{*log*} User's Manual

Version 4.9.1 Release 20

Gianfranco Rossi
Università degli Studi di Parma, Parma, Italy.
gianfranco.rossi@.unipr.it

Maximiliano Cristiá
CIFASIS and UNR, Rosario, Argentina.
cristia@cifasis-conicet.gov.a

April 30, 2016

Abstract

This is the second edition of the user's manual for {*log*}, a Constraint Logic Programming language that embodies the fundamental forms of set designation and a number of primitive operations for set management. The {*log*} interpreter is written in Prolog and the full Prolog code of the interpreter is freely available at the {*log*} WEB page `http://people.math.unipr.it/gianfranco.rossi/setlog.Home.html`.

Contents

1 Introduction

{*log*} (read 'set-log') is a Constraint Logic Programming language that embodies the fundamental forms of set designation and a number of primitive operations for set management [1, 4, 7, 9].

Sets are designated primarily by the explicit enumeration of all its elements (*extensional sets*), using *set terms*. Sets can contain not only atoms as their elements, but also other sets (*nested sets*), with no restriction over the level of set nesting.

The language provides a number of basic primitive operations for set management, such as `=` (equality), `in` (set membership), `un` (union), `inters` (intersection), etc. {*log*} can also deal with binary relations and partial functions through most of the standard relational operators, such as `dom` (domain) and `comp` (relational composition). Given that binary relations and partial functions are sets of ordered pairs they can be freely combined with sets, thus providing a uniform treatment for all these concepts.

Furthermore, {*log*} provides *Restricted Universal Quantifiers* (RUQ) and *Intensional Sets*, that is sets defined by property rather than by enumerating all their elements.

{*log*} inherits much of standard Prolog: its syntax (a part a few minor changes), the user interaction modality, input/output facilities, some extra-logical features (e.g., arithmetic). Throughout the manual, we assume the reader is familiar with Prolog and programming with Prolog techniques, as well as with the general principles and notation of Constraint Logic Programming languages. Moreover, for some part of the {*log*} language (e.g., its syntax) we will only describe those features that really differ from standard Prolog. For other parts (e.g., input/output, arithmetic) we will completely rely on the corresponding standard Prolog facilities. Finally, for all the formal results concerning {*log*} (e.g., its logical and procedural semantics, the constraint solving mechanism) we refer the reader to the {*log*} specific papers listed in the Bibliography section.

The {*log*} interpreter is written in standard Prolog and has been tested using SWI-Prolog (last releases) and can be ported to any Prolog system implementing standard Prolog with very limited effort.

The first version of the {*log*} interpreter was developed by Agostino Dovier and Enrico Pontelli, under the supervision of Eugenio Omodeo and Gianfranco Rossi, as part of their work to obtain the "Laurea" degree at the Department of Mathematics and Computer Science of the University of Udine in 1991. Later on, the {*log*} interpreter was revised at various times by Gianfranco Rossi, who is still maintaining the current version of the

interpreter. More recently (circa 2015), Gianfranco Rossi and Maximiliano Cristiá added support for binary relations and partial functions.

The Prolog code of the {*log*} interpreter is available at the {*log*} WEB page `http://people.math.unipr.it/gianfranco.rossi/setlog.Home.html`. At the same page, you can find also the PDF file of this manual, the {*log*} library file containing the {*log*} definitions of some operations on sets not provided as built-in in {*log*}, a library of definitions for binary relations and partial functions, a file containing a number of simple preprocessing rules ("filtering rules") that may help constraint solving, a few sample {*log*} programs, and the postscript files of various papers related with {*log*}.

2 Using {*log*}

Assume the {*log*} interpreter (Prolog source code) has been saved into a file named `setlog.pl`. To start working with the interpreter, invoke Prolog and then load the {*log*} interpreter, e.g., by giving the directive

```
?- consult('setlog.pl').
```

The {*log*} interpreter is loaded into the Prolog program database. At this point there are two ways to use {*log*}: interactively, much as Prolog itself; and as a Prolog predicate. We will illustrate both of them with a simple example. To access the interactive environment run the following directive:

```
?- setlog.
```

and {*log*} will show you its prompt:

```
{log}=>
```

Now you can give it goals much as in the Prolog environment. For example, you can ask {*log*} to find the solution for the following formula:

```
{log}=> un({1},{2},A).
```

in which case {*log*} answers:

```
A = {1,2}
```

and asks you if you want another solution.

The same goal can be executed from the Prolog environment, e.g. by using the predicate `setlog/1` (see Sect. 7.1 for more information):

```
?- setlog(un({1},{2},A)).
```

making Prolog to print:

```
A = {1,2}
```

In the interactive mode, you can leave the {*log*} environment by issuing:

```
{log}=> halt.
```

and you can re-enter the {*log*} environment by simply issuing again `setlog`. Note that, in the current implementation, a few run-time errors possibly detected by Prolog while executing the {*log*} interpreter may force execution to leave the {*log*} environment. To re-enter {*log*}, call the `setlog` goal.

2.1 Loading {*log*} libraries

Libraries can be loaded in any order and in any moment. Library predicates are dealt with as other user defined predicates. The standard {*log*} library can be loaded from the {*log*} environment by issuing:

```
{log}=> consult_lib.
```

The same can be done from the Prolog environment by issuing

```
?- consult_lib.
```

All other libraries, e.g. the library `'setloglibpf.slog'` concerning partial functions, can be consulted using the {*log*} predicate `consult/1`. For example:

```
{log}=> consult('setloglibpf.slog').
```

The same can be obtained from the Prolog environment by issuing

```
?- setlog(consult('setloglibpf.slog')).
```

or, equivalently, by exploiting the Prolog predicate `setlog_consult/1`, by: `?- setlog_consult('setloglibpf.slog').` See Section 11 for the complete list of the predicates defined in the standard {*log*} libraries.

2.2 Dealing with {*log*} programs

{*log*} programs are much like Prolog programs; that is, a collection of clauses saved in a file. For example, assume the following clause is saved in file `p.pl`:

```
un12(A) :- un({1},{2},A).
```

Then you can load it into the {*log*} environment with the `consult/1` predicate as follows:

```
{log}=> consult('p.pl').
```

and then you can use the clauses defined in it as follows:

```
{log}=> un12({1,2,3}).
```

in which case {*log*} answers `no` because the union of sets `{1}` and `{2}` is not equal to the set `{1,2,3}`.

While consulting, the interpreter shows on the standard output the number of the clause of the given program it is currently reading. If a syntax error is detected in the program, program consulting is immediately stopped. The last clause number that appeared on the output is the number of the clause where the error was detected. In the current implementation, finding out a syntax error while reading a program forces execution to leave the {*log*} environment.

Clauses stored in the {*log*} program database can be completely removed by executing the directive:

```
{log}=> abolish.
```

The current content of the program database can be printed out on the standard output by using the directive

```
{log}=> listing.
```

Note that both predicates `abolish` and `listing` refer to the whole set of stored clauses (both user defined and those of the library predicates), but not to predicates in the standard {*log*} library `setloglib.slog` which, conversely, are not removed and not listed[1].

2.3 Asking for help

Finally, simple help facilities are provided in the form of built-in predicates:

- `help/0` (or `setlog_help/0` from the Prolog environment) provides general help information on {*log*};

[1] At present, no other facility is provided to consult, remove, or listing a program in {*log*}. In particular, it is not allowed to consult a program stored in `file` by using the syntax `[file]`, as usual in Prolog. Moreover, there is no support for reconsulting a program. The standard Prolog predicates `abolish/2` and `listing/1` are not provided for now.

- `h/1` provides more detailed information on {*log*}, according to its parameter: `h(syntax)` for {*log*} syntactic convenctions; `h(constraints)` for {*log*} constraints; `h(builtins)` for {*log*} built-in predicates; `h(lib)` for {*log*} standard library predicates; `h(prolog)` for Prolog predicates for accessing {*log*} from the Prolog environment; `h(all)` to get all available help information.

3 Solving formulas with extensional sets

An extensional set is a set whose elements are enumerated. For example, `{1,a,hello}` is an extensional set with three elements. Some of the elements of an extensional set can be other extensional sets. For example, `{a,b}` is an element of the following extensional set `{51,{a,b}}`. Elements inside an extensional set can be of any sort (or type or class), as shown in the previous examples.

The most simple extensional set is the *empty set* noted in {*log*} as `{}`. The second most simple extensional set is the *singleton set*, noted in {*log*} as `{e}`, where `e` is its single element. Then we can ask {*log*} whether the empty set is equal to a singleton set:

```
{log}=> {} = {1}.
```

in which case the answer is, obviously, `no`.

As in mathematics, {*log*} extensional sets can contain variables. In {*log*}, as in Prolog, variables are denoted by words starting with an uppercase letter. Then, we can ask {*log*} to solve the following equation:

```
{log}=> {X} = {1}.
```

where `X` is a variable. In this case the answer is:

```
X = 1
```

Note that

```
{log}=> {x} = {1}.
```

results in a `no` answer because `x` is a constant (as it does not start with an uppercase letter) not equal to `1`.

A more interesting formula is the following:

```
{log}=> {X,Y} = {2,1}.
```

because it has two solutions:

```
X = 2, Y = 1
X = 1, Y = 2
```

that {*log*} is able to find by exploiting *set unification* [8].

{*log*} also provides a notation (not used in mathematics) to define sets. The expression `{x / A}` represents the set $\{x\} \cup A$. Then, A must be a set. So, for instance, we can write `{1 / {a / {}}}` which represents the set $\{1\} \cup (\{a\} \cup \emptyset)$ which is equal to the set $\{1, a\}$. Given this obvious equality, {*log*} allows a more user-friendly notation: we can write `{1, a / {}}` instead of `{1 / {a / {}}}`; and `{1 , a / {}}` can be further simplified as `{1, a}`. When combined with the fact that variables can be sets, this notation provides a new level of expressiveness to the language. For instance, it is interesting to ask {*log*} for the solutions of the following formula (which again calls into play set unification):

```
{log}=> {X/A} = {6,7,8}.
```

as it has six:

```
1 X = 6, A = {7,8}
2 X = 6, A = {6,7,8}
3 X = 7, A = {6,8}
4 X = 7, A = {6,7,8}
5 X = 8, A = {6,7}
6 X = 8, A = {6,7,8}
```

Note that, for instance, the second solution states that `{6 / {6,7,8}}` is equal to `{6,7,8}` which is true in virtue of the so-called *absorption property* of set theory [4]. Recall that `{6 / {6,7,8}}` can be written as `{6,6,7,8}` where the presence of duplicate elements becomes evident.

Note how the number of solutions increases if we want to identify more elements in the set. For example, the following equation:

```
{log}=> {X,Y/A} = {6,7,8}.
```

has 30 solutions, among which:

```
X = 6, Y = 7, A = {8}
X = 6, Y = 7, A = {7,8}
X = 6, Y = 6, A = {6,7,8}
```

In {x$_1$,...,x$_n$ / A}, x$_1$,...,x$_n$ is called *element part* and A is called *set part*. It is very important to remark that since the set part of an extensional set can be a variable (representing any finite extensional set), then {*log*}'s extensional set constructor allows users to write *unbounded finite sets*. In effect, an expression such as {x / A} represents a finite but *unbounded* set as the set denoted by A can have any number of elements.

3.1 Set operators

{*log*} supports all the classic set operators. All set operators are given as predicates. Then, for instance, we can ask {*log*} to find a value for A in:

```
{log}=> inters({1},{2},A).
```

making {*log*} to answer

```
A = {}
```

as A must be equal to the intersection between {1} and {2}.

Table 1 lists all the set operators available in {*log*} as predicates. As can be seen, most operators have their own negation. Although {*log*} implements negation in the form of Negation as Failure (see Sect.3.2), it is always advisable to use the negated predicates.

All the arguments of all these predicates can be variables and set terms. Even the in and nin predicates admit sets as the first argument because in {*log*} set elements can be sets. For example:

```
{log}=> {1} in {2,a,{1}}.
```

makes {*log*} to answer yes.

We believe all set operators are self-explanatory although set and nset require some clarifications. Users seldom need to indicate that something is a set when writing {*log*} code. In general, {*log*} automatically infers the *sort* of variables by analyzing the formulas in which they participate. Hence, predicates set and nset are generally used internally by {*log*}. Users may see a set predicate as part of the answer given by {*log*} for some formulas—see an example in the next section. Users may need to use set predicates in some formulas involving *multisets* (see Sect. 8.

3.2 Introducing formulas

In this section we will show how to write some of the formulas accepted by {*log*}.

Operator	$\{log\}$	Meaning
set	`set(A)`	A is a set
equality	`A = B`	$A = B$
membership	`x in A`	$x \in A$
union	`un(A,B,C)`	$C = A \cup B$
intersection	`inters(A,B,C)`	$C = A \cap B$
difference	`diff(A,B,C)`	$C = A \setminus B$
subset	`subset(A,B)`	$A \subseteq B$
strict subset	`ssubset(A,B)`	$A \subset B$
disjointness	`disj(A,B)`	$A \parallel B$
Negations		
equality	`A neq B`	$A \neq B$
union	`nun(A,B,C)`	$C \neq A \cup B$
intersection	`ninters(A,B,C)`	$C \neq A \cap B$
difference	`ndiff(A,B,C)`	$C \neq A \setminus B$
membership	`x nin A`	$x \notin A$
subset	`nsubset(A,B)`	$A \not\subseteq B$
disjointness	`ndisj(A,B)`	$A \not\parallel B$

Table 1: Set operators available in $\{log\}$

In {*log*} conjunction is written with the `&` character (instead of the comma as in Prolog). Then, in asking {*log*} to solve the formula:

```
{log}=> un({1/B},{j},A) & j in B.
```

it answers:

```
B = {j/_G6327},
A = {1,j/_G6327}
Constraint: set(_G6327)
```

where `_G6327` is a variable name automatically generated by {*log*} and `Constraint` is a list of *constraints* that the variables returned by {*log*} must verify. In this case the constraint is very simple and intuitive: `_G6327` must be a set as is the set part of an extensional set. Variables automatically generated by {*log*} are called *new* or *fresh* variables. In general, {*log*} will include many fresh variables in its answers. For the sake of presentation, from now on, we will replace these system generated names by simpler more readable ones.

Logical disjunction is also available in {*log*} by means of the `or` connective (instead of the semicolon as in Prolog). The same formula given above but using disjunction in place of conjunction:

```
{log}=> un({1/B},{j},A) or j in B.
```

has two solutions:

```
A = {j,1/B}
Constraint: set(B)


B = {j/C}
Constraint: set(A), set(C)
```

where `A` can be any set in the second solution.

Disjunction and conjunction can be freely combined to form complex formulas. As conjunction has higher precedence than disjunction use parenthesis to write the right formula.

{*log*} provides also negation by means of the `naf` predicate. `naf` implements Negation as Failure: `naf G`, where `G` is any {*log*} goal, fails if the goal `G` has a solution, and succeeds otherwise. Unfortunately, `naf` suffers of all the well-known problems of Negation As Failure. Generally speaking, `naf` is not able to deal correctly with goals containing uninitialized variables. As an example, the simple goal

```
{log}=> naf X = a & X in {b}.
```

answers `no`, while the logically equivalent goal

```
{log}=> X in {b} & naf X = a.
```

correctly answers `X = b`, simply because, in the first goal, when `naf` is executed `X` is still uninitialized.

When `G` is a primitive constraint, however, the problem can be bypassed by using the negated version of the primitive constraints (see Table 1). For example, the first goal above, written using `neq` in place of negated equality, that is

```
{log}=> X neq a & X in {b}.
```

correctly answers `X = b`.

Note that the logical implication can be implemented using disjunction and negation. Therefore, instead of writing, for instance:

$$x \in B \cup C \implies (x \in B \vee x \in C) \tag{1}$$

you can write

$$\neg x \in B \cup C \vee x \in B \vee x \in C$$

which in turn is equivalent to:

$$A = B \cup C \wedge (x \notin A \vee x \in B \vee x \in C)$$

which can be easily translated into $\{log\}$ as (assuming x is intended to be a variable):

```
{log}=> un(B,C,A) & (X nin A or X in B or X in C).
```

Precisely, you can use $\{log\}$ to automatically prove theorems about set theory. If you want to prove that (1) is a theorem then you can ask $\{log\}$ to try to find a solution for:

```
{log}=> un(B,C,A) & X in A & X nin B & X nin C.
```

that is, the negation of your theorem. In this case $\{log\}$ returns `no` which means that there is no value (for `X`, `A`, `B` and `C`) satisfying the formula. In turn, this implies that the negation of the formula given to $\{log\}$ (i.e. your theorem) is always satisfiable. And if a formula is always satisfiable, it is a theorem. In this sense, $\{log\}$ behaves as a decision procedure for the *theory of finite, unbounded sets* [7]

4 Solving formulas with binary relations

A binary relation is a set of ordered pairs. If X and Y are two sets then any set R such that $R \subseteq X \times Y$ is a binary relation. Given that binary relations are sets (of ordered pairs) then $\{log\}$ can be used to work with formulas involving binary relations. Such formulas, however, may involve not only set operators (cf. Table 1) but also *relational operators*. For this purpose, $\{log\}$ introduces a rich set of relational operators, such that it can determine the satisfiability of any formula including them.

For example, the following formula

```
{log}=> dom(R,{a}).
```

makes $\{log\}$ to return the most general binary relation whose domain is the set $\{a\}$. This relation is given a follows:

```
R = {[a,Y]/S}
Constraint: comp({[a,a]},S,S), [a,Y] nin S, rel(S)
```

There are several things to comment about this answer. `R` is given as an extensional set containing the ordered pair `[a,Y]` because `R` must contain at least an ordered pair (because it has a not-empty domain) whose first component must be `a` but whose second component can be anything—fact that is represented by making the second component to be a variable.

Observe that ordered pairs are noted with square brackets. Then, `[x,y]` represents the ordered pair (x, y). Note that, `[x,y] = [w,z]`, if and only if `x = w` and `y = z`. In this manual, when writing mathematics we will use parenthesis to note ordered pairs, but we will use square brackets when we show $\{log\}$ code.

Moreover, the set part of `R` (i.e. `S`) is *constrained* to be a binary relation by means of the predicate `rel(S)`. Indeed, `rel` forces its argument to be a set of ordered pairs. However, constraining `S` to be a relation is not enough for the correctness of the solution. The domain of `S` must be a subset of `{a}`. This is forced by the constraint `comp({[a,a]},S,S)`. In effect, `comp(Q,T,U)` means $U = Q \circ T$, that is U is the result of the relational composition between Q and T. Formally:

$$Q \circ T = \{(x, z) \mid \exists y : (x, y) \in Q \wedge (y, z) \in T\}$$

Then, it can be shown that $\text{dom}(\{(a, a)\} \circ T) = \{a\}$, thus guaranteeing that $\text{dom}(\{(a, y)\} \cup T) = \{a\}$, for any y.

Finally, note that the constraint `[a,Y] nin S` is generated by the solver to avoid possibly infinite computations due to the application of the absorption property, which might generate set terms with infinitely many occurrences of the same element. In fact, the formula `R = {x/S} & x nin S` ensures that `S` cannot contain any occurrence of `x`.

Since binary relations are sets of ordered pairs, they can be built by means of the same set constructors described in Section 3. In particular the empty binary relation is denoted with `{}`. Furthermore, formulas involving relational operators are built as we shown in Section 3.2 (i.e. by means of `&` and `or`). Besided, they can be freely combined with formulas involving set operators. For example:

```
{log}=> dom(R,{a/B}) & [b,X] in R.
```

is a formula combining set and relational operators and making use of the extensional set constructor.

It is very important to remark that, as can be seen in the previous formula, not only the relation is a *set* in exactly the same sense of the sets introduced in Section 3, but also its domain. This clearly shows that $\{log\}$ allows for a completely uniform treatment of sets and binary relations.

Table 2 lists all the relational operators available in $\{log\}$ as predicates. The negation of these relational operators, apart from `rel`, has not yet been defined. In turn, Table 3 gives the mathematical definition of each relational operator given in Table 2. All the arguments of all these predicates can be variables and set terms.

As with set operators, we believe all relational operators are self-explanatory. Same considerations mentioned for `set` and `nset` apply for `rel` and `nrel`. That is, in general, users do not need to indicate that something is a binary relation because $\{log\}$ automatically infers this fact.

A *partial function* is a binary relation where no two ordered pairs share the same first component. Formally, f is a partial function if and only if f is a binary relation and:

$$\forall x, y_1, y_2 : (x, y_1) \in f \wedge (x, y_2) \in f \implies y_1 = y_2 \tag{2}$$

Therefore, partial functions are a subset of binary relations. This means that $\{log\}$ can also be used to find solutions for formulas involving partial functions. These formulas, are built as formulas involving binary relations plus the addition of the predicates listed in Table 4.

Differently from `rel`, users *must* explicitly include a `pfun` predicate for all those binary relations they want to be partial functions. $\{log\}$ will only

Operator	$\{log\}$	Meaning
binary relation	`rel(R)`	R is a binary relation
domain	`dom(R,A)`	$\mathrm{dom}\, R = A$
range	`ran(R,A)`	$\mathrm{ran}\, R = A$
composition	`comp(R,S,T)`	$T = R \circ S$
inverse	`inv(R,S)`	$S = R^{-1}$
domain restriction	`dres(A,R,S)`	$S = A \lhd R$
domain anti-restriction	`dares(A,R,S)`	$S = A \lhd\!\!\!\cdot\, R$
range restriction	`rres(A,R,S)`	$S = R \rhd A$
range anti-restriction	`rares(A,R,S)`	$S = R \rhd\!\!\!\cdot\, A$
overriding	`oplus(R,S,T)`	$T = R \oplus S$
relational image	`rimg(A,R,B)`	$B = R[A]$
	Negations	
binary relation	`nrel(R)`	R is not a binary relation

Table 2: Relational operators available in $\{log\}$

Operator	Definition
domain	$\mathrm{dom}\, R = \{x \mid \exists y : (x, y) \in R\}$
range	$\mathrm{ran}\, R = \{y \mid \exists x : (x, y) \in R\}$
composition	$R \circ S = \{(x, z) \mid \exists y : (x, y) \in R \wedge (y, z) \in S\}$
inverse	$R^{-1} = \{(y, x) \mid (x, y) \in R\}$
domain restriction	$A \lhd R = \{(x, y) \mid (x, y) \in R \wedge x \in A\}$
domain anti-restriction	$A \lhd\!\!\!\cdot\, R = \{(x, y) \mid (x, y) \in R \wedge x \notin A\}$
range restriction	$R \rhd A = \{(x, y) \mid (x, y) \in R \wedge y \in A\}$
range anti-restriction	$R \rhd\!\!\!\cdot\, A = \{(x, y) \mid (x, y) \in R \wedge y \notin A\}$
overriding	$R \oplus S = (\mathrm{dom}\, S \lhd\!\!\!\cdot\, R) \cup S$
relational image	$R[A] = \mathrm{ran}(A \lhd R)$

Table 3: Definition of relational operators

Operator	{*log*}	Meaning
partial function	`pfun(F)`	F verifies (2)
function application	`apply(F,X,Y)`	$F(X) = Y$
identity function	`id(A,F)`	$\mathrm{id}\, A = F$
Negations		
partial function	`npfun(F)`	F does not verifies (2)

Table 4: Partial function operators available in {*log*}

automatically assert that `F` is a partial function if it appears as the first argument of `apply` or as the second of `id`. For example, the following formula is unsatisfiable if `F` is intended to be a partial function but it is satisfiable for a binary relation:

```
{log}=> dom(F,{a}) & [a,Y1] in F & [a,Y2] in F & Y1 neq Y2.
```

Then, {*log*} answers:

```
F = {[a,Y1],[a,Y2]/G}
Constraint: [a,Y2] nin G, comp({[a,a]},G,G), [a,Y1] nin G,
            Y1 neq Y2, rel(G)
```

for that formula but it answer `no` for the following:

```
{log}=> pfun(F) & dom(F,{a}) & [a,Y1] in F & [a,Y2] in F
        & Y1 neq Y2.
```

{*log*} behaves as a decision procedure also for the *theory of finite, unbounded binary relations*. This means, as with sets, that {*log*} is able to find out whether any formula, involving binary relations and the operators listed in Tables 2 and 4, is satisfiable or not.

5 Solving formulas including integer numbers

{*log*} deals with arithmetic expressions through a number of built-in predicates. The comparison arithmetic operators available in {*log*} are shown in Table 5. In the table, `e1`, `e2` are integer expressions and `n` is a variable or an integer constant.

An arithmetic expression is either a simple number or an arithmetic function (e.g. `+`, `-`, `*`, `mod`, etc.); the arguments of a function are in turn arithmetic expressions. Numbers can be either integer or floating-point numbers. For example, the following arithmetic formulas are solved as shown:

Operator	{*log*}	Meaning
simple equality	`n is e1`	$n = e_1$
less or equal	`e1 =< e2`	$e_1 \leq e_2$
less	`e1 < e2`	$e_1 < e_2$
greater or equal	`e1 >= e2`	$e_1 \geq e_2$
greater	`e1 > e2`	$e_1 > e_2$
equal	`e1 =:= e2`	$e_1 = e_2$
not equal	`e1 =\= e2`	$e_1 \neq e_2$

Table 5: Comparison arithmetic operators available in {*log*}

```
{log}=> X is 3 * 5.
X = 15

{log}=> 1.5 + 1 > 0.7.
yes
```

As Prolog, {*log*} does not evaluate arithmetic expressions unless they occur as parameters in one of the built-in predicates listed in Table 5. As an example, given the formula

```
{log}=> 2 + 3 in {5}.
```

{*log*} answers `no` because the expression `2 + 3` is left unevaluated and `2 + 3` does not belong to the set `{5}`. Conversely, using the `is` predicate, the formula

```
{log}=> X is 2 + 3 & X in {5}.
```

turns out to be satisfiable and the answer will be `X = 5`. In fact, the `is` predicate forces {*log*} to evaluate the expression at the right hand side as soon as possible.

If the expression `e_i` in the built-in predicates of Table 5 is a floating-point expression, then all variables possibly occurring in `e_i` must have a value. Otherwise, a problem in the arithmetic expression is detected and {*log*} answers `no`. For example:

```
{log}=> 1.5 + X > 0.7.
Problem in arithmetic expression
no
```

Conversely, if `e_i` is an integer expression, then it can contain uninitialized variables. As an example:

Operator	{*log*}	Meaning
integer	`integer(t)`	t is an integer number
integer	`ninteger(t)`	A is not an integer number

Table 6: `integer` and `ninteger` constraints

```
{log}=> 34 is X + 1.
X = 33
```

In fact, arithmetic built-in predicates over integer expressions are dealt with by a constraint solver over finite domains (namely, CLP(FD)).

As concerns solving formulas including integer numbers, {*log*}'s solver is *incomplete*. That is, given a goal G, if the answer is `no`, then G is surely unsatisfiable; otherwise, it is not guaranteed, in general, that G is satisfiable.

```
{log}=> 34 > X + 1.
***WARNING***: non-finite domain
true
Constraint: X in int(inf,32)
```

`int(inf,32)` represents the integer interval $[-\infty..32]$ (see next subsection). The warning message means that the answer *might be incorrect*—although in this particular example it is correct.

As another example:

```
{log}=> X+1 > Y & X+1 < Y.
***WARNING***: non-finite domain
true
Constraint: integer(X), integer(Y)
```

where `integer` is a {*log*} primitive constraint: `integer(X)` is true if and only if `X` is an integer number. There is also its negative version `ninteger` (see Table 6).

This goal is clearly unsatisfiable, but {*log*} (actually the underlying CLP(FD) solver) is not able to detect it.

The solver becomes complete (i.e. a decision procedure) if we provide a *finite domain* for each integer variable which occur in the formula to be checked.

5.1 Finite domains

Domains for integer variables are specified through *integer intervals*. In mathematics an integer interval is noted $[m, n]$ and represents the set $\{i \in \mathbb{Z} \mid m \leq i \leq n\}$. In $\{log\}$ intervals are noted as `int(m,n)`, where `m` and `n` can be, in general, either constants or variables and represent the same than in mathematics.

Intervals are primarily used to associate a finite domain to an integer variable. The formula:

```
X in int(1,10)
```

states that the domain of the variable `X` is the interval $[1, 10]$.

Note that when used to specify domains, the interval limits must be integer constants.

The last two goals above, give the correct answer if we provide suitable domains for the integer variables `X` and `Y`.

```
{log}=> 34 > X + 1 & X in int(1,100).
X = 1
...
Another solution?  (y/n)
X = 32
Another solution? (y/n)
no

{log}=> X+1 > Y & X+1 < Y & X in int(1,10) & Y in int(1,20).   (3)
no
```

It is important to observe that $\{log\}$, by default, always performs labeling at the end of the computation for all the integer variables which have a finite domain associated with them in the resulting final formula. Labeling a variable X with domain D means non-deterministically assigning to X one by one all possible values in D. After each value has been assigned, then the whole constraint is analyzed again to check its satisfiability.

If one wants to suppress the defalt activation of labeling one can give the goal:

```
nolabel.
```

If we, successively, give the goal

```
{log}=> 34 > X + 1 & X in int(1,100).
```

then the answer now will be

```
true
Constraint: X in int(1,32)
```

instead of generating all possible values for `X` as in the case when labeling is active.

When global labeling is deactivated we can nevertheless perform labeling on a single variable by using the built-in predicate `labeling(X)`.

Global labeling can be reactiveted in any moment by the goal

```
label.
```

The domain of an integer variable can be obtained also as the result of solving some arithmetic constraint on this variable. For example, the goal

```
{log}=> 34 > X + 1 & X >= 1 & X =< 100.
```

will produce the same result as the goal `34 > X + 1 & X in int(1,100)` shown above.

Note that labeling is performed only for variables which have a bounded domain associated with them. For example,

```
{log}=> 34 > X + 1 & X =< 100.
***WARNING***: non-finite domain
true
Constraint: X in int(inf,32)
```

where it is evident that no labelling has been performed.

Observe that in the goal (3) it is enough to specify the domain for one of the two variables, for example

```
@X+1 > Y & X+1 < Y & X in int(1,10).@
```

will produce the same result as above.

Finally note that the atom `X in {1,2,3}` is logically equivalent to `X in int(1,3)` but it processing by the {*log*}'s solver is quite different. Actually, `X in {1,2,3}` is operationally equivalent to

```
X in int(1,3) & labeling(X)
```

Thus, `X in {1,2,3}` is not used to associate a domain to the variable `X`; rather it is used to nondeterministically assign to `X` each value form a set of possible values.

5.2 Intervals as sets

Integer intervals are *sets*. Thus, most {*log*} primitive set predicates can include also terms of the form `int(m,n)` everywhere a set can be used. A few of these predicates allow intervals to be unbounded (i.e. one or both of the limits `n` and `m` can be uninitialized variables), while most of them work well only with bounded intervals.

Precisely, the following predicates allow *unbounded* intervals to be used as their parameters:

- `t1 in t2` (membership)
- `t1 nin t2` (non-membership)
- `inters(t1,t2,t3)` (intersection)

As an example:

```
{log}=> inters(int(3,N),int(10,20),A) & A neq {}
        & N in int(-1000000,1000000).
```

in which case the *first* answer is:

```
N = 21, A = int(10,20)
```

Afterwards, {*log*} will give one answer for each `N` in `int(-1000000,1000000)` that satisfies the formula.

Note that if we do not provide any domain for the integer variable `N` then the first answer is:

```
***WARNING***: non-finite domain
A = int(10,20)
Constraint: N in int(21,sup)
```

As another example (in which it is not necessary to specify a domain for the involved integer variables)

```
{log}=> int(A,B) = {1,X,3}.
A = 1, B = 3, X = 2
```

The following predicates allow *bounded* intervals to be used as their parameters:

- `un(s1,s2,s3)` (union), `nun(s1,s2,s3)` (non-union)
- `disj(s1,s2)` (disjointness), `ndisj(s1,s2)` (non-disjointness)

Operator	{*log*}	Meaning
set cardinality	`size(A,N)`	$\|A\| = N$

Table 7: The set cardinality operator

- `subset(s1,s2)` (subset), `nsubset(s1,s2)` (not subset)
- `diff(s1,s2,s3)` (difference), `ndiff(s1,s2,s3)` (not difference)
- `ssubset(s1,s2)` (strict subset)
- `ninters(s1,s2,s3)` (not intersection)

Note that all primitive predicates for relational operators (see Table 2) do not yet support intervals as their parameters.

5.3 Cardinality constraint

`size` is a set predicate that represents the cardinality of a set. It is defined in Table 7. Note that its second argument can be an integer constant or a variable. For example, the answer to the following formula:

```
{log}=> size(A,3).
```

is

```
A = {X,Y,Z}
Constraint: X neq Y, X neq Z, Y neq Z
```

because `{X,Y,Z}`, with `X`, `Y` and `Z` variables, is the most general set of three elements provided they hold different values—and from here the constraint.

Given `size(A,N)`, when `N` is a variable and the set part of `A` is a variable, then we may get the same warning message discussed above, unless we bound `N` to an interval. For example:

```
{log}=> size({1/R},N).
```

we get as first answer

```
***WARNING***: non-finite domain
true
Constraint: 1 nin R, M in int(0,sup), size(R,M), set(R), N in int(1,sup)
```

{*log*} provides also a few built-in predicates to support *aggregation functions* over sets. See Table 8.

When applied to an empty set, `smin` and `smax` return `no`, while `sum` returns 0.

Operator	{*log*}	Meaning
sum	`sum(A,N)`	sum all elements of set `A`
min	`smin(A,N)`	the least element of set `A`
max	`smax(A,N)`	the greatest element of set `A`

Table 8: Aggregation functions available in {*log*}

6 Control Predicates

{*log*} provides a number of built-in predicates that can be used by the user to interact with the control mechanisms of the interpreter. We will distinguish these predicates into three categories: general predicates, predicates for controlling constraint solving, predicates for execution monitoring.

General

- `call(G)`, `call(G,C)`: to call a goal `G`, possibly with constraint `C`
- `solve(G)`: same as `call(G)` but all constraints possibly generated by `G` are immediately solved
- `G!`: to make execution of goal `G` deterministic.

Comments about `G!`. {*log*} does not provide the general 'cut' facility of Prolog. In {*log*}, however, it is possible to make the execution of a goal G *determinate* by putting the *cut* symbol just after the goal G. G`!`, where G is any {*log*} goal, is executed exactly as G except that when G succeeds all (possibly none) alternative solutions for G are discarded. Thus, only the first solution for G is computed: if backtracking should later return to this goal, no further solutions will be found.

As an example:

```
{log}=> X in {a,b}!.
X = a
Another solution?  (y/n)y
no
```

whereas the same goal without 'cut' would return the two distinct solutions `X = a`, `X = b`.

Note that the 'cut' operator applies to any {*log*} goal, including disjunctions, conjunctions (e.g., `(X in a,b & Y in {c,d})!`), RUQs, user and system defined predicates.

Constraint solving

- `delay(G,C)`, where `G` and `C` are $\{log\}$ formulas: to delay execution of `G` until either `C` becomes true or the computation ends
- `strategy(`S`)`: to change goal atom selection strategy:
  - S = `cfirst`: select primitive constraints first
  - S = `ordered`: select all atoms in the order they occur
  - S = `cfirst(`*list_of_atoms*`))`: select atoms in *list_of_atoms* just after primitive constraints.

  Default atom selection strategy: `cfirst`
- `noneq_elim/0`, `neq_elim/0`: to deactivate/activate elimination of *neq*-constraints (default: `neq_elim`)
- `noirules/0`, `irules/0`: to deactivate/activate inference rules (default: `irules`)

Comments about `strategy(`S`)`. Atoms in a goal are executed left-to-right in the order they appear in the goal, except that the atomic constraints are executed before any other non-constraint atoms occurring in the goal. This default strategy can be changed by using the built-in predicate `strategy` as shown in the following examples.

Given the goal

```
{log}=>  write(b) & write(X) & X in {a}.
```

we get as answer (using the default strategy `cfirst`)

```
ba
X = a
```

that is, the constraint `X in {a}` is executed first (thus binding `X` to `a`), then the other atoms, `write(b)` and `write(X)`, are taken into account. Conversely, if we give first the goal `strategy(ordered)`, then the same goal as above will produce the answer (`_G2641` is the system generated name of the uninitialized variable `X`)

```
b_G2641
X = a
```

since atoms are executed in the exact order they occur in the goal[2]. Finally, if we give first the goal `strategy(cfirst([nl]))` then any predicate `nl` ("new line") possibly occurring in the next goals will be executed just before any other non-constraint atoms. For example:

```
{log}=>  write(b) & write(X) & nl & X in {a}.
```

we get as answer

```
ba
X = a
```

where the blank line before `ab` is caused by the execution of `nl` before that of the built-in predicates `write`.

Monitoring execution

- `trace(`*Mode*`)`: to activate constraint solving tracing:
  - *Mode* = `sat`: general tracing
  - *Mode* = `irules`: inference rules tracing
- `notrace/0`: to deactivate constraint solving tracing (default)
- `time(G,T)`: to get the CPU time `T` (in milliseconds) required to solve the goal `G`.

7 Prolog-{*log*} communication

7.1 From Prolog to {*log*}

Main predicates

- `setlog/0`: to enter/re-enter the {*log*} interactive environment
- `setlog(G)`, `setlog(G,C)`: to call a {*log*} goal `G`, possibly getting an (output) {*log*} constraint `C`
- `setlog(G,InCLst,OutCLst)`, `setlog_sc(C,InCLst,OutCLst)`: to solve a {*log*} goal `G` (resp. constraint `C`) with a (possibly empty) input constraint list `InCLst` and output constraint list `OutCLst`

[2]With the exception of equalities which are in any case considered before any other non-constraint atoms.

- `setlog(G,T,OutCLst,Res)`: to solve a $\{log\}$ goal `G` with an output constraint list `OutCLst` and a timeout `T` (an integer constant specifying a time in milliseconds). If `setlog(G,T,OutCLst,Res)` returns `no`, then `G` is surely unsatisfiable; otherwise, the value taken by `Res` specifies how the execution of `G` terminates: `success`, if `G` terminates successfully within the time `T`; `time_out`, if `G` does not terminate within the time `T`; `maybe`, if `G` terminates successfully within the time `T` but only after some controls which ensure completeness of the solver has been disabled (see comment below).

Comments about `setlog(G,T,OutCLst,Res)` (calling $\{log\}$ with a timeout). The answer `Res = maybe` comes from the fact that, if $\{log\}$ is not able to check satisfiability of `G` in the given time, it tries to deactivate some rewrite rules (e.g. the elimination of *neq*-constraints) in the hope to become capable of detecting possible inconsistencies. The deactivation of these rules, however, causes the solver to loss completeness. Therefore, if it terminates (within the time `T`) with `no`, then we can anyway conclude that the goal `G` is unsatisfiable; otherwise, we cannot conclude that `G` is surely satisfiable, but only that it *may be* satisfiable.

Note that, due to the fact that the execution of `setlog(G,T,OutCLst,Res)` tries three different strategies to check the satisfiability of `G` before ending definitively, it may take a total amount of time equal to $3 * \texttt{T}$ milliseconds.

Other predicates

- `setlog_clause(Cl)`: to add a $\{log\}$ clause `Cl` to the current $\{log\}$ program
- `setlog_config(list_of_params)`: to modify $\{log\}$ configuration parameters directly from the Prolog environment. Each parameter in `list_of_params` can be:
  - `strategy(S)`: see the control predicate `strategy` in Sect. 6
  - `path(Path)`: `Path` is the pathname of the directory to be used to prefix the name of any file which is loaded in $\{log\}$ through the `consult` predicates (default: `'.'`)
  - `rw_rules(File)`: `File` is the name of the file containing the "filtering rules" (default: `'setlog_rules.pl'`).
- `setlog_rw_rules`: to load the filtering rule library.

7.2 From {*log*} to Prolog

General

- `prolog_call(G)`: to call any Prolog goal `G` from {*log*}.

Specific Prolog built-in predicates The following Prolog built-in predicates are directly available in {*log*} for user convenience.

- `nl/0`
- `ground/1`
- `var/1`
- `nonvar/1`
- `name/2`
- `functor/3`
- `arg/3`
- `=../2`
- `==/2`
- `\==/2`
- `@/2`
- `@==/2`
- `read/1`
- `write/1`
- `call/1`
- `assert/1`
- `consult/1`
- `listing/0`
- `abolish/0`

8 Lists and Multisets

{*log*} provides also lists and multisets (or bags).

Lists are exactly the same as in Prolog. For example, `[a,b,c]` denotes a list of three elements, while `[]` denotes the empty list. Similarly to set terms, a list term of the form `[x / A]` denotes the list which is obtained by concatenating the list `[x]` with the list represented by `A`. As for sets, the list part `A` can be a variable.

Multisets are similar to sets, but the number of repeated elements in a multiset does matter (while it is irrelevant in a set). For example, the two multisets $/[a, b/]$ and $/[b, a, a/]$ are distinct multisets, while $/[a, b/]$ and $/[b, a/]$ are the same multiset. Multisets in {*log*} are represented by terms

Operator	{*log*}	Meaning
list	`list(A)`	A is a list
multiset	`multiset(A)`	A is a multiset

Table 9: `list` and `multiset` constraints

of the form `*(s)` where `s` is any extensional set term, different from `{}`. The empty multiset is represented as `{}`, that is with the same symbol as the empty set. For instance:

- `*({a,b,b})` is the multiset $/[a, b, b/]$, containing one occurrence of a and two occurrences of b
- `*({a,b/R})` is the multiset $/[a, b/] \cup R$

A few primitive set constraints apply to lists and multisets as well. Precisely, the following constraints are used to deal with both lists and multisets:

- `A = B` (equality); `A neq B` (non-equality)
- `x in A` (membership); `x nin A` (non-membership)

Moreover, there are also two new primitive constraints to test whether a term represent either a list or a multiset (see Table 9).

Here are a few examples showing goals involving lists and multisets.

```
{log}= *({a,b}) = *({b,a}).
yes

{log}=> X nin [1,2].
true
Constraint: X neq 1, X neq 2

{log}=> X in L & Y nin L & list(L).
true
Constraint: X in L, X neq Y, Y nin L, list(L)
```

9 Restricted Universal Quantifiers

Restricted Universal Quantifiers (RUQs) are atoms of one of the following forms:

`forall`(X `in` s , G)

`forall`(X `in` s , `exists`(V, G))
`forall`(X `in` s , `exists`($[V_1, \ldots, V_n], G$)

where: X is a variable; s is a term representing either a set or a multiset or a list or an interval `int(a,b)`; V, V_i are variables "local" to G; and G is an arbitrary goal, containing at least one occurrence of X.

Intuitively, an RUQ atom is true if and only if, for all element x in s, the instance of G where x replaces X is true.

RUQs can be used as a convenient way both for

- *iterating* over all elements of the given set s, and;
- *building* all sets s satisfying the given property G.

Example 9.1 *Iterating over a set*

- `print_elements(S)` *prints all elements of the set* `S`*, one for each line.*

```
print_elements(S) :-
      forall(X in S, write(X) & nl).
```

- `all_pair(S)` *is true if all elements of* `S` *are pairs.*

```
all_pair(S) :-
      forall(X in S, exists([X1,X2] X = [X1,X2]).
```

Sample goal:

```
{log}=> all_pair({[peter,ann],[tom,mary],[john,ann]}).

yes.
```

Example 9.2 *Building sets*

- `subset(R,S)` *is true if* `R` $\subseteq$ `S`

```
subset(R,S) :-
      forall(X in R,X in S).
```

When `subset(R,S)` *is called with* `S` *instantiated to a set and* `R` *uninstantiated, it builds in* `R` *all possible subsets of* `S`.

Sample goal:

```
{log}=> subset(R,{mary,ann}).

R = {};
R = {mary};
R = {ann};
R = {mary,ann}.
```

RUQs can occur everywhere ordinary atoms are allowed. In particular, RUQs can be nested at any depth, i.e. the goal part of a RUQ can be an RUQ itself and so on.

Example 9.3 *Nested RUQs*

`disj(S1,S2)` *is true if* `S1` *and* `S2` *are disjoint sets.*

```
disj(S1,S2) :-
      forall(X in S1, forall(Y in S2, X neq Y)).
```

Observe that RUQs are not dealt with as constraints. Executing the RUQ `forall`(X `in` s , G) always starts a computation that iteratively executes the goal G over all elements of the set s. If s and G are not enough instantiated this can lead to an infinite computation. For instance, the goal `forall(X in R,X in {a/S})`, after generating the two solutions `R =` {} and `R =` {`a`}, will go into an infinite loop trying to add the element `a` to the set R which already contains it.

10 Intensional set terms

Besides extensional set terms, that allow a set to be defined by explicitly enumerating all its elements, {*log*} also provides *intensional set terms*, that allow a set S to be defined by providing a condition G that is necessary and sufficient for an element X to belong to S.

Intensional set terms are terms of one of the following forms:

$\{X$: $G\}$

$\{X$: `exists`$(V, G)\}$

$\{X$: `exists`$([V_1, \ldots, V_n], G)\}$

where: X is a variable; s is a term representing either a set or a multiset or a list or an interval `int(a,b)`; V, V_i are variables "local" to G; and G is an arbitrary goal containing at least one occurrence of X.

Intuitively, the intensional set term denotes the set of all instances of X satisfying the formula G.

Example 10.1 *Intensional sets*

- `powerset(S,P)` *is true if* `P` *is the powerset of set* `S`.

```
powerset(S,P) :-
      P = {X : subset(X,S)}.
```

where predicate `subset` *is defined as in Example 9.2.*

Sample goal:

```
{log}=> powerset({a,b},P).
P = {{},{a},{a,b},{b}}.
```

- `cross_product(A,B,CP)` *is true if* `CP` *is the Cartesian product of sets* `A` *and* `B`.

```
cross_product(A,B,CP) :-
      CP = {P: exists([X,Y],P = [X,Y] & X in A & Y in B)}.
```

Sample goal:

```
{log}=> cross_product({a,b},{1,2},CP).
CP = {[a,1],[a,2],[b,1],[b,2]}.
```

Intensional set definitions can be used, in particular, to build a set R from a given set S by applying a given function f to each element of S:

$$R = \{Y \ : \ \texttt{exists}(X, X \ \texttt{in} \ S \ \& \ Y \ \texttt{is} \ f(X)\}.$$

Example 10.2 *Set mapping*

`squares(I,S)` *is true if* `S` *is the set of all the squares of the numbers in* `I`.

```
squares(I,S) :-
      S = {Y: exists(X, X in I & Y is X * X)}.
```

Sample goal:

```
{log}=> squares({2,3,4},S).
S = {4,9,16}.
```

Intensional set terms can occur everywhere ordinary set terms are allowed. Moreover, they can be nested at any depth, i.e. the goal part of an intensional set term can be contain other intensional set terms.

Example 10.3 *Assume predicate* `likes/2` *is defined by the following clauses:*

```
likes(john,beer).
likes(john,wine).
likes(mary,Any_drink) :- Any_drink neq wine.
likes(ann,wine).
```

Sample goal (note the intensional set term as an argument of a predicate):

```
{log}=> disj({P: likes(P,wine)},{P: likes(P,beer)}).
```

The answer will be `no` *(being* `john` *the element in the intersection of the two sets).*

11 The {*log*} library

A number of common predicates, dealing with sets, multisets, and lists, which are not implemented as built-in predicates by the interpreter, are provided as user-defined predicates by the standard {*log*} library `'setloglib.slog'`. The file `'setloglib.slog'` can be loaded as part of any {*log*} program using the built-in predicate `consult-lib/0`.

Below we list most of the predicates currently contained in `'setloglib.slog'`.

Dealing with sets

- `powerset(S,PS)`: powerset ($\mathtt{PS} = 2^{\mathtt{S}}$)
- `cross_product(A,B,CP)`: the Cartesian product of sets `A` and `B`
- `list_to_set(L,S)`: `S` is the set of all elements of the list `L`
- `int_to_set(I,S)`: `S` is the set of all elements of the interval `I`
- `dint_to_set(A,B,S)`: same as `int_to_set/2` but delayed if interval bounds are unknown
- `diff1(S,X,R)`: equivalent to `diff(S,X,R)` but more efficient
- `eq(T1,T2)`: syntactic unification between terms `T1` and `T2`
- `bun(S,R)`: generalized union: `R` is the union of all elements of the set of sets `S`
- `binters(S,R)`: generalized intersection: `R` is the intersection of all elements of the set of sets `S`

Dealing with multisets

- `int_to_bag(I,M)`: `M` is the multiset of all elements of the interval `I`
- `bag_to_set(M,S)`: `S` is the set of all elements of the bag `M`
- `msize(S,N)`: multiset cardinality ($\mathtt{N} = /\mathtt{S}/$)

Dealing with lists

- `prefix(P,L)`: list `P` is a prefix of list `L`
- `sublist(Sb,L)`: list `Sb` is a sublist of list `L`
- `take(N,L,NewL)`: list `NewL` consists of the first `N` elements of list `L`
- `drop(N,L,NewL)`: `NewL` is `L` with its first `N` elements removed
- `extract(S,L,NewL)`: `S` is a set of integer numbers, `L` is a list of elements of any type, and `NewL` is a list containing the i-th element of `L`, for all i in `S` (e.g., `extract(4,2,[a,h,g,m,t,r],L)` returns `L = [h,m]`)
- `filter(L,S,NewL)`: `L` is a list, `S` is a set, and `NewL` is a list containing the elements of `L` that are also elements of `S` (e.g., `filter([a,h,g,m,t,r],m,h,s,L)` returns `L = [h,m]`)

Remark 11.1 *The problem of redundant solutions is still an open problem. When the involved sets do not contain uninitialized variables the implementation takes care of avoiding as much as possible redundand solutions. When, in contrast, some uninitialized variable is involved there may be cases in which redundand (namely, equivalent or less general) solutions are returned. For example, as a limit case, the goal*

`{{a/N}} = {{b/N},{a/N}}`

returns over 100 repeated solutions of the form

`N = {a,b/_1}`.

While many work can be surely done to alleviate this problem (and possibly it will be exploited in next releases of {log}), it is hard to devise a general solution to completely remove it. For now, one can only try to partly overcome this incovenience by using the 'cut' operator to make the computation deterministic whenever non-determinism is not strictly required.

Acknowledgments

As mentioned in Section 1 the original design and implementation of the {*log*} interpreter is due to Agostino Dovier and Enrico Pontelli, under the supervision of Eugenio Omodeo and Gianfranco Rossi. Several other people, however, contributed at various extent to the development of next versions of the {*log*} interpreter. Among them, we wish to thank in particular: Bruna Bazzan, Silvia Manzoli, Silvia Monica, Carla Piazza and Laura Gelsomino.

Bibliography (ordered by date)

[1] A. Dovier, E. G. Omodeo, E. Pontelli, G. Rossi. {log}: A Logic Programming Language with Finite Sets. In *Proc. 8th Int'l Conf. on Logic Programming* (K. Furukawa, ed.), The MIT Press, 1991.

[2] A. Dovier, G. Rossi. Embedding Extensional Finite Sets in CLP. In *Proceedings of 1993 International Logic Programming Symposium* (D. Miller, ed.), The MIT Press, 1993, pp. 540–556.

[3] P. Bruscoli, A. Dovier, E. Pontelli, G. Rossi. Compiling Intensional Sets in CLP. In *Logic Programming: Proceedings of the Eleventh International Conference* (P. Van Entenryck, ed.), The MIT Press, 1994, pp. 647–661.

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

[5] A. Dovier, C. Piazza, E. Pontelli, G. Rossi. On the Representation and Management of Finite Sets in CLP-languages. In *Proceedings of 1998 Joint International Conference and Symposium on Logic Programming* (J. Jaffar, ed.), The MIT Press, 1998, pp. 40-54.

[6] Dovier, A., Policriti, A., Rossi, G.: A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms. Fundam. Inform. **36**(2-3) (1998) 201–234

[7] Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM Trans. Program. Lang. Syst. **22**(5) (2000) 861–931

[8] Dovier, A., Pontelli, E., Rossi, G.: Set unification. Theory Pract. Log. Program. **6**(6) (November 2006) 645–701

[9] Dal Palú, A., Dovier, A., Pontelli, E., Rossi, G.: Integrating finite domain constraints and CLP with sets. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming. PPDP '03, New York, NY, USA, ACM (2003) 219–229

[10] Cristiá, M., Rossi, G., Frydman, C.S.: {log} as a test case generator for the Test Template Framework. In Hierons, R.M., Merayo, M.G., Bravetti, M., eds.: SEFM. Volume 8137 of Lecture Notes in Computer Science., Springer (2013) 229–243

[11] Cristiá, M., Rossi, G., Frydman, C.S.: Adding partial functions to constraint logic programming with sets. TPLP **15**(4-5) (2015) 651–665