{*log*} User's Manual

Version 3.3

GIANFRANCO ROSSI
Università di Parma
Dip. di Matematica
Via M. D'Azeglio, 85/A
43100 Parma (Italy)
gianfr@prmat.math.unipr.it

September 11, 2000

Abstract

This is the first 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 SICStus Prolog and it is based on the prototype developed at the University of Udine by Agostino Dovier and Enrico Pontelli. The full Prolog code of the {*log*} interpreter is available at the {*log*} WEB page
http://www.math.unipr.it/~gianfr/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 [2, 3, 8, 7].

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.

In addition, a set can be designated in {*log*} by stating a property that must be satisfied by all its elements (*intensional sets*), by using simple *intensional set terms*.

The language also provides a number of basic primitive operations for set management: *in* (set membership), = (equality), along with their negative counterparts, *nin*, *neq*. All these operations are dealt with as *constraints*, and, hence, they can be used with no concern about the instantiations of their arguments, and irreducible constraints can occur in the computed answer.

Furthermore, {*log*} provides Restricted Universal Quantifiers (RUQ) for aggregate operations.

Other set-theoretic operations, such as intersection, difference, powerset, etc., can be effectively programmed in the language itself and are provided as part of the standard {*log*} library.

{*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 SICStus Prolog. Therefore to run it a SICStus Prolog installation (even a relatively old one) is required. However, the {*log*} interpreter uses very few facilities that are specific of SICStus Prolog and therefore can be ported with very limited effort to any other Prolog system.

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 "Tesi di Laurea" degree at the Department of Mathematics and Computer Science of the University of Udine in 1991 [1]. Later on, the {*log*} interpreter was revised at various times by Gianfranco Rossi, who is still maintaining the current version of the interpreter.

The Prolog code of the {*log*} interpreter is available at the {*log*} WEB page http://www.math.unipr.it/~gianfr/setlog.Home.html. At the same page, you can find also the postscript file of this manual, the {*log*} library file containing the {*log*} definitions of the most common operations on sets (not provided as built-in in {*log*}), 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 `interpreter.3.3`. To start working with the interpreter, invoke SICStus Prolog and then load the {*log*} interpreter, e.g., by giving the directive

?- `consult('interpreter.3.3')`.

The {*log*} interpreter is loaded into the SICStus Prolog program database and its execution started. {*log*} tells you that it is ready to accept a goal by showing on the standard output the message

`{log}=>`

Any line

`{log}=> G.`

where `G` is a {*log*} goal, is called a (top level) directive.

To exit from the {*log*} interpreter give the directive

`{log}=> halt.`

The interpreter will ask you to confirm your decision. Type `y` to confirm, `n` (or anything else different from `y`) to cancel your request.

Once you are in the SICStus execution environment and you want to re-enter the {*log*} environment give the directive

?- `setlog.`

The execution of this goal immediately moves you to the $\{log\}$ environment, showing the $\{log\}$ prompt on the standard output.

Note that, in the current implementation, any run-time error possibly detected by SICStus Prolog while executing the $\{log\}$ interpreter (e.g., an instantiation error in the `is` predicate) forces execution to leave the $\{log\}$ execution environment. To re-enter $\{log\}$, call the `setlog` goal.

To load a $\{log\}$ program into the $\{log\}$ environment you can use the `consult/1` predicate. If the program is stored in file `sample_program` then the directive

`{log}=> consult(sample_program).`

will load it. 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. To re-enter it (after the error has been repaired) use the `setlog` goal.

Note that the `consult` predicate is used also to load the $\{log\}$ libraries. If `setlog_lib` is the name of the file containing the libraries then the directive

`{log}=> consult(setlog_lib).`

allows you to consult them (see Section 9 for the complete list of the predicates defined in the standard $\{log\}$ library file). Actually, library predicates are dealt with as other user defined predicates.

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/0` and `listing/0` refer to the whole set of stored clauses (both user defined and those of the library predicates).[1]

[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.

Finally, simple help facilities are provided in the form of three built-in predicates: `help` (or simply `h`) provides general help information on $\{log\}$; `help(built)` provides information on available $\{log\}$ built-in predicates; `help(lib)` provides information on the $\{log\}$ predicates defined in file `setlog_lib`.

3 Syntax

$\{log\}$ uses most of the syntactic conventions and notations of standard Prolog. In particular, variables begin with an upper-case letter, while constants, function and predicate symbols begin with a lower-case letter. In this section we briefly describe the syntactic differences between $\{log\}$ and standard Prolog, and provide some hints on their (informal) semantics.

3.1 Terms

Terms in $\{log\}$ have the usual form as in standard Prolog. The only difference is the presence of a few interpreted function symbols used to build extensional and intensional sets.

3.1.1 Set terms

The following two function symbols are used to build extensional sets, that is sets defined by explicitly enumerating all their elements:

- $\{\}$, of arity 0, interpreted as the empty set;
- $\{\cdot/\cdot\}$, of arity 2, interpreted as the set constructor: informally speaking, $\{t\,/\,s\}$ is the set obtained by adding the element t to the set s, that is $\{t\} \cup s$.

Terms whose main functor is either $\{\}$ or $\{\cdot/\cdot\}$ are called *set terms*. Precisely, set terms are recursively defined as follows:

> Variables and $\{\}$ are <u>*set terms*</u>. If s is a set term and t is any term then $\{t\,/\,s\}$ is a <u>*set term*</u> (s is said the *tail* part of the set term).

As a syntactic convenience, the set term

$$\{t_1\,/\,\{t_2\,/\ldots\{t_n\,/\,s\}\ldots\}\}$$

where $t_1,\ldots,t_n$ are terms, and s is a set term, can be denoted simply as

$$\{t_1, t_2, \ldots, t_n \,/\, s\}$$

and, if s is the empty set $\{\}$, as

$$\{t_1, t_2, \ldots, t_n\}.$$

Example 3.1 *Ground set terms*

- $\{\texttt{mary}, \texttt{ann}, \texttt{peter}\}$ *a set term designating the set whose members are (the atoms)* `mary`*,* `ann`*, and* `peter`
- $\{\texttt{person}(\texttt{mary}, \texttt{white}), \texttt{person}(\texttt{peter}, \texttt{brown})\}$
- $\{\{\texttt{mary}, \texttt{ann}\}, \{\texttt{peter}\}, \{\}\}$ *a set term designating a set containing three sets (set terms can be nested at any depth)*[2]
- $\texttt{persons}(\{\texttt{mary}, \texttt{ann}, \texttt{peter}\})$ *a term containing a set term, but not itself a set term.*

Both ground and *non-ground* set terms are allowed, i.e., sets are allowed to be *partially specified*: a set term can contain variable elements as well as the tail part of the set term can be a variable.

Example 3.2 *Non-ground set terms*

- $\{\texttt{X}, \texttt{Y}, \texttt{Z}\}$*: a set with three variable elements, not necessarily all distinct;*
- $\{\texttt{Any}\}$*: a singleton set;*
- $\{\texttt{mary}, \texttt{john} \,/\, \texttt{Others}\}$*: a set with two elements and something else, i.e. the set* $\{\texttt{mary}, \texttt{john}\} \cup \texttt{Others}$*—only some of the elements of the set are explicitly enumerated, while the rest of the set is left unspecified.*

According to the interpretation of set terms as sets, the language guarantees that order and repetitions in a set term do not matter. Precisely, this means that in the underlying set theory the $\{\,\cdot\,/\,\cdot\,\}$ operator must exhibit the *(left) permutativity property*, $\{x, y \,/\, z\} = \{y, x \,/\, z\}$, and the *(left) absorption property*, $\{x, x \,/\, z\} = \{x \,/\, z\}$.

Thus, for example, $\{\texttt{peter}, \texttt{mary}, \texttt{peter}, \texttt{ann}\}$ denotes the same set as $\{\texttt{mary}, \texttt{ann}, \texttt{peter}\}$. Note that duplicates do not occur in the set, but they may occur in the concrete representations of it, i.e., in the set term denoting that set. This corresponds also to the observe that the set term $\{x \,/\, y\}$,

[2] The kind of sets which can be represented in this way are the so called *hereditarily finite sets.*

which denotes the set $\{x\} \cup y$, does not necessarily require $x \notin y$ to hold. As we will see in next sections, restrictions on y, if needed, have to be explicitly stated using non-membership constraints.

The only non-variable terms allowed in the tail part of a set term are set terms. Any other term built using $\{\cdot / \cdot\}$ and not satisfying this condition (e.g., $\{\mathtt{a} / \mathtt{b}\}$)[3] is considered illegal (a 'wrong set term' error message may be produced).[4]

Set terms are stored internally using the `with` function symbol, written infix, letf-associative. The set term

$$\{t_1, t_2, \ldots, t_n \,/\, s\}$$

is represented using the `with`-notation as

$$s \,\mathtt{with}\, t_n \,\mathtt{with} \ldots \mathtt{with}\, t_2 \,\mathtt{with}\, t_1.$$

For the sake of flexibility, the `with`-notation is allowed to be used also at the top-level, and can be freely mixed with the representation of sets using curly brackets. For example, the terms

$$\mathtt{f(\{\}\,with\,b\,with\,a,\ \{1,2\})}$$

and

$$\mathtt{\{a\,/\,\{\}\,with\,b\,with\,\{c\}\}}$$

are accepted as syntactically correct by the $\{log\}$ interpreter.

A difference between using the `with`-notation and the curly brackets notation is that with the latter the $\{log\}$ interpreter automatically removes ground duplicate terms possibly encountered in set terms during their input/output, whereas duplicates are always maintaned with the `with`-notation.

3.1.2 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 *intensionally*, that is, by providing a condition G that is necessary and sufficient for an element X to belong to S.

An *intensional set term* is a term of the form:

[3] An approach in which, on the contrary, such terms (called *colored* sets) are allowed has been considered in [4].

[4] In the current implementation, syntax checks on set tails are performed only during the input of programs and goals and the output of results. No check on the tail part of a set is performed at run-time. Hence, it can happen that a variable set tail gets bound to a non-set term during execution. This may lead, in general, to erroneous results. We plan to remove this kind of nuisance in future releases of our language by moving to a real multi-sorted logic and, accordingly, performing all possible sort checks.

$$\{X : G\}$$

where the main function symbol is $\{\cdot\}$, of arity 1, : is a function symbol of arity 2, used infix, X is a variable, and G is an arbitrary $\{log\}$ goal containing X.

The intuitive meaning of such a term is: "the set of all instances of X satisfying the formula G".

The goal G in an intensional set term can be any $\{log\}$ goal, in particular, one containing a built-in $\{log\}$ predicate, or another intensional set term (in fact, intensional set terms can be nested at any depth), or an arbitrary conjunction of $\{log\}$ atoms. Anyway, G must contain at least one occurrence of the control variable X.

As a special case, the goal G can be an existentially quantified formula, that is the intensional set term can take one of the forms

$$\{X : exists(Y, G')\}$$

or

$$\{X : \texttt{exists}([Y_1, \ldots, Y_n], G')\}$$

In this cases, variables $Y, Y_1, \ldots, Y_n$ will be dealt with as local to G'

Intensional set terms can occur everywhere ordinary terms are allowed. In particular, the head of a clause can contain an intensional set term, or a RUQ atom (cf. next section) can contain an intensional set term.

Example 3.3 *Intensional set terms*

- `{X: likes(X,wine)}`*;*
- `{Pair: exists([X,Y], Pair = [X,Y] & likes(X,Y))}`*;*[5]
- `intersection(S1,S2,{X: X in S1 & X in S2})`*:*
  an atom containing an intensional set term;
- `{X: X in {Y: between(Y,1,N)} & is_prime(X)}`*:*
  nested intensional set terms.

3.2 Atoms, clauses, and goals

Atoms, clauses and goals in $\{log\}$ have the usual form as in standard Prolog. The only differences are the presence in $\{log\}$ of a few new built-in predicates for set manipulation, the availability of special atoms called Restricted

[5] See Sect. 3.2.3 for the syntactic form of $\{log\}$ clauses.

Universal Quantifiers, and some syntactic sugar to represent logical connectives in clause bodies and goals. Furthermore, note that names of program defined predicates in $\{log\}$ are not allowed to begin with the prefix `setlog`.[6]

3.2.1 Predefined predicates

The following predefined predicates (all of arity 2, written infix) are provided for set manipulation:

- t = s (**Equality**): true if there is a substitution σ s.t. t^σ is equal to s^σ modulo the permutativity and absorption properties mentioned above;
- t `in` s (**Membership**): true if there is a substitution σ s.t. t^σ belongs to s^σ, i.e., there is an element r in s^σ such that r^σ is equal to t^σ modulo the permutativity and absorption properties;[7]
- t `neq` s (**Inequality**); true if there is no substitution σ s.t. t^σ is equal to s^σ modulo the permutativity and absorption properties;
- t `nin` s (**Non-membership**); true if there is no substitution σ s.t. t^σ belongs to s^σ.

In these predicates t and s are terms. In particular, s can be any term, hence, also a non-set term. If this is the case, the interpretation of s is such that no element belongs to s (like for the empty set).

Set manipulation atoms can occur in clause bodies and goals everywhere ordinary atoms (based on both built-in and user defined predicates) are allowed.

3.2.2 Restricted Universal Quantifiers

A $\{log\}$ atom can be also a Restricted Universal Quantifiers (RUQs), that is a formula of the form:

$$\texttt{forall}(X \texttt{ in } s\,,\, G)$$

where X is a variable, s a term, and G an arbitrary goal containing X.

[6] Actually only names of the form `setlogRUQ`xx and `setlogSF`xx, where xx is a system generated number, are used at present by the $\{log\}$ interpreter, to implement Restricted Universal Quantifiers and intensional set formers, respectively. However other names prefixed by `setlog` could be used for internal purposes in future implementations.

[7] More precisely, one should say: if there is a substitution σ s.t. the object associated by the selected interpretation to t^σ belongs to the object associated by this interpretation to s^σ.

This form stands for the quantified implication

$$\forall X((X \in s) \to G).$$

RUQs atoms can occur in clause bodies and goals everywhere ordinary atoms are allowed.

Similarly to intensional set terms, also the goal G in a RUQ can be any $\{log\}$ goal, in particular, one containing another RUQ (RUQs can be nested at any depth). Like intensional set terms, G must contain at least one occurrence of the control variable X. Moreover, the goal G can be an existentially quantified formula, that is the RUQ can take one of the forms

$$\texttt{forall}(X \ \texttt{in} \ s\,,\ \texttt{exists}(Y, G'))$$

or

$$\texttt{forall}(X \ \texttt{in} \ s\,,\ \texttt{exists}([Y_1, \ldots, Y_n], G')).$$

In these cases, the variables $Y, Y_1, \ldots, Y_n$ will be dealt with as local to G'.

Example 3.4 *Restricted Universal Quantifiers*

- `forall(X in {1,2,4}, p(X,Y))`*;*
- `forall(X in {f(a),f(b),f(c)}, exists(Y,X = f(Y)))`*;*
- `forall(X in S1, X in S2 & p(X,Y))`*;*
- *(nested RUQ's)* `forall(X in {1,2}, forall(Y in S, p(X,Y)))`.

RUQs can be freely combined with intensional set terms, that is, the goal part of an intensional set term can contain a RUQ, as well as the set in a RUQ can be defined intensionally rather than extensionally. For example:

`forall(X in {Y : between(Y,1,M)}, p(X)).`

3.2.3 Clauses and goals

The only syntactic differences w.r.t. standard Prolog clauses and goals are:

- '`&`' is used in place of '`,`' to represent goal conjunction;
- '`or`' is used in place of ';' to represent goal disjunction.

Example 3.5 *Clauses and goals*

- `singleton({_}).`

- `in_difference(X,Set1,Set2) :-`
  `X in Set1 & X nin Set2.`

- `{log}=> forall(X in S, X = a or X = b) & S neq {}.`

3.2.4 Operator precedences

The following operator declarations are provided for standard {*log*} operators:

```
:- op(1000, xfx, /).
:- op(980, xfx, :).
:- op(970, xfy, or).
:- op(950, xfy, &).
:- op(900, fy, neg).
:- op(800, xf, !).
:- op(700, xfx, in).
:- op(700, xfx, neq).
:- op(700, xfx, nin).
:- op(650, yfx, with).
```

4 Program execution

A top level goal which has more than one computed answer will generate, as usual, one answer at a time. After the first answer is shown on the standard output the system will ask the user whether he/she wants to get the other answers or not:

`'Another solution? (y/n)'`

Typing `y` followed by END-OF-LINE causes the system to look for (possible) alternative solutions.

Each computed answer for a goal G is composed of two (possibly empty) parts: a sequence of equalities which represents the computed substitution for the given goal; a list of irreducible constraints (see Section 6), prefixed by the keyword `Constraints:`.

Example 4.1 *Computed answer*

```
{log}=> a nin {X,Y/S} & Y in {a,b}.
Y = b - Constraints: [X neq a, a nin S]
```

Note that whenever either the list of irreducible constraints or the sequence of substitutions are empty they are not printed at all.

$\{log\}$ 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.

Example 4.2 *Determinate goals*

```
{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.

Also, note that using the 'cut' operator one can emulate more complex usages of Prolog's 'cut' facility, such as, for instance, the if_then_else control structure:

Prolog program:

```
p(X) :- q(X),!,r(X).
p(X) :- s(X).
```

$\{log\}$ program:

```
p(X) :- (q(X) & R = 1 or R = 2)! & p_cont(R,X).
p_cont(1,X) :- r(X).
p_cont(2,X) :- s(X).
```

Atoms in a goal are executed left-to-right in the order they appear in the goal (like in Prolog), except that the atomic constraints are executed before any other non-constraint atoms occurring in the goal. This reordering can be useful whenever the goal involves atoms that work by side-effects or have special requirements on the instantiation of their variables. For example, the goal

```
write(X) & X in {a,b}
```

will successfully output the constants `a` and `b`, disregarding the order of the two atoms in the goal.

There are cases, however, in which it can be more convenient to execute both constraint and non-constraint atoms in a prefixed order. Tipically, this can be used to reduce the number of redundant solutions possibly generated by solving constraints with uninstantiated variables. To force also atomic constraints to be executed as encountered in the text of the goal a viable solution is to replace atomic constraints with calls to equivalent program defined predicates or to use the `call` built-in predicate. For example, in the goal

```
read(X) & X = {A/X} & X = {B/X},
```

to force the `read` predicate to be executed before executing the rest of the goal,[8] we can rewrite it as:

```
read(X) & aux(X,A,B)
```

where `aux` is defined in the program as

```
aux(X,A,B) :- X = {A/X} & X = {B/X},
```

or, using `call`, we can write the goal as

```
read(X) & call(X = {A/X} & X = {B/X}).
```

Finally, observe that in its current version $\{log\}$ does not provide any support for *commands*, that is directives that start with the symbol ':-' and executed as encountered in the consulted file. To use such kind of directives one should put them into a separate Prolog file which is consulted from within the $\{log\}$ environment using the `prolog_call` built-in predicate (see also Section 10).

5 Set unification

Basic operations on sets include primarily the ability to compare two sets. When considering the more general case where also non-ground and partially specified sets are allowed this operation amounts to solve a non-trivial *set unification problem.*

Set unification differs from standard unification in that the former must account for the properties of the set constructor operator (namely, the permutativity and absorption properties).

Example 5.1 *Without tail variables*

- *Using permutativity:*

[8] Without any special care, the equality part of the goal is executed first—with the `X` variable uninstantiated—generating 13 redundant solutions.

- `{log}=> {john,mary} = {mary,john}`

  `yes`

- `{log}=> {X,Y} = {mary,john}`

  `X = mary, Y = john`*;*
  `X = john, Y = mary`*.*

  Note that there are two solutions, neither of which is more general than the other (non-uniqueness of mgu's).

* *Using absorption:*

  - `{log}=> {mary} = {mary,mary}`

    `yes`

  - `{log}=> {X} = {mary,mary}`

    `X = mary`*.*

* *Using permutativity and absorption:*

  - `{log}=> {X,Y,Z} = {mary,mary,john}`

    `X = mary, Y = john, Z = mary`*;*
    `X = john, Y = mary, Z = john`*;*
    `X = mary, Y = mary, Z = john`*;*
    `X = john, Y = john, Z = mary`*;*
    `X = john, Y = mary, Z = mary`*;*
    `X = mary, Y = john, Z = john`*.*

Example 5.2 *With tail variables*

* *Distinct tail variables:*

  - `{log}=> {john/R} = {mary/S}`

    `R = {mary/_1}, S = {john/_1}`*.*

    where `_1` *is a new variable.*

  - `{log}=> {john/R} = {john/S}`

    `R = S`*;*
    `R = {john/S}`*;*
    `S = {john/R}`*.*

* *Same tail variable:*

– {log}=> `{john/S} = {mary/S}`

`S = {john,mary/_1}`.

- *A special case:*

  – {log}=> `S = {john/S}`

  `S = {john/_1}`.

  Note that this goal has no solution using standard unification due to the cycle detected by the occur-check test.

6 Constraints

{*log*} can be seen as an instance of the general CLP scheme [4]. Basic set operations (namely, `in` and `nin`), along with disequality (`neq`), are dealt with as *constraints.*

Satisfiability of the current set of active constraints is checked at each step during the computation. Irreducible constraints (i.e., those which are not enough instantiated) are passed to the next computation step, waiting for a sufficint instantiations of variables occurring in them. Constraints which, at the end of the computation, are still irreducible are returned as part of the computed answer.

Viewing these operations as constraints definitively enhances program declarativeness (the order of execution of constraints does not matter) and support a very flexible way of dealing with partially specified sets.

6.1 Irreducible constraints

Irreducible constraints are defined to be conjunctions of atomic constraints of one of the forms:

- t `nin` X, X variable, and X dos not occur in t;
- X `neq` t, X variable, and X dos not occur in t.

Irriducible constraints are not further processed and, hence, are returned unaltered as part of the computed answer. As a matter of fact, it is guaranteed that an irreducible constraint is always satisfiable, that is one can find a suitable instantiation of its variables that makes it true. The *computed answer*, therefore, is defined as a pair:

- *substitutions*, i.e, conjunctions of equations of the form $X = t$, X variable, and X does not occur elsewhere;
- *irreducible constraints.*

6.2 Examples

Example 6.1 *Membership*

- `{log}=> X in S.`

  `S = {X/_1}.`

  Note that, differently from other constraints, membership constraints are always eliminated by reducing them to equalities.

- `{log}=> X in {john,mary/Others}.`

  ```
  X = john, Others = _1;
  X = mary, Others = _1;
  Others = {X/_1}.
  ```

Example 6.2 *Negated equality*

- `{log}=> person(Name,white) neq person(peter,Surname).`

  ```
  Constraints: [Name neq peter] ;
  Constraints: [Surname neq white].
  ```

- `{log}=> X neq mary & {X,Y} = {mary,ann}.`

  `X = ann, Y = mary.`

  Note that without the `neq`*-constraint in the given goal, also* `X = mary, Y = ann` *would be a correct solution. Also note that the order of atomic constraints in the goal does not matter.*

Example 6.3 *Negated membership*

- `{log}=> X in S & X nin S.`

  `no.`

- `{log}=> mary nin {X,Y/S}.`

  `Constraints: [X neq mary, Y neq mary, mary nin S].`

- `{log}=> mary nin {X,Y/S} & Y in {mary,john}.`

  `Y = john - Constraints: [X neq mary, mary nin S].`

- `{log}=> {mary} = {X/S} & X nin S.`

  `X = mary, S = {}.`

  Note that without the `nin`*-constraint in the goal, also* `X = mary, S = {mary}` *would be a (distinct) correct solution. Therefore, non-membership constraints can be used in this way to explicitly avoid considering repetitions whenever one wants to access, through set unification, to the elements of a set.*

Example 6.4 *A simple program*

- `in_difference(X,Set1,Set2)` *is true when* $\texttt{X} \in \texttt{Set1} \setminus \texttt{Set2}$.

  ```
  in_difference(X,Set1,Set2) :-
       X in Set1 & X nin Set2.
  ```

  Sample goal:

  ```
  {log}=> in_difference(X,{mary,ann},{john,mary}).
  X = ann.
  ```

  Sample goal:

  ```
  {log}=> in_difference(X,S,{john,mary}).
  S = {X/_1} - Constraints: [X neq john, X neq mary].
  ```

7 Restricted Universal Quantifiers

Restricted Universal Quantifiers (RUQs) are formulae of the form:

$$\texttt{forall}(X \texttt{ in } s\,,\, G)$$

where X is a variable, s a term, and G an arbitrary goal containing X, which can occur everywhere ordinary atoms are allowed.

An RUQ $\texttt{forall}(X \texttt{ in } s\,,\, G)$ 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 7.1 *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 7.2 *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 be nested at any depth, i.e. the goal part of a RUQ can be an RUQ itself and so on.

Example 7.3 *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.

8 Intensional set terms

In the practice of mathematics very often a set S is denoted *intensionally*, that is, by providing a condition φ that is necessary and sufficient for an element X to belong to S. This *intensional* definition of a set is achieved in $\{log\}$ by use of an *intensional set term* of the form:

$$\{X \ : \ G\}$$

whose intuitive meaning is: "the set of all instances of X satisfying the formula G".

The availability of intensional set terms considerably enhances the expressive power of the language, and provides the user with a powerful tool for supporting a more declarative programming style.

Example 8.1 *Intensional sets*

- `intersection(S1,S2,S3)` *is true if* $\texttt{S3} = \texttt{S1} \cap \texttt{S2}$.

```
intersection(S1,S2,S3) :-
      S3 = {X: X in S1 & X in S2}.
```

- `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 7.2.*

  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 8.2 *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}.
```

As already mentioned, intensional set terms can be freely combined with RUQs and other intensional set terms (e.g., the goal part of an intensional set term can contain another intensional set term, or an intensional set term can be used within a RUQ, and so on).

Example 8.3 *Assume predicate* `disj/2` *is defined as in Example 7.3. 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).
```

Then we can be interested in knowing for instance whether there are people who like both wine and beer.

Sample goal:

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

With the above definition of `likes`*, the answer will be* `no` *(being* `john` *the element in the intersection of the two sets).*

Intensional sets are implemented in {*log*} by replacing each intensional set term with the corresponding {*log*} goal and clauses which are able to perform the necessary set collection operations [5]. Thus, for example, the following atom

```
S = {X : p(X,Y)}
```

occurring in a $\{log\}$ goal is always replaced—at compile time—by the equivalent conjunction

```
setlogSF_1(S,Y) & Y = S
```

where `setlogSF_1` is defined by the following clauses that are also added to the program:

```
setlogSF_1(S,Y) :-
   forall(X in S,p(X,Y)) &
   neg setlogSF_2(S,Y).
setlogSF_2(S,Y) :-
   Z nin S &
   p(Z,Y).
```

These clauses implement the *set grouping* mechanism that collects in the set `S` all the instances of `X` such that `p(X,Y)` is true, rejecting all those `S` which are only partial, i.e., they do not contain all the instances of `X`. This corresponds to the idea that in $S = \{X : p(X,Y)\}$, $p(X,Y)$ represents both a necessary and sufficient condition for X to belong to S.

To reject partial sets the implementation of set grouping is required to use some form of negation. `neg` P implements a simple Negation As Failure (NAF) mechanism. That is, `neg` P fails if the goal P has a solution, and succeeds otherwise.

Unfortunately, `neg` suffers of all the well-known problems of Negation As Failure. More generally, it is not able to deal correctly with goals whose solutions contain constraints (in particular, equality constraints).[9] This in turn may lead to incompleteness when computing solutions for goals containing intensional sets. For example, assume that predicate `p/2` is defined by the two clauses

```
p(a,1). p(b,2).
```

Then the answers to the goal `S = {X : p(X,Y)}` will be

```
Y = 1, S = {a};
Y = 2, S = {b}.
```

Actually, also the solution

```
S = {}- Constraints: [Y neq 1, Y neq 2]
```

[9] `neg` in $\{log\}$ is intended to be used maily to implement intensional sets, not as a general control facility. This is also justified by the fact many usages of negation can be replaced in $\{log\}$ by the (safe) use of the negative constraints `neq` and `nin`

is a correct one, but it is not computed. To obtain also such kind of solutions a different form of negation, e.g., *Constructive Negation*, should be used in place of NAF to implement set grouping. (see, for instance, [6, 11])

However, observe that, due to the particular usage of negation within set grouping clauses, problems with intensional terms like the one mentioned above may arise only if the goal G in $\{X : G\}$ contains uninstantiated free variables when set grouping is performed. Moreover, also in this case, problems are limited to the cases in which either the set denoted by $\{X : G\}$ is the empty set or the execution of G instantiates one of its free variables to a non-ground term (possibly containing a constrained variable). For example, with the definition of `likes` given in Example 8.3, the goal

```
S = {X: likes(X,Y)}
```

also admits the solution

```
S = {mary} - Constraints: Y neq wine, Y neq beer
```

but this solution is never found. In all other cases, the computation of $\{X : G\}$ can be performed safely—provided $\{X : G\}$ denotes finite sets.[10]

9 The {*log*} library

The usual set-theoretic operations, such as union, intersection, etc., can be easily programmed in {*log*}. The definition of many of these operations is provided as part of the standard {*log*} environment in a file named `setlog.lib`. It can be loaded as part of any {*log*} program using the built-in predicate `consult`, as any other user defined program.

Below we show the complete definition of many of the most important predicates contained in `setlog.lib`[11]

- `less(R,X,S)`: true if $S = R\backslash\{X\}$

```
less(R,X,S) :-
      R = {X/S} & X nin S.
```

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

[10] {*log*} does not perform any (partial) check to try to find out whether an intensional set term denotes an infinite sets. This task is completely let to user's accurancy. To this regard, notice that, given $\{X : G\}$ if G does not contain any free variable and X is instantiated to some non-ground term then the denoted set is necessarily an infinite set. Again, in this abnormal case, the application of NAF within set grouping may turn out to be not safe.

[11] For a discussion about primitive vs. non-primitive set operations see [10].

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

- `union(R,S,T)`: true if $T = R \cup S$

```
union({},X,X).
union(R,S,{A/U}) :-
      less(R,A,RR) & union(RR,S,U).
```

- `int(R,S,T)`: true if $T = R \cap S$

```
int({},_,{}).
int(R,S,{X/U}) :-
      less(R,X,RR) & X in S & int(RR,S,U).
int(R,S,T) :-
      less(R,X,RR) & X nin S & int(RR,S,T).
```

- `diff(R,S,T)`: true if $T = R \backslash S$

```
diff({},_,{}).
diff(R,S,T) :-
      less(R,X,RR) & X in S & diff(RR,S,T).
diff(R,S,{X/U}) :-
      less(R,X,RR) & X nin S & diff(RR,S,U).
```

- `disj(S1,S2)`: true if `S1` and `S2` are disjoint sets

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

- `powerset(S,PS)`: true if `PS` is the power set of `S`

```
powerset(S,PS) :-
      PS = {SS : subset(SS,S)}.
```

- `size(S,N)`: true if `N` is the number of elements in the set denoted by `S`

```
size(S,N) :-
      in_size(S,0,N).
in_size({},N,N).
in_size(S,N,M) :-
      N neq M &
      less(S,_,SS) &
      NN is N + 1 &
      in_size(SS,NN,M).
```

Besides the predicates listed above, the current version of `setlog_lib` provides also the definition of the following predicates:

- `nsubset(S1,S2)`: not subset—true if `S1` $\not\subseteq$ `S2`
- `ssubset(S1,S2)`: strict subset—true if `S1` $\subset$ `S2`
- `nint(R,S,T)`: not intersection—true if `T = R` $\not\cap$ `S`
- `ndiff(R,S,T)`: not difference—true if `T = R` $\not\emptyset_3\setminus$ `S`
- `sdiff(R,S,T)`: symmetric difference—true if `T = R` $\triangle$ `S.`

Furthermore, `setlog_lib` provides also an alternative implementation of some of the basic set-theoretic operations that behaves exactly as the previous one but it lowers the number of repeated equivalent solutions when the involved sets are ground terms.

- `xless(R,X,S)`
- `xsubset(S1,S2)`
- `xun(R,S,T)`
- `xint(R,S,T)`
- `xdiff(R,S,T)`
- `xndiff(R,S,T)`
- `xsdiff(R,S,T)`
- `xsize(S,N)`

Example 9.1 *Using the* {log} *library predicates*

- {log}=> `xsize({a,b,a},N).`

  `N = 2.`

- {log}=> `xsize({X,Y},N).`

  `N = 2 - Constraints: [X neq Y]` *(2 times)*
  `N = 1, Y = X.`

- {log}=> `xint({a,b},{X,Y},S).`

```
S = {a,b}, X = a, Y = b;
S = {a}, X = a - Constraints: [Y neq b];
S = {a,b}, X = b, Y = a;
S = {a}, Y = a - Constraints: [X neq b];
S = {b}, X = b - Constraints: [Y neq a];
S = {b}, Y = b - Constraints: [X neq a];
S = {} - Constraints: [X neq a,Y neq a,X neq b,Y neq b].
```

The problem of redundant solutions is still an open problem. When the involved sets are ground the implementation takes care of avoiding as much as possible redundand solution. When, in contrast, some non-ground set 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.

10 Mixing Prolog and {*log*}

{*log*} provides a few simple facilities for mixing {*log*} code with Prolog code. Specifically, it is possible to add Prolog code to a {*log*} program and to use it from within the latter, as well as it is possible to add {*log*} code to a Prolog program and to use it from within Prolog.

10.1 Calling Prolog from {*log*}

{*log*} allows the user to call any SICStus Prolog goal from within a {*log*} goal or program. To call the SICStus Prolog goal G simply call the goal

`prolog_call(G).`

When G is `consult(file)`, by calling the goal

`prolog_call(consult(file))`

the user can consult the SICStus Prolog program stored in `file` adding it to the current {*log*} program. Any predicate `p` defined in this program can be subsequentely called from within {*log*} by giving the goal `prolog_call(p(···))`.

Note that Prolog clauses stored in the program database using

`prolog_call(consult(file))`

are not "seen" at all by the {*log*} predicates such as `abolish/0` and `listing/0`. These predicates consider only {*log*} programs. To list, for example, the Prolog clauses for predicate `p` one can issue the goal

`prolog_call(listing(p))`.

When mixing Prolog and {*log*}, one should take care also of the fact that {*log*} predicates work on the internal representation of terms and clauses. In particular, set terms are represented using the `with` notation described in Section 3.1.1. To facilitate conversion from the external to the internal representation of terms and vice versa the following two Prolog predicates—defined within the {*log*} interpreter—are available:

- `preproc(ExtTerm,IntTerm)`: true if `IntTerm` is the internal {*log*} representation of term `ExtTerm`. For example, the answer to ?- `preproc({a,b / S},TInt)` is `TInt = S with b with a`.

- `postproc(IntTerm,ExtTerm)`: true if `ExtTerm` is the external {*log*} representation of term `IntTerm`. For example, the answer to ?- `postproc(S with b with a with b,TExt)` is `T = {a,b / S}`.

Note that both `preproc` and `postproc` eliminate duplicate elements which possibly occur in set terms.

Example 10.1 *Define a predicate that reads from a file* `F` *a sequence of terms into a set* `Set`*. Since* {log} *does not provide any facility for file manipulation we define a Prolog predicate* `fread_set` *that performs the desired task.*

```
fread_set(F,Set) :-
    var(Set),
    seeing(OldInput),
    see(F),
    readSet(ExtSet),
    preproc(ExtSet,Set),
    seen,
    see(OldInput).

readSet({X/S}) :-
    read(X),
    X \== end_of_file, !,
    readSet(S).
readSet({}).
```

Note the use of predicate `preproc`*: it is necessary to ensure that the computed result has the form of a set term in the usual* {log} *external representation.*

Assuming the Prolog program for `fread_set` *has been stored into the file* `fread_set.pl`*, it can be consulted by the following directive*

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

and then predicate `fread_set` *can be executed by issuing the goal (assuming data to read are stored into the file* `'data.txt'`*).*

```
{log}=> prolog_call(fread_set('data.txt',S)).
```

If the data stored in `'data.txt'` *are the three terms* `a`*,* `b`*, and* `c` *the answer will be*

```
S = {a,b,c}.
```

10.2 Calling {*log*} from Prolog

When writing SICStus Prolog programs or goals one can exploit the full power of {*log*} (e.g., to solve some set-based formula) by calling the {*log*} interpreter, passing data to it and receiving results from it.

To facilitate the task of communicating with the {*log*} interpreter, the standard {*log*} configuration supplies also a Prolog file, named `prolog_to_setlog`, which contains a few predicates designed for that purpose.

In particular, predicates `setlog_call/(1-2)` are intended to be used to prove any {*log*} goal from within a Prolog program or goal.

`setlog_call(Goal,Constr_ext)`

is true if the {*log*} goal `Goal` is proved true—using the {*log*} interpreter—with the computed constraint `Constr_ext`. Whenever the constraint is not important one can use `setlog_call/2` which is like `setlog_call/2` except that it has no constraint parameter.

Example 10.2 *Calling* {log}

Read a set term and check whether all its elements are positive numbers

```
?- read(S), setlog_call(forall(X in S, X > 0)).
```

For instance, with input `{1,2,3}` *the answer will be* `yes`*.*

Surely, in order to use `setlog_call/(1-2)` the {*log*} interpreter must be already loaded into the Prolog program database. To make this task easier, file `prolog_to_setlog` provides also the predicates `setlog_consult/(0-1)` and `setlog_load/(0-1)`. `setlog_consult/0` and `setlog_load/0` assume that the name of the file containing the interpreter is `setlog`. The other two, instead, take the name of the file as a parameter. All these predicates allow the $\{log\}$ interpreter (either source code or compiled code) to be consulted without entering the $\{log\}$ environment—that is, at the end of the operation the Prolog prompt, not the $\{log\}$ one, will appear. However, the $\{log\}$ environment can be subsequently entered at any moment by issuing the goal `setlog`.

Here is a complete example showing how to exploit the $\{log\}$ capabilities to write a (non-trivial) Prolog program. The problem at hand is the well-known problem of coloring a geographical map.

Given a map of n regions $r_1,\ldots,r_n$, and a set $\{c_1,\ldots,c_m\}$ of colors, the map coloring problem consists of finding an assignment of colors to the regions of the map such that no two neighbouring regions have the same color. A map is represented as an undirected graph where the nodes are the regions and the arcs are the pairs of neighboring regions. In turn an *undirected labeled graph* can be conveniently represented by the set N of nodes and a finite set of edges $E = \{\{\mu_1,\nu_1\},\{\mu_2,\nu_2\},\ldots\}$, $\mu,\nu \in N$ (i.e., a set of nested sets).

Similarly, an *assignment* of colors to regions is represented as a set of ordered pairs $[r,c]$ where r is a region and c is the color assigned to it.

In the *coloring* program, all input/output—required to obtain from the user the name of all regions, the map and the colors to be used—will be performd using Prolog. In contrast, all set-based operations—including the predicate that solve the core part of the coloring problem—will be performed using $\{log\}$.

Example 10.3 *Coloring program*

```
coloring :-
      nl, region_input(Regions,{},Map),
      color_input(Colors),
      setlog_call(solve_coloring(Regions,Map,Colors,Ass),_),
      nl, ass_output(Ass),
      nl, write('or'),nl,
      fail.
coloring :-
      write('nothing else'),nl,nl.
```

```
region_input({Region/Regions},Map,FinalMap) :-
      write('Give the name of a region (or eof to finish)'),nl,
      read(Region),
      Region \== end_of_file,!,
      neighbouring(Region,Map,NewMap),
      region_input(Regions,NewMap,FinalMap).
region_input({},Map,Map).

neighbouring(Region,R,RR) :-
      write('Give the set of neighbouring regions for '),
      write(Region), write(':  '), read(S),
      setlog_call((build_map(Region,S,NewR))!),
      setlog_call((xun(NewR,R,RR))!).

color_input(Colors) :-
      write('Give the set of colors to be used:  '),
      read(Colors).

ass_output() :- !.
ass_output(Assignments) :-
      setlog_call(Assignments = {[Region,Color]/Ass}!),
      write('Color of region '), write(Region),
      write(' is '), write(Color),nl,
      ass_output(Ass).

:- setlog_call(consult('libraries.3.3')).
:- setlog_call(consult('coloring')).
:- coloring.
```

Assume file `'prolog_to_setlog'` *and the* {log} *interpreter have been already loaded. As soon as the* `coloring` *program has been completely consulted it starts its execution by asking to the user the name of all regions in the map and for each of them the set of neighbouring regions (represented as a* {log} *set term).*

To build the map as a set of unordered pairs, we use the {log} *predicates* `build_map/3` *and* `xun/3`*. The definition of* `build_map/3` *is shown below (actually it is very similar to the definition of the* `cross_product` *predicate shown in Example 8.1).*

```
build_map(A,B,M) :-
      M = {X: exists([Y,Z], X = {Y,Z} & Y in A & Z in B)}.
```

The definition of `xun/3` *is given in* `setlog_lib` *(see Section 9). Hence, this file is also consulted before activating the* `coloring` *program.*

When the map input has been completed, the user is asked to provide the set of colors to be used. When also this information is available the coloring problem can be finally solved calling the {log} *predicate* `solve_coloring/4`.

```
solve_coloring(Regions,Map,Colors,Ass) :-
    assign(Regions,Colors,Ass),
    forall(P in Map, exists([R1,R2,C],
      P = {R1,R2} & [R1,C] in Ass & [R2,C] nin Ass).
```

The first subgoal of `coloring` *is used to generate a possible assignment of colors to regions, whereas the second subgoal tests whether this assignment is an admissible one or not, i.e., no two adjacent regions in the map have the same color.*

The predicate `assign(S1,S2,A)` *is true if* `A` *is an assignment that assigns an element of the set* `S2` *(selected non-deterministically) to each element of the set* `S1`*. Its* {log} *definition is shown below.*

```
assign({},_,{}).
assign(Reg,Colors,{[R,C]/Assignments}) :-
    (Reg = {R/Regions} & R nin Regions)!  &
    C in Colors &
    assign(Regions,Colors,Assignments).
```

Finally, the computed assignments of colors to regions is printed out.

Observe that, for the sake of simplicity, we assume here to consider only the substitutions computed by `solve_coloring` ignoring all other constraints possibly computed by it. This turns out to request that no input data (e.g. the set of colors) is left partially specified .

Also observe that to print the computed assignments (predicate `ass_output`) we need to be able to access one by one all elements of the set of assignments returned by the {*log*} predicate `solve_coloring`. Since this set is represented as a {*log*} set term (see the representation of set terms described in Section 3.1), we need to use {*log*} unification to obtain the first element and the rest of the set.[12]

As a final remark, note that the possibility to use {*log*} directly from any Prolog programs allows us to view {*log*} not only as a stand-alone CLP language but also—and perhaps more realistically—as an extension to exiting Prolog system, in particular to SICStus Prolog.

[12] Beware of the fact that the set terms {a,/{b,/{c,{}}}} and {a,b,c} do unify in {*log*}—they are both translated to the same internal `with`-representation—but they surely do not unify in standard Prolog.

11 The {*log*} interpreter

The {*log*} interpreter is written in SICStus Prolog. Basically, it is composed of four main parts:

- Top-level. This part provides all the interaction with the user (input of top-level goal and output of computed answers), as well as the main {*log*} predicate, `solve/2`, which implements the {*log*} proof procedure. Solving a goal G, `solve/2` further disfinguishes among: higher-level built-in predicates (e.g., `call/1`, RUQs, negation, `or`,...), standard Prolog built-in predicates, {*log*} specific built-in predicates (e.g., `consult/(0-1)`, `listing/0`, ...), and, finally, program defined predicates.

- Unification algorithm. This part implements the {*log*} extended set unification algorithm [2, 3, 8].

  Observe that, for efficiency reasons we have suppressed the occur-check test. As well-known this can lead to unpleasant behaviour of the interpreter. The presence of sets can make critical situations even harder to catch. For example, the goal `{X,{X}} = {Z / S}` causes the interpreter to go into an infinite loop just becouse it is not able to detect a cyclic definition (precisely, the second solution for the given goal would require to solve the equation `Z = {Z}` which is clearly not well-founded). To avoid this kind of inconvenience it is enough to reactivate the occur-check test by uncomment those lines that apply it in the interpreter source code.

- Constraint solver. This part is devoted to constraint management. Predicate `can/2` is used to check if a given constraint C is satisfiable and, in the affirmative case, to return non-deterministically all the possible solved form constraints equivalent to C which are derivable from C (remember that the solved form is a simplified form of constraints containing only irreducible constraints).

- Preprocessing. This part implements all those actions that are performed to transform a {*log*} program or goal into the corresponding program or goal in its internal representation and vice versa. In particular this part provides: two predicates for translating set terms occurring in any {*log*} expression from the external representation (using curley brackets) to the `with`-representation and vice versa; predicates for eliminating RUQs and intensional set terms from any {*log*} clause or goal containing them (see Section 8); predicates for consulting $\{log\}$ programs and asserting $\{log\}$ clauses in the program database.

When the interpreter has to execute a goal G it first eliminates any intensional set terms and RUQs from G, possibly adding new atoms to G and new clauses to the current program, then it replaces set terms with equivalent set terms but using the internal `with`-based representation, and finally it tries to solve the resulting goal. At the end of this operation, if the goal turns out to be solvable, then the interpreter prints on the standard output in a suitable format both the computed substitution and the computed constraint.

Last but not least, it is important to remind that the current implementation of the $\{log\}$ language is intended mainly to provide a tool for testing the expressive power of the language. As such, it does not devote much attention to execution efficiency. Surely, also this aspect could be improved in many respects, some with a small effort, some with deeper and more complex changes. Anyway, the whole efficiency issue is left for future releases of the $\{log\}$ interpreter.

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, I wish to thank in particular: Davide Aliffi, Bruna Bazzan, Paola Bruscoli, Roberta Carmona, Daniela Fogli, Mauro Gaspari, Andrea Magnani, Silvia Monica, and Carla Piazza.

This work is partially supported by MURST project *Certificazione automatica di programmi mediante interpretazione astratta.*

Bibliography (ordered by date)

[1] A. Dovier and E. Pontelli. La Programmazione Logica con Insiemi. "Tesi di Laurea", Università di Udine (In italian), 1991.

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

[3] A. Dovier, E. Omodeo, E. Pontelli, G. Rossi. Embedding Finite Sets in a Logic Programming Language. In *Extension of Logic Programming '92* (E. Lamma e P. Mello, eds), Lecture Notes in Artificial Intelligence, Vol. 660, Springer Verlag, 1992, pp. 150–167.

[4] 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.

[5] 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.

[6] A. Dovier, E. Pontelli, G. Rossi. The CLP language {log} and the relation between intensional sets and negation. Technical Report *NMSU-CSTR*-9503, Department of Computer Science, New Mexico State University, USA, 1995.

[7] A. Dovier. Computable Set Theory and Logic Programming. TD–1/96, Università di Pisa, Dipartimento di Informatica, March 1996.

[8] 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.

[9] G. Rossi. Program Specification and Programming with Sets in Logic. *7th Int'l Conference on Artificial Intelligence and Information-Control Systems of Robots* (I.Plander, ed.), World Scientific Publishing, 1997.

[10] 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.

[11] A.Dovier, E. Pontelli, G. Rossi. Constructive Negation and Constraint Logic Programming with Sets. Rapporto di Ricerca, Dipartimento di Matematica, Universitá di Parma, n.183, Sept. 1998 (also to appear in *New Generation Computing*, Springer-Verlag)