{log}: A Logic Programming Language with Finite Sets

Agostino Dovier, **Eugenio G. Omodeo**[+],
Enrico Pontelli, **Gianfranco Rossi**
Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via Zanon 6, 33100 Udine (Italy)

Abstract

An extended logic programming language embodying sets is developed in successive stages, introducing at each stage simple set dictions and operations, and discussing their operational as well as declarative semantics. First, by means of special set terms added to definite Horn Clause logic, one is enabled to define enumerated sets. A new unification algorithm which can cope with set terms is developed and proved to terminate. Moreover, distinguished predicates representing set membership and equality are added to the base language along with their negative counterparts, and SLD resolution is modified accordingly. It is shown that the resulting language allows restricted universal quantifiers in goals and clause bodies to be defined quite simply within the language itself. Finally, abstraction set terms are made available as intensional designations of sets. It is shown that also such terms become directly definable within the language, provided the latter is endowed with negation, which may occur in goals and clause bodies.

Introduction

General agreement exists about the usefulness of set abstractions in **specification languages** (e.g. the Z language [Z86]) and as high-level representations of complex data structures [AHU75]. However, only relatively few real (i.e. executable) programming languages embody sets as primitive objects in a satisfactory way. A plausible reason for this is the difficulty to find out a single representation of sets which supports efficiently as required by practical applications the most common operations on sets. One exception on the *procedural programming* side is the language SETL [SD*86] which has demonstrated to be a flexible rapid prototyping tool for many applications [DF89]. Other notable efforts along these lines but in the framework of *functional programming* languages are MIRANDA [Tur86] (in which sets are defined via the so called ZF-expressions) and ME TOO [Naf86].

However, sets seem to fit better in the framework of a *declarative style* of programming, since the latter is recognized as a most adequate style for rapid prototyping. It is well known that the *logic programming* paradigm well supports this programming style (and, as such, it is often used as a prototyping language); therefore, introducing sets

---

[+] Current address: Univ. di Roma *La Sapienza*, DIS, Via Salaria, 113, 00198 ROMA.

in logic programming looks quite promising.

Actually, the two built-in predicates `setof` and `bagof` of Prolog provide a somewhat rudimentary support for sets; however these predicates have no precise logical semantics at all, and no specific support (such as ad hoc representations of sets, basic operations on sets, etc.) is provided by the language.

Our aim is, on the contrary, to define an extended logic programming language (called {log}) which supplies special **set former terms** as part of the language, and which allows basic operations on sets, including the `setof` predicate, to be defined within the language itself, so that a clear **logical semantics** can be given to them. Works along these lines are also reported in [BN*87] and [Kup87] (see Sect. 4).

The starting point of our work is a **pure** logic programming language, that is *Definite Horn Clauses* with no extra-logical constructs. Then, in successive steps, we add to this language simple set constructs (namely, enumerated set terms, set membership and equality and their negative counterparts) and we show that they have a well-defined **operational** and **declarative semantics**. Usual operations on sets, such as union, intersection, etc. can be defined in this extended language. Moreover we introduce **restricted universal quantifiers** and show that they can be defined quite simply within the language. The last step is the introduction of **negation** in clause bodies. It is then shown how **intensional set formers** can be interpreted in this final language. Notice that unlike [Sig89] we consistently restrain our investigation to finite sets only.

The paper is organized as follows. *Section 1* introduces **set terms** and the predefined predicates $\in$ and = to represent enumerated sets and the set membership and equality operations, respectively. Declarative and operational semantics of the resulting language (called {log}) are defined. This presupposes, among others, the design of a **new unification algorithm** which can deal with set terms. *Section 2* enhances $\{\log_1\}$ with the predicates $\notin$ and $\neq$ to represent **set non-membership** and **inequality**. In this language, called $\{\log_2\}$, are then introduced **restricted universal quantifiers**. *Section 3* defines $\{\log_3\}$, the extension of $\{\log_2\}$ with **negation**. In $\{\log_3\}$ are then introduced **intensional set formers** and it is shown how our language supports this new syntactic feature. Finally, a comparison with some other related proposals is carried out in *Section 4*.

Due to space limitations we omit formal proofs; in particular we skip the various soundness and completeness theorems holding for our extended resolutions methods (sections 1.4, 2.1 and 3.2) as well as all model-theoretic and fixpoint results. All of these, together with a description of the Prolog implementation of {log}, are detailed in [DP90] and in a forthcoming research report.

1 $\{\log_1\}$

1.1 Syntax of $\{\log_1\}$

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

- an interpreted constant, {}, for the empty set;
- a binary function symbol, set_cons, to be interpreted as follows: set_cons(t,s) stands for the set that results from adding t as a new element to the set s.

Apart from these two symbols, our first language, $\{\log_1\}$, contains the usual equipment of clausal Horn logic (cf. [Llo87]) and two infix predicates, $\in$ and =, endowed with their *usual* set-theoretic meaning. The extensional notation 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;
- set_cons(t,s) belongs to G for any ground term t, when s 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. If t is a set term, this is witnessed by the particular substitution σ that replaces every variable in t by {}.

Using the syntactic conveniences $\{t_1,...,t_n|s\}$ for set_cons(t_1,set_cons(t_2,..., set_cons(t_n,s)) and $\{t_1,...,t_n\}$ for set_cons(t_1,set_cons(t_2,...,set_cons(t_n,{})), where $t_1,...,t_n$ are arbitrary terms and s is a set term, we have, for instance:

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

For the rest of this paper, we will freely exploit the syntactic features available in Edinburgh Prolog in addition to the constructs discussed above. Two sample $\{\log_1\}$ clauses are:

- `q(X) :- X∈{a,b,c,d} & p(X)`
- `singleton(X) :- X = {Y}.`

1.2 Declarative semantics of $\{\log_1\}$

To conveniently interpret the language $\{\log_1\}$ introduced so far, a richer semantical structure than the one ordinarily associated with definite Horn Clauses is needed. Basically the focal point of this extension - which will now be discussed - is the interpretation of the functional symbol set_cons, which must be interpreted as a set constructor.

Since set_cons is assumed, in our intended interpretation, to fulfill the identities

- set_cons(t_1,set_cons(t_2,s)) = set_cons(t_2,set_cons(t_1,s)),
- set_cons(t_1,set_cons(t_1,s)) = set_cons(t_1,s),

the classical definitions of **Herbrand universe** and **interpretation** (see [Llo87] for instance) need some recasting.

We might proceed to consider the coarsest equivalence relation $\equiv$ over an ordinary Herbrand universe H (generated, as usual, by a collection F of functors containing at least one constant) which satisfies these two assumptions (with $\equiv$ in place of =), and then define $U_H = H / \equiv$ to be the desired domain. As a taste matter, however, we prefer our enriched Herbrand universe U_H to be constructed as

$$U_H = \bigcup_{i \geq 1} U_i$$

where the U_i s form the following hierarchy:

- U_0 = {ground terms (in H) where neither { } nor set_cons occurs}
- $U_{i+1} = U_i \cup P_{fin}(U_i) \cup \{f(T_H(t_1),..,T_H(t_n)): f \in F \setminus \{set\_cons\}$ and $t_1,...,t_n \in U_i\}$

where $P_{fin}(U_i)$ is the set of all finite subsets of U_i.

The construction of our modified *Herbrand term interpretation function* T_H is interleaved with this definition of the U_is so as to fulfill:

$T_H(c) = c$ if c is a constant, $c \neq \{\}$;
$T_H(\{\}) = \varnothing$;
$T_H(f(t_1,..,t_n)) = f(T_H(t_1),..,T_H(t_n))$;
$T_H(set\_cons(t,s)) = \{T_H(t)\} \cup T_H(s)$.

It is worthwhile observing that since the last of these identities yields no value for $T_H(set\_cons(t,s))$ unless s is a ground set term, T_H is only partially defined over U_H.[1]

With the above definitions, the usual results regarding model-theoretic semantics in the standard case (namely, existence of an Herbrand model, model intersection property, least Herbrand model, equivalence between the least Herbrand model and logical consequences) can be proved to be still valid in the case of our language. As usual, given a program P, it is possible to define also the **immediate consequence operator** T_P:

$T_P(I) = \{p(T_H(t_1),...,T_H(t_n)): t_1,...,t_n$ are ground terms and there exists a ground instance $p(t_1,..,t_n):-L_1\&...\&L_k$ of a clause in P so that $I \models L_i$ for each i, $1 \leq i \leq k\}$,

for every interpretation I of the language. The deviation from the standard definition of T_P here is the presence of T_H (the Herbrand term interpretation function), which allows us to abstract from the syntactic representation of sets.

It is then possible to prove the main result regarding the fixpoint semantics, which expresses - as usual - the equivalence between the least fixpoint of T_P and the least Herbrand model**.**

1.3 Unification

The development of a procedural semantics for $\{log_1\}$ requires first that the unification algorithm is refined in order to deal with sets, and second 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 assume the definitions (Herbrand system, substitution, solution, etc.) given in [LMM86].

First of all it is interesting to point out a new situation created by sets: the inherent lack of order inside a set causes the decay of the main result about standard unification, that is the 'uniqueness' of the most general unifier. This is clear from the following

[1] En passant, we notice that an entirely analogous interpretation universe based on a domain D of constants instead of on H, could be defined by replacing the base induction clause by $U_0 = D$ and by replacing the inductive step of the definition by $U_{i+1} = U_i \cup P_{fin}(U_i)$. We will not explicitly resort to structures of this kind in this paper, although they lie in the background.

example: consider the singleton Herbrand system E = {{X,Y} = {1,2}}; there are only two solutions, namely $\sigma_1 = \{X \rightarrow 1, Y \rightarrow 2\}$ and $\sigma_2 = \{X \rightarrow 2, Y \rightarrow 1\}$, neither of which is more general than the other.

In this connection, the following definition, referring to a Herbrand system E and a set Γ of idempotent substitutions on the variables of E, will prove helpful. Γ is said to be a **minimal exaustive set of mgus** if:

- the unifiers of E are precisely all substitutions of the form $\mu \circ \sigma$ with $\mu \in \Gamma$;
- no $\mu \in \Gamma$ is more general than any other unifier $\sigma \in \Gamma$.

A Γ with these properties can be computed, for any given E, via the non-deterministic algorithm below. The latter is indeed able to produce through suitable choices, any $\mu \in \Gamma$ (harmless additional unifiers could also be produced, as we will see). It is convened here that each idempotent substitution be represented as a Herbrand system E* in solved form.

SET UNIFICATION ALGORITHM

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;
case e **of**
1) X = X: **return** unify(E \ {e});
2) t = X, t is not a variable: **return** unify({X = t} ∪ (E \ {e})));
3) X = t, t is not a set term and X occurs in t: **fail**;
4) X = {t_0,...,t_n|Y}, X ≠ Y and X occurs in t_0 or in t_1 or ... or in t_n: **fail**;
5) X = {t_0,...,t_n|X} and X does not occur in t_0,...,t_n:
return unify({X = {t_0,...,t_n|N}} ∪ (E \ {e})σ) where σ is the substitution {X → {t_0,...,t_n|N}} and N is a new variable;
6) X = f(t_1,...,t_n), f ≠ set_cons and X is marked as a set-variable (see below for an explanation): **fail**;
7) X = t, X does not occur in t , X occurs somewhere else in E and case 6) does not apply:
return unify({X=t} ∪ (E \ {e})σ) where σ is the substitution {X → t};
8) f(t_1,...,t_m) = g(s_1,...,s_m), f ≠ g: **fail**;
9) f(t_1,...,t_n) = f(s_1,...,s_m), f ≠ set_cons:
return unify({$t_1 = s_1$,...,$t_n = s_n$} ∪ (E \ {e}));
10) {t_0,...,t_n|R} = {s_0,...,s_k|S}:
set_expansion(e,Leaves);
select a system E' in Leaves;
return unify(E'' ∪ ((E' \ E'') ∪ (E \ {e}))σ), where E'' is the set of equations for R and/or S and σ is the corresponding substitution
end.

Aim of the function set_expansion is the reduction of set-set equations. This function

produces a *set* of Herbrand systems, each representing one of the several ways in which the elements of the two given sets can be matched. We will not provide here a full description of this function, due to space limitations; just the flavor of it will be conveyed through a quick informal presentation (see [DP90] for details).

Given the set-set equation $\{t_0,...,t_n|R\} = \{s_0,...,s_k|S\}$, for example $\{A,f(X)|R\} = \{f(1)|S\}$, a certain number of parallel computations are opened by the set_expansion function (actually a tree of systems is generated; the leaves of the tree are returned as the result of the set_expansion function). The set-set reductions performed by these computations can be classified into five classes:

- **a)** t_0 is compared with one of the elements of the second set and the rest of the first set is compared with the rest of the second one; in the example, A = f(1) and {f(X)|R} = S;
- **b)** same as a), but the rest of the first set is compared with the entire second set; in the example, A = f(1) and {f(X)|R} = {f(1)|S};
- **c)** same as a), but the whole first set is compared with the rest of the second set; in the example, A = f(1) and {A,f(X)|R} = S;
- **d)** t_0 is supposed to belong to S, so S becomes $\{t_0|N\}$, with N a new variable, and the rest of the first set is compared with the rest of the second set; in the example, S = {A|N} and {f(X)|R} = {f(1)|N};
- **e)** same as d) but the rest of the first set is compared with the whole second set; in the example, S = {A|N} and {f(X)|R} = {f(1),A|N}.

In the last two cases, if R and S are the same variable, the reduction step must be modified by substituting R with N in the first set term.

Then, in each computation, the set-set reduction is repeatedly applied until a system with no set-set equations in it is obtained; from here on it is easy to compute an appropriate substitution for the variables R and S. In the example, applying a reduction of type d) we get:

{S = {A|N}, {f(X)|R} = {f(1)|N}}.

Then, by applying a reduction of type a) to the second equation we get:

{S = {A|N}, f(X) = f(1), R = N}.

This system of equations is one of the systems returned by set_expansion.

A remark on the set variable marking mentioned in rule 6 of the set unification algorithm. Variables which appear as the tail of a set can only be instantiated as set terms (for this reason, we call them **set variables**). Set variables could be distinguished from other variables by associating to them a (static) type specification (perhaps implicit in a special naming convention syntax). However we have preferred not to burden the language with such a type specification, assuming that set variables can easily be identified (and marked) at run time. In particular we assume that before the function unify is called all the variables appearing as the rest of a set in the system E are marked as set variables; moreover, any new variable created during the unification process is marked in the same way.

It has been proved (see [DP90]) that for any given system E of equations, unify(E) terminates. Furthermore, the set of substitutions computed by the unification algorithm is exhaustive and correct even though possibly not minimal. However, the number of redundancies (i.e. substitutions that are less general than other unifiers of E) is in any case

finite and the output of the algorithm could be 'filtered' in order to obtain the desired result. We have also proved that the problem of deciding whether two set terms are unifiable is NP-complete. This ensues from a reduction of 3-SAT to the the problem at hand.

An algorithm for unifying set expressions is presented in [JP89]. However as the algorithm is only sketched there, not all cases appear to have been covered equally well. For instance, given the equation {a|X} = {b|X} the algorithm in [JP89] does not terminate. Our understanding, based on [JN84], is that the purpose there was simply to unify two terms one of which was supposed to be ground.

1.4 The resolution procedure

The resolution procedure developed for $\{log_1\}$ is an extension of the usual SLD resolution, 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 as predicates with a pre-assigned meaning.

a) An equality **t = s** is solved by unifying the two terms by the new unification algorithm.
b) A membership literal **t ∈ s** is transformed into a set of equations in accordance with the following algorithm:
    1) if s is not a set term or s = {} then FAIL;
    2) if s is a variable X then replace t ∈ s by s = {t |R} where R is a new variable;
    3) if s is of the form $\{t_1,...,t_n|R\}$ then open n+1 new computations replacing t ∈ s by the following n+1 equations: $t = t_1,...,t = t_n,R = \{t|S\}$, where S is a new variable.

2 $\{log_2\}$

The expressive power of the language $\{log_1\}$ is not high enough yet as to represent many common situations in a truly declarative way, a major obstacle being the impossibility to use negative information in defining sets, in order - say - to specify that an item does not belong to a set. To make an example, if `disj(D,S1,S2)` is to state that D is a subset of S1 \ S2, the straightforward definition

```
disj({},X,Y)
disj({A|R},X,Y):- (A∈X) & (A∉Y) & (A ∉R) & disj(R,X,Y)
```

is not viable in $\{log_1\}$.[1]

Let us therefore proceed to extending $\{log_1\}$ into $\{log_2\}$ by addition of the new relators ≠ and ∉. From the semantical point of view, it is just necessary to modify the notion of interpretation in order to constrain ∉ and ≠ to their intended meaning. Proving the existence of an Herbrand model and the intersection property in the resulting richer

[1] As will turn out from section 2.1, the order of the two last subgoals is immaterial.

framework is not more problematic than before; the other model-theoretic and fixpoint semantics results that hold for $\{\log_1\}$ can also be transferred to $\{\log_2\}$ with no need of special adaptations.

2.1 Procedural semantics of $\{\log_2\}$

The main idea behind the development of the new resolution algorithm for $\{\log_2\}$ is the use of a simple constraint logic programming scheme [JL87].

In our context, a constraint is a disequation of the form $t \neq s$ or $t \notin s$. The structure of the $\{\log_2\}$-resolution process stands to $\{\log_1\}$-resolution as the CLP scheme proposed in [JL87] stands to ordinary SLD-resolution. We proceed here to illustrate the basic resolution step of our constraint method, based on the so-called **disequation analyzer** *Can* (a nondeterministic algorithm complementary, in a sense, to unification). Given a system Γ of disequations, *Can* generates a set $\{<\Delta_1,\theta_1>,...,<\Delta_\delta,\theta_\delta>\}$ of systems Δ_i in **canonical form**, each paired with a substitution θ_i which keeps track of the meaning of auxiliary variables created by the canonization process. *Can* is proved to always converge; also, Γ admits solutions if and only if $\delta \neq 0$ and some Δ_i has solutions.

Assume
G = (:- C_1&...& C_n & B_1 &...& B_k) is a goal
(the C_cs are constraints, the B_bs are $\{\log_1\}$-atoms);
Let
B_i ($i \in \{1,...,k\}$) be the selected atom,
R = (H :- C_1'&...& C_m' & B_1'&...& B_h') be the selected clause.
Choose
a unifier μ of B_i = H (cf. section 1.3);
$<\{D_1,...,D_d\}, \theta>$ from *Can*($\{C_1,...,C_n,C_1',...,C_m'\}^\mu$) (fail if empty);
Then
G' = :- D_1 &...& D_d& (B_1 &...& B_{i-1} & B_1'&...& B_h' & B_{i+1} &...& B_k)$^\sigma$,
with $\sigma = \mu \circ \theta$, is a **resolvent** of G w.r.t. R in our sense.

Let us see how the disequation analyzer works on the simple example

```
day(X) :- X ∈ {monday,...,sunday}
holy(X):- X ∉ {monday,...,saturday} & day(X).

:- holy(Day)
   Can({Day ∉ {monday,...,saturday}) =
   {<{Day ≠ monday,...,Day ≠ saturday},∅>}
:- Day ≠ monday &...& Day ≠ saturday & day(Day)
   Can({monday ≠ monday &...& monday ≠ saturday}) = {}
   --> Fail
   . . .
   Can({sunday ≠ monday &..& sunday ≠ saturday})={<{},∅>}
   --> Day = sunday.
```

2.2 Restricted Universal Quantifiers

It has been shown by several authors (e.g. [Kup87], [CFO89]) that Restricted Universal Quantifiers (RUQs) are an useful feature of the representation language in many applications, in particular to express in a clear and concise way basic set-theoretic operations (such as subset, union, intersection and so on). A RUQ formula has the form

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

and stands for the quantified implication

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

An **extended Horn clause** is a formula of the following form:

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

where each B_i can either be an atom or a RUQ $(\forall X_1 \in t_1 ... \forall X_n \in t_n)G$, G atom, satisfying the following properties:

1) if B_i is of the form $(\forall X_1 \in t_1 ... \forall X_n \in t_n)G$, then the variables $X_1,...,X_n$ can occur only in G;
2) if $i \neq j$ then $X_i \neq X_j$.

Note that the first restriction, motivated (cf. [PP91]) by our set finiteness requirement, is not present in [Kup87] since nesting of sets is not allowed there.

For example, using RUQs, it is easy to define:

1) `subset(A,B) :- (∀X ∈ A)(X∈B)`
2) `disj(D,X,Y) :- (∀Z∈D)(Z ∉Y & Z∈X).`

One might proceed as in [Kup87] by enhancing resolution so as to deal *directly* with RUQs. However, both for conceptual simplicity and for soundness concerns (which do not seem entirely fulfilled in [Kup87]) we prefer to transform extended Horn clauses into equivalent $\{\log_2\}$ clauses without RUQs (hints about a similar idea can be found in [Kup88]). We have proved that such as transformation is always possible, and have developed the algorithm to perform it. For example 1) gets transformed into the equivalent two clauses

```
subset({},Y)
subset({A|R},Y):- (A ∉R) & (A ∈Y) & subset(R,Y)
```

while 2) clearly corresponds to the $\{\log_2\}$ definition given in the previous sections; note that in $\{\log_1\}$ it is impossible to introduce RUQs: in the example, the constraint $A \notin R$ is necessary to avoid answers like {1,1,1,..} (a totally unpractical way to describe the set {1} !).

In conclusion, restricted quantifiers are introduced in $\{\log_2\}$ only at a syntactic level, as a convenient notation, with no extension at the semantic level.

3 $\{\log_3\}$. Why is negation necessary?

So far we have only considered extensions to definite Horn clause logic. A limited form of negative information has been introduced in $\{\log_2\}$ by the inclusion of the predefined predicates $\neq$ and $\notin$. However, it is easy to realize that the full power of negation (i.e. negation in both goals and clause bodies) is required in order to define other important operations on sets, without disrupting a declarative programming style.

In particular, negation in clause bodies is required to define a `setof` predicate, as the following example shows:

```
setof_p(S) :- ((∀Y ∈ S) p(Y)) & not r(S)
r(S) :- (Y ∉ S) & p(Y).
```

In turn, setof predicates can be exploited (see below) as the fundamental tool to define a set by specifying the properties of its elements, that is in an **intensional** way.

The intensional representation of sets is a common and very useful feature of a set-based language; therefore we want the final version of our language (called $\{\log_3\}$) to supply the necessary machinery (both at the syntactical and at the semantical level) to support this feature (however, still restricting our attention to finite sets).

To this purpose not only we must introduce in the language suitable intensional set formers but also we must extend Horn clause logic with some form of negation in goals and clauses which will be used in the evaluation of the set formers.

3.1 Intensionally defined sets

In $\{\log_3\}$ we allow set terms to take the form of **set abstraction terms** (also called intensional or abstract set formers). The following three different abstract set dictions (derived from SETL) are allowed:

- {X : p[X]}
- {X : p[X] | G}
- {X ∈ set term | G}

where X is a variable, p[X] denotes a positive atomic formula (including the predefined predicates =, ≠, ∈, ∉) where p is the predicate name and X occurs in one of its arguments, and G is a $\{\log_3\}$ goal (i.e. a conjunction of either positive or negative atoms). Notice that the scope of the variable X in the set term is the term itself.

Set abstraction terms have the usual well-known semantics of naive set theory. Actually, in our approach, they exist only at the syntactic level since they are translated at compile time into the corresponding set terms (extensionally defined). Thus, for instance, unification between two abstract set formers is not required at all, since set expressions are always first evaluated so that the corresponding extensional representation of a set is built and the set unification algorithm presented in Sect. 1 is used. For example, the goal

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

is first translated into the equivalent goal

```
:- setof_p(S1) & setof_q(S2) & S1 = S2
```

where `setof_p` and `setof_q` are defined in the same way as shown above.

Notice that the goal G in an abstract set former can be as complex as necessary. For example

```
{Z : Z = [Obj,People] | (∀X ∈ People) (X likes Obj)}
```

defines a set of pairs of the form `[Obj,People]` where `Obj` is an object and `People` is a set of people who like that object.

As another example let us consider the traditional 'goat-and-cabbage' problem adapted from [LM86,Pol90]. The world is represented by quadruples: `[Farmer, Goat, Cabbage, Wolf]` where each variable can take the value 0 or 1 depending on the side of the river we are considering.

```
transitions: {[[0,G,C,W],[1,G,C,W]],% farmer goes alone
             [[0,0,C,W],[1,1,C,W]],  % farmer and goat
             [[0,G,C,0],[1,G,C,1]],  % farmer and wolf
             [[0,G,0,W],[1,G,1,W]]}  % farmer and cabbage
riskyStates: {[1,0,0,_],[0,1,1,_],[1,0,_,0],[0,1,_,1]}.
```

Note that <name>:<set> is just a "sugared" notation which allows the programmer to associate a constant name to a set.

```
trips({X∈transitions | (X = [Y,Z]) & (Z ∉ riskyStates)})
plan(X,Y,[[Z,Y]|R])  :- trips(S) & [Z,Y] ∈S & plan(X,Z,R)
                         & flat(R,T) &  Y ∉ T
flat([],{})
flat([[A,B]|R],{A,B|S}) :- flat(R,S).
```

It is easy to transform the clause defining `trips` into the equivalent $\{\log_3\}$ clauses using negation:

```
setof_trips(S) :- (∀X ∈S) tr(X,S) & not r(S)
r(S) :- (X ∉S) & tr(X,S)
tr(X,S) :-  (X ∈transitions) & (X = [Y,Z]) &
            (Z ∉riskyStates).
```

Of course, some syntactic restrictions must be imposed on the predicate p and on the goal G which occur in an abstract set former, due to the set finiteness requirement and to the use of negation in clauses. Initial experimentation suggests that the class of predicates which can be accepted is nevertheless large enough to be still useful in practice. Restrictions induced by negation will be hinted at in the next section (a more detailed analysis of this issue, along with a discussion of the set finiteness problem can be found in [DP90]).

3.2 Negation

Negation has been introduced in our language in order to be able to define predicates (such as `setof`) which can be used to eliminate abstract set formers. A general form of programs (called *normal* programs in [Llo87]) is required in which negation can occur also in clause bodies.

From among the many approaches concerning negation in logic programming (see [Kun90] for a review of the main proposals that have been put forward in recent years), we have first chosen adapting to {log} the classical *negation by failure* [Cla78]. The extension of SLD-resolution with this kind of negation (SLDNF) is complete only for a restricted class of programs: those satisfying a number of syntactic restrictions, which are known in the literature (cf. [ABW87], [CL89]) as *stratification*, *allowedness* and strictness.

Completeness of {log}+NF-resolution has been proved by adapting the analogous results for the standard SLDNF-resolution to our language, in particular to deal properly with the notions of set terms and constraints as defined in the previous sections.

What is important to consider here is how these syntactical restrictions affect the expressive power of our language. It turns out that the main problem is the treatment of

RUQs, since their translation may produce some non-allowed clauses. To manage this difficulty we have tried to find out syntactic restrictions on RUQs which assure that clauses generated by their translation do not cause problems with completeness even though some of them may be non-allowed. To this purpose the definition of **confined variable** is modified so as to cover the case of extended Horn clauses. Ordinarily, a variable is said to be confined in a clause if it occurs in a positive literal in the body of the clause itself. When a RUQ of the form (∀X∈S) G occurs, essentially three new cases must be considered:

a) X is confined in the clause if S is confined in the rest of the clause or if X is confined in G;
b) if S is a variable, then it must be confined in the rest of the clause or X must be confined in G;
c) each variable that occurs in G must be confined outside the RUQ.

With these restrictions it is possible to introduce negation as failure in {log} without losing completeness. Unfortunately certain {log} definitions are no longer viable. For instance, the definition of the predicate `disj` shown in section 2.2 does not satisfy the allowedness conditions given above and should be rejected.

Other approaches to negation in logic programming are under exploration at present. In particular **constructive negation** ([Cha88], [BM*90]) seems to be very attractive, especially because it does not require allowedness for the completeness result to hold.

4 Related work

Not many proposals exist about how to deal with the introduction of sets in a logic programming language along lines such as those we have considered in this paper (i.e. by a 'deep' integration of the two paradigms, based on a semantical extension of Horn clause logic). We can mention [BN*87], [Kup87] and [Sig89] as strongly related to our work.

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 combination **set_cons + negation** 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 Sect. 3).

Kuper's proposal basically consists in extending logic programming with RUQs (see Sect. 2.2 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 [Sig89] 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.

Conclusions

In this paper we have presented {log}, a logic programming language supplying finite sets (both extensionally and intensionally defined), restricted universal quantifiers (RUQs) and basic operations on sets. As regards enumerated sets and RUQs, it has been shown that their introduction only presupposes extending Horn clause logic with set terms and a few distinguished predicates dealing with sets, namely $=, \in, \notin, \neq$. The resolution method and unification have been modified accordingly. As regards intensionally defined sets, their introduction requires negation in goals and clause bodies.

{log} has been fully implemented in Prolog and tested on simple examples. RUQs and abstract set formers are translated via preprocessing into equivalent $\{\log_2\}$ clauses which, in their turn, are executed by a Prolog metainterpreter. Nondeterminism in the unification algorithm and in the extended resolution procedure are taken into charge by Prolog's backtracking mechanism.

Much work still awaits to be done in the direction of effective implementations of the {log} language. As a first step, the unification algorithm should be implemented in a lower level language than Prolog (e.g. in C), and further investigations should be devoted to detecting special cases (ground set terms, flat set terms, etc.) in which set unification can be performed more efficiently .

Acknowledgments

We wish to thank Alberto Policriti and Francesco Zambon (Enidata) for their interesting suggestions and Ron Sigal for the first idea and the precious discussion. This work has been partially supported by the AXL project of ENI and ENIDATA, and by MURST 60%.

References

[ABW87] K.Apt, H.Blair, A.Walker. Towards a theory of declarative knowledge. In *Foundations of deductive databases and Logic Programming* (J.Minker, ed). Morgan Kaufmann, Los Altos, CA, 1987.

[AHU75] A.Aho, J.Hopcroft, J.Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1975.

[BM*90] R.Barbuti, P.Mancarella, D.Pedreschi, F.Turini. A transformational approach to negation in Logic Programming. *The Journal of Logic Programming*, (8), 1990.

[BN*87] C.Beeri, S.Naqvi et al. Set and negation in a Logic Database Language (LDL1). *Proceedings 6th ACM SIGMOD Symposium*, 1987.

[CFO89] D.Cantone, A.Ferro, E.G.Omodeo. *Computable set Theory*. Oxford University Press, International Series of Monographs on Computer Science, 1989.

[Cha88] D.Chan. Constructive negation based on the completed databases. *Proceedings 1988 Conference and Symposium on Logic Programming*, Seattle, Washington, 1988.

[Cla78] K.L. Clark. Negation as Failure. In: *Logic and Data Bases* (H.Gallaire and J.Minker eds.), Plenum Press, 1978.

[CL89] L.Cavedon, J.W.Lloyd. A completeness Theorem for SLDNF resolution. *The Journal of Logic Programming*, (7), 1989.

[DF89] E.E.Doberkat, D.Fox. *Software Prototyping mit SETL*. B.G.Teubner Stuttgart, 1989.

[DP90] A.Dovier, E.Pontelli. Logic Programming with sets (in Italian). Tesi di Laurea. University of Udine, 1991.

[JN84] B.Jayaraman, A.Nair. Subset-Logic Programming: application and implementation. *Proceedings International Conference on the Fifth Generation Computer Systems*, Tokyo, 1984.

[JP89] B.Jayaraman, D.A.Plaisted. Programming with Equations, Subsets and Relations. *Proceedings of NACLP89*, Cleveland, 1989.

[JL87] J.Jaffar, J.L.Lassez. From Unification to Constraints. *Proceedings Fifth Conference on Logic Programming*, Tokyo, 1987.

[Kun90] K.Kunen. Completeness results for SLDNF. Tutorial presented at *GULP90: Fifth National Conference on Logic Programming*, Padova, 1990.

[Kup87] G.M.Kuper. Logic Programming with Sets. *Proceedings 6th ACM SIGMOD Symposium*, 1987.

[Kup88] G.M.Kuper. On the Expressive Power of Logic Programming with Sets. *Proceedings 7th ACM SIGMOD Symposium*, 1988.

[Llo87] J.W.Lloyd. *Foundations of logic programming*. Springer Verlag, 2nd ed., 1987.

[LM86] R.M.Lee, L.W.Miller, A logic programming framework for planning and simulation. *Decision support systems*, 1986.

[LMM86] J.L.Lassez, M.J.Maher, K.Marriot. Unification revisited. *Lecture Notes in Computer Science*, Vol. 306, Springer Verlag, 1986.

[Naf86] M.Naftalin. An experiment in practical semantics. *ESOP 86*, LNCS n.213, Springer Verlag, 1986.

[PP91] F.Parlamento, A.Policriti. Expressing infinity without foundation. J*ournal of*

Symbolic Logic (to appear).

[Pol90] A.Policriti. Notes on SETLOG and 'Desiderata for Logic Programming with sets'. Università di Udine, Dip. di Matematica e Informatica, research report n.32, 1990.

[Sig89] R.Sigal. Desiderata for Logic Programming with sets. *Proceedings GULP89: Fourth National Conference on Logic Programming*, Bologna, 1989.

[SD*86] J.T.Schwartz, R.B.K.Dewar, E.Dubinsky, E.Schonberg. *Programming with sets, an introduction to SETL.* Springer-Verlag, 1986.

[Tur86] D.Turner. An overview of Miranda. *SIGPLAN Notices*, Vol.21, n.12, 1986.

[Z86] *Z handbook*, Oxford University Computing Laboratory, Oxford 1986.