In particular a new unification algorithm which is able to deal with hypersets (as well as with conventional sets) has been designed [18] and potentialities offered by this kind of abstraction are under investigation.

Intensional sets and the use of constructive negation for their definition have been analyzed in detail in [8]. Work is in progress at present, aimed at obtaining a precise characterization of the kind of intensional sets which can be represented safely in the extended language [21].

Finally, the presence of many redundant (though correct) answers may represent a serious drawback to the effective use of our language for real applications. This problem too is under investigation at present, and we are experimenting with the application to our language of operational strategies based upon memoization [55].

APPENDIX

PROOF OF THEOREM 5.1. To keep the proof simple we assume that the algorithm selects the constraints by the following priority criteria:

1. Select first all the $\in$-constraints.
2. Select then all the `ker_of`-constraints.
3. Finally select arbitrarily the remaining ones.

The initial series of actions of *Can* eliminates (in a finite number of steps) all $\in$-constraints given from the outset. The successive series of actions simplifies the `ker_of`-constraints. This phase clearly terminates, too. In fact:

- In all actions, but in action 14, the constraint is removed.
- In action 14 the constraint is replaced by a "simpler" one (and no new constraints are generated).

After this, we get a situation in which no $\in$-constraints are present and all the `ker_of`-constraints are in normal form. In the meantime the original $\neq$- and $\notin$-constraints may have been modified by the substitutions generated during the initial series of reductions. We refer as C to the constraint so reached.

For each constraint C, we define the complexity of C to be the pair $\langle A, B\rangle$, where

- A is the sum of the sizes of all terms occurring in the right-hand side of each atomic non-canonical constraints of C;
- B is the sum of the sizes of all terms occurring in both sides of each atomic non-canonical constraints of C.

We will show that any non-failing sequence of actions lowers (w.r.t. the lexicographic order) such complexity. Since this ordering is well-founded, *Can* terminates.

One immediately sees that the application of actions 1–9 causes the complexity to decrease. Furthermore, neither $\in$- nor `ker_of`-constraints are generated during the execution of such actions.

The only action that deserves a complete analysis is the application of step 10.

Let us consider the 3 different cases separately:

10.(c): The original constraint $\{t|r\} \neq \{s|r'\}$ is reduced to the constraints, $Z\,\texttt{ker\_of}\,r$, $W\,\texttt{ker\_of}\,r'$, and $Z \neq W$. First of all observe that Z and W are strictly new variables and, as such, they do not appear anywhere else. The immediately ensuing reduction steps will simplify the first two new constraints, finally producing two constraints $Z\,\texttt{ker\_of}\,h_1$ and $W\,\texttt{ker\_of}\,h_2$, where h_1 and h_2 are variables or memberless terms. If h_1 and h_2 are both variables, then the constraints produced are already in normal form and nothing else needs to be done (note that every new constraint has size strictly lower than the initial one). If either h_1 or h_2 is a memberless term, e.g. $h_1 \equiv f(t_1, \dots, t_k)$, then the new substitution $\{Z \leftarrow f(t_1, \dots, t_k)\}$ is produced by action 15. In the system there is only one occurrence of Z, in the constraint $Z \neq W$, which will turn into $f(t_1, \dots, t_k) \neq W$ once the substitution is applied. Thus we have again that the original constraint has been replaced by some constraints whose size is strictly lower than the original size.

10.(a): We are considering an atomic constraint of the form $\{t_0|r\} \neq \{s_0|r'\}$, where $r \equiv \{t_1, \dots, t_n|h\}$ with h a memberless term or a variable. Action 10(a) replaces the constraint by the two new constraints $Z \in \{t_0|r\}$ and $Z \notin \{s_0|r'\}$. Since the $\in$-constraints are always considered first, the first new constraint will be immediately solved at the next step. Two cases are then possible:

1. The new constraint $Z \in \{t_0|r\}$ is solved by computing one of the mgus of the equation $Z = t_i$ $(0 \leq i \leq n)$, which has the form $\{Z \leftarrow t_i\}$. The only other occurrence of Z is in the constraint $Z \notin \{s_0|r'\}$, which will be transformed by the substitution to $t_i \notin \{s_0|r'\}$. Doing this we managed to replace the original constraint by one of strictly lower size.
2. The new constraint $Z \in \{t_0|r\}$ is solved replacing it by the constraint $Z \in h$. If h is a memberless term, then the constraint $Z \in h$ fails. Otherwise a new substitution of the form $\{h \leftarrow \{Z|N\}\}$ is generated and applied.
   Having already handled all $\in$- and `ker_of`-constraints, actions 12 and 13(b) can generate only substitutions of the form $\{X \leftarrow \{Z\,|\,N\}\}$ and $\{Z' \leftarrow t\}$, Z, Z', N new variables. Moreover, action 15 cannot be fired anymore. It follows that, for each variable X, the number k of substitutions of the form
   $$\{X \leftarrow \{Z_1\,|\,N_1\}\}, \{N_1 \leftarrow \{Z_2\,|\,N_12\}\}, \dots, \{N_{k-1} \leftarrow \{Z_k\,|\,N_k\}\}$$

possibly introduced by action 12 is less than or equal to the number, $\#(X)$, of occurrences of X in C.
Let $X_1, \ldots, X_m$ are all distinct variables occurring in C as set remainder variables, and let θ be the substitution
$\{X_1 \leftarrow Z_1^1, \ldots, Z_{\#(X_1)}^1, \ldots, X_m \leftarrow Z_1^m, \ldots, Z_{\#(X_m)}^m\}$,
where Z_i^js are pairwise distinct new variables.

The termination of $Can(C)$ follows from the termination of $Can(C^\theta)$; actually, we are testing the longest possible sequences of substitutions for the variables generated by successive applications of action 12—the latter action cannot be fired when C gets replaced by C^θ.

To complete the analysis of this step we are to make sure that the substitutions generated do not affect the `ker_of`-constraints (in particular, action 15 is never fired). If the h variable appears in a `ker_of`-constraint, we can have the following cases (remembering that the `ker_of`-constraints are all in normalized form):

(a) The constraint is t `ker_of` h and h does not appear in t. After the application of step 12 the constraint becomes t `ker_of` $\{Z|N\}$. Only one further application of step 14 is required to bring the constraint back to a normal form, t `ker_of` N.

(b) The constraint is t `ker_of` h and h appears in t. By the definition of canonical form, this may happen only if $t \equiv h$. After the substitution generated in step 12 is applied, the constraint turns into $\{Z|N\}$ `ker_of` $\{Z|N\}$, which leads to a failure.

(c) The constraint is t `ker_of` X and h appears in t. If t is h itself, then the application of the substitution leads to a failure (step 16). If t properly contains the variable h, then the substitution leaves the constraint in normal form (i.e., no further actions are required).

10.(b): The situation is perfectly symmetrical to the one just discussed. □

PROOF OF THEOREM 5.2.

A) C is in canonical form. Let $X_1, \ldots, X_m$ be the variables occurring in C, and let f be a unary function symbol not occurring in C. It is easy to show that the substitution

$$\gamma = \{ X_1 \leftarrow f^1(\{\}), \ldots, X_m \leftarrow f^m(\{\}) \},$$

where $f^n(x)$ stands for $f(f(\cdots(f(x))\cdots))$, f occurring n times, $n \geq 1$, is a ground solution of C.

B) It suffices to show that at each step of the recursive call of *Can* the result holds. We can see that every step preserves the desired property:

- Case 1: follows from the axioms of `with`;
- Case 2: follows from axiom $(\mathbf{K}_1)$;
- Case 3: follows by axioms $(\mathbf{R})$ and $(\mathbf{R}')$;
- Cases 4 and 5: follows from the freeness axioms;
- Cases 6 and 7: follows from equality axioms;
- Case 8: follows from freeness and regularity axioms;
- Case 9: follows from axioms $(\mathbf{W}_{1,2})$ and $(\mathbf{L})$;
- Case 10: follows from the axioms of `with` and the kernel and extensionality axioms;

- Cases 11, 12, and 13: follows from the axioms of `with`;
- Cases 14, 15, and 16: follows from the kernel axioms;
- Case 17: follows from $(\mathbf{R}')$. □

PROOF OF THEOREM 5.3 (hint) As in the proof of Theorem 5.2, it suffices to analyze a single step of the algorithm *Can*. The axioms used are the same as in the proof of Theorem 5.2, with the roles of the freeness and equality axioms interchanged. □

PROOF OF THEOREM 5.5. The statement is proved by induction on the length n of the refutation of G.

Base, $n = 0$: $G \equiv C$, C in canonical form (perhaps empty). The result holds by the soundness of the unification and the canonization algorithm.

Step: Suppose that the result holds for every refutation of length $n \leq g$. We show that it holds also for $n = g + 1$. Let $G \equiv (:\!-\, C_1 , \cdots , C_m , L_1 , \cdots , L_s)$ be the goal and let L_i be the literal selected at the first step.

1. L_i is $t = s$: the result ensues from the soundness of the unification and constraint canonization algorithms and from the inductive hypothesis;
2. L_i is $t \in s$: the result ensues from the soundness of the canonization algorithm and from the inductive hypothesis;
3. L_i is $p(t_1, \ldots, t_k)$. Let
   $p(s_1, \ldots, s_k) :\!-\, D_1 , \cdots , D_r , B_1 , \cdots , B_p$
   be the corresponding selected clause and μ be an mgu of
   $\{t_1 = s_1, \ldots, t_k = s_k\}$.
   If $\langle\{C'_1, \ldots, C'_j\}, \tau\rangle$ is one of the pairs returned by
   $Can(\langle\{C_1, \ldots, C_m, D_1, \ldots, D_r\}^{\mu}, \varepsilon\rangle)$,
   then
   $$G' \equiv (:\!-\, C'_1 , \cdots , C'_j , (L_1 , \cdots , L_{i-1} , B_1 , \cdots , B_p , L_{i+1} , \cdots , L_s)^{\mu\circ\tau}) .$$
   The computation, then, proceeds from G'. The computed answer $\langle C'', \beta\rangle$ is obtained by a g-step computation. It follows from the inductive hypothesis that

   (a) $P \models_{Set} (G')^{\beta\circ\sigma} \equiv$
   $(:\!-\, C'_1 , \cdots , C'_j , (L_1 , \cdots , L_{i-1} , B_1 , \cdots , B_p , L_{i+1} , \cdots , L_s)^{\mu\circ\tau})^{\beta\circ\sigma}$
   for every solution σ of C''. Moreover $\tau \circ \beta \circ \sigma$ is a solution of
   $\{C_1, \ldots, C_m, D_1, \ldots, D_r\}^{\mu}$ (by the soundness of *Can*) and hence

   (b) $\mu \circ \tau \circ \beta \circ \sigma$ is a solution of $\{C_1, \ldots, C_m, D_1, \ldots, D_r\}$. Now, (a) and (b) imply

   (c) $\models_{Set} (C_1 , \cdots , C_m , D_1 , \cdots , D_r)^{\mu\circ\tau\circ\beta\circ\sigma}$. By the inductive hypothesis we also have $P \models_{Set} ((L_1 , \cdots , L_{i-1}, B_1 , \ldots , B_p , L_{i+1} , \ldots , L_s)^{\mu\circ\tau})^{\beta\circ\sigma}$. Since the selected clause belongs to P, we obtain

   (d) $P \models_{Set} (L_1 , \cdots , L_{i-1} , p(t_1, \ldots, t_k) , L_{i+1} , \cdots , L_s)^{\mu\circ\tau\circ\beta\circ\sigma}$.

   From (c) and (d) we get the desired result. □

PROOF OF LEMMA 5.1. It suffices to prove the result for singleton constraints. The proof is based on an exhaustive analysis of the various actions of the algorithm.

Since we are assuming a pair $\langle C_1, \theta_1\rangle$ to be generated by the first step of *Can*, we are not interested in actions 2, 3, 4, 6 and 8. Moreover, action 7 and actions 11–17 do not create any trouble.

The proof for actions 1, 5 and 9 can be carried out quite straightforwardly choosing $\gamma = \eta = \sigma$. For instance, let C^σ be the constraint $\{X \neq \{t_0, \ldots, t_n \,|\, X\}\}$, X not occurring in $t_0 \ldots t_n$, and let $\langle C_1, \theta_1\rangle$ be $\langle\{t_i \notin X\}, \varepsilon\rangle$, that is one of the pairs obtained by applying action 9 to C^σ. Now,

> σ will have the form $\{Y \leftarrow X, Z \leftarrow \{v_1, \ldots, v_r \,|\, X\}, \ldots\}$ with
> $C = \{Y \neq \{s_1, \ldots, s_{n-r} \,|\, X\}\}$ and
> $v_1 = t_0, \ldots, v_r = t_{r-1}, s_1^\sigma = t_r, \ldots, s_{n-r}^\sigma = t_n$.

The application of *Can* to C may produce:

a. *fail* when $\sigma = \{Y \leftarrow X, Z \leftarrow X\}$, and $C = \{Y \neq Y\}$ (action 6);
b. $\langle\emptyset, \varepsilon\rangle$ when Y occurs in $s_1 \ldots s_{n-r}$ (action 8): then X would occur in $t_0 \ldots t_n$;
c. $\langle C, \varepsilon\rangle$ when Y and Z are not the same variable (C is in canonical form): then

   (i) if μ is a solution of $\{t_i \notin X\}$ then $\sigma \circ \mu$ is clearly a solution of C;
   (ii) trivial, since $\theta_1 = \theta_2 = \varepsilon$, with η as σ;
   (iii) by applying *Can* to C^σ, we get $\langle\{t_i \notin X\}, \varepsilon\rangle$ by hypothesis;

d. $\langle\{s_j \notin X\}, \varepsilon\rangle$ when Y and Z are the same variable and $\sigma = \{Y \leftarrow X, Z \leftarrow X\})$ (action 9): trivial.

Let us finally consider action 10. Let c be the constraint $\{s_0, \ldots, s_m \,|\, s\} \neq \{t_0, \ldots, t_n \,|\, t\}$. The effect of action 10 of *Can*, plus a number of applications of action 1 and/or action 14, is to rewrite $\langle C, \sigma\rangle$ into one of the following answers:

- **return** $Can(\langle((C \setminus \{c\}) \cup \{s_i \neq t_0, \ldots, s_i \neq t_n, s_i \notin t\}), \sigma\rangle)$, $i = 0, \ldots, m$ ($m+1$ actions);
- if s is a variable then **return** $Can(\langle((C \setminus \{c\}) \cup \{Z \neq t_0, \ldots, Z \neq t_n, Z \notin t\})^\delta, \delta \circ \sigma\rangle)$, $\delta = \{s \leftarrow \{Z \,|\, N\}\}$;
- **return** $Can(\langle((C \setminus \{c\}) \cup \{t_j \neq s_0, \ldots, t_j \neq s_m, t_j \notin s\}), \sigma\rangle)$, $j = 1, \ldots, n$ ($n+1$ actions);
- if t is a variable then **return** $Can(\langle((C \setminus \{c\}) \cup \{Z \neq s_0, \ldots, Z \neq s_m, Z \notin s\})^\delta, \delta \circ \sigma\rangle)$, $\delta = \{t \leftarrow \{Z \,|\, N\}\}$.
- **return** $Can(\langle((C \setminus \{c\}) \cup \{Z\, \mathtt{ker\_of}\, s, W\, \mathtt{ker\_of}\, t, Z \neq W\}), \sigma\rangle)$ ($m+n+2$ actions).

In the last case, only a substitution for Z or W (occurring only there) can be infered. The only only other subcases in which a substitution different from ε is introduced are the ones in which s and/or t are variables. For the sake of brevity, we will concentrate on one case only, the one in which both s and t are variables. The validity of the theorem for simpler cases can be obtained from this one quite trivially.

Let C be the constraint $\{s_0, \ldots, s_m \,|\, X\} \neq \{t_0, \ldots, t_n \,|\, Y\}$ and σ be the substitution $\{X \leftarrow \{r_0, \ldots, r_p \,|\, M\}\,,\, Y \leftarrow \{v_0, \ldots, v_q \,|\, N\}\,,\, \ldots\}$. Then C^σ will be $\{r_0, \ldots, r_p, s_0^\sigma, \ldots, s_m^\sigma \,|\, M\} \neq \{v_0, \ldots, v_q, t_0^\sigma, \ldots, t_n^\sigma \,|\, N\}$.

- Suppose that

  $C_1 = \{r_i \neq v_0, \ldots, r_i \neq v_q, r_i \neq t_0^\sigma, \ldots, r_i \neq t_n^\sigma, r_i \notin N\}$ and
  $\theta_1 = \varepsilon$;

and consider the pair

$$C_2 = \{Z \neq t_0, \dots, Z \neq t_n, Z \notin Y\}\,,$$
$$\theta_2 = \{X \leftarrow \{Z \,|\, R\}\,,$$

which is returned by the first step of *Can* when applied to C.

(i) Let μ be a solution of C_1; then $\{Z \leftarrow r_i\} \circ \sigma \circ \mu$ is a solution of C_2.
(ii) Let η be $\{Z \leftarrow r_i, R \leftarrow \{r_0, \dots, r_{i-1}, r_{i+1}, \dots, r_p \,|\, M\}\} \cup \sigma|_{dom(\sigma)\setminus\{X\}}$.
Then $\theta_2 \circ \eta = \{Z \leftarrow r_i, R \leftarrow \{r_0, \dots, r_{i-1}, r_{i+1}, \dots, r_p \,|\, M\},$
$X \leftarrow \{r_i, r_0, \dots, r_{i-1}, r_{i+1}, \dots, r_p \,|\, M\}\} \cup \sigma|_{dom(\sigma)\setminus\{X\}}$.
By the axioms of `with`, the claim is true.
(iii) With η as in (ii), we have
$C_2^\eta = \{r_i \neq t_0^\sigma, \dots, r_i \neq t_n^\sigma, r_i \notin \{v_0, \dots, v_q \,|\, N\}$.
Clearly $\langle C_1, \varepsilon\rangle$ is returned by *Can* applied to C_2^η.

- Suppose now that

$$C_1 = \{W \neq v_0, \dots, W \neq v_q, W \neq t_0^\sigma, \dots, W \neq t_n^\sigma, W \notin N\} \text{ and}$$
$$\theta_1 = \{M \leftarrow \{W \,|\, S\}\},$$

and consider $\langle C_2, \theta_2\rangle$ as above.

(i) Let μ be a solution of C_1; of course $\{Z \leftarrow W\} \circ \sigma \circ \mu$ is a solution of C_2.
(ii) Put $\eta = \{Z \leftarrow W, R \leftarrow \{r_0, \dots, r_p \,|\, S\} \cup \sigma|_{dom(\sigma)\setminus\{X\}}$. Then $\theta_2 \circ \eta =$ $\{Z \leftarrow W, R \leftarrow \{r_0, \dots, r_p \,|\, M\},\ X \leftarrow \{W, r_0, \dots, r_p \,|\, S\}\} \cup \sigma|_{dom(\sigma)\setminus\{X\}}$ and $\sigma \circ \theta_1 = \{M \leftarrow \{W \,|\, S\}, X \leftarrow \{W, r_0, \dots, r_p \,|\, S\}\} \cup \sigma|_{dom(\sigma)\setminus\{X\}}$.
By the axioms of `with`, the claim is true.
(iii) With η as in (ii), we have
$C_2^\eta = \{W \neq t_0^\sigma, \dots, W \neq t_n^\sigma, W \notin \{v_0, \dots, v_q \,|\, N\}\}$.
Clearly $\langle C_1, \varepsilon\rangle$ is returned by *Can* applied to C_2^η. □

PROOF OF LEMMA 5.3. The proof is carried out by induction on the length n of the refutation of the goal $:\!-\,(C\ ,\ R)^\sigma$.

Base, $n = 0$: The goal has the form $:\!-\,C^\sigma$ and C^σ is in canonical form. The result ensues from Lemma 5.1.

Step: Suppose that the result holds for $n \leq g$. We show that it holds also for $n = g + 1$. Suppose that the goal has the form $:\!-\,(C\ ,\ A\ ,\ R)^\sigma$, where

C is a constraint,
A is the selected literal,
R is a conjunction of atoms none of which is a constraint.

A may be of the form $t = s$, or of the form $p(t_1, \dots, t_k)$, with p an ordinary predicate. The most significant case is the latest one. We will concentrate on this one, being the other quite straightforward. Let

$p(s_1, \dots, s_k) :\!-\,D\ ,\ S$ be the selected clause, where D is a constraint and S is a conjunction of ordinary atoms;
μ_0' be one of the most general unifiers of $\{t_1^\sigma = s_1, \dots, t_k^\sigma = s_k\}$, and
$\langle C', \theta_0'\rangle$ be one of the pairs returned by $Can(\langle C^{\sigma\circ\mu_0'} \cup D^{\mu_0'}, \varepsilon\rangle)$.

Since D does not contain any variable occurring in $dom(\sigma)$, we have $D^{\mu_0'} = D^{\sigma\circ\mu_0'}$ and $\langle C', \tau\rangle$ is also returned by $Can(\langle C^{\sigma\circ\mu_0'} \cup D^{\sigma\circ\mu_0'}, \varepsilon\rangle)$. Hence the resolvent of the given goal with the selected clause is

$$\begin{aligned} G' \equiv\ & :- C', R^{\sigma\circ\mu_0'\circ\theta_0'}\ ,\ S^{\mu_0'\circ\theta_0'} \\ & \quad \text{(since } S \text{ does not contain any variable occurring in } dom(\sigma)\text{)}, \\ \equiv\ & :- C'\ ,\ R^{\sigma\circ\mu_0'\circ\theta_0'}\ ,\ S^{\sigma\circ\mu_0'\circ\theta_0'}. \end{aligned}$$

Let us now consider the goal $:- C\,,\,A\,,\,R$; let μ_0'' be an mgu of $\{t_1 = s_1, \ldots, t_k = s_k\}$ which is more general than $\sigma \circ \mu_0'$ and let ξ be the substitution such that $\mu_0'' \circ \xi = \sigma \circ \mu_0'$. By Lemma 5.2, there is a pair $\langle C'', \theta_0''\rangle$ returned by $Can(\langle C^{\mu_0''} \cup D^{\mu_0''}, \varepsilon\rangle)$ such that

(1) if γ is a solution of C' then there exists a β such that $\beta \circ \gamma$ is a solution of C'';
(2) there exists an η such that $\theta_0'' \circ \eta|_{vars(C^{\mu_0''} \cup D^{\mu_0''})} = \xi \circ \theta_0'|_{vars(C^{\mu_0''} \cup D^{\mu_0''})}$;
(3) with η as in (ii), either $C' = C''^{\eta}$ holds, or $\langle C', \varepsilon\rangle$ is returned by $Can(\langle C''^{\eta}, \varepsilon\rangle)$.

With such a choice, we get the resolvent

$$G'' \equiv\ :- C''\ ,\ R^{\mu_0''\circ\theta_0''}\ ,\ S^{\mu_0''\circ\theta_0''}. \tag{4}$$

Now,

since $\sigma \circ \mu_0' = \mu_0'' \circ \xi$,

we have $\sigma \circ \mu_0' \circ \theta_0' = \mu_0'' \circ \xi \circ \theta_0'$;

since $\theta_0'' \circ \eta|_{vars(C^{\mu_0''} \cup D^{\mu_0''})} = \xi \circ \theta_0'|_{vars(C^{\mu_0''} \cup D^{\mu_0''})}$, we have $\mu_0'' \circ \xi \circ \theta_0'|_{vars(C)} = \mu_0'' \circ \theta_0'' \circ \eta|_{vars(C)}$.

Therefore,

$$\sigma \circ \mu_0' \circ \theta_0'|_{vars(C)} = \mu_0'' \circ \theta_0'' \circ \eta|_{vars(C)}. \tag{5}$$

Now

$$\begin{aligned} G' \equiv\ & :- C'\ ,\ R^{\sigma\circ\mu_0'\circ\theta_0'}\ ,\ S^{\sigma\circ\mu_0'\circ\theta_0'} \text{ and, by (5),} \\ \equiv\ & :- C'\ ,\ (R^{\mu_0''\circ\theta_0''}\ ,\ S^{\mu_0''\circ\theta_0''})^{\eta}. \end{aligned}$$

By (3) we may replace C' with C''^{η} without affecting the derivation, so G' is equivalent (in the sense of refutability) to

$$(G'')^{\eta}\ \equiv\ :- (C''\ ,\ R^{\mu_0''\circ\theta_0''}\ ,\ S^{\mu_0''\circ\theta_0''})^{\eta}\ .$$

Then $(G'')^{\eta}$ has a g-step refutation with computed answer $\langle C_1, \theta_R'\rangle$ such that $\theta_1 = \mu_0' \circ \theta_0' \circ \theta_R'$.

Now it is possible to apply the induction hypothesis; so there exists a refutation for G'' with computed answer

$$\langle C_2, \theta_R''\rangle \tag{6}$$

such that

there exists a γ_R such that $\theta_R'' \circ \gamma_R|_{vars(C'')} = \eta \circ \theta_R'|_{vars(C'')}$ and (7)

for each solution α of C_1, there existsβ such that $\beta \circ \alpha$ is a solution of C_2.

[End of proof of part (ii) of the thesis]

By (4) and (6), then, there exists a refutation for $:- C\,,\,A\,,\,R$ with computed answer $\langle C_2, \theta_2\rangle$. Moreover, considering only the variables in C,

$$\sigma \circ \theta_1 \quad = \quad \sigma \circ \mu_0' \circ \theta_0' \circ \theta_R' \quad \text{(by construction)};$$

$$
\begin{array}{lll}
= & \mu_0'' \circ \theta_0'' \circ \eta \circ \theta_R' & \text{(by (5))} \\
= & \mu_0'' \circ \theta_0'' \circ \theta_R'' \circ \gamma_R & \text{(by (7))} \\
= & \theta_2 \circ \gamma_R & \text{(by construction)}
\end{array}
$$

[End of proof of part (i) of the thesis] □

PROOF OF LEMMA 5.4.
$\subseteq$) Straightforward from $M_P = T_P \uparrow \omega$.
$\supseteq$) Let $A \in M_P = T_P \uparrow \omega$; then there exists an $n > 0$ such that $A \in T_P \uparrow n$. The proof is by induction on n.

Base, $n = 1$: $p(a_1, \ldots, a_k) \in (T_P \uparrow 1) = T_P(\emptyset)$ and there is a clause

$$p(t_1, \ldots, t_k) :- C \ , \ B$$

(where C is a constraint and B is a conjunction of atoms $t = s$) together with a ground substitution γ such that $p(t_1, \ldots, t_k)^\gamma = p(a_1, \ldots, a_k)$ and $\models_{Set} \ C^\gamma \ , \ B^\gamma$. Then it is possible to build a refutation for $P \cup \{A\}$ whose first step is:

$$
\begin{array}{c}
:- \ p(a_1, \ldots, a_k) \\
| \\
\langle C', \varphi \rangle \in (C^{\gamma|_{vars(p(t_1,\ldots,t_k))}})_{Can} \\
| \\
:- \ C' \ , \ B^{\gamma|_{vars(p(t_1,\ldots,t_k))}} \ .
\end{array}
$$

The result ensues from unification results.

Step: Let $p(a_1, \ldots, a_k) \in ((T_P \uparrow n + 1) \setminus (T_P \uparrow n))$; then there exists a clause

$$p(s_1, \ldots, s_k) :- C \ , \ B \ , \ L_1 \ , \cdots , \ L_h$$

in P (C and B as above, and the L_is ordinary atoms) and a ground substitution γ such that

- $p(a_1, \ldots, a_k) = p(s_1, \ldots, s_k)^\gamma$;
- $\models_{Set} \ C^\gamma \ , \ B^\gamma$;
- for $i \in \{1, \ldots, h\}$, L_i^γ is ground and $L_i^\gamma \in (T_P \uparrow n)$.

By the induction hypothesis, $L_i^\gamma \in Succ(P)$; moreover it is possible to produce a refutation for B^γ (by unification results); since all atoms in the refutations of the L_i^γs and of B^γ are ground, we can combine these refutations to obtain a refutation of $:- (C \ , \ B \ , \ L_1 \ , \cdots , \ L_h)^\gamma$ in P.

The desired refutation starts with

$$
\begin{array}{c}
:- \ p(a_1, \ldots, a_k) \\
| \\
:- \ C^\mu \ , \ B^\mu \ , \ L_1^\mu \ , \cdots , \ L_h^\mu \ ,
\end{array}
$$

where μ is an mgu of $\{p(a_1, \ldots, a_k) = p(s_1, \ldots, s_k)\}$ such that $\gamma = \mu \circ \alpha$ for some α.

Then there exists a refutation for $:- (C \ , \ B \ , \ L_1 \ , \cdots , \ L_h)^{\mu \circ \alpha}$ in P; hence, by Lemma 5.3, there is a refutation for $:- (C \ , \ B \ , \ L_1 \ , \cdots , \ L_h)^\mu$ in P, and therefore there is one for $:- p(a_1, \ldots, a_k)$. □

PROOF OF LEMMA 5.5. If

$P \cup \{\neg C \vee \neg B \vee \neg L_1 \vee \cdots \vee \neg L_h\}$ is unsatisfiable in the theory *Set*,
then

$P \cup \{\neg C \vee \neg B \vee \neg L_1 \vee \cdots \vee \neg L_h\}^{\gamma}$ is false for each γ ground. This means that
$P \models_{Set} (C\ ,\ B\ ,\ L_1, \cdots,\ L_h)^{\gamma}$; so that, in particular,
$M_P \models_{Set} (C\ ,\ B\ ,\ L_1\ , \cdots,\ L_h)^{\gamma}$; and hence, by Lemma 5.4,
$L_i^{\gamma} \in Succ(P)$ for every $i = 1, \ldots, h$, and B^{γ} has a refutation too (by unification results).

Then $:\!-\ (C\ ,\ B\ ,\ L_1\ , \cdots,\ L_h)^{\gamma}$ has a refutation in P with computed answer $\langle \emptyset, \varepsilon \rangle$. By Lemma 5.3, $:\!-\ (C\ ,\ B\ ,\ L_1, \cdots,\ L_h)$ has a refutation in P. $\square$

Proof of Theorem 5.6. If $P \models_{Set} (\neg G^{\sigma})^{\forall}$, then $P \cup \{(G^{\sigma})^{\forall}\}$ is unsatisfiable. Let γ be a ground substitution acting on the (free) variables of G^{σ}; then $P \cup \{G^{\sigma \circ \gamma}\}$ is unsatisfiable. By Lemma 5.5, there exists a refutation for $G^{\sigma \circ \gamma}$ in P, and therefore, by Lemma 5.3, G has a refutation in P. $\square$

This work originates from a project, named AXL, funded by ENI and ENIDATA. Partial support came from the Italian National Research Council (CNR grants 94.00472.CT12 and 95.00502.CT12) and from Compulog-2 (Esprit project 6810). Enrico Pontelli was partially supported by NSF Grant CCR 92-11732 and by a fellowship from Phillips Petroleum.

We have enjoyed useful discussions with Gopal Gupta, Mario Rodríguez-Artalejo, and with Alberto Policriti, who contributed to the axiomatization. An anonymous referee gave helpful suggestions for improving the presentation. John Sabini helped us in revising a preliminary version of the manuscript.

REFERENCES

1. Abiteboul, S. and Grumbach, S., A Rule-Based Language with Functions and Sets, *ACM Trans. Database Syst.*, 16(1):1–30 (1991).
2. Aczel, P., *Non-well-founded sets.* Lecture Notes Vol. 14, Center for the Study of Language and Information, Stanford, CA, 1988.
3. Adi, M. and Kirchner, C., AC-Unification Race: The System Solving Approach and Its Implementation, in: A. Miola (ed.), *Proc. of the Intl. Symp. DISCO'90*, Lecture Notes in Computer Science, Vol. 429, Springer-Verlag, New York, 1990.
4. Aït-Kaci, H., *Warren's Abstract Machine—A Tutorial Reconstruction*, The MIT Press, Cambridge, MA, 1991.
5. Barwise, J. and Moss, L., Hypersets. *The Mathematical Intelligencer*, 13(4):31–41 (1991).
6. Beeri, C., Naqvi, S., Ramakrishnan, R., Shmueli, O., and Tsur, S., Sets and Negation in a Logic Database Language (LDL1), in: *Proceedings of the 6th ACM SIGMOD Symposium*, 1987.
7. Beeri, C., Naqvi, S., Shmueli, O., and Tsur, S., Set Constructors in a Logic Database Language. *J. Logic Programming*, 10(3):181–232 (1991).
8. Bruscoli, P., Dovier, A., Pontelli, E., and Rossi, G.-F., Compiling Intensional Sets in CLP, in: P. Van Hentenryck (ed.), *Proceeding of the 11th International Conference on Logic Programming*, The MIT Press, Cambridge, MA, 1994.
9. Burckert, H.-J., *A Resolution Principle for a Logic with Restricted Quantifiers.* Springer-Verlag, New York, 1991.
10. Büttner, W., Unification in the datastructure sets, in: J. H. Siekmann (ed.), *Proceedings of the 8th International Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 230, Springer-Verlag, New York, 1986.
11. Cantone, D., Ferro, A., and Omodeo, E. G., *Computable set theory*, Vol. 1. Clarendon Press, Oxford, 1989.

12. Cavedon, L. and Lloyd, J. W., A completeness Theorem for SLDNF-resolution. *J. Logic Programming*, 7(3):177–191 (1989).
13. Chan, D., Constructive Negation based on the Completed Database, in R. A. Kowalski and K. A. Bowen (eds.), *Proceedings of the North American Conference on Logic Programming'89*, The MIT Press, Cambridge, MA, 1989.
14. Clark, K. L., Negation as Failure, in H. Gallaire and J. Minker (eds.), *Logic and Data Bases*, Plenum Press, New York, 1978.
15. Dincbas, M., Van Hentenryck, P., Simonis, H., and Aggoun, A., The Constraint Logic Programming Language CHIP, in: *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, Tokyo, 1988.
16. Dovier, A., Omodeo, E. G., Pontelli, E., and Rossi, G.-F., {log}: A Logic Programming Language with Finite Sets, in: K. Furukawa (ed.), *Proceedings of the 8th International Conference on Logic Programming*, The MIT Press, Cambridge, MA, 1991.
17. Dovier, A., Omodeo, E. G., Pontelli, E., and Rossi, G.-F., Embedding Finite Sets in a Logic Programming Language, in: E. Lamma and P. Mello (eds.), *Proceedings of the 3rd Workshop on Extensions of Logic programming*, Lecture Notes on Artificial Intelligence, Vol. 660, Springer-Verlag, New York, 1993.
18. Dovier, A., Omodeo, E. G., Policriti, A., and Rossi, G.-F., Solving systems of equations over hypersets. In M. Alpuente, R. Barbuti, and I. Ramos (eds.), *GULP-PRODE'94: Joint Conf. on Declarative Programming*, Peñiscola (Spain), 1994.
19. Dovier, A. and Pontelli, E., La programmazione logica con insiemi. "Laurea" thesis, University of Udine, 1991.
20. Dovier, A. and Pontelli, E., A WAM Implementation of a Logic Programming Language with Sets, in: M. Bruynooghe and J. Penjam (eds.), *Proceedings of the Programming Language Implementations and Logic Programming'93*, Lecture Notes in Computer Science, Vol. 714, Springer-Verlag, New York, 1993.
21. Dovier, A., Pontelli, E., and Rossi, G. F., The CLP Language {log}, and the Relation Between Intensional Sets and Negation. Technical Report NMSU-CSTR-9503, Dept. of Computer Science, New Mexico State University, 1995.
22. Dovier, A. and Rossi, G. F., Embedding extensional finite sets in CLP, in: D. Miller (ed.), *Proceedings of the 4^{th} Int'l Logic Programming Symp.*, Vancouver, The MIT Press, Cambridge, MA, 1993.
23. Enderton, H. B., *A mathematical introduction to logic.* Academic Press, New York, 1973.
24. Fogli, D., Gaspari, M., and Rossi, G.-F., *Adding Sets and Multisets to Gödel.* Presented at the workshop on programming in Gödel, University of Manchester, 14–15 April, 1994.
25. Gervet, C., Conjunto: constraint logic programming with finite set domains, in: *Proceedings of the 1994 International Symposium on Logic Programming*, The MIT Press, Cambridge, MA, 1994.
26. Hill, P. and Lloyd, J., *The Gödel Programming Language.* The MIT Press, Cambridge, MA, 1994.
27. Jaffar, J. and Lassez, J. L., From Unification to Constraints, in: *Proceedings of the 5th Conference on Logic Programming*, Tokyo, 1987.
28. Jayaraman, B. and Plaisted, D. A., Programming with Equations, Subsets and Relations, in: *Proceedings of the North American Conference on Logic Programming'89*, The MIT Press, Cambridge, MA, 1989.
29. Jayaraman, B., Implementation of Subset-Equational Programs. *J. Logic Programming*, 12(4):229–324 (1992).
30. Jech, T. J., *Set theory.* Academic Press, New York, 1978.
31. Kapur, D. and Narendran, P., NP-Completeness of the Set Unification and Matching Problems, in: J. H. Siekmann (ed.), *Proceedings of the 8th International Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 230, Springer-Verlag, New York, 1986.

32. Kuper, G. M., On the Expressive Power of Logic Programming with Sets, in: *Proceedings of the 7thACM SIGMOD Symposium*, 1988.
33. Kuper, G. M., Logic Programming with Sets. *J. Comput. Syst. Sci.*, 41(1):44–64 (1990).
34. Lassez, J. L., Maher, M. J., and Marriot, K., Unification revisited, in M. Boscarol, L. Carlucci Aiello, and G. Levi (eds.), *Foundations of Logic and Functional Programming*, Lecture Notes in Computer Science, Vol. 306, Springer-Verlag, New York, 1986.
35. Legeard, B. and Legros, E., CLPS: A Set Constraints Logic Programming Langage. Research Report, Laboratoire d'Automatique de Besançon, Institut de Productique, Besançon, France, February 1991.
36. Legros, E., Tests de satisfiabilité dans un language de programmation en logique avec contraintes ensemblistes. PhD Thesis. Université de Franche-Comté, 1993.
37. Livesey, M. and Siekmann, J., Unification of Sets and Multisets, Internal report MEMO SEKI-76-II, Institut für Informatik I, Universität Karlsruhe, 1976.
38. Lloyd, J. W., *Foundations of logic programming*, 2nd edition, Springer-Verlag, New York, 1987.
39. Martelli, A. and Montanari, U., An efficient unification algorithm, *ACM TOPLAS*, 4:258–282 (1982).
40. Munakata, T., Notes on Implementing Sets in Prolog. *Communications of the ACM*, 35(3):112–120 (1992).
41. Naftalin, M., An experiment in practical semantics, in Lecture Notes in Computer Science, Vol. 213, Springer-Verlag, New York, 1986.
42. Naqvi, S. and Tsur, T., *A Logical Language for Data and Knowledge Bases*, Computer Science Press, Rockville, MD, 1989.
43. Paterson, M. S. and Wegman, M. N., Linear unification, IBM Thomas J. Watson Research Center, Yorktown Heights, 1976.
44. Pontelli, E., Logic Programming with Sets: Theory and Implementation, M.Sc. Thesis, University of Houston, Houston, TX, 1992.
45. Schwartz, J. T., Dewar, R. B. K., Dubinsky, E., and Schonberg, E., *Programming with sets, an introduction to SETL*, Springer-Verlag, New York, 1986.
46. Shepherdson, J. C., Negation in Logic Programming. In J. Minker (ed.), *Foundations of deductive databases and Logic Programming*. Morgan Kaufmann, Los Altos, CA, 1987.
47. Shmueli, O., Tsur, S., and Zaniolo, C., Compilation of set terms in the logic data language (LDL). *J. Logic Programming*, 12(1):89–119 (1992).
48. Siekmann, J. H., Unification Theory, *J. Symbolic Computat.*, 7(1):207–274 (1989).
49. Sigal, R., Desiderata for Logic Programming with Sets, in: P. Mello (ed.), *Proceedings of the GULP89: 4th National Conference on Logic Programming*, Bologna (Italy), 1989.
50. Spivey, J. M., *The Z Notation: A reference Manual*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1992.
51. Stickel, M. E., A Unification Algorithm for Associative-Commutative Functions. *J. of the ACM*, 28(3):423–434 (1981).
52. Stuckey, P. J., Constructive Negation for Constraint Logic Programming, in: *Proceedings of the 6th IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, New York, 1991.
53. Turner, D., An overview of Miranda, *SIGPLAN Not.*, 21(12):158–166 (1986).
54. Ullman, J. D., *Principles of Database and Knowledge-base Systems.* Vols. 1 and 2, Computer Science Press, Rockville, MD, 1989.
55. Warren, D. S., Memoing for Logic Programs. *Communications of the ACM*, 35(3):93–111 (1992).