$\{log\}$ User's Manual

Version 4.9.8 Release 9c

GIANFRANCO ROSSI
E-mail: gianfranco.rossi@unipr.it
Università di Parma
Italy

MAXIMILIANO CRISTIÁ
E-mail: cristia@cifasis-conicet.gov.ar
Universidad Nacional de Rosario and CIFASIS
Argentina

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.dmi.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, 3, 4, 5].

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 extended $\{log\}$ in various aspects, adding new features for dealing with binary relations, partial functions, Cartesian products, and Restricted Intentional Sets.

The Prolog code of the $\{log\}$ interpreter is available at the $\{log\}$ web page:

```
http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html
```

At the same page, you can find also the PDF file of this manual, various $\{log\}$ library files containing the $\{log\}$ definitions of operations on sets, binary relations and lists not provided as built-in in $\{log\}$, a file containing a number of simple preprocessing rules ("filtering rules") that may help constraint solving, and a few sample programs and applications written in $\{log\}$.

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 using `consult/1`:

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

The $\{log\}$ interpreter is loaded into the Prolog program database. While loading, the interpreter tries to consult three additional files from the working directory (see also the `setlog_config/1` built-in predicate in Sect. 12.1):

- `setlog_rules.pl`: this file provides a number of additional constraint rewriting rules that are not strictly necessary for the solver to work correctly, but can be useful to simplify processing of the input formulas.
- `size_solver.pl`: this file provides the implementation of a constraint solving procedure that allows you to extend the basic constraint solver of $\{log\}$, making it able to deal with cardinality constraints in a correct and complete way; without loading this file you can still use cardinality constraints but the solver is no longer guaranteed to be a decision procedure (see Sect. 8).
- `setlog_tc.pl`: this file provides a number of predicates that allow you to activate (optional) type checking for your program; if this file is not loaded no type checking is performed, as usual in Prolog (see Sect. 10).

Once loaded, 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 execute the following goal:

```
?- 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 solve the following formula (as regards terminology, note that saying "executing the goal G" is the same as saying "asking $\{log\}$ to solve the formula G"):

```
{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 built-in predicate `setlog/1` (see Sect. 12.1 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 `setlog`.

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, must be consulted using the $\{log\}$ predicate `add_lib/1`. For example:

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

The same can be obtained from the Prolog environment by issuing

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

See Section 13 for the complete list of the predicates defined in the standard $\{log\}$ library.

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 it is currently reading or an error message if a syntax error is detected. All clauses previously stored in the $\{log\}$ program database are removed.

User defined clauses currently stored in the $\{log\}$ program database can be printed out on the standard output by executing:

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

These clauses can be completely removed by executing:

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

Note that both predicates `abolish` and `listing` ignore library clauses that have been added by either `consult_lib` or `add_lib`; thus, library clauses cannot be removed nor listed.

$\{log\}$ programs can be loaded also by using `consult/2`. Precisely:

```
{log}=> consult('file.slog',mute).
```

loads the program in `file.slog` just like `consult('file.slog')` but without showing the number of the clause it is currently reading; while:

```
{log}=> consult('file.slog',app).
```

loads the program in `file.slog` just like `consult('file.slog')` but without removing clauses previously stored in the program database. $\{log\}$ programs can be loaded also directly from the Prolog environment by issuing:

```
?- setlog_consult('file.slog').
```

whose behavior is exactly the same as `consult('file.slog',mute).`[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 $\{log\}$;
- `h/1` provides more detailed information on $\{log\}$, according to its parameter: `h(syntax)` for $\{log\}$ syntactic conventions; `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 identifiers starting with an uppercase letter or an underscore. Then, we can ask $\{log\}$ to solve the following equation:

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

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 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* [6].

$\{log\}$ 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 [3]. 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 $\{x_1, \dots, x_n$ `/ A}`, $x_1, \dots, x_n$ 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.

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 $\{log\}$

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. 3.3), 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.

3.2 Considerations on set membership and not membership

It is easy to prove the following:

$$x \in A \Leftrightarrow \exists B : A = \{x\} \cup B \wedge x \notin B \tag{1}$$

This equivalence is used by $\{log\}$ when it finds a constraint of the form `X in A`. In fact $\{log\}$ transforms `X in A` into `A = {X/_N1} & X nin _N1`, where `_N1` is a fresh variable, which is aligned with the semantics of the `{_/_}` set constructor.

On the other hand, when $\{log\}$ finds a constraint of the form `A = {X/B}` it *does not* assume `X nin B`. This may lead to a degraded performance as $\{log\}$ will open two computation branches: one in which `X nin B` holds, and another one in which `B = {X/_N1} & X nin _N1` holds. Then, in general, $\{log\}$ will need twice the time to solve a formula including `A = {X/B}` than the same formula but including `X in A` instead of the latter.

Observe that in most situations, what you want to say is `X in A` rather that `A = {X/B}`, for some `B`. In these situations it is advisable to use the former over the latter. There are situations, however, where `B` appears in some other constraint of the formula, meaning that the scope of `B` is not the sub-formula representing the membership relation like in (1), but the whole formula. In these situations conjoining `X nin B` might not be what the formula is intended to state. So $\{log\}$ leaves this to the user's discretion.

3.3 Introducing formulas

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

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/_N1},
A = {1,j/_N1}
Constraint: j nin _N1, set(_N1)
```

where `_N1` 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: `_N1` 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.

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/_N1}
Constraint: set(A), j nin _N1, set(_N1)
```

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 $\{log\}$ formula, fails if `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 formulas containing uninitialized variables. As an example, the simple goal:

```
{log}=> naf(X = a) & p(X).
```

where `p` is a user predicate simply defined as `p(b)`, answers `no`, while the logically equivalent goal:

```
{log}=> p(X) & naf(X = a).
```

correctly answers `X = b`. Actually, `naf(X = a)`, when `X` is an uninitialized variable, implements the formula $\forall X : \bar{} X = a$ which is trivially false; hence the first goal fails. Instead, in the second formula, `X` is first bound to `b`, hence `naf(b = a)` is true and the whole goal succeeds.

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 & p(X).
```

correctly answers `X = b`. In fact, in `X neq a`, `X` is (correctly) assumed to be an existentially quantified variable and `X neq a` is trivially found to be true.

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

$$x \in B \cup C \Rightarrow (x \in B \vee x \in C) \tag{2}$$

you can write:

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

which in turn is equivalent to (for some A):

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

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).
```

3.4 Using $\{log\}$ as an automated theorem prover

$\{log\}$ can also be used to automatically prove theorems about set theory. If you want to prove that (2) 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* [4].

3.5 Modes of operation

Users can run $\{log\}$ in one of two modes of operation: the *prover* mode and the *solver* mode. The default mode when $\{log\}$ is loaded is the prover mode. Users can switch from one mode to the other by issuing:

```
{log}=> mode(solver).
```

or

```
{log}=> mode(prover).
```

The prover mode is better when the formula passed to $\{log\}$ is supposed to be unsatisfiable. In general, in these cases $\{log\}$ will be more efficient in concluding that the formula is indeed unsatisfiable, than in solver mode. This means that if users want to use $\{log\}$ as an automated theorem prover (cf. Section 3.4), they should use it in prover mode.

The solver mode is better when the formula passed to $\{log\}$ is supposed to be satisfiable. In general, in these cases $\{log\}$ will return solutions with fewer constraints and more equalities, than in solver mode. This means that if users want to use $\{log\}$ as a prototyping environment, they should use it in solver mode.

For example, executing the following goal

```
{log}=> ssubset(X,{a,b}).
```

in prover mode, makes $\{log\}$ to return the following answer:

```
true
Constraint: subset(X,{a,b}), set(X)
```

meaning that the input formula is anyway satisfiable. If the same goal is executed in solver mode, instead, $\{log\}$ will return the following four solutions:

```
X = {a,b}
X = {a}
X = {b}
X = {}
```

In general, it is hard to predict in advance which mode of operation will be the best for a given formula. However, if you want to use $\{log\}$ as a programming language, then the first choice would be the solver mode; if you want to use it as an automated prover, then use it in prover mode.

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 [10]. 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. Besides, $\{log\}$ provides a new set term, `cp(A,B)`, whose semantics is the Cartesian product between sets `A` and `B` [9]. Note that a Cartesian product is a binary relation.

For example, asking $\{log\}$ to solve the following formula:

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

makes the solver to return the most general binary relation whose domain is the set $\{a\}$. This relation is given as 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 one ordered pair (because it has a not-empty domain) whose first component must be `a`, while the second component can be anything—which is represented by making the second component to be a variable.

Observe that ordered pairs are noted with square brackets. Then, `[a,b]` represents the ordered pair (a,b). Note that, `[a,b] = [c,d]`, if and only if `a = c` and `b = d`. 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:

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

Then, it can be shown that $\mathrm{dom}(\{(a,a)\} \circ T) = \{a\}$, thus guaranteeing that $\mathrm{dom}(\{(a,y)\} \cup T) = \{a\}$, 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 Cartesian products are binary relations, a term `cp(A,B)` can be passed to any predicate expecting a binary relation as its argument. Hence, we can use Cartesian products with relational operators, for example, as follows:

```
{log}=> dom(cp({a/A},B),{a}).
```

making $\{log\}$ to return `A = {}` and `A = {a}`.

In turn, since binary relations are sets of ordered pairs, they can be built by means of the same set constructors described in Section 3 and they can be used in the same places as any other set terms. In particular the empty binary relation is denoted with `{}`. Furthermore, formulas involving relational operators are built as we shown in Section 3.3 (i.e., by means of `&` and `or`). Besides, they can be freely combined with formulas involving set operators. For example:

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

are formulas combining set and relational operators and making use of the extensional set and Cartesian product constructors.

OPERATOR	$\{log\}$	MEANING
binary relation	`rel(R)`	R is a binary relation
domain	`dom(R,A)`	$\operatorname{dom} R = A$
range	`ran(R,A)`	$\operatorname{ran} R = A$
composition	`comp(R,S,T)`	$T = R \circ S$
inverse	`inv(R,S)`	$S = R^{-1}$
identity relation	`id(A,F)`	$\operatorname{id} A = F$
domain restriction	`dres(A,R,S)`	$S = A \lhd R$
domain anti-restriction	`dares(A,R,S)`	$S = A \ntriangleleft R$
range restriction	`rres(R,A,S)`	$S = R \rhd A$
range anti-restriction	`rares(R,A,S)`	$S = R \ntriangleright A$
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
domain	`ndom(R,A)`	$\operatorname{dom} R \neq A$
range	`nran(R,A)`	$\operatorname{ran} R \neq A$
composition	`ncomp(R,S,T)`	$T \neq R \circ S$
inverse	`ninv(R,S)`	$S \neq R^{-1}$
identity relation	`nid(A,F)`	$\operatorname{id} A \neq F$
domain restriction	`ndres(A,R,S)`	$S \neq A \lhd R$
domain anti-restriction	`ndares(A,R,S)`	$S \neq A \ntriangleleft R$
range restriction	`nrres(R,A,S)`	$S \neq R \rhd A$
range anti-restriction	`nrares(R,A,S)`	$S \neq R \ntriangleright A$
overriding	`noplus(R,S,T)`	$T \neq R \oplus S$
relational image	`nrimg(A,R,B)`	$B \neq R[A]$

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

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 (including Cartesian products).

Table 2 lists all the relational operators, along with their negations, available in $\{log\}$ as predicates. 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:

$$\forall x, y_1, y_2 : (x, y_1) \in f \wedge (x, y_2) \in f \Rightarrow y_1 = y_2 \tag{3}$$

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 [7]. These formulas, are built as formulas

OPERATOR	DEFINITION
domain	$\operatorname{dom} R = \{x \mid \exists y : (x,y) \in R\}$
range	$\operatorname{ran} R = \{y \mid \exists x : (x,y) \in R\}$
composition	$R \circ S = \{(x,z) \mid \exists y : (x,y) \in R \land (y,z) \in S\}$
inverse	$R^{-1} = \{(y,x) \mid (x,y) \in R\}$
identity relation	$\operatorname{id} A = \{(x,x) \mid x \in A\}$
domain restriction	$A \lhd R = \{(x,y) \mid (x,y) \in R \land x \in A\}$
domain anti-restriction	$A \ntriangleleft R = \{(x,y) \mid (x,y) \in R \land x \notin A\}$
range restriction	$R \rhd A = \{(x,y) \mid (x,y) \in R \land y \in A\}$
range anti-restriction	$R \ntriangleright A = \{(x,y) \mid (x,y) \in R \land y \notin A\}$
overriding	$R \oplus S = (\operatorname{dom} S \ntriangleleft R) \cup S$
relational image	$R[A] = \operatorname{ran}(A \lhd R)$

Table 3: Definition of relational operators

OPERATOR	$\{log\}$	MEANING
partial function	`pfun(F)`	F verifies (3)
function application	`apply(F,X,Y)`	$F(X) = Y$
NEGATIONS		
partial function	`npfun(F)`	F does not verifies (3)
function application	`napply(F,X,Y)`	$F(X) \neq Y$

Table 4: Partial function operators available in $\{log\}$

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 automatically assert that `F` is a partial function if it appears as the first argument of `apply`. 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 semi-decision procedure for the *theory of finite, unbounded binary relations* (including Cartesian products). This means that in general $\{log\}$ will return the right answer but for some formulas it will not return; or it will return some solutions and then will block; or it will return an infinite number of solutions. In any case the solutions are correct but it will fail in proving that some formulas are unsatisfiable.

Note that Cartesian products can be used to assert that a binary relation is of a particular *type*. If you want binary relation R to be of type $A \times B$ you can state:

```
{log}=> subset(R,cp(A,B)).
```

So, for instance, $\{log\}$ will answer `no` if the following formula is provided:

```
{log}=> subset(R,cp(A,{1,2})) & R = {[X,1],[Y,9]}.
```

but it will find solutions if the following one is given:

```
{log}=> subset(R,cp(A,{1,2})) & R = {[X,1],[Y,Z]}.
```

5 Intensional sets

Intensional sets, also called *set comprehensions*, or *set-builder notation*, are sets described by a property whose elements must satisfy rather than by explicitly enumerating their elements. Intensional sets are widely recognized as a key feature to describe complex problems.

The main way to express intensional sets in $\{log\}$ is by means of the so called *restricted intensional sets* (RIS) [8]. Another way is by using *general intensional sets* (GIS). The next six subsections deal with RIS, while GIS are briefly described in subsection 5.2.

5.1 Solving formulas with restricted intensional sets

A restricted intensional set (RIS) denotes a *finite* intensional set. In the language of mathematics a RIS is noted as:

$$\{x : D \mid F(x) \bullet P(x)\} \tag{4}$$

where x, called *control variable*, is a bound variable whose scope is the RIS itself; D, called *domain*, is a set; F, called *filter*, is a formula; and P, called *pattern*, is an expression. The semantics of a RIS is the following:

$$\{y : \exists x(x \in D \wedge F(x) \wedge y = P(x))\} \tag{5}$$

that is, the elements of the RIS are of the form $P(x)$ for all those $x \in D$ satisfying $F(x)$.

In $\{log\}$ a RIS such as (4) is written as follows:

```
Ris(X in D,[],F(X),P(X))
```

where `D` can be any kind of set except for `cp`, variable-intervals (see Section 9) and RIS themselves; `F` is a $\{log\}$ formula and `P` is a $\{log\}$ term. The second argument (i.e., `[]`) will be explained shortly. A $\{log\}$ RIS term can be more complex but for the moment we will focus on this simpler construction.

In the current version, $\{log\}$ admits RIS in all the set operators of Table 1. This means that RIS cannot be used as arguments or as part of arguments passed in to any relational operator (cf. Tables 2 and 4).

The following formula uses a RIS to find out if `N` is a prime number or not (`int(m,n)` is a set term denoting the integer interval $[m,n]$, see Section 7 for further details):

```
{log}=> N > 1 & MD is N div 2 &
        ris(X in int(2,MD),[],0 is N mod X,X) = {}
```

The idea is to check if the set of proper divisors of N (i.e., $\{x : [1,MD] \mid 0 = N \bmod x\}$) is empty or not. Then, if N is bound to, say, 20, $\{log\}$ answers no; but if it is bound to 101 it answers `N = 101, MD = 50`.

Note that in the last example the pattern is the control variable (i.e., X). When this is the case the pattern can be omitted. Similarly, when the second argument is the empty list it can be omitted. Thus, the RIS above can be written more concisely as:

```
ris(X in int(2,MD),0 is N mod X)
```

It is important to observe that atomic predicates occurring in a RIS formula can be not only any of the predefined operators available in $\{log\}$, but also any user-defined predicate, provided its negated version is also defined in the program. If p is the name of a user-defined predicate with arity $n \geq 0$, the *negated version* of p must be named n_p, with arity n, and the clauses defining it must be added to the user program. If p occurs in the formula of a RIS and the solver needs to negate the formula, then it will look for a clause defining n_p: if it finds it, then it uses it; otherwise, it uses Negation as Failure, printing the warning message `unsafe use of negation for predicate p`.

5.1.1 Parameters and the functional section

Now we are going to explain the meaning of the second argument of a RIS term and through it we will present one more argument of RIS terms. Say you want to specify a function mapping sets to their cardinalities provided they are greater than one. Then, you can use the following RIS:

```
CF = ris(S in D,[C],C > 1,[S,C],size(S,C))
```

In effect, CF is the set of ordered pairs of the form [S,C] where S belongs to D, C is the cardinality of S and C is greater than one. Hence, CF is a function as is a set of ordered pairs where no two pairs have the same first component. The following are two formulas that can be proved to hold using $\{log\}$:

```
D = {{X},{Y,Z}} &
CF = ris(S in D,[C],C>1,[S,C],size(S,C)) & CF = {}.

[{1,2},N] in ris(S in D,[C],C>1,[S,C],size(S,C)).
```

yielding `Z = Y` and `N = 2`, respectively, as part of their computed answers.

Note that in CF the second argument is no longer the empty list but the list containing variable C. This kind of variables are called *parameters*. Parameters are local to the RIS where they appear. The semantics of a parameter is an existentially quantified variable. Then, the semantics of CF is given by the following intentional set:

$$\{y : \exists s(\exists c(s \in D \land |s| = c \land c > 1 \land y = (s,c)))\} \tag{6}$$

The second argument of a RIS term is called the *parameters list*. In the parameters list you can introduce as many parameters as you need. All of them have the same semantics.

In general, $\{log\}$ may give wrong answers to formulas including RIS with parameters. However, if parameters are used to get the 'results' of some predicates, then they are safe. This is what we did with C in the above RIS because we use it to get the cardinality of `S` through the predicate `size(S,C)`. In doing so we placed `size(S,C)` in the fifth argument of the RIS term and not as part of the filter. By placing `size(S,C)` in the fifth argument $\{log\}$ can treat it in a different way than a regular filter predicate; had we placed it as a regular filter constraint $\{log\}$ might have given wrong answers.

The fifth argument of a RIS term is called the *functional section*. In the functional section you can place a conjunction of constraints each of which captures its 'result' in a parameter declared in the RIS. This means that only constraints that behave as functions can be placed in the functional section. We call these constraints *functional predicates*. For example, assuming `A` is a parameter, you cannot place `size(A,N)` in the functional section because for a given `N` there are many sets whose cardinality is `N`. In other words, `size` behaves as a function only w.r.t. its second argument. Along the same lines, if `A` and `B` are parameters, `un(A,B,C)` is not allowed in the functional section because it depends on two parameters; if only `A` is a parameter, `un(A,B,C)` is not allowed neither because for any given sets `B` and `C` there are many `A` for which `un(A,B,C)` is true; but `un(D,C,A)` can be placed in the functional section because for any given sets `D` and `C` there is only one `A` for which `un(D,C,A)` is true.

5.1.2 Parameters and control expressions

Some times we need parameters but we cannot express what we want with functional predicates. If we persist in using parameters the formula becomes unsafe. However, $\{log\}$ offers another safe way that avoids many of those unsafe parameters.

In fact, some parameters can be avoided by using *control expressions* instead of a control variable. A control expression is an expression of the following forms:

- Any nested closed list whose elements are all distinct variables (e.g., `[X]`, `[X,[Y],Z]`, etc.).
- `[X|Y]`, where `X` and `Y` are different variables.
- `{X/Y}`, where `X` and `Y` are different variables.

In all cases the variables in the control expression are variables bound to the RIS. Then, for example, if `R` is a set and we want the subset of `R` whose elements are ordered pairs such that their first components are greater than or equal to the second components, we can use the following RIS:

```
ris([X,Y] in R, X >= Y)
```

where the pattern is omitted as is expected to be the control expression. Observe that this set cannot be expressed with the set and relational operators of Sections 3 and 4 nor with functional predicates (because $\geq$ is not such a predicate).

As the above RIS does not introduce parameters, every formula including it will always return the right answer (i.e., those formulas are *safe* because they lay inside the decision procedure).

When control expressions are used in place of control variables, only the elements of the domain of the RIS that unify with the control expression are processed (all the others are simply ignored). For example, consider the RIS above where `R` is instantiated with `{1,[0,3],[5,1]}`. Then, we have:

```
{[5,1]} = ris([X,Y] in {1,[0,3],[5,1]}, X >= Y)
```

because `1` is ignored as it does not unify with `[X,Y]`; `[0,3]` does not pass the filter; and `[5,1]` is the only element of the domain of the RIS which unifies with the control expression and passes the filter.

5.1.3 Encoding sets of structured elements

Control expressions can be used to extract elements with particular structures. The above is such an example but elements with more complex structure can also be considered. In effect, the structure of an element can be given by an appropriate nesting of functors. For example, `p(X,q(X,Y,Z))`, where `p` and `q` are functors and `X`, `Y` and `Z` are variables. Such an element can be encoded as follows: `[p,[X,[q,[X,Y,Z]]]]`. Then, if we want the subset of `A` whose elements are of the form `p(X,q(X,Y,Z))` we can use the following RIS:

```
ris([P,[X1,[Q,[X2,Y,Z]]]] in A,P = p & Q = q & X1 = X2)
```

where `X1` and `X2` are introduced because all the variables in a control expression must be different from each other.

Furthermore, if we also want elements of the form `p(E)` we can use union:

```
un(ris([P,[X1,[Q,[X2,Y,Z]]]] in A,P = p & Q = q & X1 = X2),
   ris([P,[E]] in A,P = p),
   Result)
```

5.1.4 Safe patterns

As we have said at the beginning of this section, a pattern in a RIS term is a $\{log\}$ term. However, for a formula to be safe the patterns involved in the RIS participating in the formula must verify a couple of conditions. In order to precisely state those conditions, we need the following definitions.

Definition 5.1 (Bijective pattern) *Let $\{x : D \mid F(x) \bullet P(x)\}$ be a RIS, then its pattern is* bijective *if $P : \{x : x \in D \wedge F(x)\} \to Y$ is a bijective function, where Y is the set of images of P.*

Definition 5.2 (Co-injective patterns) *Two patterns, P and Q, are said to be* co-injective *if for any x and y, if $P(x) = Q(y)$ then $x = y$.*

Then, for a formula to be safe all its patterns must be bijective and pairwise co-injective.

Checking whether a formula verifies these conditions is, in general, not decidable. Hence, $\{log\}$ does not perform this check: checking whether the patterns of the RIS included in input formulas are bijective and pairwise co-injective or not is left to the user's responsibility. Fortunately, at least the following patterns *always* verify Definitions 5.1 and 5.2:

- terms of the form `[X,f(...,X,...)]`, where `X` is the control expression and `f` is any function;
- terms of the form `[f(...,X,...),X]`, where `X` is the control expression and `f` is any function;
- the formula contains patterns of either form but not a mix of them.

As an example, executing the following goal:

```
{log}=> [4,N] in ris(X in D,[Y],X>=0,[X,Y],Y is X*X).
```

where the RIS contains a safe pattern, correctly answers

```
N = 16,
D = {4/_N1}
Constraint: 4 nin _N1, set(_N1)
```

On the other hand, executing the goal:

```
{log}=> ris([X,Y] in R,true,X) = {1}.
```

where the RIS pattern is not a safe one, will yield the following answer:

```
R = {[1,_N2]/_N1}
Constraint: ris([X,Y]in _N1,[],true,X,true) = {}
```

stating that R cannot contain more than one pair whose first component is 1, which is incorrect.

Observe that most of the RIS that you will need can be defined with these patterns. So, in general, you will not need to check whether your patterns verify Definitions 5.1 and 5.2.

Furthermore, if your formula is not going to end up resolving a constraint such as:

```
ris(X in D, G(X), Q(X)) = {t / ris(Y in D, F(Y), P(Y))}
```

where D is a variable and t stands for any $\{log\}$ term, then the condition on the pairwise co-injectivity of the patterns can be dropped. Note that D is the same variable used as RIS domain at both sides of the equation.

5.1.5 Enumerating the elements of a RIS

Given an equation of the form `S = ris(X in D,[],F(X),P(X))`, where S and D are variables, the RIS is not evaluated and thus remains as it is. However, when the domain is a ground set (i.e., `{}`, or `{t1,...,tn}` with `t1,...,tn` ground), or a ground interval, then it is possible to enumerate the elements of the RIS by means of the `is` operator, which forces the evaluation of its term. For example, when the following is executed:

```
Sqrs is ris(X in int(1,100),[Y],true,[X,Y],Y is X*X)
```

$\{log\}$ returns:

```
Sqrs = {[1,1],[2,4],...,[100,10000]}
```

Note, however, that if `Y is X*X` is written as part of the filter:

```
Sqrs is ris(X in int(1,100),[Y],Y is X*X,[X,Y])
```

$\{log\}$ first prints a series of warning messages and only after them it prints the correct answer. This means that the answer is not fully reliable (although in this case is correct).

If parameter Y is eliminated by introducing a control expression:

```
Sqrs is ris([X,Y] in D,[], X in int(1,100),[X,Y],Y is X*X)
```

the domain is a variable and thus $\{log\}$ simply returns:

```
Sqrs = ris([X,Y] in D,[], X in int(1,100),[X,Y],Y is X*X)
```

5.1.6 Automated proofs

As with extensional sets and binary relations, $\{log\}$ can be used as an automated theorem prover for formulas involving RIS *provided the formula is safe*. Remember that a formula is safe if all RIS terms possibly occurring in it contain only safe patterns (i.e., they are bijective and pairwise co-injective) and, if they contain parameters, then they are safe parameters (i.e., they are used only as the result of functional predicates). For instance, $\{log\}$ can prove that:

```
inters(A,B,C) & D = ris(X in A,X in B) & C neq D
```

is false which means that

$$A \cap B = \{x \mid x \in A \land x \in B\}$$

is a theorem.

5.2 General intensional set terms

$\{log\}$ provides also another way to express intensional sets, by means of general intensional set (GIS) terms.

GIS terms are terms of one of the following forms:

$\{X : (G)\}$
$\{X : \texttt{exists}(V, G)\}$
$\{X : \texttt{exists}([V_1, \dots, V_n], G)\}$

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

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

The following are two simple examples using GIS.

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

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

  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]}.
  ```

GIS can occur everywhere ordinary set terms are allowed. Moreover, they can be nested at any depth, i.e., the formula of a GIS term can contain other GIS terms.

Note that the control expression of a GIS can be only a single variable, whereas control expressions of RIS can be also compound terms. Apart from this, GIS are more general than RIS. Hence, from a logical point of view, RIS can be always replaced by the equivalent GIS. From an operational point of view, however, their behavior can be quite different.

In fact, internally, atoms containing RIS terms are dealt with as primitive constraints, while atoms containing GIS terms are always replaced by new atoms whose definition (given in terms of automatically generated {*log*} clauses) implements the set grouping mechanism which allows to collect into an extensional set all values satisfying a given intensional definition.

As a consequence, when the intensional set term denotes an infinite set or even an unbounded set, the use of GIS may lead to possibly incorrect computations. For instance, given the goal of Section 5.1.6 written using GIS instead of RIS, i.e.:

```
inters(A,B,C) & D = {X : (X in A & X in B)} & C neq D
```

{*log*} is not able to find it is unsatisfiable (actually it generates wrong solutions).

Hence, GIS can be used safely only when the corresponding extensional sets can be effectively constructed. In other word, using RIS one surely stays inside the {*log*} decision procedure, while using GIS may lead {*log*} to fail to decide the satisfiability of the input formula.

6 Quantifiers

{*log*} provides a form of *restricted universal quantifier* (RUQ) where the quantified variable ranges over a {*log*} set. The main way to express RUQ in {*log*} is by means of the `foreach/2` and `foreach/4` predicates:

- `foreach(X in A,F)`

  where `X` can be either a variable or a control expression (cf. Section 5.1.2), `A` is any {*log*} set admitted as RIS domain, and `F` is any {*log*} formula. The semantics of such a constraint is the expected:

  $$\forall x : x \in A \Rightarrow F$$

  That is, F is a formula true for every element of A.

- `foreach(X in A,[params],F,functional predicates)`

  which behaves as the first form while allowing the introduction of parameters and functional predicates (cf. Section 5.1).

{*log*} provides also the negated version `nforeach` of `foreach/4`. In particular, note that the logical meaning of `nforeach(X in A,[],F,true)` is $\neg\, \forall x : x \in A \Rightarrow F$, i.e., $\exists x : x \in A \wedge \neg F$.

Formulas including `foreach/2` and `foreach/4` are always *safe* (i.e., solving the formula always returns the right answer), whereas formulas including `nforeach/4` are safe provided they use only safe parameters.

The following are two simple examples where RUQ are used to iterate over all elements of the given set:

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

```
    print_elements(S) :_
        foreach(X in S, write(X) & nl).
```

- `all_pair(S)` is true if all elements of `S` are pairs (i.e., `all_pair(S)` ⇔ `rel(S)`).

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

  Sample goal:

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

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

```
foreach(X in S1,foreach(Y in S2, X neq Y))
```

can be used to state that `S1` and `S2` are disjoint sets (i.e., `disj(S1,S2)` is true). As an example, executing the goal:

```
{log}=> S1={a,b} & S2={Z} &
        foreach(X in S1,foreach(Y in S2, X neq Y)).
```

returns the constraint:

```
Z neq a, Z neq b
```

while executing:

```
{log}=> S1={a/R} & S2={c} &
        foreach(X in S1,foreach(Y in S2, X neq Y)).
```

returns the constraint:

```
subset(R,ris(X in R,[],foreach(Y in {c},[],X neq Y,yes),X,yes)),
set(R)
```

The last answer is motivated by the fact that, internally, the constraint `foreach(X in A,F)` is translated into:

```
 subset(A,ris(X in A,[],F,X,true))
```

which is enough to guarantee that all the elements of `A` satisfy `F`.

{*log*} provides also another way to express RUQ by means of the `forall` predicates:

`forall`(X in A, F)
`forall`(X in A,`exists`(V,F))
`forall`(X in A,`exists`([V_1,...,V_n],F)

where `X` is a variable, `A` is any {log} set admitted as RIS domain, `F` is an arbitrary {*log*} formula, containing at least one occurrence of X, and `V`, `V_i` are variables “local” to `F`.

The meaning of `forall(X in A,exists([V_1,...,V_n],F)` is the same as `foreach(X in A,[V_1,...,V_n],F`
Hence, from a logical point of view, `forall` can be always be replaced by the equivalent `foreach` predicate. In this respect, `forall` is motivated mainly for compatibility with previous versions of {*log*}.

From an operational point of view, however, `forall` behaves differently from `foreach` whenever the set over which the control variable ranges is a variable or a set containing a variable set part. As an example, executing the goal:

```
{log}=> foreach(X in R,X in {a,b}).
```

simply returns the constraint:

```
subset(R,ris(X in R,[],X in S,X,true)), set(R), set(S)
```

(note that this constraint is trivially true for `R = {}`). On the other hand, executing the goal:

```
{log}=> forall(X in R,X in {a,b}).
```

explicitly generates all possible solutions:

```
R = {}
R = {a}
R = {a,b}
R = {b}
```

Actually, the `forall` predicates are not dealt with as constraints. Executing `forall`(X `in` A , F) always starts a computation that iteratively executes the goal F over all elements of the set A. If A and F are not enough instantiated this can lead to an infinite computation. For instance, executing the goal:

```
{log}=> forall(X in R,X in {a/S}) & a nin R.
```

after generating the first solution `R = {}`), will go into an infinite loop trying to add more and more elements to `R` which anyway contains `a`.

Hence, when using `foreach` one surely stays inside the {*log*} decision procedure, while using `forall` may lead {*log*} to fail to decide the satisfiability of the input formula.

7 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 arithmetic expressions and `n` is a either a variable or a numeric constant.

An arithmetic expression is either a variable or a numeric constant or an arithmetic function (e.g., `+`, `-`, `*`, `mod`, etc.) applied to its arguments, which are in turn arithmetic expressions. Numbers can be either integer or floating-point numbers. For example, the following arithmetic formulas are solved as shown:

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

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}=> 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 constant value when they are evaluated, as in Prolog. Otherwise, a problem in the arithmetic expression is detected and {*log*} answers `no`. Arithmetic predicates containing real numbers are not dealt as constraints, but exactly as in Prolog. For example:

```
{log}=> 1.5 + X > 0.7 & X is 2*3.0.
Problem in arithmetic expression
no
```

while

```
{log}=> X is 2*3.0 & 1.5 + X > 0.7.
X = 6.0
```

yields the correct answer.

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

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

OPERATOR	$\{log\}$	MEANING
integer	`integer(t)`	*t* is an integer number
integer	`ninteger(t)`	*t* is not an integer number

Table 6: `integer` and `ninteger` constraints

In fact, arithmetic built-in predicates over integer expressions are dealt with by a constraint solver. Specifically, one can use either a constraint solver over finite domains (namely, CLP(FD)) or a constraint solver over rationals, but restricted to integers (namely, CLP(Q) using `bb_inf/4`) By default, $\{log\}$ starts solving arithmetic predicates by calling CLP(Q). Users can change this by issuing `int_solver(clpfd)` and can reset the default with `int_solver(clpq)`.

Both solvers have their advantages and disadvantages. We briefly analyze them in the next sections. See Section 7.3 for a few consideration on which solver should be used.

7.1 CLP(FD)

The CLP(FD) 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. For example:

```
{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)
```

This goal is clearly unsatisfiable, but $\{log\}$ (actually the underlying CLP(FD) solver) is not able to detect it. `integer(X)` is a $\{log\}$ primitive constraint that is true if and only if `X` is an integer number. There is also its negated version `ninteger` (see Table 6).

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.

7.1.1 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 integer 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 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).                    (7)
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 (provided CLP(FD) is the active integer solver). 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 default activation of labeling one can give the goal:

```
{log}=> 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 reactivated at any moment by issuing the goal:

```
{log}=> 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 labeling has been performed.

Observe that in goal (7) 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 its 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 from a set of possible values.

7.2 CLP(Q)

The CLP(Q) solver is complete for *linear arithmetic* (be it real, rational or integer). In other words, it can give the right answer whenever the formula is linear, i.e., multiplication is restricted to expressions of the form $x * y$ where either x or y are constants.

{*log*} uses CLP(Q) restricted to integer solutions. Thus, for instance, the goal

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

which has solutions over the rational numbers, is found to be unsatisfiable over the integers; hence, {*log*} answers `no`.

The CLP(Q) solver can be activated at any time by issuing `int_solver(clpq)`.

When CLP(Q) is the active integer solver, {*log*} does not perform automatic labeling as with CLP(FD). You can nevertheless perform labeling on a single variable by using the built-in predicate `labeling`. If `X` is constrained to range over a bounded interval, then `labeling(X)` non-deterministically assign to `X` one by one all possible values in the interval; otherwise, i.e., the domain of `X` is unbounded, `labeling(X)` does nothing. For example, executing the goal

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

will produce the answer

```
true
Constraint: 34>X+1, X>=1, X=<100, integer(X), ...
```

meaning that the input formula is satisfiable.[2] If we conjoin `labeling(X)` to the input formula, then the answer will be:

```
X = 1
...
Another solution?  (y/n)
X = 32
Another solution? (y/n)
no
```

i.e., the same result as with CLP(FD) using the default labeling.

7.3 Which integer solver should be used?

As we have said, each integer solver has its own advantages and disadvantages. Hence, you should use CLP(FD) or CLP(Q) depending on how you are using $\{log\}$. In general, if you are using $\{log\}$ as a programming language, then CLP(FD) should be your first choice. Recall that in this situation the solver mode should be preferred over the prover mode—cf. Section 3.5. Conversely, if you are using $\{log\}$ as an automated theorem prover, then CLP(Q) is definitely the integer solver to be used—because it is complete for linear arithmetic thus turning `no` answers into real unsatisfiability proofs if linear arithmetic constraints are in the formula. Recall that if you are using $\{log\}$ as an automated theorem prover then you should use it in prover mode.

7.4 An aggregation function: sum of a set

$\{log\}$ provides also the built-in predicate `sum(Set,Sum)`, to compute the sum of a set. This is one of the classic *aggregation functions*.

The first argument of `sum` can be either a variable or an extensional set or an an integer interval with constant limits (integer intervals are introduced in Section 9). The second argument can be either a variable or an integer constant. When bound, the first argument must denote either the empty set or a set of non-negative integer numbers. In particular, when applied to an empty set it returns 0.

Note that both elements of an extensional set and its set part can be uninitialized variables. For example, executing the goal

```
{log}=> sum({X1,X2},2).
```

we get the three solutions:

```
X1 = 0, X2 = 2
X1 = 2, X2 = 0
X1 = 2, X2 = 2
```

However, when both the first and the second arguments are variables occurring in some other constraints in the input formula, the solver may be unable to determine the satisfiability/unsatisfiability of the formula. For example, given the formula:

[2]The constraints not shown in the computed answer are (negligible) `integer` constraints over fresh internal variables.

```
{log}=> sum(A,S1) & sum(A,S2) & S1 < S2.
true
Constraint: sum(A,S1), S1 >= 0, sum(A,S2), S2 >= 0, S2 > S1
```

the answer is clearly wrong.

In order to get the correct answer one should activate CLP(FD), bound integer variables to ground intervals of the form `X in int(m,n)` for some constants `m` and `n`, and force labeling for at least some of them. For example, $\{log\}$ gives the correct answer in the following case:

In order to get the correct answer one should bound integer variables to ground intervals of the form `X in int(m,n)` for some constants `m` and `n`, and force labeling for at least some of them (remember that, if CLP(FD) is activated, then labeling is performed automatically at the end of the computation). For example, $\{log\}$ gives the correct answer in the following case:

```
{log}=> int_solver(clpfd).
{log}=> sum(A,S1) & sum(A,S2) & S1 < S2 & S1 in int(1,10) & S2 in int(1,10).
no
```

Hence, in the current version, $\{log\}$ does not provide a decision procedure for formulas involving `sum`. Removing this limitation is a goal for future releases. However, $\{log\}$ does provide a decision procedure for other aggregation functions as is explained in Section 9.

8 Cardinality constraints

`size` is a set predicate that represents the cardinality of a set. `size`, and its negated version `nsize`, are defined in Table 7.

The first argument of both predicates can be either a variable, or a set term, or a non-variable-interval, or a `cp` term, but not a `ris` term. The second argument can be either a variable or an integer constant. As an example, given the following goal:

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

we get as first answer:

```
true
Constraint: 1 nin R, size(R,_N1), _N1>=0, M>=1, _N1 is M-1,
            set(R)
```

If the second argument of `size` is a constant k and the first is a variable then the computed answer depends on the operation mode of the solver (cf. Section 3.5). In *solver* mode the most general set of k elements is explicitly shown. As an example, the answer to the following goal:

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

is

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

OPERATOR	{*log*}	MEANING
set cardinality	`size(A,N)`	$|A| = N$
not set cardinality	`nsize(A,N)`	$|A| \neq N$

Table 7: The set cardinality operators

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.

Conversely, if we are in *prover* mode, then the solver may check the satisfiability of the input formula without explicitly generating sets involved in `size` constraints of the form `size(A,`k`)`. For example, given the following goal:

```
{log}=>  size(A,10) & subset({1,2,3},A).
```

we get as first answer:

```
A = {1,2,3/_N1}
Constraint: 1 nin _N1, size(_N1,7), 2 nin _N1, 3 nin _N1,
            set(_N1)
```

whereas in *solver* mode, `A` would be bound to `{1,2,3,_N7,_N6,_N5,_N4,_N3, _N2,_N1}`, along with the necessary constraints to ensure that elements in the set are all distinct from each other.[3]

Although the second argument of a `size` constraint can only be an integer constant or variable, users can link it to more complex (linear) expressions by means of the `is` or the ordering operators, as shown in the following examples:

```
{log}=> size(S,N) & N > 1.
{log}=> size({1/R},N) &
        N is 2*X + 3*Y + 4 & X > -6 & 2*Y + 5 < 10.
```

{*log*} provides a decision procedure for formulas involving `size` constraints provided the following conditions are met:

1. The only constraints in the formula are those of Tables 1, 5 and 6.
   That is, no relational constraints are allowed in the formula.
2. The first argument of any `size` constraint in the formula is either the empty set, a variable or an extensional set. This means that `size` will, in general, not work well when the first argument is a Cartesian product (i.e., a `cp` term), a RIS (i.e., a `ris` term) or when the set part of the first argument is one of these.
3. All integer constraints in the formula are linear.

For example, {*log*} is able to detect that the last two formulas given above are satisfiable. Similarly, {*log*} can detect that the formula:

```
{log}=> subset(A,B) & size(A,CA) & size(B,CB) & CA = CB &
        A neq B.
```

is unsatisfiable.

[3] As a more practical solution, in the current version, if k is less or equal to a given threshold (now fixed to 6), then solving the constraint `size(A,`k`)` causes the set `A` to be anyway generated, disregarding the solver execution mode.

8.1 The solved form of formulas involving `size` constraints

Some solutions returned by $\{log\}$ when the formula involves `size` constraints might be too abstract. For example the answer to the following formula:

```
{log}=> size(A,M) & 1 =< M & M =< 2 & size(B,N) & 5 =< N &
        subset(C,B) & size(C,K) & 7 =< K.
```

is

```
true
Constraint: size(A,M), M>=0, 1=<M, M=<2, size(B,N), N>=0,
            5=<N, subset(C,B), size(C,K), K>=0, 7=<K
```

that is, the formula itself. This means the formula is satisfiable and that all the possible solutions can be obtained by fixing values for the variables as long as all the constraints are met. However, this answer does not point out an evident concrete solution for the formula.

In general, when the `size` constraint is present in the answer, substituting all set variables by the empty set can lead to unsound solutions. Manually computing a concrete solution from such an answer can be cumbersome and error prone. Therefore, for these cases, $\{log\}$ provides the `fix_size` and `nofix_size` built-in predicates. The latter is active by default. When `fix_size` is issued, the answer to the above goal is a more concrete solution:

```
A = {_N8},
M = 1,
B = {_N7,_N6,_N5,_N4,_N3,_N2,_N1},
N = 7,
C = {_N7,_N6,_N5,_N4,_N3,_N2,_N1},
K = 7
Constraint: _N7 neq _N6, _N7 neq _N5, _N7 neq _N4, _N7 neq _N3,
_N7 neq _N2, _N7 neq _N1, _N6 neq _N5, _N6 neq _N4, _N6 neq _N3,
_N6 neq _N2, _N6 neq _N1, _N5 neq _N4, _N5 neq _N3, _N5 neq _N2,
_N5 neq _N1, _N4 neq _N3, _N4 neq _N2, _N4 neq _N1, _N3 neq _N2,
_N3 neq _N1, _N2 neq _N1
```

Recall to reactivate `nofix_size` if you are expecting abstract solutions.

If one wants simply to know which are the smallest cardinalities of the set variables occurring in *size* constraints as to satisfy the formula, without explicitly computing the relevant sets, then it is possible to use the `show_min` and `noshow_min` built-in predicates. As an example, by executing:

```
{log}=> show_min.
{log}=> size(A,M) & 1 =< M & M =< 2 & size(B,N) & 5 =< N &
        subset(C,B) & size(C,K) & 7 =< K.
```

we get

```
true
Constraint: M=1, N=7, K=7,
            size(A,M), M>=0, 1=<M, M=<2, size(B,N), N>=0, 5=<N,
            subset(C,B), size(C,K), K>=0, 7=<K, set(A), set(B),
            set(C)
```

where `M=1, N=7, K=7` represent the smallest cardinalities of sets `A`, `B` and `C` that make the input formula `true`.

9 Finite integer intervals

$\{log\}$ allows to represent *finite integer intervals* and to deal with them as sets of integer numbers.[4] The integer interval $[m,n]$ is written in $\{log\}$ as `int(m,n)`; `m` and `n`, called limits, can be either variables or integer constants. Note that an interval `int(m,n)` where `m > n` denotes the empty set.

If the input formula fulfills the following conditions:

- only operators of Tables 1 and 7 are involved;
- only linear integer arithmetic is involved;
- CLP(Q) is the active integer solver,

then $\{log\}$ will always compute the right answer even when set arguments are terms of the form `int(m,n)` (i.e., {log} provides a decision procedure for those formulas).

In this case:

- extensional sets and intervals can be freely combined, e.g. `un({X,Y},{V,W},int(M,5));`
- the limits of intervals can be variable or constants;
- intervals can also occur as the set part of extensional sets, e.g., $\{$`-1/int(1,N)`$\}$.

Although the limits of intervals can only be variables or constants, they can participate in arithmetic constraints. For example:

```
{log}=> un({X,Y},{V,W},int(M,5)) & M is X - Y.
X = 4,
Y = 2,
V = 5,
W = 3,
M = 2
```

Some powerful set operators can be defined and $\{log\}$ can automatically reason about them within the decidable fragment. In particular, some classic *aggregation functions* are definable as $\{log\}$ formulas. These are gathered in the $\{log\}$ library `setloglibIntervals.slog`. Here we comment on a couple of them.

The first one is the aggregation function that computes the minimum element of a set:

```
setmin(S,M) :- M in S & subset(S,int(M,_)).
```

Then, we can compute the minimum of a given set:

```
{log}=> setmin({2,5,-1,9,0},M).
M = -1
Constraint: 2=<_N1, 5=<_N1, -1=<_N1, 9=<_N1, 0=<_N1

{log}=> setmin({2,X,-1,9,0},M).                    [one element is a variable]
```

[4]From now on, we will say integer interval or just interval meaning finite integer intervals.

```
M = X
Constraint: X=<2, 2=<_N1, X=<_N1, X=< -1, -1=<_N1, X=<9, 9=<_N1,
            X=<0, 0=<_N1

Another solution?  (y/n)y
M = -1
Constraint: 2=<_N1, -1=<X, X=<_N1, -1=<_N1, 9=<_N1, 0=<_N1
```

And we can prove properties true of `setmin` as long as the formula remains in the decidable fragment:

```
{log}=> setmin(S,M) & X in S & X < M.
no
```

`setloglibIntervals.slog` includes a predicate that computes the maximum of a set.

Along these lines we can define a predicate stating when an element of a set is the successor of another element of the same set.

```
ssucc(A,X,Y) :-
  X < Y &
  A = {X,Y/A1} & X nin A1 & Y nin A1 &
  un(Inf,Sup,A1) & disj(Inf,Sup) &
  M is X - 1 & subset(Inf,int(_,M)) &
  N is Y + 1 & subset(Sup,int(N,_)).
```

And as with `setmin` we can get the successor of a given element in a given set:

```
{log}=> ssucc({2,5,-1,9,0},5,M).
M = 9
Constraint: _N1=<2, _N1=< -1, _N1=<0
```

But also the predecessor:

```
{log}=> ssucc({2,5,-1,9,0},M,0).
M = -1
Constraint: 2=<_N1, 5=<_N1, 9=<_N1
```
[the second argument is a variable]

An we can prove properties true of `ssucc`:

```
{log}=> ssucc(S,X,Y) & Z in S & X < Z & Z < Y.
no
```

Note that some of the predicates listed in `setloglibIntervals.slog` can also be encoded with RIS and RUQ (cf. Sections 5.1 and 6). Which encoding is the best cannot be told because it depends on the context where they are used.

Concerning intervals, when operators other than those in Tables 1 and 7 are involved or when the formula contains non-linear integer arithmetic, the answer returned by {*log*} is unreliable (i.e., {*log*} is no longer guaranteed to provide a decision procedure for those formulas). It can be made reliable if the limits of all the intervals in the formula are constants. If this is the case, intervals can be safely used as the domain of a RIS, the domain of a RUQ and as arguments of the operators of Tables 2 and 4. Intervals cannot be the arguments of `cp` terms.

10 Types in $\{log\}$

As we have said $\{log\}$ accepts untyped formulas. For example, the following is a possible value for a $\{log\}$ set:

```
{a, 1, {2}, [5,b]}
```

It is also possible to operate with those sets:

```
{log}=> un({a, 1, {2}, [5,b]},{X},{Y/R}).
Y = a,
R = {1,{2},[5,b],X}
Constraint: X neq a
```

In this way, variables are not declared because they can assume values of any type. $\{log\}$ is able to distinguish between variables representing sets and variables representing non set objects, in particular integer numbers. This distinction is made according to the constraints where a variable participates in. For instance:

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

makes $\{log\}$ to classify `X` and `Y` as integer variables. This means that:

```
{log}=> X > Y & un(X,{1,2,3},Z).
```

will fail just because `X` cannot be an integer and a set at the same time.

In general the lack of types works well but it allows $\{log\}$ to accept formulas that cause undesired behaviors in some cases. For example:

```
{log}=> id({X/A},R) & id(R,A).
```

makes $\{log\}$ to enter an infinite loop that the user can interrupt by typing `Ctrl+c`. Due to the first constraint, `[X,X]` belongs to `R` and by the second constraint `[[X,X],[X,X]]` belongs to `A` which initiates the loop again. Internally, $\{log\}$ builds an increasingly larger ordered pair which eventually will consume all the available memory.

In a sense, this problem is caused because the formula is ill-typed. In effect, in a way, the first constraint states that `A` is a set of some elements and `R` is the identity function on `A`; but the second constraint states quite the opposite: `A` is the identity function on `R`. A typing system would deem this formula ill-typed and would reject it before any attempt on deciding its satisfiability is made.

Besides, types can help the programmer in avoiding certain errors as is acknowledged by the programming languages community. On the other hand, types can complicate programs and formulas by imposing strong restrictions on some operations. In an attempt to resolve this tension between type safety and *typeless* freedom, $\{log\}$ accepts untyped formulas but the user can activate a type checker at will. If the type checker is active all variables in formulas and clauses must be declared to be of a certain type and the type checker is called before a formula is executed (in interactive mode) or when a file is consulted. The type checker is activated by issuing:

```
{log}=> type_check.
```

and is deactivated with:

```
{log}=> notype_check.
```

Before going into the details of $\{log\}$'s type system, we present the typed version of the formula based on the `id` constraint analyzed above:

```
{log}=> id({X/A},R) & id(R,A) &
        dec(X,t) & dec(A,stype(t)) & dec(R,stype([t,t])).
```

Each `dec` constraint states that a variable is of a certain type. For instance, `dec(R,stype([t,t]))` states that `R` is a set of ordered pairs whose components are both of type `t`. If this formula is run while the type checker is *not* active, $\{log\}$ will simply ignore the `dec` constraints. This would cause a loop, as explained above. Instead, if the type checker is active, $\{log\}$ will report a type error and it will not execute the formula:

```
type error: in id(R,A)
            R is of type stype([t,t])
            A is of type stype(t)
```

The error comes from the fact that the type of `id` is expected to be:

```
id(stype(T),stype([T,T]))
```

for some type `T`. The error is clearly informing that this type is not verified given the types of the arguments.

Typing formulas is good but rejects harmless formulas such as the first one seen in this section:

```
{log}=> un({a, 1, {2}, [5,b]},{X},{Y/R}).
```

just because it is impossible to define a type for `{a, 1, {2}, [5,b]}`.

10.1 The type system

$\{log\}$ defines a type system based on those enforced by the Z and B notations. In this sense, the type system is oriented towards a typed set theory.

As we have said, when the type checker is active all variables must be declared to have *exactly one* type. These declarations are made by means of the `dec/2` constraint, called a *type constraint*. In `dec(V,t)`, `V` must be a variable and `t` must be a type—as defined right afterwards. Type constraints can be anywhere in the formula—that is, it is not necessary to declare the type of a variable before its first use. There is available also a `dec` constraint whose first argument is a list of variables:

$$\texttt{dec}([V_i,\dots,V_n],\texttt{t}) \Leftrightarrow \texttt{dec}(V_i,\texttt{t}) \land \cdots \land \texttt{dec}(V_n,\texttt{t})$$

which helps in reducing the size of typed formulas.

In $\{log\}$ types are not sets. That is, if `t` is a type you cannot write `X in t`. This is simply an ill-formed, incorrectly sorted constraint. $\{log\}$ will fail immediately if such constraint is provided.

In $\{log\}$ type identifiers and type constructors begin with a lowercase letter. The types and type constructors available in $\{log\}$ are the following.

10.1.1 Integers

`int` is a type representing the set of integer numbers ($\mathbb{Z}$). Then a formula such as `X > Y` can be typed as follows:

```
{log}=> dec(X,int) & X > Y & dec(Y,int).
```

Numbers are automatically typed as expected. Then a formula such as `X > Y` can be typed as follows:

```
{log}=> dec(X,int) & X > 10.
```

`int` is a reserved word of the language when in type checking mode. It can only be used where a type is allowed.

10.1.2 Character strings

`str` is a type representing the set of character strings. There are no special purpose operators defined on `str` so strings can be used mainly as elements of sets, as components of ordered pairs, etc. That is, for instance, the following are all type-correct:

```
{log}=> dec(X,str) & X = "Messi".
{log}=> "Pele" nin {"Messi","Maradona","Di Stefano"}.
{log}=> ["Di Stefano","Argentina"] = ["Maradona","Argentina"].
```

Atoms are not strings:

```
{log}=> dec(X,str) & X = messi.
type error: 'messi' should belong to some enumerated type
{log}=> "messi" = messi.
type error: 'messi' should belong to some enumerated type
```

`str` is a reserved word of the language when in type checking mode. It can only be used where a type is allowed.

10.1.3 Uninterpreted types

Any atom can be used as a type. All these types are called *uninterpreted types*—`int` is interpreted. In the Z notation these are called basic or given types. However, in {*log*} we have added a programming-oriented feature to uninterpreted types, as described below.

For example:

```
dec(A,address)
dec(N,name)
dec(Zip,zipcode)
dec(C,country)
dec(H,city)
```

are all possible type constraints declaring variables with uninterpreted types. Then, assuming these declarations:

```
{log}=> A neq N.
type error: in A neq N
            A is of type address
            N is of type name
```

In the Z notation the structure or form of the elements of uninterpreted types is unknown. This provides an abstraction mechanism. For instance, the programmer do not want to say, at the moment, whether or not an `address` is a character string, or a number (house) and a character string (street). But (s)he wants to be able to distinguish between `address`'es and `name`'s, so (s)he uses two different uninterpreted types.

In $\{log\}$, because is a programming language, uninterpreted types are associated to a known set of elements. If `t` is an uninterpreted type then all its elements are of the form `t?elem`, for any atom `elem`. Hence, we have:

```
{log}=> t?A = u?a.
type error: in t?A, t and A must be atoms
{log}=> t?a = u?a.
type error: in t?a = u?a
            t?a is of type t
            u?a is of type u
{log}=> t?a = t?a.
yes                                          [type correct, satisfiable]
{log}=> t?a = t?b.
no                                         [type correct, unsatisfiable]
{log}=> {t?a,t?b} = {t?b,t?a,t?b}.
yes
{log}=> dec(X,t) & X = t?a.
X = t?a
{log}=> dec(X,t) & X = t.
type error: 't' should belong to some enumerated type
{log}=> dec(X,t) & X = u?a.
type error: in X = u?a
            X is of type t
            u?a is of type u
```

In type-checking mode '?' is a reserved symbol. It can only be used as described in this section. See Section 10.7 to learn more about admissible terms in type-checking mode.

Uninterpreted types vs. strings (`str`). Note that replacing uninterpreted types with `str` might cause some problems. Using `str` instead of uninterpreted types is fine as long as you are aware that you are using the same type for things that might be quite different. For example[5]:

```
{log}=> dec_type(city,str).
{log}=> dec_type(name,str).
```

[5]`dec_type` is explained in Section 10.3.

make `city` and `name` the same type, `str`. This might be confusing:

```
{log}=> dec(N,name) & dec(C,city) & N = "Leo" & C = "Leo".
N = Leo,
C = Leo
```

That is `"Leo"` can be a `name` and a `city`. This is not the case if `city` and `name` are uninterpreted types.

```
{log}=> city?'Leo' = name?'Leo'.
type error: in city?'Leo' = name?'Leo'
            city?'Leo' is of type city
            name?'Leo' is of type name
```

10.1.4 Enumerated types

`etype(`[$e_1,\dots,e_n$]`)` is an *enumerated type* whose elements are all the e_i.

For example:

```
etype([red,blue,green])
etype([normal,warning,failure,stop])
etype([messi,maradona,distefano,carlovich])
```

are all enumerated types. So a declaration such as:

```
dec(A,etype([red,blue,green]))
```

will constrain `A` to be bound to only those three values.

Note that `etype([yes,no])` is a different type than `etype([no,yes])`. Actually, the type checker will issue a type error when the second type is used for the first time because ‘`no`’ is an element of another enumerated type.

In an enumerated type, each e_i must be an atom, different from all the uninterpreted types in scope and from all other atoms declared in other enumerated types—and from `int` and `str` which can only be used where a type can be used. All e_i in the list must be different from each other. The list must contain at least two elements.

See Section 10.3 to learn how to give a name to enumerated types so you do not have to repeat the enumeration in each type constraint.

10.1.5 Product types

If `t` and `u` are two types then `[t,u]` is a type interpreted as the Cartesian product between `t` and `u`. This means that the elements of `[t,u]` are ordered pairs whose first component is of type `t` and the second is of type `u`.

For example:

```
dec(B,[city,etype([red,blue,green])])
```

forces `B` to be bound only to ordered pairs whose first component is of type `city` and the second is `red`, `blue` or `green`. In this way:

```
B = [A,blue]
```

will type-check if `dec(A,city)` is in context. Conversely,

```
B is X + 7
```

will fail because `is` is of type `is(int,int,int)`.

Product types can be generalized to any number of products and can be nested at any level:

```
[t,str,[int,v]]
```

10.1.6 Set types

If `t` is a type then `stype(t)` is a type representing all the sets whose elements are of type `t`. In other words, `stype(t)` represents the powerset of `t`. Hence, if `X` is of type `stype(t)`, then `X` is a set whose elements are of type `t`.

Obviously, set types are everywhere in $\{log\}$. Most of the set constraints available in $\{log\}$ are typed by means of the `stype` type constructor. For example the following is the type of the `un` constraint:

```
un(stype(T),stype(T),stype(T))
```

for any type `T`. In other words, `un` is a polymorphic operator accepting sets of any type as long as all its elements are of the same type. The empty set is a polymorphic set term. Hence, `A = {}` is a correctly typed formula provided `A` has been declared to be of an `stype`.

The extensional set constructor is typed in such a way as to accept elements of the same type. Then, a set such as `{`$e_1, \dots, e_n / S$`}` is correctly typed if and only if every e_i is of type `T` and S is of type `stype(T)`, for some type `T`.

10.2 Types for binary relations and partial functions

By combining a product type and a set type it is possible to construct types representing binary relations. Typically:

```
stype([t,u])
```

corresponds to the type of all binary relations whose ordered pairs have a first component of type `t` and a second component of type `u`. Precisely, all the relational operators available in $\{log\}$ have types based on this construct. For example the following is the type of the `comp` constraint:

```
comp(stype([T,U]),stype([U,V]),stype([T,V]))
```

for any types `T`, `U` and `V`. Again, `comp` is a polymorphic operator.

The differences between `rel(R)` and `dec(R,stype([t,u]))` are the following:

1. `rel` is part of $\{log\}$'s inference engine; `dec(R,stype([t,u]))` is used only by the type checker when it is active.
2. `rel(R)` is automatically added by $\{log\}$ in certain situations. For example, if you assert `id(A,R)`, $\{log\}$ automatically adds `rel(R)`. Only the user can assert `dec(R,stype([t,u]))`.
3. `rel(R)` forces the elements of `R` to be ordered pairs but it does not state what the type of those pair is. Then, if only `rel(R)` is asserted, `{[1,a],[{t},[x,1]]}` is a possible value for `R`.

   Instead, if `dec(R,stype([t,u]))`, for any types `t` and `u`, is asserted then `R = {[1,a], [{t},[x,1]]}` will not type-check and the formula containing it will not be executed.

4. In this sense `rel(R)` is weaker than `dec(R,stype([t,u]))`, but it is automatic.

Note that a `cp` term is a set of ordered pairs. Then the type of `cp(A,B)` is `stype([u,t])` if and only if `A` is of type `stype(t)` and `B` is of type `stype(u)`.

Observe that there is no type for partial functions. Then, if `F` is meant to be a partial function taking values from some type `t` and returning values of some type `u`, you have to assert the following:

```
{log}=> dec(F,stype([t,u])) & pfun(F) & ...
```

Furthermore, `pfun(F)` is usually a consequence or a property of a program and so you can use {*log*} to automatically prove that this is actually the case. For example in:

```
{log}=> dec([F,G],stype([t,u])) & dec(D,stype(t)) &
        dec(X,t) & dec(Y,u) &
        pfun(F) & dom(F,D) & X nin D @ G = {[X,Y] / F}.
```

`pfun(G)` is a consequence of that formula. Then, you do not need to assert it, instead you can prove it:

```
{log}=> dec([F,G],stype([t,u])) & dec(D,stype(t)) &
        dec(X,t) & dec(Y,u) &
        pfun(F) & dom(F,D) & X nin D & G = {[X,Y] / F} &
        npfun(G).
no
```

In this way, the formula is lighter but as stronger as if you were conjoined `pfun(G)` to it.

10.3 Type declarations

Some times a type is defined by means of a long, complex type expression. For example, the following type is taken from a {*log*} program implementing the Bell-LaPadula security model:

```
stype([obj,[int,stype(cat)]])
```

Then, if you have to declare a couple of variables of that type the `dec` constraint becomes annoying:

```
dec([O1,O2],stype([obj,[int,stype(cat)]]))
```

For these situations {*log*} offers the `dec_type/2` constraint:

```
{log}=> dec_type(t,stype([obj,[int,stype(cat)]])).
{log}=> dec([O1,O2],t) & ...
```

That is, `dec_type(t,texpr)` declares that `t`, an atom, is a name for type `texpr`. Aftewards it holds that:

$$\texttt{dec}(V,\texttt{t}) \Leftrightarrow \texttt{dec}(V,\texttt{texpr})$$

In `dec_type(t,texpr)`: `t` cannot occur in `texpr`; cannot be an element of an enumerated type (in scope); cannot be the first argument of another `dec_type` constraint; and cannot be `int` or `str`. In turn, `texpr` cannot be an atom—because in this case you would be giving an uninterpreted type another name. However, `t` *can* be used in other type definitions. For example, `ac` is used to define `oac` and `sac`:

```
{log}=> dec_type(ac,[int,stype(cat)]).
{log}=> dec_type(oac,stype([obj,ac])).
{log}=> dec_type(sac,stype([sub,ac])).
{log}=> dec(A,ac) & dec(F,oac) & comp(F,{[A,A]},{}).
```

Naming enumerated types is another convenient use of `dec_type` constraints:

```
{log}=> dec_type(thebest,
                 etype([messi,maradona,distefano,carlovich])).
{log}=> dec([P1,P2],thebest) & P1 neq P2.
```

See Section 10.8 to learn how type declarations can be consulted and managed.

10.4 Typing clauses

If you activate the type checker and you want to consult a file containing {*log*} clauses, then you first need to type all the clauses of that file.

For instance, you may want to define a predicate adding a $(name, address)$ pair to the function holding the addresses of your acquaintances:

```
add_person(P,N,A,P_) :-
   pfun(P) & dom(P,D) & N nin D & P_ = {[N,A] / P}.
```

If you put this predicate in file `misc.pl` then, before loading `misc.pl` in type-check mode, you have to type `add_person/4` inside `misc.pl` as follows:

```
:- dec_type(na,[name,address]).
:- dec_p_type(add_person(stype(na),name,address,stype(na))).
add_person(P,N,A,P_) :-
    dec(D,stype(name)) &                        [D is typed inside the clause]
    pfun(P) & dom(P,D) & N nin D & P_ = {[N,A] / P}.
```

`dec_p_type/1` declares the type of `add_person` by giving the type of each and every argument. It can be read as "declare predicate type". Then, for example, the first `stype(na)` states that the first argument of `add_person` must be of that type (note that in turn `na` is a name for the type `[name,address]`); `name` states that the second argument of `add_person` must be of that type; and so on and so forth. Given that `D` is an internal variable we declare its type inside `add_person`.

A `dec_p_type` declaration must precede the definition of the corresponding predicate. If a predicate is defined in more than one clause, only one `dec_p_type` predicate must be in place. Hence, before re-consulting a file you have to reset all type declarations; see Section 10.8 to learn how to do this. `dec_p_type/1` is a reserved word of the language when the type checker is active. It cannot be used in other contexts.

10.5 Typing polymorphic operators

We have already shown that some operators are polymorphic. For example, we already know the type of these operators:

```
un(stype(T),stype(T),stype(T))
comp(stype([T,U]),stype([U,V]),stype([T,V]))
id(stype(T),stype([T,T]))
```

where T, U and V are *type variables*. That is, T, U and V are not types but they can be bound to types—recall that types and type constructors always begin with a lowercase letter.

You can define your own polymorphic operators and type them. For example, if you want to define the predicate un3 which performs the union of three sets, you can write that definition in some file as follows:

```
:- dec_pp_type(un3(stype(T),stype(T),stype(T),stype(T))).

un3(A,B,C,D) :-
    dec(X,stype(T)) &                          [X typed with same type variable, T]
    un(A,B,X) & un(X,C,D).
```

dec_pp_type/1 is the equivalent to dec_p_type for polymorphic predicates. It can be read as "declare polymorphic predicate type". The difference between a dec_pp_type and dec_p_type declaration is that the former accepts type variables, while the latter does not. {*log*} understands that predicate p is polymorphic if it is preceded by a dec_pp_type(p(...)) declaration, where the arity of p must coincide with that of p(...) inside the dec_pp_type declaration.

Note that X in un3 is typed with the same type variable, T, used in the dec_pp_type declaration; otherwise a type error will be informed. dec accepts type variables only when placed inside polymorphic predicates.

10.6 Running formulas in type-checking mode

When formulas are run with the type checker active, all variables must be declared to have exactly one type.

For example, if you want to call the un3 predicate defined in the previous section you will have to give the type of each actual parameter:

```
{log}=> dec([W,X,Y,Z],stype(t)) & un3(W,X,Y,Z).
```

Note that in the type constraint we use a type, i.e. t, and not a type variable, e.g. T. Recall that elements of type t are of the form t?elem for any atom elem. So you can call un3 as follows:

```
{log}=> dec([X,Y,Z],stype(t)) & un3({t?a,t?aa},X,Y,Z).
```

Compound terms can also be used as expected. For example:

```
{log}=> dec([X,Y,Z],stype([t,u])) & dec(I,t) &
        un3({[I,u?abc]},X,Y,Z).
```

Similarly, the elements of int and str can be used in formulas:

```
{log}=> dec(Z,stype(int)) & dec([I,J],int) &
        un3({1,2,3},{I,J},{},Z).
{log}=> dec(Z,stype(str)) & dec([I,J],str) &
        un3({"Messi","Maradona","Di Stefano"},{I,J},{},Z).
```

10.7 Admissible terms in type-checking mode

In type-checking mode only terms that can be assigned a type are admissible. Then, for instance, `a:b` is not an admissible term in type-checking mode:

```
{log}=> a:b = 3.
type error: a:b isn't an admissible term in type-checking mode
```

Clearly, the admissible terms are correlated with the type system as follows:

- Variables – any type
- Integer numbers – type `int`
- Strings – type `str`
- Atoms – enumerated type
- Terms of the form `type?elem` where `type` and `elem` are atoms – type `type`
- Nested lists – product type
- `{}` – any set type
- Sets – set type
- Any syntactically correct term recursively constructed using the terms listed above

All the other terms are not admitted.

Note, however, that you can represent any Prolog term with a product type, much as the `=../2` predicate works. For example, if you want to represent terms of the form `_:_` where the left argument is of type `t` and the right of type `int`, then you can use ordered pairs typed as `[t,int]`. If you want to represent terms of the form `p(_,_,_)` then you can use lists of four elements: `[p,_,_,_]`.

10.8 User commands to explore types

The following user commands are available to deal with types.

- `dec_type/2`. Besides the function already described, this command can be used to get the type expression associated to a type name by passing in a variable as a second argument.

  For example, assuming that `t` is a type name for some type expression:

  ```
  {log}=> dec_type(t,E).
  E = stype([obj,ac])
  Another solution?  (y/n)
  no
  ```

- `reset_types/0` deletes all type declarations currently in scope. These include declarations made through the following predicates: `dec_type`, `dec_p_type` and `dec_pp_type`. This command should be used if a file is going to be re-consulted. Note that all the type declarations made in other loaded files or in interactive mode are also deleted.
- `type_of(p,T)` where `p` is an atom and `T` is a variable. If `p` is the functor of a declared predicate, `T` is bound to its type as given by either a `dec_p_type` or a `dec_pp_type` declaration.

  For example:

```
{log}=> type_of(dres,T).
T = [stype(T),stype([T,U]),stype([T,U])]
Another solution?  (y/n)
no
```

- `type_decs/1` where the parameter can be `td`, `pt` or `ppt`. It shows all the pairs (*type_name*, *type_expression*) (`td`); the type of all non-polymorphic predicates (`pt`); and the type of all polymorphic predicates (`ppt`); in all three cases currently in scope.
- `expand_type(t,E)` where `t` is a type expression and `E` is a variable. If `t` is given in terms of type names, these are recursively replaced by the corresponding type expressions.

  For example:

  ```
  {log}=> dec_type(t,stype(a)).
  {log}=> dec_type(a,stype(b)).
  {log}=> expand_type(t,E).
  E = stype(stype(b))
  Another solution?  (y/n)
  no
  {log}=> expand_type([t,a],E).
  E = [stype(stype(b)),stype(b)]
  Another solution?  (y/n)
  no
  ```

11 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 execute goal `G`, possibly getting constraint `C`.
- `solve(G)`: same as `call(G)` but all constraints possibly generated by `G` are immediately solved. `solve` can be used to impose an order in the execution of the formula. For example in:

  ```
  solve(G1) & solve(G2)
  ```

  `G1` is executed *before* `G2`.
- `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` and `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 (`_5044` is the system generated name of the uninitialized variable `X`)

```
b_5044
X = a
```

since atoms are executed in the exact order they occur in the goal[6]. 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:

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

```
{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`.

12 Prolog-$\{log\}$ communication

12.1 From Prolog to $\{log\}$

Main predicates

- `setlog/0`: to enter/re-enter the $\{log\}$ interactive environment.
- `setlog(G)`, `setlog(G,C)`: to execute a $\{log\}$ goal `G`, possibly getting an (output) $\{log\}$ constraint `C`.
- `setlog(G,InCLst,OutCLst)`: to execute a $\{log\}$ goal `G` with a (possibly empty) input constraint list `InCLst` and output constraint list `OutCLst`.
- `setlog(G,T,OutCLst,Res,OptsLst)`: to execute a $\{log\}$ goal `G` with an output constraint list `OutCLst` and a timeout `T` (an integer constant specifying a time in milliseconds), with a (possibly empty) list `OptsLst` of execution options. The `Res` argument is used to specify how 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 the computed result is not guaranteed to be reliable (e.g., CLP(FD) is the active integer solver) and some integer variables have no finite domain associated with them; or, the elimination of *neq*-constraints has been disabled through the option `neq_elim`).

  The possible execution modes are those specified by the built-in control predicates mentioned in the previous sections (in particular in Sect. 11). As an example, the following goal:

  ```
      ?- setlog(X in int(1,5),1000,C,R,
                 [int_solver(clpfd),nolabel]).
  ```

  requires the goal `X in int(1,5)` to be executed, with a `1000` millisecond time-out, using CLP(FD) as the integer solver but disabling the final automatic labeling step (see Sect. 7.1.1).

  A complete list of the available execution options can be found in Appendix A.

- `setlog(G,T,OutCLst,Res,try([OptsLst`$_1$`,...,OptsLst`$_n$`]))` ($n \geq 0$): same as `setlog/5` in the previous item, but possibly attempting goal `G` as many times as the number of `OptsLst`$_i$ occurring in the argument of term `try`. Precisely, for each $i \in 1..n-1$, if the call

  `setlog(G,T,OutCLst,Res,OptsLst`$_i$`)`

  terminates with a time-out, then the call

  `setlog(G,T,OutCLst,Res,OptsLst`$_{i+1}$`)`

  is attempted next; otherwise, the initial call terminates. As an example, the following goal:

  ```
  ?- setlog(dom(R,S) & ran(R,{1/S}),2000,C,Res,
            try([[],[noran_elim]])).
  ```

  requires the goal `dom(R,S) & ran(R,{1/S})` to be executed, the first time, with a 2000 millisecond time-out and no execution mode options enforced; since this call terminates with a time-out, then the goal `dom(R,S) & ran(R,{1/S})` is executed again, with a 2000 millisecond time-out but activating the `noran_elim` option.

  Note that, when `setlog/5` is called with the `try` option, the total execution time might be as large as `T` * n, where n is the number of `OptsLst` occurring in the `try` list.

  {*log*} provides also `setlog/4` which is exactly as `setlog/5` but using a default value for the fifth argument, namely:

  ```
  try([[int_solver(clpfd)],[int_solver(clpq),final],
       [noirules],[noneq_elim],[noran_elim]]).
  ```

- `rsetlog(G,T,OutCLst,Res,Opts)`: same as `setlog/5`, but with "reification" on `Res`; specifically, if the execution of goal `G` terminates with failure, then `Res` is unified with `failure`; otherwise, `Res` is unified with either `success` or `time_out` or `maybe` as with `setlog/5`.

Other predicates

- `setlog_clause(Cl)`: to add a {*log*} clause `Cl` to the current {*log*} program
- `setlog_config(list_of_params)`: to modify some {*log*} configuration parameters directly from the Prolog environment. Each parameter in `list_of_params` can be:
  - `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'`);
  - `strategy(S)`: see the control predicate `strategy` in Sect. 11.
- `setlog_rw_rules`: to load the filtering rule library (automatically executed when accessing the {log} environment or calling a {*log*} goal from the Prolog environment).

12.2 From {*log*} to Prolog

General

- `prolog_call(G)`: to execute 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
-   @=</2
-   @>=/2
```

13 The {*log*} library

A number of common predicates, dealing with sets 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 ($\mathtt{PS} = 2^{\mathtt{S}}$)
- `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
- `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 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]`)

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] 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) (1996), 1–44.

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

[5] 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, ACM (2003), 219–229.

[6] Dovier, A., Pontelli, E., Rossi, G.: Set unification. Theory and Practice of Logic Programmming **6**(6) (2006), 645–701.

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

[8] Cristiá, M., Rossi, G.: A decision procedure for restricted intensional sets. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th Int'l Conf. on Automated Deduction, Lecture Notes in Computer Science, vol. 10395 (2017), 185–201

[9] Cristiá, M., Rossi, G.: A set solver for finite set relation algebra. In: Desharnais, J., Guttmann, W., Joosten, S. (eds.) Relational and Algebraic Methods in Computer Science - 17th Int'l Conf., RAMiCS 2018, Lecture Notes in Computer Science, vol. 11194, (2018), 333–349.

[10] Cristiá, M., Rossi, G.: Solving quantifier-free first-order constraints over finite sets and binary relations. J. Autom. Reasoning 64(2) (2020), 295–330.

A {*log*} commands

- `comp_elim/nocomp_elim` (default: `comp_elim`): to activate/deactivate complete rewriting of `comp` constraints
- `final/nofinal` (default: `nofinal`): to activate/deactivate final mode for constraint solving
- `fix_size/nofix_size` (default: `nofix_size`): to activate/deactivate fixed set cardinality mode for `size` constraint solving
- `int_solver(S)`, `S` = `clpq` | `clpfd` (default: `clpq`): to select the integer constraint solver
- `irules/noirules` (default: `irules`): to activate/deactivate inference rules
- `label/nolabel` (default: `label`): to activate/deactivate labeling on integer variables

- `mode(M)`, `M` = `prover` | `solver` (default: `prover`): to change the solver operation mode
- `neq_elim`/`noneq_elim` (default: `neq_elim`): to activate/deactivate `neq` elimination
- `ran_elim`/`noran_elim` (default: `ran_elim`)): to activate/deactivate complete rewriting of `ran` constraints
- `strategy(Str)`, `Str` = `cfirst` | `ordered` | `cfirst(list_of_atoms)`: to change goal atom selection strategy to `S`
- `show_min`/`noshow_min` (default: `noshow_min`): to activate/deactivate showing minimal set cardinalities making the input formula satisfiable
- `trace(T)`/`notrace` `T` = `sat` | `irules` (default: `notrace`): to deactivate/activate constraint solving tracing
- `type_check`/`notype_check` (default: `notype_check`): to activate/deactivate $\{log\}$ type-checking.