Embedding Finite Sets in a Logic Programming Language

Agostino Dovier
Dipartimento di Informatica
Università di Pisa
C.so Italia 40, 56100 PISA
dovier@di.unipi.it

Eugenio G. Omodeo
Dip. di Informatica e Sistemistica
Università di Roma La Sapienza
Via Salaria 113, 00198 ROMA
omodeo@assi.ing.uniroma1.it

Enrico Pontelli
New Mexico State Univ.
Dept. of Computer Science
Las Cruces, NM 88003
epontell@nmsu.edu

Gianfranco Rossi
Dip. di Matematica
Università di Bologna
P.zza di Porta S.Donato 5,
40127 BOLOGNA
gianfr@dm.unibo.it

Abstract. A way of introducing simple (finite) set designations and operations as first-class objects of an (unrestricted) logic programming language is discussed from both the declarative and the operational semantics viewpoint. First, special set terms are added to definite Horn clause logic and an extended Herbrand Universe based on an axiomatic characterization of the kind of sets we are dealing with is defined accordingly. Moreover, distinguished predicates representing set membership and equality are added to the base language along with their negative counterparts (π and $\not\sqsupset$). A new unification algorithm which can cope with set terms is developed and proved to terminate. Usual SLD resolution is modified so as to incorporate the new unification algorithm and to properly manage the distinguished predicates for set operations (in particular, conjunctions of atoms containing π and $\not\sqsupset$ are dealt with as constraints, first reduced to a canonical form through a suitable canonization algorithm). Finally, the application of the resulting language to the definition of Restricted Universal Quantifiers is discussed.

1 Introduction

General agreement exists about the usefulness of sets as very high-level representations of complex data structures (cf. [2]) and of set abstractions in code specifications. In particular, sets and set abstractions can be conveniently used in rapid software *prototyping*, where efficiency is not a primary requirement while the availability of high level data and operation abstractions are key features of the prototype implementation language.

Only relatively few programming languages embody sets as primitive objects. Among them, the (non-executable) specification language Z [26], the procedural language SETL [5, 24] and the functional languages MIRANDA [25] and ME TOO [18]. A number of recent proposals have preferred, however, a more declarative programming framework. In particular, *logic programming* languages have been advocated by many as a suitable setting for *declarative programming* with sets. Among them, the proposals of the logic database languages LDL [3] and LPS [11]. Also the recent work by Legeard and Legros [13] introduces sets in a Prolog-like language dealing with set expressions as constraints, while [23] analyzes the problem

of logic programming with sets in its generality (including infinite sets). A similar framework but stressing the notion of equational programming is also advocated in [9].

Actual Prolog systems 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. In fact, no precise logical semantics can be attributed to them. Sets are simply represented *as lists* and therefore are dealt with as ordinary terms (e.g. by an implicit order among their elements). On the contrary, a logic language embodying sets ought to supply special *set former terms* along with a few basic operations on them as part of the language, and allow other more complex operations on sets (possibly including the *setof* operation), to become definable within the language itself, without having to further extend its semantics.

Aim of this paper is to define such an extended logic language, called {log} (read set-log). Unlike some of the above cited proposals, however, we do not restrict ourselves to any specific application domain. So our starting point is a *pure* logic programming language, that is Definite Horn clauses with no extra-logical constructs. Then simple *set constructs* are added to this language (namely, enumerated set terms, set membership and equality and their negative counterparts) and suitable *operational* and *declarative semantics* are defined for them. It is shown that the usual set operations, such as union, intersection, etc., can be defined in this extended language. Also Restricted Universal Quantifiers are shown to be definable within the language itself; so they can be added to it as a simple syntactic extension (dealt with by a suitable preprocessor).

The paper is organized as follows. Section 2 briefly describes a few extensions to ordinary logic programming language syntax that can support the introduction of set terms. Declarative semantics of the resulting language is presented in section 3 by defining an extended Herbrand Universe based on an axiomatic characterization of the kind of sets we are dealing with. The problem of unification of set terms is then addressed in section 4 and a new unification algorithm which can deal with set terms is described in detail. 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. Finally, a comparison with some other related proposals is carried out in section 7.

2 {log} syntax

To begin with, we introduce in this section the *extensional* representation of *finite* sets. All that is presupposed is the availability of:

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

Apart from these two symbols, {log} contains the usual equipment of clausal Horn logic (cf. [14]) along with distinguished predicates for set membership and equality ($\in$ and $=$) and their negative counterparts ($\neq$ and $\notin$).

The extensional representation for sets referred to above is provided by a collection *G* of ground terms: this is the smallest collection such that

- the constant *{}* belongs to *G* ;
- *s with t*, *s*, *t* ground terms (not necessarily in *G*), belongs to *G* .

In view of the intended interpretation, any *s* in *G* will be called a *set term*. A non-ground term *t* is called a set term if there exists an instantiation σ of the variables in *t* such that t^σ belongs to *G*; in particular a variable is a set term. Set terms of the

form *t with* t_n *with ... with* t_1 wherein *t* is *not* a set term are intended to designate sets based on a *kernel t* other than *{}* (also called *colored sets*).

For the sake of simplicity we introduce special syntactic forms to designate set terms: $\{t_1,...,t_n|s\}$ stands for *s with* t_n *with ... with* t_1 and $\{t_1,...,t_n\}$ stands for *{} with* t_n *with ... with* t_1 where $n \geq 1$ and $s, t_1,...,t_n$ are terms. For example:

- `f(a,{5})`, i.e. `f(a,{}with 5)`, is a term, but not a set term;
- `{2,g(3),a}`, i.e. `{}with a with g(3)with2`, is a ground set term;
- `{}`, `{1,X,Y,2}`, `{1,1,{2,{}},f(a,{b})}` and any term $\{t_1,\ldots,t_n|R\}$ with a 'tail' variable R, are set terms
- `{a|f({b})}` is a colored set term based on the kernel `f({b})`.

For the rest of this paper, we will freely exploit the syntactic features available in Edinburgh Prolog in addition to the constructs discussed above. Three sample {log} clauses are:

```
- q(X) :- X ∈ {a,b,c,d} & p(X)

- singleton(X) :- X = {Y}.

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

Colored set terms (i.e. terms of the form $\{t_1,...,t_n|t\}$ wherein *t* is not a set term) do not designate sets of any conventional kind; one might therefore contend that such terms ought to be forbidden as they are unlikely to serve any serious purpose in programming. Nevertheless, we deem it convenient to always regard $\{t_1,...,t_n|t\}$ as a legal set term when $t_1,...,t_n$, *t* are legal, to make the language structure absolutely uniform and the inference mechanisms (e.g. unification) more straightforward.

3 {log} declarative semantics

To convey the meaning of our language, we are to formally characterize the legal interpretations of {log}. 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. [8]).

Next we will focus on a privileged interpretation domain formed by terms, U_H, resulting from suitable modifications to the classical Herbrand universe [14]; moreover, we will designate a fixed binary relation over U_H as the privileged interpretation of $\in$. (The predicate = will be interpreted as syntactic equality).

The interpretation of the predicate symbols of {log} other than $\in$ or = is left totally unspecified for the time being, as it must take into account a {log} program, viz. a set of application-specific clausal axioms enriching the initial axiom endowment we are presenting now.

3.1 Set axioms

Here are our basic axioms (*X*, *Y*, *V*, *S* and the X_is distinct variables ranging over the whole interpretation domain):

(Z) $V \in \{\}$;

(W1) $V \pi Y \varnothing (V \in X \text{ with } Y \times V \in X)$;

(W2) $Y \in X \text{ with } Y$;

(L) $Y \in X \varnothing \exists z (Y \in z \,\&\, X = z \text{ with } Y)$;

(K_0) $(\forall z\ z \in X) \varnothing (X = ker(X))$;

(K_1) $V \in ker(X)$;

(K_2) $ker(V \text{ with } Y) = ker(V)$;

(E) $(ker(X) = ker(Y) \;\&\; \forall z\,(z \in X \times z \in Y)) \varnothing X = Y$;

(R) $\exists z\, \forall w\,(w \in X \varnothing (z \in X \;\&\; w \in z))$;

(U) $f(X_1,...,X_n)\, \pi\, \{\}$ & $V \in f(X_1,...,X_n)$, *where n is the arity of f and* $f/n \in \{\{\}/0, ker/1, with/2\}$.

Although rather conventional in the overall, and somewhat narrow in comparison to well-established set theories such as Zermelo-Fraenkel or von Neumann-Bernays-Gödel, our theory slightly deviates from the classical ones under two respects:

- Presence of *urelements*. One refers by this word to member-less entities distinct from $\{\}$. It readily follows from (U) that a single function symbol, say $f/1$, can be used to generate infinitely many such entities, namely
  $f(\{\}), f(\{\{\}\}), f(\{\{\{\}\}\}), \ldots$.
- Each term t in the interpretation domain has an associated kernel, $ker(t)$, which can be an urelement or $\{\}$. Intuitively speaking, we think of t as resulting from repeated (possibly none) insertions of members into this initial kernel, insertions being achieved by the operation *with*. (Note that, $ker(f(a)) = f(a)$).[1]

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 (W_1), (W_2) and (K_2) that *with* enjoys the following properties:

$(X \text{ with } Y_1) \text{ with } Y_2 = (X \text{ with } Y_2) \text{ with } Y_1$, *(permutativity)*

$(X \text{ with } Y) \text{ with } Y = X \text{ with } Y$. *(absorption)*

Another very useful consequence of (W_1), (W_2), (K_2) and (E) is that

$Y \in X \times \exists z\,(X = z \text{ with } Y)$,

whereby one can express membership in terms of equality. By exploiting the element removal axiom (L) too, one obtains

$(X \text{ with } Y = V \text{ with } S \;\&\; Y \neq S) \varnothing \exists w\,(w \text{ with } S = X \;\&\; w \text{ with } Y = V)$,

which will prove crucial in our unification algorithm.

The regularity axiom (R) states that from each non-empty set X one can choose a member z which belongs to X and does not intersect X: this is a well-known expedient way to state that membership forms no cycles (and a little more).

It goes without saying that standard equality axioms (cf. [8]) are being postulated here. The Clark *freeness axioms* [21] are being adopted too:

$f(X_1,...,X_n) \neq g(Y_1,...,Y_m)$;

$f(X_1,...,X_n) = f(Y_1,...,Y_n) \varnothing (X_1 = Y_1 \;\&...\&\; X_n = Y_n)$;

$t[X] \neq X$.

As usual f/n and g/m differ from one another in the first of these; we must further require that they differ from $ker/1$ and from $with/2$. Moreover, in addition to requiring 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 about sets of the *occur axiom* $t[X] \neq X$. Actually, we must strengthen (R) for our purposes, by adding to it

(R') $t[X] \in X \;\&\; ker(X) \neq t[X]$.

---

[1] These also are the main differences, so far, between our theory and the theory *T0* in [19].

3.2 An adapted Herbrand universe

In order to define the *privileged interpretation domain* U_H one considers the ordinary Herbrand universe H (generated as usual by the collection F of functors, which is assumed to fulfil $\{\}/0$, $with/2 \in F$ and $ker/1 \in F$), and takes the smallest equivalence relation $\int$ over H that fulfils the above permutativity and absorption properties (with $\int$ in place of =). Then, one takes a representative term from each one of the equivalence classes forming $H/\int$ (according to specific criteria to be hinted at below), and finally one puts $U_H = \{representative\ terms\}$ by definition.

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 that a total ordering of F is given from the outset. By exploiting this ordering, < can be defined antilexicographically. This means – among others – that *r with t < s with u* holds, recursively, when either $t < u$ or t coincides with u and $r < s$.

A ground term g is said to be *canonical* if either it is a constant or every one of its subterms is canonical and, moreover, $t < u$ holds for every subterm of the form *s with t with u* of g. U_H will be formed by all canonical terms. Let τ be the function that maps a ground term t to its canonical representative. If t has the form $f(t_1,...,t_n)$ with $f/n \neq with/2$, $\tau(t)$ will be $f(t_1',...,t_n')$ where $t_i' = \tau(t_i)$ for each i, $1 \le i \le n$. If t has the form *k with* t_1 *with ...with* t_n – where the main functor of k differs from $with/2$ – $\tau(t)$ is the term k' *with* t_{π_1}' *with...with* t_{π_m}' where k', $t_{\pi_1}',...,t_{\pi_m}'$ are the distinct canonical representatives of k, $t_1,...,t_n$ arranged so that $t_{\pi_1}' < ... < t_{\pi_m}'$. Of course 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 that for any $t, u \in U_H$:

- the kernel $k = ker(t)$ is obtained by decomposing t in the form *k with* t_1 *with ... with* t_n, n≥0, where the main functor of k differs from $with/2$;
- 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.

The definition of the term canonization function τ described above can be extended so as to encompass atoms and clauses in the following way:

- for each ground atom $A = p(t_1,...,t_n)$, $\tau(A) = p(\tau(t_1),...,\tau(t_n))$;
- for each ground clause $C = A$:- B_1 &...& B_n, $\tau(C) = \tau(A)$:- $\tau(B_1)$ & ... & $\tau(B_n)$;
- for each set I of either ground atoms or ground clauses, $\tau(I) = \approx_{A \in I} \{\tau(A)\}$.

As anticipated at the beginning of the section, we are taking $U_H = \tau(H_P)$ as the interpretation domain of any given program P. Also, functors are so interpreted that each ground term t denotes $\tau(t)$. Then 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, because 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 *ground(P)*.

With the semantics thus restricted to set interpretations only, we can easily prove the usual model-theoretic and fixpoint semantics results. In particular, with

$T_P(I) = \{a \mid a$:- b_1 & ... & $b_n \in \tau(ground(P))$ *and, for each* $i \in \{1,...,n\}$, *either* $b_i \in \tau(I)$ *or* b_i *is* $s \pi t$ *where* $\pi \in \{=, \in, \pi, \ni\}$ *and* b_i *is true w.r.t. the axioms*$\}$

and

$$M_P = \underset{M \text{ model of } P}{\leftrightarrow} M$$

one will have

$M_P = T_P \neq \omega$.

3.3 Side remarks about set complementation

If we included in the {log} language a set complementation operator, *comp*/1, so that – among others –

- *comp({})* designates the whole interpretation domain, and
- *comp*(*comp*(*X*) *with Y*) designates *X* deprived of the element *Y*,

then co-finite sets ought to be taken into charge along with finite sets, which would significantly enrich the structure of any legitimate interpretation of {log}. We would then be forced to accept certain membership cycles – since $comp(\{\}) \in comp(\{\})$ –, and would have to modify the set axioms accordingly.

We refrain from this extension mainly because it would conflict with another very useful extension of {log}: the introduction of **intensional set formers** [6,7]. As a matter of fact, by adopting the two extensions together, one would run into paradoxical expressions like

$$\{ X \mid comp(\{\}) : X \in X \}.$$

Should the latter be regarded as an acceptable set expression, it would designate a set ξ fulfilling both $\xi \in \xi$ and $\xi \notin \xi$, as was first remarked by B. Russell in 1901.

4 Set unification

The development of a procedural semantics for {log} requires that the unification algorithm is refined in order to deal with sets, and that the SLD procedure is modified in such a way as to include set unification and proper management of the equality and membership relations. We cope in what follows with the first of these two points, postponing the second one to the next section.

4.1 The set unification problem

As regards the unification problem we assume all the definitions (e.g., Herbrand system, substitution, solution, etc.) given in [15] and [16].

Standard unification is no more adequate to deal with set terms. A 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 $E = \{\{X,Y\} = \{1,2\}\}$; there are only two solutions, namely $\sigma_1 = \{X \mapsto 1, Y \mapsto 2\}$ and $\sigma_2 = \{X \mapsto 2, Y \mapsto 1\}$, 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 non-unifiable. Furthermore, the equation $X = \{1|X\}$, which does not admit any solution in the standard case (unless infinite terms are taken into account), has the solution $\sigma = \{X \mapsto \{1|N\}\}$ 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 functional symbols (*with* and *{}* in our case) by means of a set of equations (axioms). In this connection, two terms *s* and *t* are said to be T-unifiable iff there exists a substitution σ such that $s^\sigma =_T t^\sigma$; such a σ is called a *T-unifier*. The set of all T-unifiers of two terms *s* and *t* is denoted by $\approx\Sigma_T(s,t)$ (cf. for instance [22]). Of course, we are interested in the *minimal set of unifiers* $\mu\Sigma_T(s,t)$, that is, the set of substitutions satisfying:

(i) $\mu\Sigma_T(s,t) \prod \Sigma$;

(ii) $\forall \delta \in \Sigma\ (\exists \sigma \in \mu\Sigma_T(s,t)\ (\sigma \leq_T \delta))$;

(iii) $\forall \sigma,\delta \in \mu\Sigma_T(s,t)\ (\sigma \leq_T \delta \Leftrightarrow \sigma = \delta)$

for each complete set of unifiers Σ of s and t (and in particular for $\Sigma = \approx\Sigma_T(s,t)$)[2].

The characterizing properties of our set constructor *with* are absorption and permutativity, as seen in section 3. Other properties for which a number of *extended unification* algorithms[3] have been developed in the past, such as associativity, commutativity, distributivity and their combinations, are unfortunately not satisfied by the *with* operator. In particular, ACI-unification cannot be directly applied in our case. In fact, the identity *(X with Y) with Z = X with (Y with Z)*, representing associativity, does not hold, for instance, under the substitution $\{X \blacklozenge \{\}$ *with c*, $Y \blacklozenge \{\}$ *with b*, $Z \blacklozenge a\}$, because the two sets $\{a,\{b\},c\}$ and $\{\{a,b\},c\}$ are distinct in our interpretation. Similarly, the idempotency property *X with X = X* does not hold under the substitution $\{X \blacklozenge \{\}$ *with a*$\}$ since $\{\{a\},a\} \neq \{a\}$.

A viable approach could be to use a *universal unification algorithm* for a class of theories $\mathbb{E}$ (i.e. "an algorithm which takes as its input a pair of terms (s,t) and a theory $E \in \mathbb{E}$ and generates a complete set of unifiers for $<s=t>_{\mathbb{E}}$" [22]), such as those based on narrowing or on some different rewriting technique (e.g. [17]). According to these approaches, the theory at hand should be represented as a canonical rewrite system. This is not the case of the theory we are interested in, since the permutativity rule for the *with* operator is non-terminating. By using suitable techniques (e.g. lazy evaluation) one can circumvent the ensuing problem of non-termination, but universal unification algorithms seem too inefficient for our purposes.

Therefore, we have developed a new unification algorithm that extends standard unification so as to embody the set axioms presented in section 3. For any given Herbrand system *E* involving set terms, the algorithm is able to compute through non-determinism each element of a complete set of unifiers of *E*.

4.2 Complexity

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

$$\Phi = (l^{(1);1} \,/\, l^{(1);2} \,/\, l^{(1);3}) \,\&\ldots\&\, (l^{(m);1} \,/\, l^{(m);2} \,/\, l^{(m);3})$$

with

$l^{(j);i} = a^{(j);i}$ *or* $l^{(j);i} = \neg a^{(j);i}$, where

$a^{(j);i} \in \{d_1,\ldots,d_k\},\ \forall i \in \{1,2,3\},\ \forall j \in \{1,\ldots,m\},\ k \leq 3*m,$

find a truth-value assignment to the propositional variables $d_1,\ldots,d_k$ such that $\Phi \times$ *true*.

We define the transformation function *f* as follows:

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

and translate Φ into the equation

$$\{\{X_1,Y_1\},\ldots,\{X_k,Y_k\},\ f(l^{(1);1} \,/\, l^{(1);2} \,/\, l^{(1);3}),\ldots,f(l^{(1);1} \,/\, l^{(1);2} \,/\, l^{(1);3})\} = \{\{false,$$

---

[2] Recall that: two substitutions δ, σ are **T-equal** (written $\delta =_T \sigma$) iff $X^\delta =_T X^\sigma$ for each variable *X*; a substitution δ is **more general than a substitution** σ (written $\delta \leq_T \sigma$) iff there exists a substitution μ such that $\delta \circ \mu =_T \sigma$ (for instance, $\delta = \{X \blacklozenge \{f(X),b\}\}$ is more general than $\sigma = \{X \blacklozenge \{b,f(a)\}\}$ in the theory we are considering here); two substitutions δ, σ are **T-equivalent** (written $\delta \in_T \sigma$) iff $\delta \leq_T \sigma$ and $\delta \geq_T \sigma$ (for instance, $\{X \blacklozenge \{a\}\}$, $\{X \blacklozenge \{a,a\}\}$, $\{X \blacklozenge \{a,a,a\}\}$ are all equivalent substitutions in our theory). Note that, if the set $\mu\Sigma_T(s,t)$ exists, it is unique up to the equivalence $\in_T$ and therefore it is enough to compute just one $\mu\Sigma_T(s,t)$ as a representative of the equivalence class $[\mu\Sigma_T(s,t)]_{\in_T}$ (cf. [Sie86]).

[3] I.e. algorithms for special theories embodying the equality axioms of the theory itself.

true}},
which requires each of the sets $\{X_i,Y_i\}$ to become *{false, true}*, and at least one of $f(l^{(i)};^1)$, $f(l^{(i)};^2)$, $f(l^{(i)};^3)$ 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).

It is hence clear that an algorithm for unifying two sides of such an equation in polynomial time could also be exploited for solving 3-SAT in polynomial time.

The NP-completeness of the set unification problem ensues from the polynomial behaviour of standard unification. In fact, a set-set equation can be replaced by a set of equations between the elements of the two sets. This is a non-deterministic equivalent of the term decomposition action of standard unification algorithms (see, for instance, [16]).

4.3 Set unification algorithm

Let *E* be a Herbrand system, i.e. a set of equations $\{t_i = t_i', i = 1,...,n\}$, where the t_i's are terms (variable or not). A Herbrand system is said to be in *solved form* if all equations have the form $x_i = s_i$ $(i = 1,...,m)$, and every variable x_i occurs exactly once in *E*. Such a system has the obvious solution $\{x_1 \blacklozenge s_1,...,x_n \blacklozenge s_m\}$. Aim of the following algorithm is to bring any given system *E* to solved form or to report a failure if *E* has no solution.

UNIFICATION ALGORITHM

Let *F* be a set of functional symbols, *V* a denumerable set of variables and *T* be the set of first order terms over $F \approx V$. Let *X*, *Y* be generic variables (i.e. $X,Y \in V$), *t*, t_i, t_i' be terms in *T*, and *r*, *s* represent set terms.

function *unify*(*E*: Herbrand_system): Herbrand_system;
begin
 if *E* is in solved form
 then return *E*
 else select arbitrarily an equation *e* in *E* so that:
  case *e* of
  1) $X = X$: return *unify*$(E \setminus \{e\})$;
  2) $t = X, t \notin V$: return *unify*$((E \setminus \{e\}) \approx \{X = t\})$;
  3) $X = t$, *t* is not a set term and *X* occurs in *t*: fail;
  4) $X = t$ *with* t_n*...with* t_0 and *X* occurs in t $(t \notin V)$ or in t_0 or...or in t_n: fail;
  5) $X = X$ *with* t_n*...with* t_0 and *X* does not occur in $t_0,...,t_n$:
  return *unify*$((E \setminus \{e\}) \approx \{X = N$ *with* t_n *... with* $t_0\})$, *N* new variable;
  6) $X = t$, *X* does not occur in *t*, *X* occurs somewhere else in *E*:
  return *unify*$((E \setminus \{e\})\sigma \approx \{X = t\})$, where σ is the substitution $\{X \blacklozenge t\}$;
  7) $f(t_1,...,t_n) = g(t_1',...,t_m')$, $f \neq g$ or $n \neq m$: fail;
  8) $f(t_1,...,t_n) = f(t_1',...,t_n')$, *f*/n ≠ *with*/2:
  return *unify*$((E \setminus \{e\}) \approx \{t_1 = t_1',...,t_n = t_n'\})$;
  9) $r = s$, where $r \int h$ *with* t_n *with...with* t_0 and $s \int k$ *with* t_m' *with...with* t_0',
  h, *k* terms with main functor ≠ *with*/2, using the notation $r \backslash t_i$
  (resp., $s \backslash t_i$) to denote the term obtained from *r* (resp. *s*) by taking away its i-th element t_i:
  if *h*, *k* are not the same variable then
  9.1) choose one from among the following actions:

a) return *unify*$((E \setminus \{e\}) \approx \{t_0 = t_0', r\backslash t_0 = s\backslash t_0'\})$
b) return *unify*$((E \setminus \{e\}) \approx \{t_0 = t_0', r = s\backslash t_0'\})$
c) return *unify*$((E \setminus \{e\}) \approx \{t_0 = t_0', r\backslash t_0 = s\})$
d) return *unify*$((E \setminus \{e\}) \approx \{r\backslash t_0 = N$ *with* t_0', N *with* $t_0 = s\backslash t_0'\})$,
N new variable;

else $h, k \in V, h \neq k \neq X$

9.2) select arbitrarily *i* from among 0,...,*m*;
choose one from among the following actions:
a) return *unify*$((E \setminus \{e\}) \approx \{t_0 = t_i', r\backslash t_0 = s\backslash t_i'\})$
b) return *unify*$((E \setminus \{e\}) \approx \{t_0 = t_i', r = s\backslash t_i'\})$
c) return *unify*$((E \setminus \{e\}) \approx \{t_0 = t_i', r\backslash t_0 = s\})$
d) return *unify*$((E \setminus \{e\}) \approx \{X = N$ *with* t_0,
N *with* t_n *with...with* $t_1 = N$ *with* t_m' *with...with* $t_0'\})$,
N new variable;

end.

The following theorems state the termination, correctness and completeness of *unify(E)* for any given system *E* of equations.

Theorem 4.1 (termination). **Let *E* be a Herbrand system. Then *unify(E)* terminates, provided a suitable strategy is adopted for selecting *e* across recursive levels.**

Theorem 4.2 (soundness and completeness). **Given a system *E*, let $E_1,...,E_n$ be all the systems in solved form produced by the unification algorithm. Then *Soln(E)* = $Soln(E_1)' \simeq ... \simeq Soln(E_n)'$, where *Soln(X)* is the set of all the ground T-unifiers of *X* and $Soln(E_i)'$ is $Soln(E_i)$ restricted to the variables of *E*.**

Let us briefly comment upon action 9 of the algorithm. Its aim is the reduction of set-set equations. In particular, cases (b), (c) and (d) take care of duplicates in the left-hand side term, duplicates in the right-hand side term and permutativity of the set constructor *with*, respectively.

As an example, let us consider the following system

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

The algorithm applies action 9.1 to it, requiring one of the following systems to be solved

a) $\{a=b, X=\{a|Y\}\}$
b) $\{a=b, \{a|X\}=\{a|Y\}\}$
c) $\{a=b, X=\{b,a|Y\}\}$
d) $\{X=\{b|N\},\{a|Y\}=\{a|N\}\}$.

The first three clearly have no solution, whereas system (d) can be further transformed by applying again action 9.1 to its second equation, which leads to the following new systems:

a) $\{X=\{b|N\}, Y=N\}$,
from which, by variable substitution, we get $\{X \blacklozenge \{b|Y\}\}$;
b) $\{X=\{b|N\}, \{a|Y\}=N\}$,
from which, by variable substitution, we get $\{X \blacklozenge \{b,a|Y\}\}$;
c) $\{X=\{b|N\}, Y=\{a|N\}\}$,
from which we get $\{X \blacklozenge \{b|N\}, Y \blacklozenge \{a|N\}\}$;
d) $\{X=\{b|N\}, Y=\{a|N'\}, N=\{a|N'\}\}$
from which, by variable substitution, we get $\{X \blacklozenge \{b,a|N'\}, Y \blacklozenge \{a|N'\}\}$.

The substitutions we have got constitute the set of unifiers for the initial system

we were looking for. Notice that this set is not *minimal* even though correct and complete. For instance, the fourth solution $\theta = \{X \blacklozenge \{b,a|N'\}, Y \blacklozenge \{a|N'\}\}$ can be obtained, apart from duplicates, from the second solution $\sigma = \{X \blacklozenge \{b,a|Y\}\}$ by applying the substitution $\{Y \blacklozenge \{a|N'\}$ to σ. 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 so as to detect cases in which not all the alternatives of the algorithm are required.

Equations of the form X *with* t_n *with...with* $t_0 = X$ *with* t_m' *with...with* t_0', where the two sides are set terms with the same variable tail element, are dealt with as a special case by action 9.2. The problem here is how permutativity of the *with* operator can be guaranteed. In fact, it is easy to check that by applying action 9.1, and in particular case (d), to an equation of this form (e.g. $\{a|X\} = \{b|X\}$) the algorithm might go into an infinite loop. The solution we have adopted is to avoid action 9.1.d of the general case by requiring the algorithm non-deterministically considers each element of one of the two sets involved in the given set-set equation, thus trying all possible combinations.

Of course, this solution opens a great (though finite) number of alternatives, generating many redundant solutions. An alternative formulation of action 9.2 which is less uniform with the rest of the algorithm but more efficient as for the number of generated solutions is shown below.

9.2) X *with* t_m *with ... with* $t_0 = X$ *with* t_n' *with...with* t_0', X variable:
return $unify((E \setminus \{e\}) \approx T)$, where the set T is generated as follows:
choose $I = \{(i_1,j_1),...,(i_p,j_p)\}$ from among the subsets of $\{0,...,m\} \infty \{0,...,n\}$
such that either I or $\{(j,i)|(i,j) \in I\}$ is a single-valued map;
T is $\{t_i = t_j' \mid (i,j) \in I\} \approx \{X = N \text{ with } r_1 \text{ with ... with } r_{m+n+2-p}\}$
where the r_ks are all terms t_i and t_j' such that $(\forall a \in \{0,...,n\}\ (i,a) \in I)$
and $(\forall b \in \{0,...,m\}\ (b,j) \in I)$ (the ordering of the r_ks is immaterial).

Finally, we want to remark that our unification algorithm is close in spirit to the algorithm proposed by Jayaraman and Plaisted in [9] but it manages to solve a larger number of cases. In particular the algorithm in [9] intentionally does not take into account duplicates in a set (e.g. the equation $\{a,a\} = \{a\}$ fails in that algorithm). Furthermore, as far as can be drawn from the presentation of the algorithm sketched in [9], that algorithm requires one of the terms in a set-set equation to be ground, and accordingly is not concerned at all with the situations dealt with by action 9.2 in our algorithm (e.g. $\{a|X\} = \{b|X\}$).

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 π and $\notin$ is the use of a simple constraint logic programming scheme [10]. In our context, an *atomic constraint* is any disequation of the form $t_1 \pi t_2$ or $t_1 \notin t_2$; a *constraint* is a conjunction of atomic constraints. We proceed here to illustrate the basic resolution step of our constraint handling method, based on the so-called *disequation analyzer Can* (a non-deterministic algorithm complementary, in a sense, to unification).

The main purpose of *Can* is to transform a given constraint into an equivalent set of constraints in canonical form. A constraint is in *canonical form* if all of the atomic constraints in it (if any) have the form:

a) $X \pi t$ and X does not occur in t (X variable, t term), or
b) $t \sqsupset X$ and X does not occur in t.

Given a constraint C, *Can* non-deterministically generates every element of an equivalent finite set $C_{Can} = \{<\Gamma_1,\theta_1>,...,<\Gamma_\delta,\theta_\delta>\}$ where each Γ_i is a constraint in canonical form, and θ_i is a substitution which keeps track of the bindings for the auxiliary variables created by the canonization process. We will describe the canonization algorithm in detail in section 5.2.

5.1 Extended SLD Resolution Procedure

Let P be a {log} program and G be a goal $:- C_1 \& ... \& C_n \& B_1 \& ... \& B_k$ (where the C_is, are atomic constraints and the B_js are {log} atoms). The goal G' is *derived* from G with substitution σ if the following conditions hold:

- if $k = 0$ then μ is the empty substitution ε and C' is $\emptyset$;
- if $k > 0$ then
  - B_i ($i \in \{1,...,k\}$) is the selected atom;
  - μ and C' are computed as follows:
    case B_i of
    a) $p(t_1,...,t_k)$, with p an ordinary predicate and there exists a clause $p(t_1',...,t_k') :- C_1' \& ... \& C_m' \& B_1' \& ... \& B_h'$ in P:
    μ is an mgu of the system $\{t_1 = t_1',...,t_k = t_k'\}$; C' is $\{C_1',...,C_m'\}$;
    b) $t = t'$: μ is an mgu of the system $\{t = t'\}$; C' is $\emptyset$;
    c) $t \sqsupset \{t_1,..,t_n|h\}$: μ is an mgu of one of the systems $\{t = t_1\},...,\{t = t_n\}$ or, alternatively, if h is a variable X not occurring in t, μ is the substitution $\{X \blacklozenge \{t|N\}\}$, N new variable; C' is $\emptyset$;
- $<\{D_1,...,D_d\},\theta>$ is one of the pairs generated by applying *Can* to $(\{C_1,...,C_n\} \approx C')^\mu$;
- G' is $:- D_1 \& ... \& D_d \& (B_0 \& ... \& B_{i-1} \& B_1' \& ... \& B_h' \& B_{i+1} \& ... \& B_k)^\sigma$, where σ is $\mu \circ \theta$.

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

Finally, a *computed answer* for a refutation $G, G_1,...,G_n$ of $P \approx \{G\}$ is a pair $<C,\sigma>$ where C is the constraint (in canonical form) occurring in G_n and σ is the substitution $\sigma_1 \circ \sigma_2 \circ ... \circ \sigma_n$ restricted to the variables of G where $\sigma_1, \sigma_2,...,\sigma_n$ are the substitutions generated at consecutive steps of the refutation.

It is interesting to notice that such a resolution algorithm involves four kinds of *non-deterministic* choice:

- which atom in the goal to select (don't care non-determinism);
- which clause in the program to select (don't know non-determinism);
- which mgu in the set of unifiers computed by the unification algorithm to select (don't know non-determinism)
- if the selected atom has the form $t \sqsupset \{t_1,...,t_n|X\}$, which one of the $n+1$ systems $\{t = t_1\},...,\{t = t_n\}, \{X = \{t|N\}\}$ to solve first (don't know non-determinism).

5.2 Constraint canonization algorithm

In this section we describe a non-deterministic algorithm which is able to compute, for any given constraint C, the corresponding set of constraints in canonical form C_{Can}. The algorithm starts with the pair $<C,\varepsilon>$ and generates, through non-determinism, each element of C_{Can}. Notice that a constraint is represented here as a set of atomic constraints; in particular, Ø is a constraint in canonical form.

CONSTRAINT CANONIZATION ALGORITHM

Let t, t', t_i, t_i' be first order terms, X, Y be generic variables (i.e. $X, Y \in V$), and r, s be set terms.

function *Can*(<*C*: constraint, σ: substitution>): <constraint,substitution>;
begin
  if C is in canonical form
  then return $<C,\sigma>$
  else select arbitrarily an atomic constraint c in C;
    case c of
    1) $t \in s$ *with* t': return *Can*$(<(C \setminus \{c\}) \approx \{t \pi t', t \in s\}, \sigma>)$;
    2) $t \in f(t_1,..,t_n), f \pi$ *with*: return *Can*$(<(C \setminus \{c\}), \sigma>)$;
    3) $t \in X$, X variable, and X occurs in t: return *Can*$(<(C \setminus \{c\}), \sigma>)$;
    4) $f(t_1,...,t_n) \pi g(t_1',...,t_m'), f \pi g$: return *Can*$(<(C \setminus \{c\}), \sigma>)$;
    5) $f(t_0,...,t_n) \pi f(t_0',...,t_n'), f \pi$ *with*: choose i from among $0,...,n$;
       return *Can*$(<(C \setminus \{c\}) \approx \{t_i \pi t_i'\}, \sigma>)$;
    6) $f \pi f$ or $X \pi X$, X variable: fail;
    7) $t \pi X$ and t is not a variable: return *Can*$(<(C \setminus \{c\}) \approx \{X \pi t\}, \sigma>)$;
    8) $X \pi t$, t is not a set term and X occurs in t, or t is
       h *with* t_n...*with* t_1, h kernel or variable, and X occurs in t_1 or...or t_n:
       return *Can*$(< (C \setminus \{c\}), \sigma >)$;
    9) $X \pi X$ *with* t_n...*with* t_0: choose i from among $0,...,n$;
       return *Can*$(<(C \setminus \{c\}) \approx \{t_i \in X\}, \sigma >)$;
    10) r *with* $t \pi s$ *with* t': choose one from among the following actions:
       a) δ is *member_solve*$(Z \in r$ *with* $t)$;
          return *Can*$(<(C \setminus \{c\}) \approx \{Z \in s$ *with* $t'\})^{\delta}, \delta \circ \sigma >)$, Z new variable;
       b) δ is *member_solve*$(Z \in s$ *with* $t')$;
          return *Can*$(<(C \setminus \{c\}) \approx \{Z \in r$ *with* $t\})^{\delta}, \delta \circ \sigma>)$, Z new variable;

where *member_solve* is a function which is able to solve set membership atoms (in the same way they are solved within the extended resolution procedure presented above)

function *member_solve*(*M*: membership atom): substitution;
begin
  case M of
    1) $Z \in s$ *with* t: choose one from among the following actions:
         a) return $\{Z \blacklozenge t\}$;
         b) return *member_solve*$(Z \in s)$;
    2) $Z \in X$, X variable:
         return $\{X \blacklozenge N$ *with* $Z\}$, N new variable;
    3) $Z \in f(t_1,..,t_n), f \pi$ *with*: fail
end.

Let us see how the {log} resolution procedure works on a simple example involving also a negative answer.

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

Given 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 above) the two alternative substitutions *{X♦1}*, *{X♦2}*; then *Can* is applied non-deterministically to either the constraint *{1 ∈ {1,3}}* or *{2 ∈ {1,3}}*. The first generates (action 1 of *Can*) the new constraint *{1 ≠ 1, 1 ∈ {3}}* that clearly fails; the second generates the constraint *{2 ≠ 1, 2 ∈ {3}}* from which, after few iterations of *Can*, we get the pair $<\emptyset,\varepsilon>$. 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 following substitution *{Set1♦{X|N}}*, whereas applying *Can* to the constraint *{X ∈ {1,3}}* generates the pair $<\{X \neq 1, X \neq 3\},\varepsilon>$. So the final computed answer is

```
-->   Set1 = {X|N}, X π 1, X π 3.
```

The following theorems state the termination, correctness and completeness of the canonization algorithm for any given constraint *C* (formal proofs of these theorems and of those of the following section can be found in [7]).

Theorem 5.1 (termination). Let *C* be a constraint; then *Can*(*C*) always terminates.

Theorem 5.2 (soundness). Let *C* be a constraint such that
$C_{Can} = \{<C_1,\theta_1>,...,<C_n,\theta_n>\}$. Then
1) if $n = 1$, $C_1 = C$, and θ_1 is empty, then *C* admits a solution;
2) if θ is a solution of C_i ($i \in \{1,...,n\}$) then θ is a solution of C^{θ_i}.

Theorem 5.3 (completeness). Let *C* be a constraint such that $C_{Can} = \{<C_1,\theta_1>,...,<C_n,\theta_n>\}$. If θ is a solution of *C* then there exists an *i* among *1* and *n* and a solution σ of such 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 1 of theorem 5.2. In consequence of this fact, one has:

Theorem 5.4. The axiomatic set theory specified in Section 3.1 makes it possible to prove either *C* or ¬*C* for any constraint *C*.

5.3 Soundness and completeness of the resolution procedure

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 $<C,\theta>$ and σ is a solution for *C* then we have $P \models_{Set} G^{\theta\circ\sigma}$.

Lemma 5.1 (lifting lemma for {log}). Let *P* be a {log} program, *G* be a non empty conjunction of {log} atoms, *C* be a constraint, and σ be a substitution for the variables in *(C,G)*. If *:- (C & G)*$^\sigma$ does have a refutation in *P* with computed answer $<C',\theta'>$ then *:- (C & G)* has a refutation in *P* with computed answer $<C'',\theta''>$ such that

(i) θ'' is more general than $\sigma \circ \theta'$ and
(ii) for each α solution of C' there exists β such that $\beta \circ (\alpha|_{vars(C)})$ is a solution of C''.

Lemma 5.2. Let B be an atom of the form
(i) $t = s$ or (ii) $t \sqsubseteq s$.
If $\models_{Set} B^{\sigma}$ for some σ ground then there exists a refutation for $\{:\text{-}\ B^{\sigma}\}$ in any program P.

Lemma 5.3. Let $Succ(P) = \{\tau(a) \mid Pred(a) \in \{=,\sqsubseteq,\pi,\sqsubset\}$ *and there exists a refutation for* $P \approx \{:\text{-}\ a\}\}$. Then $Succ(P) = M_P$, for each {log} program P.

Lemma 5.4. Let P be a {log} program, $G = (:\text{-}\ C\ \&\ L_1\ \&\ \ldots\ \&\ L_h)$ be a goal such that $P \approx \{\sqsubseteq C \nearrow \sqsubseteq L_1 \nearrow \ldots \nearrow \sqsubseteq L_h\}$ is unsatisfiable in the set theory; then there is a refutation for G in P.

Theorem 5.6 (Completeness). Let P be a {log} program, G be a goal and $P \models_{Set} (G^{\sigma})^{\forall}$; then G has a refutation in P.

6 Restricted Universal Quantifiers

Restricted Universal Quantifiers (*RUQ*s) are formulas of the form $(\forall X \sqsubseteq s)\ F$, F formula, which stand for the quantified implication $\forall X\ ((X \sqsubseteq s) \varnothing F)$. Usefulness of providing RUQs as part of the representation language has been demonstrated by several authors (e.g. [4,11]). 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 so as to encompass RUQs.

An *extended Horn clause* is a formula:

$$p(t_1,\ldots,t_n) :\text{-}\ B_1\ \&\ \ldots\ \&\ B_n$$

where each B_i can either be an atom or an RUQ formula of the form $(\forall X_1 \sqsubseteq t_1)\ldots(\forall X_n \sqsubseteq t_n)G$, G atom, satisfying the following properties:

- the variables $X_1,\ldots,X_n$ can occur only in G;
- if $i\ \pi\ j$ then $X_i\ \pi\ X_j$.

These two restrictions ensure that a B_i of the form $(\forall X_1 \sqsubseteq t_1)\ldots(\forall X_n \sqsubseteq t_n)G$ is logically equivalent to $\forall X_1 \ldots \forall X_n (X_1 \sqsubseteq t_1\ \&\ldots\&\ X_n \sqsubseteq t_n \varnothing G)$. (Note that the first restriction, motivated (cf. [20]) by our set finiteness requirement, is implicitly present in [11] since nesting of sets is not allowed there.)

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

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

(B) disj(D,S1,S2) :-
                    (∀Z ∈ D)(Z ∈ S2 & Z ∉ S1)
```

where the second clause is intended to state that D is a subset of S1 \ S2.

One might proceed as in [11], by enhancing resolution so as 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 [12]). 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 &...& B_k & B_{k+1} &...& B_n be an extended Horn clause, where $B_1,...,B_k$ ($k \leq n$) are {log} formulas and $B_{k+1},...,B_n$ are formulas containing RUQs.

1) Replace C by the set of clauses

$$I = \{H \text{ :- } B_1 \,\&...\&\, B_k \,\&\, D_1 \,\&...\&\, D_{n-k},$$
$$D_1 \text{ :-} B_{k+1},$$
$$...,$$
$$D_{n-k} \text{ :- } B_n\}$$

where each D_j is obtained by taking a new predicate symbol (different from all the others in the program) and applying it to all variables in B_{k+j} which are not quantified by an RUQ of B_{k+j}.

2) Replace each element of I of the form

$$p(t_1,...,t_n) \text{ :- } (\forall X_1 \in s_1)(\forall X_2 \in s_2)\, G$$

by the two clauses:

$$p(t_1,...,t_n) \text{ :- } (\forall X_1 \in s_1)\, r(Y_1,...,Y_k)$$
$$r(Y_1,...,Y_k) \text{ :- } (\forall X_2 \in s_2)\, G,$$

where $Y_1,...,Y_k$ are all variables (different from X_2) occurring in s_2 or free in G, and r is a new predicate symbol; repeat this step as long as possible.

3) Replace each extended Horn clauses of the form

$$p(t_1,...,t_n) \text{ :- } (\forall X \in \{t_1',...,t_m'|h\})G.$$

by

$$p(t_1,...,t_n) \text{ :- } G^{\{X \blacklozenge t_1'\}} \,\&\, ... \,\&\, G^{\{X \blacklozenge t_m'\}}$$

if h is a term whose main functor differs from *with*/2, or by

$$p(t_1,...,t_n) \text{ :- } G^{\{X \blacklozenge t_1'\}} \,\&\, ... \,\&\, G^{\{X \blacklozenge t_m'\}} \,\&\, D$$
$$D \text{ :- } (\forall X \in h)G$$

if h is a variable, where D is put together in the same way as in step 1).

4) Replace each simple extended Horn clause

$$p(t_1,...,t_n) \text{ :- } (\forall X \in Y)G[X,Z_1,...,Z_m],$$

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

$$p(t_1,...,t_n) \text{ :- } r(Y,Z_1,...,Z_m)$$
$$r(K,Z_1,...,Z_m) \text{ :- } K \neq \_ \text{ with } \_ \qquad \text{(i.e., } K \text{ is a set kernel)}$$
$$r(\{A \mid R\},Z_1,...,Z_m) \text{ :-}$$
$$(A \in R) \,\&\, G^{\{X \blacklozenge A\}} \,\&\, r(R,Z_1,...,Z_k),$$

with r a new predicate symbol.

For example, the extended Horn clause (A) for the subset operation given at the beginning of the section gets transformed into the equivalent three {log} clauses (action 4 of the algorithm is applied):

```
subset(S1,S2) :- r(S1,S2)

r(S1,_) :- S1 ≠ _ with _
r({A|R},S2) :-
          (A ∈ R) & (A ∈ S2) & r(R,S2).
```

Similarly, clause (B) for the predicate *disj*/3 gets transformed into:

```
disj(D,S1,S2) :- r(D,S1,S2)
```

```
r(K,S1,S2) :- K ≠ _ with _
r({A|R},S1,S2):-
                (A ∉ S1) & (A ∉ S2) & (A ∉ R) &
                r(R,S1,S2).
```

Note that it is impossible to implement RUQs in a usable manner without resorting to ≠ and ∉. In particular, the constraint $A \notin R$ in the last step of the algorithm is needed to avoid that the program may loop forever trying to generate a set with infinite many occurrences of the same element (e.g. *{1,1,1,...}* instead of *{1}*).

In conclusion, RUQs are introduced in {log} only at a syntactic level, as a convenient notation, with no extension at the semantic level. The same approach can be adopted to introduce intensional set formers and the *setof* predicates by which such abstractions are implemented (see [6]). However, it turns out that in this case the language must be extended so as to provide either a built-in set collection mechanism (see [3]) or some form of negation in goals and clause bodies.

7 Related 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.

Among the proposals that have addressed this problem with a similar approach we briefly recall [3], [11] and [23].

The first paper defines LDL, a logic based language oriented to the manipulation of deductive databases. The main differences between LDL and {log} are:

- the procedural semantics: bottom-up in LDL, top-down (with set unification) in {log};
- in LDL the set manipulators (union, intersection etc.) are built-in predicates, while in {log} they are programmer-defined;
- the 'collector capability' is expressed in different ways: via set-grouping in LDL, using the *with* + negation combination in {log}.

It is interesting to note that the syntactic restrictions enforced in LDL are very similar to those necessary to introduce classical negation by failure in {log} (see [7]).

Kuper's proposal [11] basically consists in extending logic programming with RUQs (see section 6 above). Kuper shows the usefulness of this extension, but does not offer a full-blown semantics for the language.

Quite interesting is a comparison between our proposal and Sigal's work [23] which outlines, from a theoretical point of view, a complete logic language with sets, where set-theoretic operations are built-in. A model-theoretic semantics is developed for a subset of this language which bears some resemblance to {log}. Sigal's proposal copes with the rather intriguing task of manipulating infinite sets (even repeatedly nested one inside another). Although theoretically appealing, this approach leads to difficulties that are hard to surmount: whence the lack of a realistic procedural semantics for the proposed language.

8 Future developments of {log}

The interpreted operator *ker*, which helped us in keeping the form of our set axioms simple (cf. section 3.1), has not been included among the symbols of {log}. Adding *ker* to the language would be useful, enabling one e.g. to define the *subset* predicate as

```
subset(S1,S2) :-
        ker(S1) = ker(S2) &
        (∀X ∈ S1)(X ∈ S2),
```

instead of in the less satisfactory way seen in section 6. Similarly one could define the element removal operation by the clause

```
less(A,X,B) :-
        ker(A) = ker(B) &
        (∀Y ∈ B)(Y ∈ A & Y ≠ X) &
        (∀Y ∈ A)(Y = X ∨ Y ∈ B).
```

This extension calls for a modification of the unification algorithm which ought to produce along with each substitution a conjunction of atomic constraints of the new form *ker(X) = t*. Our extended SLD resolution procedure ought to be modified too, to take into proper charge the new kind of constraints.

We have already announced a couple of times in this paper that {log} will be extended with intensional set formers. This extension poses problems of various kinds, one of the most obvious being that infinite sets can easily be described by means of intensional set formers, e.g.

```
natS({X : nat(X)})

nat(0)
nat(succ(X)) :- nat(X).
```

Suitable criteria could be adopted for rejecting expressions (such as `{X:nat(X)}` in the example) that denote (or might denote) infinite sets. Unfortunately such criteria must be, out of necessity, very conservative, and prone to refuse useful set formers together with dangerous ones. Alternatively infinite sets could be accepted as part of the language. In this case, however, a coherent attitude should be taken to deal with infinite sets at all levels: by having an explicit infinity axiom, by accordingly enriching the privileged interpretation, by enhancing the procedural semantics, etc. (cf. [23]).

We also envisage generalizations of {log} for the treatment of non-standard sets: for instance, non-well-founded sets (cf. [1]) among which membership can form cycles of all kinds.

Acknowledgments

This work originates from a project, named AXL, funded by ENI and ENIDATA. Partial support came from MURST 60%. We have enjoyed useful discussions with Alberto Policriti, in particular concerning the axiomatization in section 3 and the proof of termination of our unification algorithm.

References

1. P.Aczel. Non-well-founded sets. Vol.14, Lecture Notes, Center for the study of Language and Information, Stanford, 1988.
2. A.Aho, J.Hopcroft, J.Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1975.
3. C.Beeri, S.Naqvi et al. Set and negation in a Logic Database Language (LDL1). *Proceedings 6th ACM SIGMOD Symposium*, 1987.
4. D.Cantone, A.Ferro, E.G.Omodeo. *Computable set Theory*. Oxford University

Press, International Series of Monographs on Computer Science, 1989.
5. E.E.Doberkat, D.Fox. *Software Prototyping mit SETL*. B.G.Teubner Stuttgart, 1989.
6. A.Dovier, E.G.Omodeo, E.Pontelli, G.F.Rossi. {log}: A Logic Programming Language with Finite Sets, in *Logic Programming: Proceedings of the Eighth International Conference* (K.Furukawa, ed.), The MIT Press, 1991.
7. A.Dovier, E.G.Omodeo, E.Pontelli, G.F.Rossi. {log}: A Language for Programming in Logic with Finite Sets, Research Report, in preparation.
8. H.B.Enderton. *A mathematical introduction to logic*. Academic Press, 2nd printing, 1973.
9. B.Jayaraman, D.A.Plaisted. Programming with Equations, Subsets and Relations. *Proceedings of NACLP89*, Cleveland, 1989.
10. J.Jaffar, J.L.Lassez. From Unification to Constraints. *Proceedings Fifth Conference on Logic Programming*, Tokyo, 1987.
11. G.M.Kuper. Logic Programming with Sets. *Proceedings 6th ACM SIGMOD Symposium*, 1987.
12. G.M.Kuper. On the Expressive Power of Logic Programming with Sets. *Proceedings 7th ACM SIGMOD Symposium*, 1988.
13. B.Legeard, E.Legros. CLPS: A Set Constraints Logic Programming Language. Research Report, Laboratoire d'Automatique de Besançon, Institut de Productique, Besançon, France, Feb. 1991.
14. J.W.Lloyd. *Foundations of logic programming*. Springer Verlag, 2nd edition, 1987.
15. J.L.Lassez, M.J.Maher, K.Marriot. Unification revisited. *Lecture Notes in Computer Science*, Vol. 306, Springer Verlag, 1986.
16. A.Martelli, U.Montanari. An efficient unification algorithm. *ACM TOPLAS*, 4, April 1982.
17. A.Martelli, C.Moiso, G.F.Rossi. Lazy Unification Algorithms for Canonical Rewrite Systems, in *Resolution of Equations in Algebraic Structures,* vol II (H.Ait-Kaci and M.Nivat, eds), Academic Press, 1989.
18. M.Naftalin. An experiment in practical semantics. *ESOP 86 - Lecture Notes in Computer Science*, Vol. 213, Springer Verlag, 1986.
19. F.Parlamento, A.Policriti. Decision procedures for elementary sublanguages of set theory. IX. Unsolvability of the decision problem for a restricted subclass of Δ_0-formulas in set theory. *Comm. of Pure and Applied Mathematics*, 41, 1988.
20. F.Parlamento, A.Policriti. Expressing infinity without foundation. *Journal of Symbolic Logic*, 56(3), 1991.
21. J.C.Shepherdson. Negation in Logic Programming. In *Foundations of deductive databases and Logic Programming* (J.Minker, ed). Morgan Kaufmann, Los Altos, CA, 1987.
22. J.H.Siekmann. Unification Theory. *Journal of Symbolic Computation*, 7, 1989.
23. R.Sigal. Desiderata for Logic Programming with sets. *Proceedings GULP89: Fourth National Conference on Logic Programming*, Bologna, 1989.
24. J.T.Schwartz, R.B.K.Dewar, E.Dubinsky, E.Schonberg. *Programming with sets, an introduction to SETL*. Springer-Verlag, 1986.
25. D.Turner. An overview of Miranda. *SIGPLAN Notices*, Vol.21, n.12, 1986.
26. *Z handbook*, Oxford University Computing Laboratory, Oxford 1986.