{*log*} User's Manual

Version 4.9.9 Release 1c

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://www.clpset.unipr.it/setlog.Home.html.

Contents

WHAT'S NEW

Release 1c

- Bug fixed in `applysp`; now the implementation covers more cases.
- Bug fixed in the typechecker when a variable was declared more than once.
- `groundsol` in typechecking mode extended to generate constants of product types of any number of types.

Release 1b

- Bug fixed concerning preprocessing of `let` predicates.
- Bug fixed in typechecking of `let` predicates.
- Bug fixed in standard partitions of `dres` and `dares`.
- Bug fixed in `exporttt`.
- Bug fixed in `gentc`.
- Bug fixed in the workflow of {*log*}-TTF.

Release 1a

- Bug fixed concerning the addition of sort constraints to `let` predicates.
- Bug fixed concerning the execution of simulations in the NEXT environment.

Version 4.9.9 Release 1

- Bug fixed concerning sort constraints for local variables in existential quantifiers.
- NEXT, a simple environment to assist in scenario execution (Sect. 13.2.2).
- TTF, test case generation for state machine specifications (Sect. 13.6).
- Negation of user-defined predicates is now automatic in many cases (Sect. 3.5).
- The help information has been updated and improved (Sect. 2.3).
- Sect. 3.5 has been extensively rewritten.
- State machines must declare at least one invariant and the `initial` declaration is mandatory (Sect. 13).
- Some *named singleton variables* are ignored in the computed answer. Helps in simplifying answers (Sect. 3.2) and when running prototypes (Sect. 13.2.1).
- Bug fixed concerning variable names. Now variable names of the form `_N`⟨*number*⟩ and `_X`⟨*number*⟩ are forbidden in the user input (Sect. 3).
- Several minor changes on how fresh variables are printed in the computed answer.
- {*log*} commands don't ask for more solutions.
- Bug fixed concerning input preprocessing and ground solutions.

1 Introduction

{*log*} (read 'set-log') is a Constraint Logic Programming language that embodies the fundamental forms of set designation and a number of primitive operations for set management [1, 4, 5, 7].

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

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 a property rather than by enumerating all their elements.

{*log*} inherits much of standard Prolog: its syntax (apart 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.[1]

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://www.clpset.unipr.it/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*}.

[1] As a minor difference with standard Prolog, note that the precedence of the module qualification operator (`:`)—see `op/3` and Defining a meta-predicate, in SWI-Prolog's on-line user's manual—has been changed from 600 to 350.

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 four additional files from the working directory (see also the `setlog_config/1` built-in predicate in Sect. 15.1):

- `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 programs; if this file is not loaded no type checking can be performed (see Sect. 12).
- `setlog_vcg.pl`: this file provides predicates that permit to work with state machines as in formal notations such as B and Z (see Sect. 13).

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 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. 15.1 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 16 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)`.[2].

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 the table of contents to get help on the following topics:

[2] 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.

syn	{*log*} syntax w.r.t. Prolog and syntax of set terms
cons	{*log*} constraints
q	quantifiers provided by {*log*}
type	{*log*} type system
sm	specification of state machines
next	Next, a simple environment for running state machines
opt	execution options
cmd	{*log*} user commands
lib	{*log*} library predicates
all	to get all available help information

- `h/1` (both from {*log*} and from the Prolog environment) provides more detailed information on the topic corresponding to the key shown in the first column above. For example, `h(syn)`, `h(cons)`, `h(cmd)`, etc. In turn the help displayed by some of these commands present their own tables of contents which lead to more detailed help.

When SWI-Prolog is called with `swipl`, help commands use a pager such as `less` to display help information when it's too long to fit in the terminal. In this way users can call all the pager's commands. For instance if `less` is used, its search function (e.g. /⟨*pattern*⟩) can be called. On the contrary, when SWI-Prolog is called with `swipl-win` help information is displayed without piping it to a pager.

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}.
```

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* [8].

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

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

as it has six:

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

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

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

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

has 30 solutions, among which:

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

In $\{x_1, \dots, x_n\ /\ A\}$, $x_1, \dots, x_n$ 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.

In addition to extensional set terms, sets can also be denoted by other types of set terms, specifically, Cartesian products (Sect. 4), intensional set terms (called RIS, Sect. 5), and integer intervals (Sect. 9).

Forbidden variable names

Variable names of the form `_N`⟨*number*⟩ (e.g. `_N3`) and `_X`⟨*number*⟩ (e.g. `_X341`) are forbidden in the user input. {*log*} raises an exception if this happens.

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 (see Sect. 3.4), 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. In addition to sorts, {*log*} defines a type system, see Section 12.

All these operations are dealt with as *constraints*, and, hence, they can be used with no concern about the instantiations of their arguments. The predicates listed in Table 1, along with those listed in Tables 3, 5, 6, 8, and 9, represent all and only the *atomic constraints* available in {*log*}.

3.2 Answers to goals, ground solutions and named singleton variables

{*log*} is a *satisfiability solver*. That is, it's a program that tries to find values for the free variables appearing in the goal as to make it to be true. In {*log*} goals are formulas combining sets, set operators and arithmetic. If {*log*} is able to find values for the free variables appearing in the goal as to make it to be true the goal is said to be *satisfiable*; otherwise the goal is said to be *unsatisfiable*. Whenever {*log*} answers `no` to a goal it means the goal is unsatisfiable. For example:

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)`	$A \subseteq B$
disjointness	`disj(A,B)`	$A \parallel B$
strict subset	`ssubset(A,B)`	$A \subset B$
symmetric difference	`sdiff(A,B,C)`	$C = A \vartriangle B$
NEGATIONS		
set	`nset(A)`	A is not a set
equality	`A neq B`	$A \neq B$
membership	`x nin A`	$x \notin A$
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)`	$C \neq A \setminus B$
subset	`nsubset(A,B)`	$A \nsubseteq B$
disjointness	`ndisj(A,B)`	$A \nparallel B$

Table 1: Set operators available in {*log*}

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

no
```

That is, given that {*log*} is unable to find a value for `B` as to make `un({1,2},B,{2})` (i.e. $\{2\} = \{1,2\} \cup B$) true it answers `no`.

Whenever the answer to a goal is different from `no` it means the goal is satisfiable. For instance, the first answer to the following goal:

```
{log}=> diff({1},A,B).
```

is the following:

```
A = {1/_N1}
Constraint: subset(B,{1}), 1 nin _N1, un(B,_N1,_N2), 1 nin _N2, set(_N2),
            1 nin B, set(_N1), disj(B,_N1), set(B)
```

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. Variables automatically generated by {*log*} are called *new* or *fresh* variables. {*log*} will include many fresh variables in its answers. In general, the answer returned by {*log*} is composed of:

- A (possibly empty) list of equalities of the form $v = t$ where v is a variable and t a term which can contain other variables. These equalities are called *bindings*.

- A (possibly empty) list of constraints which contain variables.

The constraints occurring in the computed answer are called *irreducible constraints*.

The above answer is interpreted as follows: `B` is the result of computing the set difference between `{1}` and `A` if and only if *there exist* sets `_N1` and `_N2` such that `A = {1/_N1}`, `B` is a subset of `{1}`, `1` doesn't belong to `_N1`, `_N2` is the union of `B` and `_N1`, `1` doesn't belong to `_N2` nor to `B` and `B` and `_N1` are disjoint. Hence, `_N1` and `_N2` are existentially quantified variables—where the existential quantifier is implicit. This interpretation can be easily and naturally generalized to every answer returned by {*log*}.

In many academic papers about {*log*} it has been shown that the conjunction of constraints returned by {*log*} is always satisfiable. In other words, there are always values that can be bound to the free variables appearing in the returned answer satisfying the conjunction of equalities and constraints. In many answers it is quite easy to find these values but this is not always the case. When there are no integer variables involved, a possible value for set variables is the empty set. That is, if set variables are substituted by the empty set the conjunction of the returned equalities and constraints is satisfied. For example, let's substitute `B`, `_N1` and `_N2` in the above answer by the empty set:

```
A = {1/{}}
Constraint: subset({},{1}), 1 nin {}, un({},{},{}), 1 nin {}, set({}),
            1 nin {}, set({}), disj({},{}), set({})
```

It's clear that `A = {1}` and that all the constraints are true. For example, `subset({},{1})`, interpreted as $\emptyset \subseteq \{1\}$, is obviously true; and `1 nin {}`, interpreted as $1 \notin \emptyset$, is also evidently true.

As we have said, {*log*} may return several answers to a given goal (just press letter 'y' when {*log*} asks you for more solutions). We have proved that *the disjunction of all these answers is equivalent to the original formula*. Since an answer may contain variables, it represents a (possibly infinite) set of ground (or concrete) solutions of the original formula. A *ground solution* is a solution where variables appear only at the left-hand side of the equalities composing a {*log*} answer. For example, we have just seen that `A = {1}, B = _N1 = _N2 = {}` is a ground solution of the first answer returned by {*log*} after executing `diff({1},A,B)`. Whereas the bindings `A = {1,2}, B = {}, _N1 = _N2 = {2}` represent another ground solution. So by changing `2` by any number different from `1` we get one more solution. When the initial formula belongs to one of the fragments of set theory for which {*log*} implements a decision procedure, {*log*} returns a finite number of answers. Hence, in these cases, *the disjunction of the answers returned by* {*log*} *is a finite representation of all the (possibly infinite) ground solutions of the initial formula*.

A `no` answer means the goal is unsatisfiable; any other answer means the goal is satisfiable.

Ground solutions. Just above we showed how to get a concrete solution from an answer returned by {*log*}. Basically, we substituted all set variables by the empty set. However, when integer variables are involved, finding a ground solution satisfying the computed answer is more complex. For these cases, and even for complex formulas not involving integer variables, {*log*} provides the predicate `groundsol` which forces {*log*} to produce *ground solutions*—i.e., solutions where variables occur only at the left-hand side of bindings.

Consider the following examples.

```
{log}=> groundsol.
{log}=> diff({1},A,B).

A = {1},
B = {}

{log}=> X in A.

X = n0,
A = {n0}
```

When `groundsol` is active, {*log*} binds set and integer variables to (ground) sets and integer numbers, respectively. For the remaining free variables, {*log*} binds constants of the form `n`⟨*number*⟩, where *number* is an integer number starting from zero.

To restore the default behavior issue `nogroundsol`.

`groundsol` also works (somewhat differently) when {*log*} is running in typechecking mode (see Section 12).

Hereafter, if not explicitly mentioned, all sample queries to {*log*} will be intended to be issued not in `groundsol` mode.

Named singleton variables. The answers to queries can also be modified due to the presence of some *named singleton variables*[3] (or just *named singletons*) in the user input. In {*log*} a named singleton is a variable whose name is of the form `_`⟨*Letter*⟩, where *Letter* is an uppercase letter[4]. For instance, `_W`, `_Q2` and `_Ta` are named singletons, but `__W`, `_v` and `_4` are not. In SWI-Prolog named singletons aren't included in the output under the default configuration[5]:

```
?- length([a,b,c],_L),functor(T,f,_L).     % _L named singleton, not printed
T = f(_, _, _).
```

{*log*} provides the same functionality as long as the answer remains correct. For example in the following goal, {*log*} can hide `_Q`:

```
{log}=> un({a,b},{1,2},_Q) & un(_Q,{x},A).

A = {a,b,1,2,x}
```

The idea is that we aren't interested in `_Q`'s value because we're forced to use it as an intermediate variable or partial result. Intermediate variables are pervasive in logic programming. For those habituated to functional-like programming languages, intermediate variables can be cumbersome. Named singletons may provide some relief in that regard.

However, {*log*} will be unable to hide named singletons when the terms bound to them have variables occurring in the constraints returned in the computed answer. For example in:

[3] SWI-Prolog: Singleton variable checking.

[4] {*log*}'s definition of named singletons is more restrictive than SWI-Prolog's.

[5] SWI-Prolog: `toplevel_print_anon` in Environment Control (Prolog flags).

Connective	{*log*}	Meaning
conjunction	`&`	$\wedge$
disjunction	`or`	$\vee$
negation	`neg`	$\neg$
implication	`implies`	$\Rightarrow$
not implication	`nimplies`	$\nRightarrow$
negation as failure	`naf`	$\neg$

Table 2: Propositional connectives available in {*log*}

```
{log}=> un(_A,B,{X/U}).

_A = {X/_N1}
Constraint: X nin _N1, un(B,_N1,U), X nin U
```

{*log*} can't hide `_A` because otherwise the solution would be wrong as we wouldn't know where `_N1` comes from. In this case `_A` is not being used as an intermediate variable, though.

Named singletons will be hidden when goals are executed in `groundsol` mode, as ground solutions don't return a list of constraints.

3.3 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.

3.4 Introducing formulas

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

The propositional connectives available in {*log*} are listed in Table 2. Formulas are built from these connectives in the usual way. Arguments of the connectives can be either atomic constraints, restricted quantifiers (see Sect. 6), user-defined or built-in predicates, as well as other {*log*} formulas, built in the same way.

As can be seen, 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)
```

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. Disjunction is managed as in Prolog, i.e., through non-determinism and backtracking. Hence, if a variable name appears in two or more disjuncts then it actually represents a different variable in each disjunct.

3.5 Negation and the let predicate

Negation is available through predicates naf and neg. naf computes the classical "Negation as Failure" of Prolog: naf(G), fails if G has a solution, and succeeds otherwise. neg computes the propositional negation of its argument, possibly applying Boolean laws. In particular, if the argument is a {*log*} constraint, then the language itself provides its negation. For example, neg(X in A & Z nin C) becomes X nin A or Z in C.

Both forms of negation, however, may work incorrectly in general. More specifically, naf works well only when variables in the formula are properly instantiated; neg may not work well when the negated formula contains unrestricted existentially quantified variables *inside the formula*.

As an example of the latter, consider the following user-defined predicate:

```
singleton_set(X) :- X = {Y}.                                    (2)
```

which is true when `X` is a singleton set. In $\{log\}$, as in Prolog, variables that occur only in the body of a clause (e.g., `Y`) are all implicitly existentially quantified (excluding *local* or *bound* variables occurring in intensional sets—Sect. 5—and restricted quantifiers—Sect. 6). These variables are called *existential variables*. Hence, solving the body of this clause amounts to solve the following formula:

$$\exists X : (\exists Y : X = \{Y\}) \tag{3}$$

If we want also the negative version of this predicate, then we can define a new predicate, say `notsingleton_set(X)`, which is true whenever `X` is not a singleton set (e.g., `{}`, `{1,2}`, `1`,...). Hence, the clause body of `notsingleton_set` should express the formula:

$$\exists X : \neg (\exists Y : X = \{Y\}) \equiv \exists X : (\forall Y : X \neq \{Y\}) \tag{4}$$

Observe the existentially quantified variable inside the formula to be negated. The definition of `notsingleton_set(X)` using `neg` is as follows:

```
notsingleton_set(X) :- neg(X = {Y}).
```

In this case, however, solving the clause body amounts to solve the formula:

$$\exists X : (\exists Y : X \neq \{Y\}) \tag{5}$$

Note the difference between the formula obtained by using `neg` (i.e. (5)) and the actual negation of (3) (i.e. (4)). Thus, for example,

```
{log}=> notsingleton_set({}).
```

correctly answers `yes`, but issuing

```
{log}=> notsingleton_set({1}).
```

we get

```
true Constraint: S neq {_N1}
```

instead of `no`.

Alternatively, one can use `naf` for expressing negation instead of `neg`. In this case, however, the formula obtained by solving the clause body of `notsingleton_set` is:

$$\forall X, Y : X \neq Y \tag{6}$$

since, in general, all variables in a predicate which is negated through `naf` are dealt with as universally quantified. Thus, for example,

```
{log}=> notsingleton_set({}).
{log}=> notsingleton_set({1}).
```

correctly answer `yes` and `no`, respectively, but issuing:

```
{log}=> notsingleton_set(S).
```

we get a `no` answer which is evidently wrong.

Note that implementing the other form of formula (4), i.e. $\exists X : (\forall Y : X \neq \{Y\})$, is not feasible in the current version of $\{log\}$, since the general form of universal quantification required by this formula is not supported yet.

`singleton_set` can be written, however, in such a way that negating it by means of `neg` works correctly.

```
singleton_set(X) :- size(X,1).                              (7)
```

where `size` computes the cardinality of its first argument (see Sect. 8). Note that the body of this clause does not contain any existential variable. In this way:

```
notsingleton_set(X) :- neg(size(X,1)).
```

is equivalent to:

```
notsingleton_set(X) :- nsize(X,1).
```

which coincides with the logical negation of `singleton_set`. Yet another encoding of `singleton_set` that doesn't introduce existential variables is the following:

```
singleton_set(X) :- X neq {} & foreach([A in X, B in X],A = B).         (8)
```

where `foreach` is introduced in Sect. 6 and `A` and `B` are local variables, not existential variables, because they are bound or local to the quantifier.

Nevertheless, in general, is not always possible to find a $\{log\}$ encoding of a given mathematical formula whose negation is correctly computed by `neg`.

The `let` predicate. Say we want to define a predicate to compute the following:

$$(A \cup B) \cap C$$

In $\{log\}$ it can be encoded as follows:

```
ex(A,B,C,R) :- un(A,B,U) & inters(U,C,R).
```

See that we need to introduce an existential variable (`U`) to get a partial result.

Now, say we want to compute the negation of `ex`. As we have explained in the introduction to this section, in general, this is not currently feasible in $\{log\}$ if the predicate includes existential variables (e.g. `U`). However, sort to speak, the 'nature' of `U` here is different from the nature of `Y` in (2). In `ex` `U` is simply a "name" for $A \cup B$. Actually, the interpretation of `ex` is the following:

$$\exists A,B,C,R : (\exists U : U = A \cup B \wedge R = U \cap C) \tag{9}$$

By virtue of the one-point rule[6] this formula is equivalent to the following:

$$\exists A,B,C,R : (R = U \cap C[A \cup B/U]) \tag{10}$$

[6] One-point rule: $(\exists x_1, \dots, x_n : x_1 = e_1 \wedge \cdots \wedge x_n = e_n \wedge \phi) \equiv \phi[e_1/x_1, \dots, e_n/x_n]$, because the existential quantifier can be realized only by substituting each x_i by e_i due to the equalities $x_i = e_i$. $\langle e_1, \dots, e_n \rangle$ is the only (one) point where the formula can be true.

which in turn reduces to $\exists A,B,C,R : R = (A \cup B) \cap C$. Therefore, the negation of `ex` is:

$$\exists A,B,C,R : R \neq (A \cup B) \cap C \tag{11}$$

which *doesn't need the introduction of universal variables*.

Hence, for these cases (which appear frequently in {*log*}) the language provides the `let/3` predicate. We can use it to capture the fact that we are naming a subexpression or capturing a partial result as follows:

```
ex(A,B,C,R) :- let([U], un(A,B,U), inters(U,C,R)).
```

When `let/3` is used, {*log*} can easily compute the negation of a predicate by means of `neg`. In fact, {*log*} works as follows when negating `ex`:

```
neg(let([U], un(A,B,U), inters(U,C,Res)))
```

which is rewritten as:

```
un(A,B,U) & ninters(U,C,Res)
```

which is {*log*}'s encoding of (11) (recall that `ninters` is the negated version of `inters`, see Table 1).

The `let/3` predicate is inspired in the following logical version of the widely used 'let' expression:

$$\text{LET } x_1,\dots,x_n \text{ BE } x_1 = e_1,\dots,x_n = e_n \text{ IN } \phi \mathrel{\widehat{=}} \exists x_1,\dots,x_n : x_1 = e_1 \land \cdots \land x_n = e_n \land \phi$$

where x_i are variables, e_i terms such that x_i does not occur in e_i and ϕ is a formula. In turn, by one-point rule, the existential formula is equivalent to $\phi[e_1/x_1,\dots,e_n/x_n]$. Therefore, the general form of `let/3` in {*log*} is as follows:

$$let([x_1,\dots,x_n],\psi,\phi)$$

where:

- $x_1,\dots,x_n$, called *let-variables*, are all distinct variables appearing only inside the *let*.
- ψ is a conjunction of *functional predicates* and equalities of the form $X = t$, where X is a variable, t a term and X appears free in ϕ. A functional predicate is a predicate that behaves as a function for its last argument. For example, union, intersection, set difference, etc. are functional predicates. The last argument is called *result*. The `is/2` predicate is functional but on its first argument. The results of functional predicates and the variables occurring at the right-hand side of the equalities in ψ, must be let-variables. Each let-variable can appear as the result of only one functional predicate or at the right-hand side of only one equality[7].
- ϕ is a {*log*} formula.

[7] The current version of {*log*} doesn't implement all these restrictions, so users are responsible of enforcing them.

Using a let variable more than once in ψ. As we said above, each let-variable can appear as the result of only one functional predicate or at the right-hand side of only one equality. This doesn't forbid a let-variable to be used as a non-result argument of another functional predicate in ψ. The following example is a valid use of `let/3` and should clarify this point.

```
ex2(A,B,C,D,R) :- let([U,V], un(A,B,U) & un(U,C,V), inters(V,D,R)).
```

That is, `U` is the result (or name) of $A \cup B$ but is also used to compute `un(U,C,V)`. It would be a mistake to put `U` also in place of `V`.

Equalities. Equalities of the form $X = t$ in ψ are useful in situations as the following one.

```
sp(P) :- P = [X,Y] & X < 0 & Y > 0.
```

It is clear that `neg` will not work well when applied to `sp` due to the presence of `X` and `Y`. We can use `let/3` to remedy this situation.

```
sp(P) :- let([X,Y], P = [X,Y], X < 0 & Y > 0).
```

What are the subexpressions `X` and `Y` are naming in this case? In some mathematical notations one would write $P.1$ and $P.2$ to refer to the first and second components of P. So X (Y) is the name for $P.1$ ($P.2$).

Rewrite rules for the let predicate. $\{log\}$ implements two simple rewrite rules for `let/3`:

$$let([x_1,\dots,x_n],\psi,\phi) \longrightarrow \exists x_1,\dots,x_n : \psi \wedge \phi$$
$$neg(let([x_1,\dots,x_n],\psi,\phi)) \longrightarrow \exists x_1,\dots,x_n : \psi \wedge neg(\phi)$$

□

By now it should be clear that we can't use `let/3` in `singleton_set/1` as defined in (2) to avoid the introduction of an existential variable. Indeed, in `singleton_set/1` variable `Y` doesn't represent the name or result of some subexpression appearing in the predicate; sort to speak, in that case, `Y` is an authentic existential variable. Hence, although `let/3` helps to avoid the introduction of existential variables in many cases, its application is not universal. When `let/3` can't be applied try to find an alternative, free-of-existential-variables encoding of your predicate as we did with `singleton_set/1`—in (7) and (8). If this fails too, then you'll have to manually compute the negation of your predicate. If this still fails, then $\{log\}$ can't express your problem.

Negation of user-defined predicates. $\{log\}$ is able to compute the negation of user-defined predicates by means of `neg` if the following conditions are met:

- The predicate doesn't contain existential variables.

  An existential variable is a non-local variable occurring only in the body of a clause. A local or bound variable is a variable occurring only *inside* intensional sets (Sect. 5) and restricted quantifiers (Sect. 6).

- The predicate is defined in only one clause.

  This doesn't mean the predicate cannot be defined in terms of other predicates as long as they are defined in a single clause. For example, p/2 satisfies this condition.

  ```
  p(X,Y) :- q(X) & t(Y).

  q(X) :- X >= 1.

  t(Y) :- Y < 0.
  ```

- The predicate's head contains only variables.

  For instance if we define the following:

  ```
  sp([X,Y]) :- X < 0 & Y > 0.
  ```

  neg(sp(P)) won't work well as it expects sp's head to contain only variables.

In general, the hardest condition to met is the first one. For this reason, $\{log\}$ provides the let/3 predicate (see above) and restricted quantifiers—Sect. 6. In particular, restricted existential quantifiers can be used to turn some existential variables into local variables.

If the above conditions can't be met for a given user-defined predicate, the user must provide its negation by means of another user-defined predicate. If the signature[8] of a user-defined predicate is p/k, then its negation is a user-defined predicate with signature n_p/k. That is the prefix $n\_$ is added to p. In this way $\{log\}$ considers that n_p is the negation of p whereas the latter is the negation of the former: $neg(p)$ is n_p and $neg(n\_p)$ is p. Therefore, users are advised to refrain from using the prefix $n\_$ in other contexts. As an example, if we keep (2) as the definition of singleton_set then we need to manually define its negation. If we name the negation n_singleton_set, instead of notsingleton_set as we did, then we can apply neg directly, as for instance:

```
{log}=> neg(singleton_set({1})).
{log}=> neg(n_singleton_set({1})).
```

Note that if we define singleton_set as in (7) or (8), then we don't need to define n_singleton_set because $\{log\}$ will be able to automatically compute their negations.

If $\{log\}$ can't automatically negate a user-defined predicate p and does not find any clause defining n_p when calling $neg(p(t_1,\dots,t_n))$, it will use $naf(p(t_1,\dots,t_n))$ in its place and, if $p(t_1,\dots,t_n)$ is not ground, then it will print the warning message: using unsafe negation.

Implication. Finally, note that implication is defined in $\{log\}$ as follows:

```
P implies Q :- neg(P) or Q.
```

This means that implies works correctly much as neg does. $\{log\}$ also introduces nimplies as follows:

```
P nimplies Q :- P & neg(Q).
```

[8] Predicate indicator.

If `implies` or `nimplies` are used with user-defined predicates, recall the above discussion on the negation of user-defined predicates. A common pattern when {*log*} is used to prove unsatisfiability is:

$$\neg (p \Rightarrow q)$$

which {*log*} rewrites as:

$$p \land \neg q$$

Then, if q is a user-defined predicate, make sure {*log*} can automatically compute q's negation or there's a predicate named n_q in scope representing the negation of q.

`neg` is also implicitly used when solving some constraints on Restricted Intensional Sets (Sect. 5.1).

3.6 Proving unsatisfiability (i.e., proving theorems)

{*log*} can be used to prove that a given formula is unsatisfiable. The following is a trivial example.

```
{log}=> {X,Y} = {Z} & X neq Y.

no
```

As can be seen, {*log*} answers `no` meaning the formula is unsatisfiable (i.e., there are no values for `X`, `Y` and `Z` that can make the formula true).

The following is a more interesting example as we are proving a property of set union.

```
{log}=> neg(un(A,B,C) implies un(B,A,C)).

no
```

Clearly, if the above formula is unsatisfiable *for all finite sets* `A`*,* `B` *and* `C`, then the inner formula (`un(A,B,C) implies un(B,A,C)`) is valid (i.e., it's a theorem) *for all finite sets* `A`*,* `B` *and* `C`. So we have just proved that union is commutative *for all finite sets*. However, have a look at Sect. 3.5 to see some limitations of `neg`.

Hence, if you want to check if p is a theorem, call {*log*} on $\neg\, p$ and wait for a `no` answer.

In general, the capability of {*log*} to prove unsatisfiability depends on whether or not the formula belongs to a class of formulas for which {*log*} implements a decision procedure. {*log*} implements a decision procedure for formulas involving extensional sets, the operators of Table 1, and the first two connectives of Table 2. In coming sections we will show other fragments of set theory for which {*log*} implements decision procedures.

Although the use of `neg` in the above goal is safe, it can also be written without `neg` as follows:

```
un(A,B,C) & nun(B,A,C).
```

Indeed, note that $\neg (p \Rightarrow q) \equiv \neg (\neg p \vee q) \equiv (p \wedge \neg q)$. As nun is exactly $\neg$ un, then both formulas are equivalent.

Precisely, the introduction of other connectives besides conjunction and disjunction may compromise decidability, as indicated in Sect. 3.5. For example, as shown in that section, if negation is used in combination with existential variables, {*log*} may be not able to work correctly. Furthermore, if the user-defined predicates possibly involved in the formula are defined through recursive definitions, the proof of the formula may be non-terminating. Also, if some extra-logical built-in predicates are involved in the formula, the solver is not guaranteed to work correctly whenever variables are not enough instantiated.

Other cases in which {*log*} may be not able to behave as a decision procedure, e.g., when intensional sets are involved (Sect. 5.1), will be pointed out in next sections.

When {*log*} is used to prove unsatisfiability we say that it's used as an *automated theorem prover*. Proving unsatisfiability can be computationally very hard. The *prover* mode of operation (see next section) can help on that. Nevertheless, see Sect. 11 for a detailed account of user commands that might overcome this difficulty.

3.7 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).
```

Although it is hard to predict in advance which mode of operation will be the best for a given formula, as a rule of thumb, we can say that if you want to use {*log*} as a programming language, then the first choice would be the solver mode; and if you want to use it as an automated prover, then use it in prover mode.

In general, in prover mode {*log*} will be more efficient in concluding that the formula is unsatisfiable, than in solver mode. The prover mode has more advanced options that can further improve {*log*}'s efficiency when attempting to prove unsatisfiability, as is explained in Section 11. In turn, 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 in prover mode:

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

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 = {}
```

Note that if `groundsol` is activated when in prover mode, the `subset` goal above will return only one ground solution whereas in solver mode it will return the same four solutions.

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 [12]. 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 (CP) between sets `A` and `B` [11]. 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. {*log*} provides the predicate `pair(t)` (resp., `npair(t)`) to constrain a term `t` to be (resp., not to be) an ordered pair.

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 3.4 (i.e., by means of `&`, `or` and the other logical connectives). 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.

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).

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]}.
```

See more about types in $\{log\}$ in Section 12.

4.1 Relational operators

Table 3 lists all the relational operators, along with their negations, available in $\{log\}$ as atomic constraints. In turn, Table 4 gives the mathematical definition of each relational operator given in Table 3. All the arguments of all these predicates can be variables and set terms, but not RIS terms (Sect. 5.1) nor integer intervals (Sect. 9).

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)`	$S = A ⩤ R$
range restriction	`rres(R,A,S)`	$S = R \rhd A$
range anti-restriction	`rares(R,A,S)`	$S = R ⩥ A$
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)`	$S \neq A ⩤ R$
range restriction	`nrres(R,A,S)`	$S \neq R \rhd A$
range anti-restriction	`nrares(R,A,S)`	$S \neq R ⩥ A$
overriding	`noplus(R,S,T)`	$T \neq R \oplus S$
relational image	`nrimg(A,R,B)`	$B \neq R[A]$

Table 3: Relational operators available in $\{log\}$ (`R`, `S` and `T` are binary relations)

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	$A ⩤ R = \{(x,y) \mid (x,y) \in R \land x \notin A\}$
range restriction	$R \rhd A = \{(x,y) \mid (x,y) \in R \land y \in A\}$
range anti-restriction	$R ⩥ A = \{(x,y) \mid (x,y) \in R \land y \notin A\}$
overriding	$R \oplus S = (\operatorname{dom} S ⩤ R) \cup S$
relational image	$R[A] = \operatorname{ran}(A \lhd R)$

Table 4: Definition of relational operators

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. Thus, strictly speaking, the definitions in the first part of Table 3 should be extended by conjoining the predicates `rel` for all the relations occurring as arguments in the operators. For example, the precise meaning of `dom(R,A)` is $rel(R) \wedge \operatorname{dom} R = A$. Of course, similar observations apply to the negative versions of these predicates. For example, the precise meaning of `ndom(R,A)` is $nrel(R) \vee \operatorname{dom} R \neq A$. For example:

```
{log}=> ndom({[a,1]/R},{a}).
```

has two solutions:

```
R = {[_N3,_N2]/_N1} Constraint:  set(_N1), _N3 neq a

R = {_N2/_N1} Constraint: set(_N1), npair(_N2)
```

where the second one simply states that `R` in not a binary relation because it contains something that is not an ordered pair.

4.2 Partial functions

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:

$$\forall x, y_1, y_2 : (x, y_1) \in f \wedge (x, y_2) \in f \Rightarrow y_1 = y_2 \qquad (12)$$

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 [9]. These formulas, are built as formulas involving binary relations plus the addition of the predicates listed in Table 5.

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 an argument of one of the predicates implementing partial function operators, as shown in the first part of Table 5. 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*} gives as a first solution:

```
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 answers `no` for the following:

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

Operator	$\{log\}$	Meaning
partial function	`pfun(F)`	F verifies (12)
function application	`apply(F,X,Y)`	$F(X) = Y$
function application	`applyTo(R,X,Y)`	$\{(X,X)\} \circ R = \{(X,Y)\}$
domain	`dompf(F,A)`	$\operatorname{dom} F = A$
composition	`comppf(F,G,H)`	$F \circ G = H$
domain restriction	`drespf(A,F,G)`	$A \lhd F = G$
Negations		
partial function	`npfun(F)`	F does not verifies (12)
function application	`napply(F,X,Y)`	$F(X) \neq Y$
domain	`ndompf(F,A)`	$\operatorname{dom} F \neq A$

Table 5: Partial function operators available in $\{log\}$ (`F`, `G` and `H` are partial functions)

Note that, as observed for relational operators, strictly speaking, the definitions of the predicates listed in Table 5 should include the constraints `pfun` and `npfun` that are automatically added by $\{log\}$. Thus, for instance, the precise meaning of `dompf(F,A)` is $pfun(F) \land \operatorname{dom} F = A$; while the precise meaning of `ndompf(F,A)` is $npfun(F) \lor \operatorname{dom} R \neq A$.

$\{log\}$ provides also a less restrictive form of function application through the predicate `applyTo`. `applyTo(R,X,Y)` is true if `R` is a binary relation containing exactly one pair whose first component is `X` and whose second component is `Y`. Formally:

$$applyTo(R,X,Y) \Leftrightarrow \{(X,X)\} \circ R = \{(X,Y)\} \tag{13}$$

This means that one can use `applyTo` on a binary relation which is not a function but which is a function only for certain values of its domain. For example:

```
{log}=> applyTo({[1,2],[2,3],[2,4]},1,Y).
```

makes $\{log\}$ to answer `Y = 2`, even if `{[1,2],[2,3],[2,4]}` is not a function. Conversely:

```
{log}=> applyTo({[1,2],[2,3],[2,4]},2,Y).
```

makes $\{log\}$ to answer `no`, since `{[1,2],[2,3],[2,4]}` is not a function in `2`.

Observe that the negations of predicates `comppf`, `drespf`, and `applyTo` are not available (at present) as $\{log\}$ constraints, but they are provided as library predicates by the libraries `setloglibpf.slog` and `setloglib_tc.slog` (as such, they take the names `n_comppf`, `n_drespf`, and `n_applyTo`, respectively).

4.3 Decidable formulas involving binary relations

$\{log\}$ 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.

More precisely, such undesired behaviors may appear when `comp` constraints are involved, either explicitly or implicitly. The following are two simple formulas showing non-termination of $\{log\}$:

```
{log}=> comp({[X,Z]/R},{[Z,Y]/S},R).
{log}=> comp(R,S,{[X,Z]/S}).
```

Actually, with the first goal {*log*} enters an infinite loop producing no answer, while with the second one it returns an infinite number of solutions. Intuitively, the problem with the above formulas is caused by the presence of `comp` constraints where the first or second argument shares a variable with the third argument. Generally speaking, in these cases, {*log*} is not able to compute a finite representation for the possible solutions.

The sharing of a variable between the third and the other two arguments of a `comp` constraint can be also indirect, through some other constraint. For instance:

```
{log}=> comp(R,S,{[X,Z]/U}) & un(S,T,U).
```

Moreover, the sharing can be generated during constraint solving, even starting with a formula that initially does not contain any such situation.

In our experience, however, all these "unpleasant" formulas occur very rarely in practice.

It is worth noting that the presence of a variable shared in a `comp` constraint does not necessarily implies non-termination. For instance, the following formula:

```
{log}=> comp({[X,Z]/R},{[Z,Y]/S},R) & id(A,R).
```

contains one such unsafe variable sharing, but nevertheless {*log*} terminates, returning a finite number of solutions.

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) [10]. Another way is by using *general intensional sets* (GIS). The next subsection deals 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:

$$\{x : D \mid F(x) \bullet P(x)\} \tag{14}$$

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:

$$\{y : \exists x(x \in D \land F(x) \land y = P(x))\} \tag{15}$$

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 (14) is written as follows:

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

where D can be any kind of set except for cp and variable intervals (see Section 9); 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 and dompf (only as first argument) and pfun. This means that RIS cannot be used as arguments or as part of arguments passed in to relational operators (cf. Tables 3 and 5).

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., $\{x : [1, MD] \mid 0 = N \operatorname{mod} x\}$) 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 predicates occurring in a RIS formula can be not only any of the predefined predicates available in {*log*}, but also any user-defined predicate. Since in some cases the solver needs to negate the RIS formula, then, in general, it is necessary that the negated versions of all the user-defined predicates occurring in the RIS formula are in scope. In this regard, remember that (see Sect. 3.4) if p is the name of a user-defined predicate with arity n, the negated version of p must be named n_p. If the solver does not find any clause defining n_p, it will use naf p, possibly printing the warning message using unsafe negation.

The control variable of a RIS term is an existentially quantified variable whose scope is the RIS term itself—in other words, the control variable is *local* to the RIS term. In its current form, however, {*log*} doesn't check whether or not this variable is used outside the RIS term. It's the user's responsibility to avoid such name clashes which may make {*log*} to produce unexpected behaviors.

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[9]. The semantics of a parameter is an existentially quantified variable. Then, the semantics of `CF` is given by the following intentional set:

$$\{y : \exists s(\exists c(s \in D \land |s| = c \land c > 1 \land y = (s,c)))\} \tag{16}$$

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. As an example, the following formula:

```
R = {[1,a],[1,b],[3,a],[2,c],[3,b]} & A = {1,2} &
ris(X in A,[Y],[X,Y] in R,[X,Y]) = {}.
```

is found to be satisfiable, whereas it is clearly not. In fact, the given RIS defines the domain restriction of a binary relation `R` with respect to a set `A`, and with the given values for `R` and `A` it represents the set `S = {[2,c],[1,a],[1,b]}`.

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 first RIS at the beginning of this subsection, 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.

As can be seen, parameters and the functional section in a RIS behave as the `let` construct (see Sect. 3.5). In fact we have the following:

`ris(X in A,[`$vars$`],`ϕ`,`p`,`ψ`)` $=$ `ris(X in A,[],let([`$vars$`],`ψ`,`ϕ`),`p`)`

[9] As with control variables, {*log*} doesn't check whether or not the formula respects this restriction; hence, it's the user's responsibility to avoid using parameters outside of RIS terms.

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.[10] Then, for example, the RIS in the formula with a wrong answer in the previous subsection can be alternatively written without using parameters but using a control expression:

```
R = {[1,a],[1,b],[3,a],[2,c],[3,b]} & A = {1,2} &
ris([X,Y] in R,X in A) = {}.
```

where the RIS pattern is omitted as is expected to be the control expression. In this case, {*log*} will correctly find that the formula is unsatisfiable.

As another example, if `R` is a set and we want the subset of `R` whose elements are ordered pairs of integer numbers 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)
```

Observe that this set cannot be expressed with the set and relational operators of Sections 3 and 4 nor with functional predicates (because ≥ is not such a predicate).

As the above RIS do not introduce parameters, every formula including them 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,

[10] The variables in control expressions are subjected to the same locality restriction of control variables and parameters.

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 $P : \{x : x \in D \wedge F(x)\} \rightarrow Y$ 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:

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

where: X is a variable; 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, predicates containing RIS terms are dealt with as constraints, while predicates containing GIS terms are always replaced by new predicates 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).

6 Quantifiers

{*log*} provides a form of *restricted quantifiers* (RQ) where the quantified variable ranges over a {*log*} set. There are *Restricted Universal Quantifiers* (RUQ) and *Restricted Existential Quantifiers* (REQ). RUQ are supported through predicates `foreach` and `forall`, while REQ are supported through predicate `exists`. Furthermore, the current version of {*log*} provides also a form of general existential quantifiers.

6.1 `foreach` and `exists`

The main way to express RUQ and REQ in {*log*} is by means of the `foreach/2`, `foreach/4`, `exists/2` and `exists/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 `foreach/2` while allowing the introduction of parameters and functional predicates (cf. Section 5.1). Note that quantified variables of a RQ are meant to be local to the RQ.[11] Same considerations as with RIS apply to the use of the `let` construct in RQ (see at the end of Sect. 5.1.1).

[11] In its current form {*log*} doesn't check whether or not local variables are used outside the RQ. It's the user's responsibility to avoid such name clashes which may make {*log*} to produce unexpected behaviors.

- `exists(X in A,F)`
  where arguments are as in `foreach/2`. The semantics of such a constraint is the expected:

  $$\exists x : x \in A \land F$$

- `exists(X in A,[params],F,functional predicates)`
  which behaves as `exists/2` while allowing the introduction of parameters and functional predicates (cf. Section 5.1).

{*log*} provides also the negated versions of `foreach/2`, `foreach/4`, `exists/2` and `exists/4`, namely `nforeach` and `nexists`. 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., $\exists x : x \in A \land \neg\, F$. Actually, `exists/2` is implemented in terms of `nforeach` as follows:

```
exists(X in A, F) :- nforeach(X in F,[],neg(F))
```

Note that, then, `exists` may not work well in all cases due to the presence of `neg`. The same holds for `exists/4`.

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.
  ```

RQ can occur everywhere ordinary predicates are allowed. In particular, RUQ and REQ can be nested at any depth, i.e., the formula of a RUQ can be a RUQ itself and so on; and the formula of a REQ can be a REQ itself and so on. Decidability of formulas including RQ is described below. The following is a safe formula showing a nested RUQ:

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

which 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:

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

In order to simplify the introduction of nested RQ, {*log*} allows to write a list of constraints of the form `X in A` in `foreach` and `exists`. For example:

```
foreach([X1 in A1,...,Xn in An],...)
```

For instance, the nested RUQ shown above stating that `S1` and `S2` are disjoint sets can be written more compactly as follows:

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

Why REQ? The introduction of REQ in the language deserves a special note. Since logic programming allows for the introduction of existential variables, the presence of REQ might seem unnecessary, but it is not. REQ help in further extending the fragment of the language where negation can be easily computed. Consider the following simple predicate.

```
p(S) :- X in S & X > 0.
```

The negation of `p` cannot be easily computed due to the presence of `X`, an existential variable—recall Sect. 3.5. However, `p` can be written in terms of a REQ—thus changing an existential variable for a bound one—with the same meaning.

```
p(S) :- exists(X in S, X > 0).
```

Now `neg(p(S))` can be easily computed:

$$\neg\, p(S) \equiv \neg\,(\exists X \in S : X > 0) \equiv (\forall X \in S : \neg\,(X > 0)) \equiv (\forall X \in S : X \leq 0) \tag{17}$$

where the rightmost predicate is written in {*log*} with a RUQ: `foreach(X in S, X =< 0)`.

Observe that we can't use `let/3` in `p` to avoid the introduction of `X`.

Decidability of formulas including RQ. The theory of restricted quantifiers is parametric w.r.t. some quantifier-free, decidable theory. However, the parameter theory must verify the following condition: if its solver, as a result of solving a formula, returns a conjunction of atoms including set variables, this conjunction must be satisfiable by substituting set variables by the empty set. This is not the case of a theory including `size/2` (Sect. 8) or a theory including integer intervals (Sect. 9), but theories including set and relational operators of Tables 1 and 3 can be accepted. If this condition is not met, the results provided by {*log*} are unreliable.

Once the above condition is met, the decidability of formulas including RQ is as follows. Let `{.../A}` be the domain of a RQ where `A` is a variable. In this case we will say that `A` is the *domain variable* of the RQ[12]. The innermost formula of a RQ is called its *quantifier-free formula*. For instance, `X neq Y` is the quantifier-free formula of the nested RUQ encoding `disj` shown above. Formulas including `foreach` and `exists` are safe provided at least one of the following conditions is met (for technical details see [14]):

1. The formula contains only `exists` and the quantifier-free formula belongs to a decidable fragment.
2. The formula contains only `foreach`, the quantifier-free formula belongs to a decidable fragment, and none of the domain variables are used in the quantifier-free formula.
3. The formula contains `foreach` and `exists` but all the `foreach` occur *after* any `exists`. Here 'after' means that the only nested RQ are of the form:

   ```
   exists([$X_1$ in $A_1$,...,$X_n$ in $A_n$],
     foreach([$Y_1$ in $B_1$,...,$Y_m$ in $B_m$], formula)
   )
   ```

   where `formula` is a quantifier-free formula fitting in a decidable fragment, and the nested `foreach` verifies 2.
4. The formula contains `foreach` and `exists` but doesn't belong to the above class (i.e., some `exists` occur after some `foreach`). In this case the condition for decidability is as follows: the quantifier-free formula must belong to a decidable fragment, no `exists` occurring *after* a `foreach` share the same domain variable, and no domain variable of a `foreach` is used in the quantifier-free formula. For instance, the following two formulas *do not* verify the above condition:

   ```
   foreach(X in {W / A}, exists(Y in {V / A}, formula))

   foreach(X in {H / A}, exists(Y in B, formula_1)) &
   foreach(Z in B, exists(W in {Q / A},formula_2))
   ```

   Note that in the first case there's an `exists` after a `foreach` sharing variable `A`. The second case is more complex because is harder to see that there's an `exists` after a `foreach` sharing variable `A` as they are in different RUQ. However these two RUQ are connected through domain variable `B`—see [14] from Example 5 to Theorem 3.

If these conditions aren't met then {*log*} will most likely run forever when is called on such a formula but it may not, depending on the quantifier-free formula inside the RQ.

6.2 forall

{*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)

[12] Clearly, if the domain of a RQ is just a variable then this is the domain variable.

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,true)
```

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`.

6.3 General existential quantifiers

{*log*} provides also a simple form of general existential quantifiers, where the quantified variable is not required to range over a specified domain:

```
exists(X,F)
```

where X is a (single) variable and F is any {*log*} formula.

This form of quantification can be used, for instance, to make explicit the otherwise implicit existential quantification of fresh variables in clause bodies, as well as of the so called parameters of RIS and RQ. As an example, predicates singleton_set and not_singleton_set defined in Sect. 3.5 can be equivalently defined as:

```
singleton_set(X) :- exists(Y,X = {Y}).
not_singleton_set(X) :- neg(exists(Y,X = {Y})).
```

In this case, by issuing the goal

```
{log}=> not_singleton_set({1}).
```

{*log*} recognizes it is using a form of negation that it can't handle, and so it tries to use naf in place of neg, printing the warning message: using unsafe negation. This means that the answer *might be incorrect*—although in this particular example it is correct.

As another example, the following formula

```
D = {[a,b],[1,1],[1,2]} & S is ris(Z in D,[X,Y],Z = [X,Y] & X neq Y).
```

using a RIS with two parameters X and Y, can be equivalently written without parameters but using nested existential quantifiers:

```
D = {[a,b],[1,1],[1,2]} &
S is ris(Z in D,[],exists(X,exists(Y,Z = [X,Y] & X neq Y))).
```

In both cases, the use of the existential variables X and Y inside the RIS is unsafe. In the second case, however, {*log*} tries to use naf in place of neg, printing the warning message concerning unsafe negation. Again, in this specific case, the use of naf allows the solver to obtain a correct answer (namely, S = {[1,2],[a,b]}).

It is worth noting that by using restricted existential quantifiers instead of the general ones we always get a reliable answer. For instance, the definition of not_singleton_set can be easily generalized by allowing the domain of Y to be an argument of the predicate, i.e.,

```
not_singleton_set(X,D) :- neg(exists(Y in D,X = {Y})).
```

In this case, even if both X and D are left unspecified, we still get a correct answer:

```
{log}=> not_singleton_set(S,R).

true
Constraint: set(R), foreach(_X in R,neg S={_X})
```

7 Solving formulas including integer numbers

{*log*} deals with arithmetic expressions through a number of predefined predicates. The comparison arithmetic operators available in {*log*} are shown in Table 6. In the table, e1 and e2 are arithmetic expressions, and n is a either a variable or a numeric constant. An arithmetic

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 6: Comparison arithmetic operators available in {*log*}

Function	{*log*}	Meaning
addition	`e1 + e1`	$e_1 + e_1$
substraction	`e1 - e1`	$e_1 - e_2$
product	`e1 * e1`	$e_1 \times e_2$
division	`e1 / e1`	e_1 / e_2
integer division	`e1 div e1`	$e_1 \,\mathsf{div}\, e_2$
integer module/remainder	`e1 mod e2`	$e_1 \,\mathsf{mod}\, e_2$

Table 7: Arithmetic functions available in {*log*}

expression is either a variable or a numeric constant or an arithmetic function (see Table 7) 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

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

As Prolog, {*log*} does not evaluate arithmetic expressions unless they occur as parameters in one of the predicates listed in Table 6. 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 predicates of Table 6 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
```

In fact, predefined arithmetic 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 (namely, CLP(Q)). 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 considerations 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*} constraint that is true if and only if `X` is an integer number. There is also its negated version `ninteger` (see Table 8).

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.

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

Table 8: `integer` and `ninteger` constraints

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 (see Section 9). *Finite domains* are specified through ground intervals, i.e., intervals with constant limits.

Finite domains are associated to integer variables through membership constraints. The formula:

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

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

The last two goals above, give the correct answers 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).                    (18)
no
```

7.1.2 Labeling

{*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 (18) 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 predicate `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, and division and remainder are restricted to 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 ground 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.[13] 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.

`div` and `mod` in CLP(Q). Since `div` and `mod` are not available over CLP(Q), one can use Euclid's division lemma as a way to compute them. Then, say that we want to test a number for parity:

$$0 = x \bmod 2$$

In {*log*} over CLP(Q) the correct way is to find the quotient of $x/2$:

```
{log}=> X is 2*Q.
```

where `Q` is a new variable. As can be seen, this is a linear constraint and so CLP(Q) can always return the right answer.

How {*log*} uses CLP(Q). {*log*} uses CLP(Q)'s `bb_inf/4` predicate to determine whether a conjunction of linear integer constraints is satisfiable over the integers. See the CLP(Q) documentation[14] for further details.

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

[14] https://www.swi-prolog.org/pldoc/doc_for?object=bb_inf/4

Non-linear expressions in CLP(Q). If the formula passed to CLP(Q) contains non-linear expressions the returned answer is unreliable. {*log*} alerts the user with a proper warning message. For example:

```
{log}=> A is X*Y & B is Y*X & A neq B.

***WARNING***: non-linear expression over CLP(Q); possible unreliable answer

true
Constraint: A is X*Y, B is Y*X, A neq B
```

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. In this situation the `solver` mode should be preferred over the `prover` mode—cf. Section 11. 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.

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 9.

The first argument of both predicates can be either a variable or a set term, including integer intervals and CP terms, but not RIS terms. 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. Sect. 11). 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 9: 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.[15]

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.
```

8.1 Decidable formulas involving cardinality constraints

{*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, 6 and 8.

   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 CP term, or 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 and CLP(Q) is the active integer solver.

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

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.

8.2 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
```

`groundsol` (Section 3.2) produces even more concrete solutions also for formulas involving cardinality constraints. 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.[16] 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. If both interval limits are constants we say that `int(m,n)` is a *ground interval*, while if one of the limits is a variable it is a *variable interval*. Note that limits of variable intervals 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
```

An interval `int(m,n)` where `m > n` denotes the empty set.

9.1 Decidable formulas involving integer intervals

If the input formula fulfills the following conditions:

- only operators of Tables 1 and 9 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))`;

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

- intervals can also occur as the set part of extensional sets, e.g., {-1/int(1,N)}.

When operators other than those in Tables 1 and 9 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 (i.e., only ground intervals are involved). 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 3 and 5. Intervals cannot be the arguments of CP terms.

9.2 Defining set operators using intervals

Some powerful set operators can be defined using (variable) intervals and {*log*} can automatically reason about them within the decidable fragment. These are gathered in the {*log*} library setloglibIntervals.slog. Here we comment on one of them—see Sect. 10 for more.

Integer intervals can be used to 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,_)).
```

In this way, 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).                [the second argument is a variable]
M = -1
Constraint: 2=<_N1, 5=<_N1, 9=<_N1
```

And 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.

10 Aggregation functions

{*log*} provides some aggregation or aggregate functions[17] within its decision procedures, and some others can be defined outside of them.

[17] Wikipedia.org: Aggregate function

10.1 Minimum and maximum of a set

{*log*} provides predicates `smin(S,min)` and `smax(S,max)` where `S` is a set and `min` and `max` can be integer constants or variables. These predicates are true when `min` (`max`) is the minimum (maximum) of set `S`. Both predicates are within the decidable fragments implemented in {*log*} as long as the second argument participates only in linear integer arithmetic (and the active integer solver is CLP(Q)).

The minimum of a set can be defined in terms of RUQ or in terms of integer intervals—same considerations apply to the maximum of a set. `smin` is defined in terms of RUQ.

```
smin(S,M) :- M in S & foreach(X in S, M =< X).
```

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

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

{log}=> smin({2,X,-1,9,0},M).                    [one element is a variable]
M = X
Constraint: X=<2, X=< -1, X=<9, X=<0

Another solution?  (y/n)y
M = -1
Constraint: -1=<X
```

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

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

The library `setloglibIntervals.slog` includes a predicate that computes the minimum of a set, called `setmin`, defined in terms of intervals:

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

10.2 Sum of a set

The predicate `sum(Set,Sum)` computes the sum of a set. The first argument of `sum` can be either a variable or an extensional set or an integer interval with constant limits. 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[18]:

[18] Here we assume CLP(Q) is the active integer solver; if CLP(FD) is active, labeling is unnecessary as is performed automatically.

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

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:

```
{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 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, as a first step, consider the following section.

10.3 Sum of an array

This section and its implementation in {*log*} are still experimental.

Arrays can be defined by the following {*log*} formula:

```
arr(A,N) :- 0 < N & pfun(A) & dom(A,int(1,N)).
```

That is, an array A of length N is a function whose domain is the interval $[1,N]$. For example:

```
{log}=> arr({[1,5],[2,3],[3,8],[4,2]},4).
yes
```

but also:

```
{log}=> arr(A,4).
A = {[1,_N4],[2,_N3],[3,_N2],[4,_N1]}
```

The decidability of arrays in {*log*} has not been deeply studied, yet. However, as a rule of thumb, it is possible to say that {*log*} formulas containing `arr/2` belong to a decidable fragment if the cardinality of arrays, and anything related to them, is never taken. For example, the following is a formula laying *outside* that decidable fragment:

```
arr(A,N) & ran(A,R) & size(R,K) & K < N
```

because we are computing the cardinality of the range of array A. Taking the cardinality of some set containing some of the elements of an array will be useful in some circumstances. However, according to preliminary experiments, there are many non-trivial problems where taking the cardinality of some set related to an array isn't necessary. This implies that those problems belong to the decidable fragment.

Summing the elements of an array is one of those problems. Assuming A is an array of length at least K, the following predicate computes the sum of the first K elements of A.

```
arrsum(A,K,Sum) :-
  arr(Tr,K) &
  applyTo(A,1,X) &
  Tr = {[1,X],[K,Sum] / Tr1} & [1,X] nin Tr1 & [K,S] nin Tr1 &
  foreach(I in int(2,K),[I1,Y,Z,Si],
    [I,Si] in Tr,
    I1 is I - 1 & applyTo(Tr,I1,Z) & applyTo(A,I,Y) & Si is Y + Z
  ).
```

As an example:

```
{log}=> arrsum({[1,5],[2,3],[3,8],[4,2]},2,S).
S = 8
```

The sum is performed by computing the partial sums in each component of *Tr*. In effect, the following is true of $arrsum(A, K, Sum)$:

$$Tr(1) = A(1)$$
$$Tr(2) = Tr(1) + A(2)$$
$$Tr(3) = Tr(2) + A(3)$$
$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$
$$Tr(k) = Tr(k-1) + A(k)$$

In a sense, *Tr* records the state trace of the standard imperative algorithm computing the sum of an array. Consequently, arrsum/3 would be an abstract or logical representation of that algorithm.

As can be seen, arrsum/3 never takes the cardinality of anything related to an array. So, according to our first studies, arrsum/3 belongs to the decidable fragment.

11 Using {*log*} as an automated theorem prover

As we have said in Sect. 3.6, proving unsatisfiability can be computationally very hard. {*log*} may take an unpractical time in proving that a given formula is unsatisfiable. In this section we present several user commands and facilities that can significantly improve {*log*}'s efficiency when used as an automated theorem prover.

11.1 Alternative rewrite rules and execution options

{*log*} is basically a rewrite system that applies a set of rewrite rules to the input formula until a fixpoint is reached. Users can slightly change the set of rewrite rules that {*log*} will apply to the input formula by means of so-called *execution options*. Each execution option activates an *alternative rewrite rule* for a particular constraint while at the same time deactivates the corresponding default rewrite rule. These options are meant to improve {*log*}'s efficiency when used as an automated theorem prover—although it's hard to predict which option will have a positive or a negative influence in the proof of a *particular* theorem.

There are two *independent* ways of setting the execution options.

- Using the `mode` command (Sect. 3.7) as follows:

  ```
  {log}=> mode(prover([opt_1,...,opt_n])).
  ```

  with $0 \leq n$ and where each opt_i is listed in the first column of Table 10. `mode(prover)` is equivalent to `mode(prover([]))` meaning that no execution option is activated (which in turn implies that the default rewrite rules are applied). When {*log*} is loaded, `mode(prover)` is automatically executed.

  When execution options are activated in this way they influence all the goals executed directly from the {*log*} environment.

- Using `prover_all_strategies/1` as follows:

  ```
  {log}=> prover_all_strategies([mod,[opt_1,...,opt_n]]).
  ```

  where *mod* can be the *modifiers* `all` or `all_single`, and opt_i are the same as above. This predicate can also be used to get the current set of active execution options by passing it a variable:

  ```
  {log}=> prover_all_strategies(X).

  X = [mod,[opt_1,...,opt_k]]
  ```

  The default value of `prover_all_strategies/1` is `[all,`*dftlist*`]` where *dftlist* is the list of all options of Table 10.

  When execution options are activated in this way they influence all the goals executed by means of the commands `t_solve/1`, `t_solve/3` and `p_t_solve/1`, which are explained below.

  The *modifiers* `all` and `all_single` are explained in the next section.

As can be seen in Table 10, some of the execution options implement a rewrite rule in terms of RUQ. While this considerably speeds up the execution of many goals, it also tends to slow down many others and in general produces more complex solutions when the goal is satisfiable.

As we have said, it's very hard to predict what execution options will have a positive influence when proving some theorem. Then, users should try out different combinations of execution options in the hope that one of them solves the goal in a reasonable time. Given Table 10, the current number of possible combinations is 128. However, since some rules have influence only on particular constraints, if the theorem to be proved doesn't include some such constraint then the corresponding option can be ignored. Yet the number of possible combinations could still be too high. Next section shows how to deal with this in a more automatic way.

See Section 15.1 to know how to activate execution options when {*log*} is called from Prolog.

OPTION	AFFECTS	DESCRIPTION
`subset_unify`	equality	Implements set equality as a double set inclusion instead of implementing it by exploiting set unification. In this way, for example, $\{log\}$ generates two solutions instead of four for `{X / A} = {Y / B}`. In other words, `subset_unify` is closer to one of the standard proofs techniques when it comes to set equality because it assumes the existence of an element in one set and tries to prove that it belongs to the other set, and vice versa.
`un_fe`	union	Implements $un(A,B,C)$ in terms of RUQ (Sect. 6) when C is not a variable and A or B are variables. In this way $\{log\}$ produces two answers instead of six when solving $un(A,B,\{X/C\})$, with A and B variables. These two answers encode the standard proof of $x \in A \cup B$ by considering $x \in A$ and $x \in B$.
`comp_fe`	composition	Implements $comp(R,S,T)$ (Sect. 4) in terms of RUQ when T is not a variable and R and S are variables. In this way, for example, $comp(R,S,\{[1,a],[2,b],[3,c]\})$ quickly returns only one solution if `comp_fe` has been activated whereas under the default configuration it will take longer to return the first solution and even much longer to return other solutions.
`oplus_fe`	overriding	Implements $oplus(R,S,T)$ (Sect. 4) in terms of RUQ. The default rewrite rule for *oplus* is implemented in terms of *dom*, *dares* and *un* which in many cases requires a hard computation. When `oplus_fe` is active, for example, the proof of unsatisfiability of $oplus(\{X,Y,Z/R\},S,T) \land noplus(\{X,Y,Z/R\},S,T)$ takes about one second whereas the default rule takes a very long time.
`ran_fe`	range	Implements $ran(R,B)$ (Sect. 4) in terms of RUQ.
`noirules`	several	By default $\{log\}$ applies some optional inference rules while the goal is processed. noirules deactivates the application of these inference rules.
`strategy(ordered)`	disjunction	Changes the order in which atoms are processed. It has shown to impact proofs of the form $p \land (q \lor r) \land s$, with p,q,r,s atoms. With the default strategy, $\{log\}$ first rewrites p and s, say into t, and then solves $t \land q$ and $t \land r$. Instead, with strategy(ordered) the proof is first transformed into two subproofs ($p \land q \land s$ and $p \land r \land s$) and then the atoms are rewritten in the order they appear.

Table 10: Execution options

Decidability with alternative rewrite rules. {*log*} preserves the same decidability results when one or more of the alternative rewrite rules are activated. In other words, the solver doesn't become "less" or "more" complete by activating one of the execution options of Table 10. There can be differences in the solutions provided by {*log*} and even in how it doesn't terminate, but when it terminates the set of solutions with or without execution options are equivalent. In a sense, calling {*log*} with different execution options is like calling different solvers whose completeness (or lack of it) is the same, although their execution speeds tend to be different.

Concerning the differences in non-termination when {*log*} is called with different execution options consider the following example.

```
{log}=> dom(R,A) & ran(R,{X/A}).    % default config., blocks immediately

{log}=> mode(prover([subset_unify])).
{log}=> dom(R,A) & ran(R,{X/A}).    % infinite number of solutions

R = {[X,X]/_N2},
A = {X/_N1}
Constraint: dom(_N2,_N1), set(_N1), X nin _N5, comp(_N4,{[X,X]},_N4),...
```

Hence, in the default mode one can't tell whether or not this particular formula is satisfiable, whereas when `subset_unify` is active {*log*} finds it satisfiable but it's impossible to get a finite representation of all its solutions. This last point is crucial when this formula is part of a larger, unsatisfiable one because {*log*}, in general, won't be able to prove that.

11.2 Parallel execution

The command `p_t_solve(`*G*`)` solves *G* by running *G* in parallel in multiple (operating system) threads. Each thread runs *G* with some combination of the execution options listed in `prover_all_strategies`. The number of threads and the combinations of execution options in use depend on the current value of `prover_all_strategies` as follows:

- `[all,`*list*`]`: all the possible combinations of the elements of *list* are used in an equal number of threads. More precisely: {*log*} computes the powerset of *list*, creates a thread for each element of the powerset, activates the corresponding options and runs *G* in each thread.

  For example, considering the default value of `prover_all_strategies`, `p_t_solve(`*G*`)` will create 128 threads that will run *G* in parallel, each with a different set of execution options activated.

- `[all_single,`*list*`]`: {*log*} creates a thread for each element of *list*, activates the corresponding option and runs *G* in each thread.

  For instance, if the current value is `[all_single,[un_fe,comp_fe,oplus_fe,ran_fe]]`, `p_t_solve(`*G*`)` will create 4 threads that will run *G* in parallel, each with a different execution option (among `[un_fe,comp_fe,oplus_fe,ran_fe]`) activated.

As soon as one thread terminates the whole computation terminates as well[19]. In this way, the net execution time will tend to be the time needed by the thread running the best

[19] {*log*} uses SWI Prolog's `first_solution/3` to implement parallel execution.

combination of execution options for that proof. Note that, however, if the machine has less cores than the number of threads, thread scheduling will increase the time needed also by the fastest one. Hence, either run your goal on a machine with enough cores, or reduce the number of execution options in `prover_all_strategies` as explained in the previous section, or wait longer.

Besides, `p_t_solve(`*G*`)` executes *G* for at most 1 minute[20]. This parameter can be changed with the following command:

```
{log}=> timeout(number).
```

where *number* can be a positive integer number representing an amount measured in milliseconds; or a variable, if the user wants to consult the current timeout. If {*log*} is unable to solve the goal before the timeout is met, the answer returned by `p_t_solve` will be `timeout`.

The following examples illustrates different behaviors on an eight cores standard laptop.

```
{log}=> prover_all_strategies(X).
  X = [all,[subset_unify,comp_fe,un_fe,oplus_fe,ran_fe,noirules,
            strategy(ordered)]]
{log}=> timeout(2000).
{log}=> p_t_solve(oplus({X,Y,Z/R},S,T) & noplus({X,Y,Z/R},S,T)).

timeout
```

{*log*} is unable to solve the goal in less than two seconds when the default value of `prover_all_strategies` is considered, in spite that there's at least one combination of execution options that would solve the goal in less than 2 seconds. This is because the operating system has to schedule 128 threads using 8 cores making the "good" threads to use the computer as much as the "bad" ones. Now we set three execution options, with `oplus_fe` among them, which amounts to eight possible combinations (i.e., the number of available cores). In this way the goal is solved in less than 1.5 seconds.

```
{log}=> prover_all_strategies([all,[oplus_fe,subset_unify,un_fe]]).
{log}=> timeout(1500).
{log}=> p_t_solve(oplus({X,Y,Z/R},S,T) & noplus({X,Y,Z/R},S,T)).

no
```

That is, now the operating system has to schedule just eight threads, plus other operating system processes, using eight cores. In this way our threads will be able to use the computer more time, than in the previous experiment, allowing one of them to solve the goal faster. Finally, we run the same goal with the `all_single` modifier with a one second timeout:

```
{log}=> prover_all_strategies([all_single,[oplus_fe,subset_unify,un_fe]]).
{log}=> timeout(1000).
{log}=> p_t_solve(oplus({X,Y,Z/R},S,T) & noplus({X,Y,Z/R},S,T)).

no
```

[20] For this reason, `p_t_solve` stands for *parallel, timed solve*.

Hence, by using less threads they probably can use the computer for as long as they need, allowing $\{log\}$ to solve the goal even faster. But in this case the user needs to analyze what options are likely to speed up the proof.

Extreme parallelism and timeouts. When a goal is run in parallel using many more threads than the number of available cores, threads might miss out timeout signals. As a consequence, threads will run beyond the expected timeout. This is usually the case when `p_t_solve` is called when a "high" number of execution options belong to `prover_all_strategies`. As this number goes down, threads tend to miss less timeout signals. Clearly, this number is high or not in relation to the number of available cores. Note that the number of threads created by `p_t_solve` grows exponentially w.r.t. the number of executions options in `prover_all_strategies` when the `all` modifier is used, but grows linearly when `all_single` is used. The fact that threads tend to miss timeout signals under these conditions, is outside of the control of $\{log\}$.

What if the goal is satisfiable? Parallel execution in $\{log\}$ is meant to be used to prove unsatisfiability. However, users might not know in advance whether or not the goal is unsatisfiable. In case the goal is satisfiable, $\{log\}$ will return a first solution as in the normal case but it won't be able to compute more solutions. Parallel execution in $\{log\}$ cannot deal with non-determinism[21]. Hence, you will know the goal is satisfiable but you won't be able to call for more than one solution.

11.3 Other user commands

There are two more user commands that can be exploited when using $\{log\}$ as a theorem prover.

- `t_solve(`*G*`)` executes *G* for as long as the current timeout—i.e, the only difference with the normal interactive execution is the timeout.
- `t_solve(`*G*`,`*timeout*`,`*exec_conf*`)` executes *G* for as long as *timeout* and using the configuration given by *exec_conf*. Possible values for *exec_conf* are:
  - The empty list. Behaves as setting the current timeout to *timeout* and then calling `t_solve(`*G*`)`.
  - A list whose elements are the execution options listed in Table 10 and Appendix A. The execution options passed in the list are activated during the execution of *G*.
  - `try([`$optList_1$`,...,`$optList_n$`])`, where each $optList_i$ is a list of execution options as in the previous item. *G* is first executed with $optList_1$: if the execution doesn't timeout, no more is done and `t_solve` terminates, with either success or failure; if the execution timeouts, *G* is executed with $optList_2$. This process continues until all the elements of the list are attempted. For more information see `setlog/5` with option `try` in Sect. 15.1.
  - `tryp([`$optList_1$`,...,`$optList_n$`])`, where each $optList_i$ is a list of execution options as in `try`. $\{log\}$ creates a (operating system) thread for each element of the list. Each of these threads is configured with the corresponding *optList* and tries to solve *G*. As soon as one thread terminates, the whole computation is bring to an end. All threads

[21] This is a consequence of the same limitation of SWI Prolog's `first_solution/3`.

are terminated as soon as *timeout* is reached. Since the goal is executed in parallel, only one solution will be returned if the goal is satisfiable.

- `try(prover_all)`. The powerset of the current value of `prover_all_strategies` is computed in list *power_set* and the call `t_solve(`G`,`*timeout*`,try(`*power_set*`))` is executed.
- `try(prover_all_single)`. Let $[opt_1,\dots,opt_n]$ be the current value of `prover_all_strategies`, then the call `t_solve(`G`,`*timeout*`,try(`$[[opt_1],\dots,[opt_n]]$`))` is executed.
- `tryp(prover_all)`. The powerset of the current value of `prover_all_strate-gies` is computed in list *power_set* and the call `t_solve(`G`,`*timeout*`,tryp(`*power_set*`))`is executed.
- `tryp(prover_all_single)`. Let $[opt_1,\dots,opt_n]$ be the current value of `prover_all_strategies`, then the call `t_solve(`G`,`*timeout*`,tryp(`$[[opt_1],\dots,[opt_n]]$`))` is executed.

Although, `try` and `tryp` may look similar, they aren't. $\{log\}$ will return only one solution under `tryp` if the goal is satisfiable, but it will have a normal behavior under `try` if one execution attempt doesn't timeout. `try` is purely sequential; in this sense, it's like executing all the threads of `tryp` one after the other.

12 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,"messi"]}
```

It is also possible to operate with those sets:

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

As in Prolog, variables are not declared and 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 type system would deem this formula ill-typed and would reject it before any attempt on deciding its satisfiability is made.

Besides, types can help programmers 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 typechecker at will. If the typechecker is active all variables in formulas and clauses must be declared to be of a certain type and the typechecker is called before a formula is executed (in interactive mode) or when a file is consulted. The typechecker 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,set(t)) & dec(R,rel(t,t)).
```

Each `dec` constraint states that a variable is of a certain type. For instance, `dec(R,rel(t,t))` states that `R` is a binary relation with domain and range of type `t`. If this formula is run while the typechecker is *not* active, {*log*} will simply ignore the `dec` constraints. This would cause a loop, as explained above. Instead, if the typechecker 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 set([t,t])
            A is of type set(t)
```

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

```
id(set(T),rel(T,T))
```

for some type `T`. The error clearly informs that such a type can't be found given the types of the arguments.

Typing formulas is good but complicates some formulas such as the first one seen in this section:

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

However, {*log*} provides a type system where an encoding of the set `{a,1,{2},[5,"messi"]}` can be correctly typed—see Encoding untyped sets.

In the following section the type system and other related features are introduced. In Section 13 the reader will find a complete example of a typed {*log*} program.

12.1 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 typechecker is active *all* variables must be declared to have *exactly one* type. These declarations are made by means of the `dec/2` constraint, called *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 one 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.

12.1.1 Integers

`int` is a type representing the set of integer numbers (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 > 10` 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.

12.1.2 Character strings

`str` is a type representing the set of character strings. There are no special purpose operators defined over `str` in {*log*} 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' doesn't fit in the sum type
{log}=> "messi" = messi.
type error: 'messi' doesn't fit in the sum type
```

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

12.1.3 Basic types

Any *atom* can be used as a type. All these types are called *basic types*.

For example:

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

are all possible type constraints declaring variables of basic types. Then, we have the following:

```
{log}=> dec(A,address) & dec(N,name) & A neq N.
type error: in A neq N
            A is of type address
            N is of type name
```

In some notations (e.g., Z) the structure or form of the elements of basic 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 basic types.

In {*log*}, because is a programming language, basic types are associated to a known set of elements. If t is a basic type, then all its elements are of the form t:⟨*atom*⟩, for any atom. For example, if we want to bind the atom john to variable N of type name, then:

```
{log}=> dec(N,name) & N = john.
***ERROR***: type error: 'john' doesn't fit in the sum type
```

while

```
{log}=> dec(N,name) & N = name:john.
 N = name:john
```

and

```
{log}=> dec(N,name) & N = city:john.
***ERROR***: type error: in N=city:john
        N is of type name
        city:john is of type city
```

The following are further examples of how the :/2 operator works:

```
{log}=> t:A = u:a.
type error: t:A isn't well defined
{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
```

In type-checking mode ':'/2 is a reserved symbol. It can only be used as described in this section. See Section 12.9 to learn more about admissible terms in type-checking mode.

Basic types vs. strings (str). Note that replacing basic types with str might cause some problems. Using str instead of basic 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[22]:

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

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 basic 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
```

[22] idef_type is explained in Section 12.2.

12.1.4 Enumerated types

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

For example:

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

are all enumerated types. So a declaration such as:

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

will constrain `A` to be bound to only those three values. Hence, for instance:

```
{log}=> dec(Color,enum([red,blue,green])) & Color = blue.
 Color = blue
```

while by issuing

```
{log}=> dec(Color,enum([red,blue,green])) & Color = yellow.
```

we get a type error.

In an enumerated type, each e_i must be an atom, different from all the basic types in scope and from all other atoms declared in other enumerated types, even if occurring in different goals or different clause bodies—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. In this sense, enumerated types are *persistent* and have a *global validity*. The `reset_types` command can be used to delete all type declarations currently in use (see Section 12.10) thus removing all the enumerated types being used.

Note that `enum([yes,no])` is a different type than `enum([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.

See Section 12.2 to learn how to give a name to enumerated types so you do not have to repeat the enumeration in each type constraint.

12.1.5 Sum types

Enumerated types are a particular class of *sum types*. Sum types implement the widely known notion of *variant*[23] (heavily used in functional programming languages).

For example, the following are all sum types:

```
sum([nil,some(str)])
sum([null,num(int),pair([int,int])])
sum([null,num(int),pair(int,int)])
sum([col(enum([red,blue,green])),other(sum([nil,some(str)]))])
```

[23] Wikipedia.org: Tagged union

In the first case the type is composed of the values nil and any value of the form some(s) where s is of type str. The second type is composed of the values null, all the values of the form num(i) where i is of type int, and all the values of the form pair([i,j]) where i and j are of type int and [i,j] is of a product type—see Section 12.1.6. The third type is isomorphic to the second one. The following is a possible goal using the second sum type:

```
{log}=> dec(X,sum([null,num(int),pair([int,int])])) & X = pair([1,2]).
 X = pair([1,2])
```

A sum type is given by a list of terms of any arity. The arguments of non-nullary terms must be types. The head symbol of each of these terms is called *constructor*. So, for instance, nil and some are constructors. This emphasizes the fact that elements of a sum type are built or constructed according to the constructors that define the type.

Internally, the enumerated type enum([e_1,...,e_n]) is rewritten as sum([e_1,...,e_n]). Then, enumerations are no more than sum types whose constructors are all nullary terms. For this reason, for now on, whenever we mention a sum type it includes also the case of an enumeration.

In a sum type, each constructor must be an atom, different from all the basic types in scope and from all other atoms declared in other sum types—and from int and str which can only be used where a type can be used. All constructors in the list must be different from each other. The list must contain at least two elements. Like enumerations, sum types have a global validity, which persists between one goal and another (in interactive mode) and between one loaded program and another. One can use the reset_types command to delete all type declarations currently in use—see Section 12.10.

These restrictions imply that if you want to introduce, for example, two or more *option types*[24] you must use different constructors. In other words you can't use none and some for all the option types. For example:

```
dec(X,sum([nil,some(str)])) & dec(Y,sum([nil,some(int)])).
```

produces the following type error:

```
type error: in dec(Y,sum([nil,some(int)]))
    type sum([nil,some(int)]) is not well-defined
```

because nil is part of two different sum types. In this case the correct declarations would be:

```
dec(X,sum([nils,somes(str)])) & dec(Y,sum([nili,somei(int)])).
```

This generalizes to more complex sum types.

12.1.6 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:

[24] Wikipedia.org: Option type

```
dec(B,[city,enum([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).

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

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

12.1.7 Set types

If t is a type then set(t) is a type representing all the sets whose elements are of type t. In other words, set(t) represents the powerset of t. Hence, if X is of type set(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 set type constructor. For example, the following is the type of the un constraint:

```
un(set(T),set(T),set(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 a set type.

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 set(T), for some type T.

Encoding untyped sets. The combination between the set and sum types allows to encode untyped sets. For example, the set used in the introduction to this section:

```
{a, 1, {2}, [5,"messi"]}
```

can be casted in terms of a set type combined with a sum type:

```
dec(S,set(sum([a,n(int),s(set(int)),p(int,str)]))) &
S = {a, n(1), s({2}), p(5,"messi")}
```

12.1.8 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:

```
set([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`. Since this type is frequently used in {*log*}, the `rel` type constructor is introduced as a synonym of a type based on a set and a product type:

```
rel(t,u) == set([t,u])
```

Precisely, all the relational operators available in {*log*} have types based on `rel`. For example the following is the type of the `comp` constraint:

```
comp(rel(T,U),rel(U,V),rel(T,V))
```

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

The differences between `rel(R)` (cf. Table 3) and `dec(R,rel(t,u))` are the following:

1. `rel` is part of {*log*}'s inference engine; `dec(R,rel(t,u))` is used only by the typechecker 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,rel(t,u))`.
3. `rel(R)` forces the elements of `R` to be ordered pairs but it does not state what the type of those pairs is. Then, if only `rel(R)` is asserted, `{[1,a],[{t},[x,1]]}` is a possible value for `R`.

   Instead, if `dec(R,rel(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,rel(t,u))`, but it's automatic.

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

12.2 Type declarations

Sometimes 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 [13]:

```
rel(obj,[int,set(cat)])
```

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

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

For these situations {*log*} offers the `idef_type/2` and `def_type/2` commands. The first one is used in goals (in interactive mode), whereas the second is used in program clauses. For example:

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

That is, `def_type(t,texpr)` and `idef_type(t,texpr)` state that `t`, an atom, is a name for type `texpr`. The effect of these commands is global and persistent. Afterwards it holds that:

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

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

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

Naming sum types is another convenient use of the `[i]def_type` commands:

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

See Section 12.10 to learn how type declarations can be consulted and managed.

12.3 Typing RUQ and REQ

RUQ and REQ (Section 6) are typed as any other constraint in {*log*} with the exception that bound variables need not be typed—although they can be typed if users wish to do so. For example, the following is type correct, in spite that there's no type declaration for X:

```
dec(A,set(int)) & foreach(X in A, X > 0)
```

{*log*} will infer X's type from A's. However, X can be explicitly typed as follows:

```
dec(A,set(int)) & foreach(X in A, dec(X,int) & X > 0)
```

Nevertheless, if the explicit type isn't the correct one, {*log*} will issue a type error message:

```
{log}=> dec(A,set(int)) & foreach(X in A, dec(X,t) & X > 0).

***ERROR***: type error: in dec(X,t), variable X is already declared
```

In effect, when {*log*} processes the RUQ it attempts to find out the type for X (which is int, given A's type), but when it processes the inner formula the type inferred by {*log*} for X doesn't coincide with the type declared by the user.

Same considerations apply when a control expression is used:

```
dec(A,set([int,t])) & foreach([X,Y] in A, X > 0)
```

However, parameters used inside RUQ and REQ must be typed by the user with a dec predicated inside the RQ:

```
dec(A,set(t)) & dec(F,rel(t,int)) &
foreach(X in A,[Y], Y > 0, applyTo(F,X,Y) & dec(Y,int)).
```

The typechecker doesn't support forall (Section 6.2) nor general existential quantifiers (Section 6.3).

12.4 Typing user-defined predicates

If you activate the typechecker and you want to consult a file containing a collection of {*log*} clauses (i.e., a {*log*} program), then you first need to type 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 a file then you have to type the predicate, before loading the file in type-checking mode:

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

The first line simply declares a name for the type [name,address]—this is optional. In the second line 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 set(na) states that the first argument of add_person must be of that type; 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 declaration must be in place. Directives and facts (i.e., unit clauses) are not typed.

Note that before re-consulting a file or consulting a new one, you have to reset all type declarations—see Section 12.10 to learn how to do this. The abolish/0 command does not remove global information about types.

dec_p_type/1 is a reserved word of the language when the typechecker is active. It cannot be used in other contexts.

12.5 Typing polymorphic predicates

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

```
un(set(T),set(T),set(T))
comp(rel(T,U),rel(U,V),rel(T,V))
id(set(T),rel(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 predicates 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(set(T),set(T),set(T),set(T))).
un3(A,B,C,D) :-
  dec(X,set(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 a 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.

12.6 Running formulas in type-checking mode

If formulas are run when the typechecker is 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],set(t)) & un3(W,X,Y,Z).
```

Note that in the dec 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],set(t)) & un3({t:a,t:aa},X,Y,Z).
```

Ordered pairs can also be used as expected. For example:

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

12.7 groundsol when typechecking is active

groundsol can also be used in typechecking mode (see Section 3.2). Differently from the untyped case, in typechecking mode groundsol returns constants of the appropriate type for each variable. For example:

```
{log}=> dec(X,set([int,enum([a,b])])) & size(X,3).

X = {[0,b],[1,b],[2,b]}

{log}=> dec(X,set([t,str])) & size(X,3).

X = {[t:n0,n0],[t:n1,n1],[t:n2,n2]}
```

Observe the difference in the output when the same goals are run after turning off typechecking:

```
{log}=> notype_check.
{log}=> dec(X,set([int,enum([a,b])])) & size(X,3).

X = {n0,n1,n2}

{log}=> dec(X,set([t,str])) & size(X,3).

X = {n0,n1,n2}
```

Recall that type information is ignored when the typechecker isn't active. Then, {*log*} processes only size(X,3) in both goals. Since the elements of X have no type, {*log*} binds them to constants of the form n⟨*number*⟩, as explained at the end of Section 3.2.

On the contrary, when typechecking is active {*log*} generates constants of the appropriate types. For instance, in the first goal the elements of `X` are ordered pairs whose first component is an integer number and the second is either `a` or `b`. In the second goal, the first components are of type `t`, whereas the seconds are of type `str`. Then, `t:n0` is a first component because the elements of type `t` are of the form `t:`⟨*atom*⟩, as explained in Section 12.1.3. In turn, the second components are of the form `n`⟨*number*⟩ because they are strings.

In general, {*log*} generates the following constants when `groundsol` is called in typechecking mode:

- `int` → `0,1,-1,2,-2,...`
- `str` → `n0, n1, n2,...`
- basic type `t` → `t:n0, t:n1, t:n2,...`
- sum types → the nullary constructors are used as constants; if a non-nullary constructor `x` depends on type `T` then constants of the form `x(c)`, with `c` of type `T`, are generated.
- product type → ordered pairs where the first components and the second components are constants of the corresponding types.
- set type → the constants are singleton sets where the elements are of the corresponding type.

The combination between `groundsol` and type checking can't be used to solve user-defined predicates including existential variables.

12.8 Proving goals involving finite types

Clearly, the following goal is unsatisfiable:

```
{log}=> dec(Z,enum([t,f])) & Z neq t & Z neq f.
```

because the only two values `Z` can take are exactly `t` and `f`. When working in typechecking mode, {*log*} automatically transforms that goal into:

```
Z in {t,f} & Z neq t & Z neq f.
```

thus answering `no`. However, if the typechecker isn't active, the same goal is found to be satisfiable because the `dec` predicate is ignored and `Z` can take any value beyond `t` and `f`.

As another example, consider the following goal:

```
dec(F,rel(enum([t,f]),int)) & pfun(F) & F = {X1,X2,X3} &
dec([X1,X2,X3],[enum([t,f]),int]) & X1 neq X2 & X1 neq X3 & X2 neq X3.
```

As `F` is a partial function and given the `neq` constraints, the first components of `X1`, `X2` and `X3` must be different from each other. At the same time, these first components have type `enum([t,f])`. So at least two of these first components must have the same value. Consequently the goal is unsatisfiable. As with the first goal, {*log*} identifies this situation and automatically conjoin suitable membership constraints to make type information available to the constraint solver. Note that an alternative encoding of the above goal without types is:

```
foreach([X,Y] in F, X in {t,f} & integer(Y)) &
pfun(F) & F = {X1,X2,X3} &
X1 neq X2 & X1 neq X3 & X2 neq X3.
```

which is found to be unsatisfiable, as well. These situations arise when the formula involves finite types and entails, in a way or another, "too many" `neq` constraints involving variables whose type is finite. In this context `T` is a finite type if:

1. `T` is an enumerated type (i.e., a sum type with only nullary constructors).
2. `T` is the sum of finite types. This means that all the constructors of the sum type take values belonging to finite types.
3. `T` is the product of at least one finite type. One may be tempted to define this rule as: `T` is the product of finite types. However, the second example above shows that this alternative definition is wrong as `[enum([t,f]),int]` wouldn't be finite but that formula is unsatisfiable. Conversely, according to our definition, `[enum([t,f]),int]` is a finite type because `enum([t,f])` is a finite type.
4. `T` is the set type of a finite type.

12.9 Admissible terms in type-checking mode

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

```
{log}=> [a|[]] = 1.
type error: '[a]' doesn't fit in the sum type
```

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
- Non-nullary terms – sum 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. In particular, the type checker doesn't support general intensional set terms (Section 5.2).

12.10 User commands to work with types

The following user commands are available to deal with types when using {*log*} in interactive mode.

- `idef_type/2`. Besides being used for type declarations as described in Section 12.2, 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}=> idef_type(t,E).
  E = set([obj,ac])
  Another solution?  (y/n)
  no
  ```

- `reset_types/0` deletes all currently active type declarations. These include declarations made through the following predicates: `[i]def_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(pred)` where `pred` is an atom, prints the type of `pred` as given by either a `dec_p_type` or a `dec_pp_type` declaration for `pred`.

  For example:

  ```
  {log}=> type_of(dres).
  dres(set(T),set([T,U]),set([T,U]))

  yes
  ```

- `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, with respect to the currently active declarations.

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

13 Specifying and verifying state machines

{*log*} can be used to specify state machines in a similar fashion as in set-based, state-based formal notations such as B [6] and Z [2]. These state machines can be executed as if they were functional prototypes (Sect. 13.2). Furthermore, {*log*} provides declarations that are used to automatically generate verification conditions on those state machines (Sect. 13.3 and 13.5). Later, users can use {*log*} itself to automatically prove (Sect. 13.3) or disprove (Sect. 13.4) these verification conditions. Finally, {*log*} provides the implementation of a model-based testing method to automatically generate test cases from the state machines (Sect. 13.6).

The features described in this section can only be used if files `setlog_vcg.pl` and `setlog_ttf.pl` are present in the working directory.

We will show how to work with state machines by encoding in {*log*} the classic birthday book problem used by Spivey to illustrate the Z formal notation.

13.1 Specification of state machines

In {*log*} a state machine is composed of:

1. Zero or more (model or specification) parameters
2. One or more state variables
3. Zero or more axioms
4. One or more state invariants
5. A predicate defining a set of initial states
6. One or more state operations (i.e., state transitions)
7. Zero or more theorems

Users must declare these elements in that order, except for theorems which can be declared anywhere in the specification after the first axiom.

We assume the birthday book specification is saved in a file named `bb.slog`.

Hence, we start by declaring the state variables of the birthday book. The birthday book is a system which records people's birthdays, and is able to issue a reminder when the day comes round. As Spivey, we will use two state variables: `Known`, holding the names of the people in our book; and `Birthday`, mapping names to birthdays. Then, we add the following fact at the beginning of file `bb.slog`:

```
variables([Known,Birthday]).
```

where `variables` is used to declare the state variables of a state machine. It receives as an argument a list of *distinct* variables—i.e., no constants or compound terms are allowed; the same variable cannot be more than once in the list. `variables` is a reserved word when it comes to the specification of state machines using `setlog_vcg.pl`. There can be only one `variables` declaration per state machine.

The intention is for `Known` to be the domain of `Birthday` and for the latter to be a partial function. As these properties are meant to be state invariants we introduce the following declarations and predicates:

```
invariant(birthdayBookInv).
birthdayBookInv(Known,Birthday) :- dom(Birthday,Known).

invariant(pfunInv).
pfunInv(Birthday) :- pfun(Birthday).
```

That is, before introducing the predicate stating an invariant we declare it to be an invariant by means of the `invariant` declaration—a reserved word. This declaration takes as its argument the head (without arguments) of the predicate stating the invariant.

There can be one or more `invariant` declarations. Any sensible state machine will have at least one of them. If you can't find an invariant for your specification and you still think this is right, then declare something like this:

```
invariant(sillyInv).
sillyInv(X) :- X = X.
```

where `X` is any state variable.

All the invariant declarations must appear right after the `variables` declaration and before other declarations we will shortly introduce. In turn, an invariant is a predicate that depends on at least one state variable. As can be seen, `birthdayBookInv` depends on the two state variables whereas `pfunInv` depends only on `Birthday`. Invariants may also depend on model parameters.

Declaring a predicate as an invariant *doesn't* mean that the state machine indeed verifies it. The `invariant` declaration is just a declaration of intent. However, {*log*} will use these declarations to automatically generate verification conditions that will make evident whether or not the state machine verifies these invariants—see Section 13.3. These verification conditions will eventually ask {*log*} to negate each and every invariant. As invariants are given as user-defined predicates {*log*} might not be able to compute the expected negation—recall Sect. 3.5. Hence, users should check invariants to see whether or not they met the conditions given in Sect. 3.5 for {*log*} to be able to compute their negations. In any case, if a user-defined predicate doesn't meet the conditions for negation, when {*log*} is asked to compute that negation it will issue a warning message.

> **Negation of invariants**
>
> Check invariants to see if their negations can be computed by `neg`. Review Sect. 3.5.

{*log*} will be able to compute the negation of the invariants given in the birthday book specification—they don't contain existential varibles, are given in a single clause and their signatures contain only variables.

We can combine the above declarations with type declarations as follows:

```
def_type(bb,rel(name,date)).
def_type(kn,set(name)).

invariant(birthdayBookInv).
dec_p_type(birthdayBookInv(kn,bb)).
```

```
birthdayBookInv(Known,Birthday) :- dom(Birthday,Known).

invariant(pfunInv).
dec_p_type(pfunInv(bb)).
pfunInv(Birthday) :- pfun(Birthday).
```

After introducing invariants we have to introduce a predicate defining a set of initial states. In the case of the birthday book we have the following:

```
initial(birthdayBookInit).
dec_p_type(birthdayBookInit(kn,bb)).
birthdayBookInit(Known,Birthday) :- Known = {} & Birthday = {}.
```

As `invariant`, the `initial` declaration takes as argument the head of the predicate defining the initial state. `initial` is another reserved word.

The `initial` declaration is mandatory. It must come after the last invariant and before the first operation. The predicate defining the initial states of the system must depend on at least one state variable, and can depend on model parameters.

The last component of a state machine is one or more state operations or state transitions. For example, in the birthday book we have an operation to remind the user of all the persons whose birthday is in a given date.

```
operation(remind).
dec_p_type(remind(kn,bb,date,kn,kn,bb)).
remind(Known,Birthday,Today,Cards,Known,Birthday) :-
  rres(Birthday,{Today},M) & dec(M,bb) &
  dom(M,Cards).
```

Again, the `operation` declaration takes as argument the head of the predicate defining the operation. It is a reserved word, too.

The above operation doesn't change the state of the system. We can tell that because there are no next-state variables in its head. A next-state variable is a state variable ending with the underscore character ('_'). For instance, `Known_` would be a next-state variable in the birthday book specification. In notations such as Z, `Known_` would have been written as *Known*′. The underscore character is a reserved symbol when it comes to the specification of state machines.

If an operation doesn't change the value of one of the state variables, then either:

- Include the variable *twice* in the operation's head (for example as in `remid` above). In this way, some users may see more clearly that the operation isn't changing that variable—for instance Z users because in Z one must explicitly indicate which variables are unchanged by an operation.
- Include the variable *once* in the operation's head. This may be closer to B specifications.
- Don't include the variable in the operation's head. If the variable isn't needed for that operation then just don't write it as an argument for that operation.

In any of these cases {*log*} will interpret that the operation doesn't change the value of that variable. Making explicit that an operation doesn't change a state variable allows {*log*} to infer some facts about the specification, as we will see in Section 13.3.

On the other hand, when an operation does change the value of a state variable we have to include the corresponding next-state variable in the head (and use it in the body). For example, the following operation adds a new birthday to the book:

```
operation(addBirthday).
addBirthday(Known,Birthday,Name,Date,Known_,Birthday_) :- ...
```

Then Known_ and Birthday_ are arguments of the operation's head.

The body of addBirthday is given as the disjunction between two predicates: the first, considers the case when the name whose birthday is going to be added is not in the book, and the second considers the opposite situation. The first case is covered by addBirthdayOk:

```
dec_p_type(addBirthdayOk(kn,bb,name,date,kn,bb)).
addBirthdayOk(Known,Birthday,Name,Date,Known_,Birthday_) :-
  Name nin Known &
  un(Known,{Name},Known_) &
  un(Birthday,{[Name,Date]},Birthday_).
```

The second case is covered by nameAlreadyExists:

```
dec_p_type(nameAlreadyExists(kn,name)).
nameAlreadyExists(Known,Name) :- Name in Known.
```

Note that we don't declare these predicates as operations. This is because we declare the predicate making the disjunction of them as the operation:

```
operation(addBirthday).
dec_p_type(addBirthday(kn,bb,name,date,kn,bb)).
addBirthday(Known,Birthday,Name,Date,Known_,Birthday_) :-
  addBirthdayOk(Known,Birthday,Name,Date,Known_,Birthday_)
  or
  nameAlreadyExists(Known,Name) & Known_ = Known & Birthday_ = Birthday.
```

It would be wrong to declare all three (and even two) of them as operations. Think of addBirthdayOk and nameAlreadyExists as auxiliary predicates used to define the operation. Auxiliary predicates shouldn't be declared as operations. Nonetheless, {*log*} won't be able to warn the user if this is happening. The problem with declaring auxiliary predicates as operations is that {*log*} will generate more verification conditions than necessary thus enlarging the verification process—see Section 13.3.

It is also not possible to first define addBirthday and then addBirthdayOk and nameAlreadyExists because when the typechecker finds addBirthday it cannot resolve the type of addBirthdayOk and nameAlreadyExists as it hasn't typechecked them already.

The birthday book offers an operation to find the birth date of a given person:

```
dec_p_type(findBirthdayOk(kn,bb,name,date)).
findBirthdayOk(Known,Birthday,Name,Date) :-
  Name in Known &
  applyTo(Birthday,Name,Date).
```

```
dec_p_type(notAFriend(kn,name)).
notAFriend(Known,Name) :-
  Name nin Known.

operation(findBirthday).
dec_p_type(findBirthday(kn,bb,name,date)).
findBirthday(Known,Birthday,Name,Date) :-
  findBirthdayOk(Known,Birthday,Name,Date)
  or
  notAFriend(Known,Name).
```

As can be seen, this operation doesn't change the state of the system.

If V is a state variable and V_ is an argument of an operation then V must be another argument. Including V_ as an argument when V isn't a state variable is an error. Including V_ more than once is an error. If V is a state variable it can be included at most twice in the head of the operation.

13.1.1 Parameters, axioms and theorems

In this section we will explain how parameters, axioms and theorems can be used to specify state machines. If parameters are used they must be declared at the beginning of the specification, axioms must be included right after the declaration of state variables. Theorems can be declared anywhere after the first axiom.

Parameters play the role of machine parameters or constants in B specifications, and the role of variables declared in axiomatic definitions in Z. Axioms play the role of properties in B, and the role of predicates appearing in axiomatic definitions in Z. That is, parameters serve to declare the existence of some (global) values accessible to invariants and operations, but they cannot be changed by operations. Axioms, in turn, can be used to state properties of parameters. In fact, an axiom can only depend on parameters—i.e., axioms *can't* depend on state variables or any other kind of variables.

As an example, consider a specification dealing with users and their passwords. We are interested in stating that passwords are stored in an encrypted form but we don't want to specify a particular encryption algorithm or cryptographic hash function. In this case we can work as follows:

```
parameters([Password,CryptoHash,Hash]).

axiom(axm1).
axm1(CryptoHash) :- pfun(CryptoHash).

axiom(axm2).
axm2(Password,CryptoHash) :- dom(CryptoHash,Password).

axiom(axm3).
axm3(CryptoHash,Hash) :- foreach([X,Y] in CryptoHash, Y in Hash).
```

From now on, we can use `Password`, `CryptoHash` and `Hash` as arguments in invariants, operations and theorems. However, including `CryptoHash_` (i.e., an after state variable) as an argument in an operation will make {*log*} to issue an error.

Theorems are used to state properties that can be deduced from axioms, invariants, operations or theorems that have already been declared. For this reason, theorems can only be declared after the first axiom.

For example, if we have:

```
variables([X,Y,Z]).

invariant(inv1).
inv1(X) :- 0 =< X.

invariant(inv2).
inv2(Y) :- 0 =< Y.
```

then we can declare the following theorem statement:

```
theorem(thrm1).
thrm1(X,Y) :- 0 =< X + Y.
```

for which we provide a proof:

```
proof_thrm1(X,Y) :- neg(inv1(X) & inv2(Y) implies thrm1(X,Y)).
```

The proof of a theorem statement named $th(x_1,\dots,x_m)$ must be a clause named `proof_`*th* located right after *th* and whose body must be of the form:

`neg(`h_1 `&` h_2 `& ... &` h_n `implies` $th(x_1,\dots,x_m)$`)`

where each h_i must be an axiom, invariant, operation or theorem already declared. Note that the proof of a theorem is stated as an unsatisfiability condition.

In this way, {*log*} will include verification conditions ensuring that the proofs of theorems are correct so users can use theorem statements as hypotheses for other verification conditions. As with invariants, these verification conditions will eventually ask {*log*} to compute the negation of the statements of theorems. As with invariants, users must check whether or not theorem statements satisfy the conditions given in Sect. 3.5.

Negation of theorems

Check predicates declared as `theorem` to see if their negations can be computed by `neg`.

13.2 Execution of state machines

Any predicate defined in the {*log*} specification of a state machine can be called as any {*log*} predicate. Therefore, the specification can be seen as an executable functional prototype where operations implement the prototype's functionality. Executions of the specification's operations can be performed to spot errors early on, analyze complex scenarios, analyze interactions among the operations, etc. Later, properties such as invariant preservation can be proved (Sect. 13.3).

Executions can be performed by considering operations as regular predicates (Sect. 13.2.1) but {*log*} also offers the Next environment which considerably simplifies the definition and execution of functional scenarios that are deterministic and fully instantiated (Sect. 13.2.2).

13.2.1 Operations as predicates

We start by considering operations as regular predicates. This is the most general way to execute a prototype and can serve many purposes, but, at the same time, it can be cumbersome in many situations. Let's see an example of an execution assuming the specification of the birthday book has been saved in a file called `bb.slog`.

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

{log}=> birthdayBookInit(K,B) & addBirthday(K,B,messi,2406,K_,B_).     (19)
K = {},
B = {},
K_ = {messi},
B_ = {messi,2406}
```

In this execution, the predicate setting the initial state (`birthdayBookInit`) is called followed by a call to one of the operations (`addBirthday`). Note how state variables are chained between predicates: `K` and `B` are passed in to `birthdayBookInit` and then passed in as the initial state to `addBirthday`. Actually, we can call more than one operation in the same execution:

```
{log}=> birthdayBookInit(K,B) &
        addBirthday(K,B,messi,2406,K1,B1) &
        addBirthday(K1,B1,'Pele',2310,K_,B_).

K = {},
B = {},
K1 = {messi},
B1 = {[messi,2406]},
K_ = {messi,Pele},
B_ = {[messi,2406],[Pele,2310]}
```

Note how after-state variables in the first call to `addBirthday` become before-state variables in the second one. By using state variables in this way it's possible to chain many state transitions in the same execution.

If we don't need the complete execution trace of a simulation but only its final state and outputs, then we can:

1. Use named singletons for intermediate states and any other variable we aren't interested in (Sect. 3.2).

   ```
   {log}=> birthdayBookInit(K,B) &
           addBirthday(K,B,messi,2406,_K,_B) &   % _K,_B named singletons
           addBirthday(_K,_B,'Pele',2310,K_,B_).

   K = {},                                       % _K,_B not printed
   ```

```
B = {},                                % only before and after state
K_ = {messi,Pele},
B_ = {[messi,2406],[Pele,2310]}
```

2. Define a clause for the simulation whose arguments are the variables we are interested in:

```
sim(K,B,K_,B_) :-
  birthdayBookInit(K,B) &
  addBirthday(K,B,messi,2406,K1,B1) &
  addBirthday(K1,B1,'Pele',2310,K_,B_).

{log}=> sim(K,B,K_,B_).

K = {},
B = {},
K_ = {messi,Pele},
B_ = {[messi,2406],[Pele,2310]}
```

We can use variables instead of constants for input parameters but in that case we might get a list of constraints (not only equalities) and many variables, but at the same time we'll be able to conclude more general properties about the specification.

```
{log}=> birthdayBookInit(K,B) &
        addBirthday(K,B,N1,D1,K1,B1) &
        addBirthday(K1,B1,N2,D2,K_,B_).

K = {},
B = {},
K1 = {N1},
B1 = {[N1,D1]},
K_ = {N1,N2},
B_ = {[N1,D1],[N2,D2]}
Constraint: N1 neq N2
```

Constraints and variables can be avoided in the computed answer if executions are run in `groundsol` mode (Sect. 3.2), but in this case the results are less general.

```
{log}=> groundsol.
{log}=> birthdayBookInit(K,B) &
        addBirthday(K,B,N1,D1,K1,B1) &
        addBirthday(K1,B1,N2,D2,K_,B_).

K = {},
B = {},
N1 = n0,
D1 = n3,
K1 = {n0},
B1 = {[n0,n3]},
N2 = n1,
D2 = n2,
K_ = {n0,n1},
B_ = {[n0,n3],[n1,n2]}
```

Checking invariants. Executions can be used to have a first but incomplete assessment on whether or not operations preserve invariants. All we have to do is to conjoin the invariant we want to check at the end of the scenario taking care of using the final-state variables.

```
{log}=> birthdayBookInit(K,B) &
        birthdayBookInv(K,B) &            % inv on initial state
        addBirthday(K,B,N,D,K_,B_) &
        birthdayBookInv(K_,B_).           % inv on final state
```

If as a result {*log*} returns a solution (as in the above case) then we can say that the operation *may* preserve the invariant but we can't be sure until an invariance lemma is proved—see Sect. 13.3. However, if {*log*} answers `no` then we're sure the operation doesn't preserve the invariant; either the operation or the invariant need to be fixed. For example, if by mistake we set `Known_ = Known` instead of `un(Known,{Name},Known_)` in `addBirthday`, the above scenario will fail because `birthdayBookInv` can't hold in the final state.

13.2.2 Next, a simple environment to assist in scenario execution

This section and its implementation in {*log*} are still experimental.

Next is a simple environment that allows users to execute deterministic, fully instantiated functional scenarios to analyze state machines. It can be used only if:

1. The specification has been typechecked.
2. The VCG has been called on the specification (Sect. 13.3).
3. The VC file has been consulted.

Let's see how scenario (19) is executed on Next.

```
{log}=> consult('bb.slog').
{log}=> vcg('bb.slog').          % call the VCG
{log}=> consult('bb-vc.slog').   % VC file

{log}=> initial >> addBirthday(Name:messi,Date:2406).

Final result is:
Known = {messi},
Birthday = {[messi,2406]}
```

Next works as follows:

- `initial` is a reserved word referring to the predicate declared as `initial` in the specification. In the case of the birthday book specification it refers to the `birthdayBookInit/2` predicate.

- The *then* operator (>>) indicates that addBirthday has to be executed from the state set by birthdayBookInit/2.
- Note that users don't need to (and can't) provide state variables. Next does all that job.
- Arguments like Name:messi are called *assignments*. This assignment indicates that Name is bound to messi when addBirthday is called. The left-hand side of an assignment must be the name of a non-state, non-parameter argument in the operation. For instance Known can't be at the left-hand side of an assignment because is a state variable of the specification. The right-hand side of an assignment can be a ground term, among other possibilities that we will see below. The left-hand sides of assignments are supposed to be the input arguments of the operation. Users decide what non-state, non-parameter arguments are inputs and what are outputs for each operation. However, if a call to an operation is not sufficiently instantiated {*log*} will raise an exception. For example:

```
{log}=> initial >> addBirthday(Name,Date:2406).

***ERROR***: (next) next-state/output not sufficiently instantiated
             after addBirthday(Name,Date:2406)
```

If we want to add two or more persons to the birthday book we can do as follows:

```
{log}=> initial >> addBirthday(Name:[[messi,'Pele']],Date:[[2406,2310]]).

Final result is:
Known = {messi,Pele},
Birthday = {[messi,2406],[Pele,2310]}
```

In assignments of the form ⟨*variable*⟩ : [[⟨*list*⟩]], *list* must be a proper or closed list—i.e. it has to verify SWI-Prolog's is_list/1 predicate—of length at least 2. When this happens we say that [[⟨*list*⟩]] is a *double list*. If in a call to an operation there are two or more double lists, then all of them have to have the same length. The above goal is equivalent to the following one:

```
{log}=> initial >>
        addBirthday(Name:messi,Date:2406) >>
        addBirthday(Name:'Pele',Date:2310).
```

Double lists can be combined with simple assignments:

```
addBirthday(Name:[[messi,riquelme]],Date:2406)
```

which is equivalent to:

```
addBirthday(Name:messi,Date:2406) >> addBirthday(Name:riquelme,Date:2406)
```

The right-hand side of an assignment can be the name of an argument used in some previous step of the execution.

```
{log}=> initial >>
        addBirthday(Name:[[messi,pele,riquelme]],Date:[[2406,2310,2406]]) >>
        findBirthday(Name:messi,Date) >> remind(Today:Date,Cards).
```

```
parameters([V,W]).
variables([X]).

invariant(even).
  even(X) :- 0 is X mod 2.
invariant(above).
  above(V,X) :- X >= V.

initial(init).
  init(V,X) :- X is V + 1.

operation(z).
  z(X,X_) :- 0 =< X & X_ is X + 1.
operation(w).
  w(W,X,Y,X_) :- X_ is X + Y - W.

q(A,B) :- A = 3 & B = 5.
my_init(A) :- A = -10.
other_init(Q) :- Q = 100.
```

Figure 1: A toy specification

```
Final result is:
Known = {messi,pele,riquelme},
Birthday = {[messi,2406],[pele,2310],[riquelme,2406]},
Date = 2406,
Cards = {messi,riquelme}
```

The assignment `Today:Date` in `remind` use `Date`, an (output) argument in `findBirthday`. This makes Next to assign `Today` the value of `Date` generated when `findBirthday` is executed. That's why in the above execution `Cards` equals `{messi,riquelme}`. In this way, the output values produced in one step of the execution can be used as the input values of a subsequent step.

Now consider the toy specification shown in Figure 1 where we have omitted type declarations for brevity—recall that Next requires the specification to be typechecked and analyzed by the VCG. Some predicates in the specification (`above`, `init` and `w`) depend on parameters.

Dealing with specification parameters in executions. As the meaning of parameters is to remain fixed for many runs of the system, the use of Next in these cases requires the presence of a predicate setting values for the parameters. These values will remain the same for all scenarios until the user provides another such predicate. Users can provide such predicate by means of command `setpp(`$\langle p \rangle$`)` where p is a Prolog atom. {*log*} will search for a predicate with signature p/n where n is the number of parameters declared in the specification. For instance, `q/2` is such a predicate in the specification of Figure 1. {*log*} assumes the order of p's arguments coincides with the order of parameter declaration. In the case of `q/2`, 3 will be `V`'s value while 5 will be `W`'s. The user indicates that `q/2` is the predicate setting values for parameters as follows[25]:

```
{log}=> setpp(q).
```

From this moment on, all scenarios will use the values set by `q/2` until the user sets another predicate. So, for example:

```
{log}=> initial >> w(Y:3).
Parameters: V = 3,  W = 5

Final result is:
X = 2

{log}=> initial >> w(Y:3) >> z.
Parameters: V = 3,  W = 5

Final result is:
X = 3
```

If an execution is launched before the user had called `setpp/1`, {*log*} will search for a predicate with signature `these_parameters/`*n*, where *n* is as before. Binding between arguments in `these_parameters` and the parameters is positional as explained above. If {*log*} can't find such a predicate, an exception is raised because the execution won't be sufficiently instantiated.

Indexed calls. Executions can include *indexed calls* to an operation, as shown in the following examples.

```
{log}=> initial >> 5:z.
Parameters: V = 3,  W = 5

Final result is:
X = 9

{log}=> initial >> 2:w(Y:3).
Parameters: V = 3,  W = 5

Final result is:
X = 0
```

For instance, `initial >> 2:w(Y:3)` is equivalent to `initial >> w(Y:3) >> w(Y:3)`. That is, if $n \in \mathbb{N}_2$ and p is an operation, an execution step of the form $n : p$ is defined as follows:

$$n : p \equiv \overbrace{p \gg \cdots \gg p}^{n}$$

In the birthday book adding the same name for a second time produces no state change. The following scenario confirms that.

[25] `setpp` stands for *set predicate* for *parameters*.

```
{log}=> initial >> 2:addBirthday(Name:messi,Date:2406).

Final result is:
Known = {messi},
Birthday = {[messi,2406]}
```

Execution traces. In some situations we may be interested in analyzing the *execution trace* (i.e. the sequence of states generated by the execution) and not just the final result of a scenario. This is easy to do with Next, just enclose `initial` between square brackets.

```
{log}=> [initial] >> 2:addBirthday(Name:messi,Date:2406).

Execution trace is:
Known = {},
Birthday = {}
  ----> addBirthday(Name:messi,Date:2406)
Known = {messi},
Birthday = {[messi,2406]}
  ----> addBirthday(Name:messi,Date:2406)
Known = {messi},
Birthday = {[messi,2406]}
```

In this way we can see that while the first call to `addBirthday` changes the birthday book, the second one has no effect.

Changing the starting state of an execution. The starting state of an execution can be different from the initial state of the specification. Simply define a predicate with arity equal to the number of state variables declared in the specification and begin an execution with its head. Binding between arguments and state variables is positional by following the order of declaration of the latter. For example, in the specification of Figure 1 `my_init` and `other_init` are such predicates, so the following executions have the final results shown below.

```
{log}=> initial >> z.
Parameters: V = 3,  W = 5

Final result is:
X = 5

{log}=> [my_init] >> w(Y:1) >> z >> w(Y:3).
Parameters: V = 3,  W = 5

Execution trace is:
X = -10
  ----> w(Y:1)
X = -14
```

```
  ----> z failed, execution aborted

{log}=> other_init >> z.
Parameters: V = 3,  W = 5

Final result is:
X = 101
```

As can be seen, as soon as an operation can't be executed the whole execution is aborted.

Checking invariants. As with regular predicates, Next can be used to check invariant satisfaction (not to be confused with proving invariance lemmas as in Sect. 13.3). In this regard Next is easier to use but less general than executing predicates. Invariants can be placed as arguments of Next as if they were operations. If Next finds an invariant, evaluates it in the final state yield by the previous operation or on the initial state if the invariant is the second step of the execution. If the invariant is *not* satisfied by the current state, {*log*} informs this situation to the user—if the invariant is satisfied nothing is said. Let's see a couple of examples.

```
{log}=> initial >> addBirthday(Name:messi,Date:2406) >> birthdayBookInv.

Final result is:
Known = {messi},
Birthday = {[messi,2406]}        % invariant checked ok, nothing to say

{log}=> [initial] >> z >> even >> z >> even.
Parameters: V = 3,  W = 5

Execution trace is:
X = 4
  ----> z
X = 5
  even check failed
  ----> z
X = 6                            % invariant checked ok, nothing to say
```

As can be seen in the second execution, after the first z step even (the invariant) isn't satisfied but after the second z step it is satisfied. The execution continued even though an invariant check failed.

Given that invariants can depend only on parameters and state variables, it is not necessary (and can't be done) to write those variables when the invariant is an argument of Next. Have in mind that Next doesn't check if the starting state satisfies an invariant appearing in the execution. We remark this because invariant preservation is supposed to happen when the operation departs from and arrives to a state satisfying the invariant. If you want to be sure that the starting state satisfies an invariant place it as the second step of the execution.

Perhaps we want to check some invariants *after every step* of an execution. Next provides an easy way to state that. See the following example[26].

[26] By the way, check out how negative numbers must be typed when used in assignments.

```
{log}=> [initial]:[even,above] >> 2:z >> w(Y:(-6)).
Parameters: V = 3,  W = 5

Execution trace is:
X = 4
  ----> z
X = 5
  even check failed             % above checked ok, nothing to say
  ----> z
X = 6                           % even and above checked ok
  ----> w(Y: -6)
X = -5
  even check failed
  above check failed
```

That is, if we want to check invariants $I_1, \dots, I_n$ after each step we start the execution with a term of the form $init : [I_1, \dots, I_n]$, where *init* can be either an atom or an atom enclosed between square brackets, with the meaning explained in Execution traces.

Recall that the most valuable information when checking invariants are the **failed** messages because we know that something is really wrong with the specification. Positive answers are less valuable.

13.3 Automatic generation of verification conditions

Once we have defined a state machine as described above, we can use the Verification Condition Generator (VCG) to help in its verification. The VCG generates some verification conditions that, when successfully discharged, will give some confidence on the correctness of the specification. {*log*} itself can be used to automatically prove or disprove these verification conditions.

Assuming the birthday book specification is saved in file `bb.slog`, the user can issue the following commands to generate the verification conditions associated to this specification:

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

In this way, {*log*} generates a file named `bb-vc.slog` containing the verification conditions. The file is consulted as usual:

```
{log}=> consult('bb-vc.slog').
```

> If `vcg` is called more than once on the same file, the corresponding `-vc` file will be overwritten, thus loosing any changes introduced by the user.

The main predicate in that file is named `check_vcs_bb`. When this command is run, {*log*} attempts to discharge all the verification conditions:

```
{log}=> check_vcs_bb.

Checking birthdayBookInit_sat_birthdayBookInv ... OK
Checking birthdayBookInit_sat_pfunInv ... OK
Checking addBirthday_is_sat ... OK
Checking findBirthday_is_sat ... OK
Checking remind_is_sat ... OK
Checking addBirthday_pi_birthdayBookInv ... OK
Checking addBirthday_pi_pfunInv ... ERROR
Checking findBirthday_pi_birthdayBookInv ... OK
Checking findBirthday_pi_pfunInv ... OK
Checking remind_pi_birthdayBookInv ... OK
Checking remind_pi_pfunInv ... OK
```

Note that {*log*} is able to discharge all but one of the verification conditions (addBirthday_pi_pfunInv).

In general, the command used to discharge proof obligations is named check_vcs_⟨*fileName*⟩, where *fileName* is the name of the file containing the specification of the state machine. This command expects no arguments. There are two more commands with the same name but different arities:

- check_vcs_⟨*fileName*⟩(+Timeout,+OptList) where Timeout must be a positive number indicating a timeout measured in milliseconds and OptList is any term accepted by setlog/5 in its fifth argument as documented in Sect. 15.1—see also Sect. 11. The timeout is used for *each* proof obligation—for example, in the birthday book the maximum time this command can take is 11*Timeout milliseconds because there are 11 verification conditions. Some examples of this command are the following:

  ```
  {log}=> check_vcs_bb(1000,[subset_unify,comp_fe]).

  {log}=> check_vcs_bb(2000,try(prover_all)).

  {log}=> check_vcs_bb(2000,tryp(prover_all)).
  ```

  Use different timeouts and execution options if one or more of your VC times out. try[p](prover_all) can solve some hard goals without much manual work. If you still get timeouts, try with try[p](prover_all_single) or try passing a list of options (as in the first example) or a list of lists of options to try[p].

- check_vcs_⟨*fileName*⟩(+OptList) is implemented as:

  check_vcs_⟨*fileName*⟩(60000,OptList).

- check_vcs_⟨*fileName*⟩ is implemented as:

  check_vcs_⟨*fileName*⟩(60000,[]).

13.4 Analyzing undischarged verification conditions

{*log*} may not be able to discharge a verification condition for many reasons. If a verification condition falls outside the decision procedures implemented in {*log*}, then the tool will be unable to discharge it. There's nothing to do in these cases. If the specification is wrong (e.g., a precondition is missing, an invariant is too strong, etc.) then it may be impossible to prove some verification condition. In these cases the counterexamples returned by {*log*} can be of much help (Section 13.4.1). Finally, a verification condition may require more hypotheses, as explained in Section 13.4.2.

13.4.1 Counterexamples of undischarged verification conditions

When a verification condition such as `addBirthday_pi_pfunInv` remains undischarged, {*log*} saves a counterexample. These counterexamples can be very helpful in finding out why the proof failed. There are two kinds of counterexamples: abstract and ground. Abstract counterexamples may include many variables; ground counterexamples bind a ground term to every variable (recall Section 3.2).

Let's see the abstract counterexample of `addBirthday_pi_pfunInv`:

```
{log}=> vcace(addBirthday_pi_pfunInv).

Birthday = {[Name,_N2]/_N1},
Known_ = {Name/Known},
Birthday_ = {[Name,Date],[Name,_N2]/_N1}
Constraint: pfun(_N1), comppf({[Name,Name]},_N1,{}),
            Name nin Known,  Date neq _N2
```

As can be seen, `vcace`[27] receives the name of a verification condition as its sole argument and prints a counterexample containing many variables (e.g. `_25302`) besides those used in the specification.

This counterexample helps to understand why `addBirthday_pi_pfunInv` failed. In effect, notice that `Name` is in the domain of `Birthday` but it doesn't belong to `Known` (see the constraint `Name nin Known`). This is in contradiction with invariant `birthdayBookInv` which states that `Known` must be the domain of `Birthday`. Given that we have proved that `birthdayBookInv` is indeed an invariant we can include it as an hypothesis to prove `addBirthday_pi_pfunInv`—see how to do that in Section 13.4.2.

The ground counterexample can be seen with command `vcgce` (g for ground):

```
{log}=> vcgce(addBirthday_pi_pfunInv).

Birthday = {[n2,n1]}
Known = {}
Name = n2
Date = n0
Known_ = {n2}
Birthday_ = {[n2,n0],[n2,n1]}
```

[27] `vcace` stands for *v*erification *c*ondition *a*bstract *c*ounter*e*xample.

We can see again that Known is empty while Birthday is not thus contradicting birthdayBookInv.

13.4.2 The findh command family

{*log*} was unable to discharge addBirthday_pi_pfunInv because of the way it generates verification conditions. This point is treated more deeply in Section 13.5. For the moment, it suffice to say that {*log*} generates each verification condition with the minimum number of hypotheses. In consequence, {*log*} is sometimes unable to discharge a verification condition because there are some missing hypotheses. Missing hypotheses can only be found among the axioms and invariants declared in the specification.

The findh family of commands helps users to find missing hypotheses in verification conditions whose proofs failed (as with addBirthday_pi_pfunInv). However, before calling these commands take a look at the counterexamples returned by {*log*} because in many cases they provide enough information to solve the problem. The simplest way of finding missing hypotheses (although in some cases it will take a long time), is by running the findh command:

```
{log}=> findh.
Missing hypotheses for addBirthday_pi_pfunInv: [[birthdayBookInv]]
```

Now we can add birthdayBookInv as an hypothesis of addBirthday_pi_pfunInv. In order to do that, we edit bb-vc.slog, look up addBirthday_pi_pfunInv and modify it as follows (as the comment suggests):

```
addBirthday_pi_pfunInv :-
  birthdayBookInv(Known,Birthday) &          %%% NEW HYPOTHESIS
  neg(
    pfunInv(Birthday) &
    addBirthday(Known,Birthday,Name,Date,Known_,Birthday_) implies
    pfunInv(Birthday_)
  ).
```

Later, the user should consult bb-vc.slog again and run check_vcs_bb. If the proof fails, findh can be called again in which case it will probably find other hypotheses.

> If vcg is called more than once on the same file, the corresponding -vc file will be overwritten, thus loosing all hypothesis added by the user.

findh goes through all the failed proofs searching for missing hypotheses for each of them. It first tries with each axiom and invariant as a possible hypothesis. In some cases, the conjunction of two or more axioms or invariants are necessary to prove a given verification condition. For this reason, if findh couldn't find a single axiom or invariant, it tries by conjoining two of them; if this isn't enough, findh will try with conjunctions of three axioms and invariants. This process continues until the conjunction of all axioms and invariants is tried out. In specifications with more than 10 to 15 axioms and invariants the number of conjunctions of 2, 3 or more elements will be computationally prohibitively.

As a consequence, {*log*} provides also the following findh and findhn commands based on the same idea:

- `findh(vc)` where `vc` is the identifier of a verification condition (e.g. `addBirthday_pi_pfunInv`). Works as `findh` but only for `vc`.
- `findh(vc,n,o|e,axinvs)` where `vc` is the identifier of a verification condition (e.g. `addBirthday_pi_pfunInv`), `n` is a positive natural number, `o|e` is either the atom `o` or `e`, and `axinvs` is a list of axiom or invariants identifiers. This command works as `findh` but only for `vc` and by applying the following restrictions.
  - It will attempt with conjunctions of exactly `n` axioms or invariants.
  - If the third argument is `o` it will use *only* the axioms and invariants passed in `axinvs`.
  - If the third argument is `e` it will *exclude* the axioms and invariants passed in `axinvs`.
- `findhn(n)` where `n` is a positive natural number. Works as `findh` but only with conjunctions of exactly `n` axioms or invariants.
- `findhn(vc,n)` where `vc` is the identifier of a verification condition (e.g. `add-Birthday_pi_pfunInv`), and `n` is a positive natural number. Works as `findh` but only for `vc` and only with conjunctions of exactly `n` axioms or invariants.

Observe that, in general, more than one conjunction of axioms or invariants can be a missing hypothesis. When this is the case, the `findh` commands will print a list with all of them. So, for example:

```
{log}=> findh.
Missing hypotheses for xxx_pi_yyy: [[ax1,inv3],[inv2,inv4]]
Missing hypotheses for vvv_pi_www: [[ax3],[inv3],[inv4]]
```

means that the conjunction of `ax1` and `inv3` and the conjunction between `inv2` and `inv4` are missing hypothesis for the verification condition named `xxx_pi_yyy`; and that `ax3`, `inv3` and `inv4` are missing hypothesis for `vvv_pi_www`. In these cases, the user should evaluate which of the missing hypothesis is the most promising to conclude the proof of the corresponding verification condition.

13.5 Verification conditions generated by the VCG

The VCG generates the following verification conditions. The examples given below correspond to the birthday book specification (when possible).

1. The conjunction of all axioms is satisfiable (`axioms_sat`).
2. The initial state satisfies each and every invariant (`_sat_`). As an example we have:

```
birthdayBookInit_sat_birthdayBookInv :-
  birthdayBookInit(Known,Birthday) &
  birthdayBookInv(Known,Birthday).
```

3. Each operation is satisfiable and can change the state (`_is_sat`). For example:

```
addBirthday_is_sat :-
  addBirthday(Known,Birthday,Name_i,Date_i,Known_,Birthday_) &
  [Known,Birthday] neq [Known_,Birthday_].
```

If the operation doesn't change state variables, then the proof obligation checks satisfiability of the operation. For example:

```
findBirthday_is_sat :-
  findBirthday(Known,Birthday,Name,Date).
```

4. All `applyTo` predicates appearing as a functional predicate in a `foreach` constraint are well-defined (`_is_wd_`). More precisely, if `applyTo(F,X,Y)` appears in a `foreach` constraint whose quantification variable is `X`, then the verification condition ensures that `X` belongs to the domain of `F`.
5. *Invariance lemmas*: each operation preservers each and every invariant (`_pi_`). For example:

```
addBirthday_pi_birthdayBookInv :-
  % here conjoin other ax/inv as hypothesis if necessary
  neg(
    birthdayBookInv(Known,Birthday) &
    addBirthday(Known,Birthday,Name,Date,Known_,Birthday_) implies
    birthdayBookInv(Known_,Birthday_)
  ).
```

6. *Theorems*: the proofs of user-defined theorems are included as verification conditions. Since {*log*} will try to prove their unsatisfiability, proofs must encode unsatisfiable formulas.

The most important verification conditions are the invariance lemmas. However, if the operation or the invariant are unsatisfiable the invariance lemma will trivially hold. Hence, the VCG also generates the first two satisfiability verification conditions.

13.6 Test case generation

This section and its implementation in {*log*} are still experimental.

As we have said, the state machines specified in {*log*} are considered functional prototypes. These specifications will be implemented in some other programming language. These implementations should be tested. Model-based testing[28] (MBT) indicates that implementations can be tested with test cases generated from their specifications. {*log*} includes an implementation of the Test Template Framework[29] (TTF), a MBT method originally developed for the Z notation. We strongly suggest to read the mentioned Wikipedia page before using this implementation.

Before applying the TTF on a specification, the following conditions must hold:

1. The specification has been typechecked (Sect. 12).
2. The VCG has been called on the specification (Sect. 13.3).
3. The resulting VC file has been consulted.

After consulting the VC file issue the following command in order to initialize the TTF:

```
{log}=> ttf(⟨ atom ⟩).
```

[28] Wikipedia.org: Model-based testing.

[29] Wikipedia.org: Test Template Framework.

where *atom* can be any Prolog atom. In general, this atom should be something related to the specification such as its file name. For instance, for the birthday book specification we can set:

```
{log}=> ttf(bb).
```

bb will be used to name some files after test case generation has finished.

The TTF generates test cases for each operation in a state machine. Test case generation proceeds by applying so-called *testing tactics* to the selected operation. What tactics are applied depends on the logical structure and mathematical elements used in the operation, as well as on the testing goals determined by the testing team. Each testing tactic modifies a so-called *testing tree*. Nodes in the testing tree are called *test specifications*. Each test specification is a conjunction of {*log*} constraints depending on the before-state and input variables of the operation. The root node of a testing tree is called *Valid Input Space* (VIS). Test cases are generated from the leaves of the testing tree. So the first stage of the TTF is to growth the testing tree by applying testing tactics whereas the second is to generate test cases from the leaves of the testing tree.

In {*log*} the first testing tactic to be applied is called *disjunctive normal form* (DNF). We will apply DNF to the addBirthday operation of the birthday book specification. The command is as follows.

```
{log}=> applydnf(addBirthday(Name,Date)).
```

That is, the command waits a term of the form $operation(i_1,\dots,i_n)$ where $i_1,\dots,i_n$ are the inputs of the operation, *not including the before-state variables*. In other words, the term $operation(i_1,\dots,i_n)$ must not include state variables. The TTF uses the structure of the specification to check whether the term can be accepted or not. Users decide what operation's arguments are inputs and what are not. applydnf automatically fetches state variables to be included as part of the operation's VIS.

applydnf writes the precondition of the operation into DNF and generates a test specification for each term of the DNF. Consequently, the testing tree generated as a result of the above applydnf command is the following.

```
{log}=> writett.

addBirthday_vis
    addBirthday_dnf_1
    addBirthday_dnf_2
```

addBirthday_vis is the root of the tree while addBirthday_dnf_1 and addBirthday_dnf_2 are its children—indentation is used to depict tree levels.

Test specifications can be seen by exporting the testing tree to a {*log*} file as follows.

```
{log}=> exporttt.
ttf: testing tree successfully exported to bb_addBirthday-tt.slog
```

The contents of bb_addBirthday-tt.slog is roughly the following.

```
addBirthday_dnf_1(Name,Date,Known,Birthday) :- Name nin Known.
addBirthday_dnf_2(Name,Date,Known,Birthday) :- Name in Known.
```

Once an `applydnf` command is issued for an operation, all the other testing tactics are applied to the same operation until `ttf/1` is issued again, after which `applydnf/1` can be called again for a different (or the same) operation. Every call to `ttf/1` resets the test case generation process.

It would be possible to generate test cases from this testing tree but we will apply more testing tactics in order to show the TTF more thoroughly. After DNF, tactics are applied to one or more test specifications of the testing tree. Then, we apply a second tactic known as *standard partitions* (SP) to the `addBirthday_dnf_1` test specification. The command is as follows.

```
applysp(addBirthday_dnf_1,un(Known,{Name},Known_)).
```

The first argument is the test specification and the second one is a constraint present in the operation—see `addBirthdayOk` in Sec. 13.1. SP defines partitions for key mathematical operators such as union, intersection, overriding, etc. Figure 2 shows the standard partition for union, intersection and set difference. The partition indicates conjunctions of conditions depending on the arguments of the operator. The rationale behind SP is that set and relational operators have non trivial implementations that deserve a thorough testing. The SP tactic generates a test specification for each conjunction defined in the partition.

The operators supported by SP and the partitions defined for them can be explored by looking at the file `ttf_sp.pl` located in the {*log*}'s directory. Users can extend SP by adding partitions for other {*log*} constraints. Follow the pattern of the existing partitions and check the brief documentation in the file. Compare Figure 2 with the clauses of predicate `ttf_sp/4` with first argument `un`, `inters` or `diff`. Every time the file is modified, consult it from {*log*} to ensure minimum consistency. Future versions will provide a safer way of modifying the SP file.

The testing tree generated by the above application of SP is the following.

```
{log}=> writett.

addBirthday_vis
    addBirthday_dnf_1
        addBirthday_sp_11
        addBirthday_sp_12
        addBirthday_sp_13
        addBirthday_sp_14
        addBirthday_sp_15
        addBirthday_sp_16
        addBirthday_sp_17
        addBirthday_sp_18
    addBirthday_dnf_2
```

As can be seen, `addBirthday_dnf_1` is partitioned into eight test specifications due to the eight conjunctions in the standard partition for set union. Two sample test specifications are the following—export the testing tree again with `exporttt`.

```
addBirthday_sp_11(Name,Date,Known,Birthday) :-
  Name nin Known & Known = {} & {Name} = {}.
```

$S = \emptyset, T = \emptyset$

$S = \emptyset, T \neq \emptyset$

$S \neq \emptyset, T = \emptyset$

$S \neq \emptyset, T \neq \emptyset, S \cap T = \emptyset$

$S \neq \emptyset, T \neq \emptyset, S \subset T$

$S \neq \emptyset, T \neq \emptyset, T \subset S$

$S \neq \emptyset, T \neq \emptyset, T = S$

$S \neq \emptyset, T \neq \emptyset, S \cap T \neq \emptyset, S \nsubseteq T, T \nsubseteq S, S \neq T$

Figure 2: Standard partition for $S \cup T$, $S \cap T$ and $S \setminus T$

```
addBirthday_sp_14(Name,Date,Known,Birthday) :-
  Name nin Known & Known neq {} & {Name} neq {} & disj(Known,{Name}).
```

Note that each test specification includes the predicate of `addBirthday_dnf_1`. Besides, note that the arguments of the partition (S and T) are substituted by the arguments of the constraint passed in to the `applysp` command (`{Name}` and `Known`). In other words, the predicates of these test specifications are the result of conjoining the predicate of the parent node (`addBirthday_dnf_1`) with those resulting from the application of SP.

Finally, observe that `addBirthday_sp_11` is unsatisfiable. This is a usual situation in the TTF. Given that it is impossible to generate test cases from unsatisfiable test specifications, they must be *pruned* from the testing tree. The TTF provides a command, namely `prunett`[30], that iterates over all the leaves of a testing tree and asks {*log*} whether or not they are unsatisfiable. If {*log*} finds an unsatisfiable test specification `prunett` eliminates it from the testing tree. The resulting testing tree after calling `prunett` is the following.

```
{log}=> prunett.
{log}=> writett.

addBirthday_vis
    addBirthday_dnf_1
        addBirthday_sp_12
        addBirthday_sp_14
    addBirthday_dnf_2
```

Run `prunett` right after applying the second tactic and any ensuing tactics.

Now, we apply a third tactic known as *set cardinality* (SC). The command is as follows.

```
{log}=> applysc(addBirthday_dnf_1,Birthday).
```

Note that SC is applied to `addBirthday_dnf_1` again. The second argument is an operation's variable of a set type. This means that all its children are partitioned as follows.

```
{log}=> writett.
```

[30] `prunett` stands for *prune testing tree*.

```
addBirthday_vis
  addBirthday_dnf_1
    addBirthday_sp_12
      addBirthday_sc_121
      addBirthday_sc_122
      addBirthday_sc_123
    addBirthday_sp_14
      addBirthday_sc_141
      addBirthday_sc_142
      addBirthday_sc_143
  addBirthday_dnf_2
```

We could have applied SC to, say, `addBirthday_sp_14` in which case only this test specification would have been partitioned. Likewise, we could have applied it to the root of the tree (`addBirthday_vis`) in which case also `addBirthday_dnf_2` would have been partitioned. At what test level of the testing tree or to what test specifications we apply a tactic depends on our testing plan and what we want to test or not.

SC partitions a test specification into three new test specifications each characterized by one of the following predicates: $V = \emptyset$, $V = \{X\}$ and $V = \{X, Y/W\} \land X \neq Y$, where V is the variable passed in as second argument to the `applysc` command and X, Y and W are fresh variables. In other words, SC binds V to the empty set, to a singleton set and to a set with at least two elements. The rationale behind SC is that sets are going to be implemented with some data structures that will be traversed inside of some kind of loop. Then, if we have the empty set it means the loop won't be entered; if we have a singleton set the loop will perform just one iteration; an if we have a set with two or more elements, the loop will perform at least two iterations. In this way the loop is tested with a reasonable coverage.

As indicated above, `prunett` is run again although in this case it prunes no test specification.

```
{log}=> prunett.
{log}=> writett.

addBirthday_vis
  addBirthday_dnf_1
    addBirthday_sp_12
      addBirthday_sc_121
      addBirthday_sc_122
      addBirthday_sc_123
    addBirthday_sp_14
      addBirthday_sc_141
      addBirthday_sc_142
      addBirthday_sc_143
  addBirthday_dnf_2
```

The following is a sample test specification after applying SC.

```
addBirthday_sc_143(Name,Date,Known,Birthday) :-
  Name nin Known & Known neq {} & {Name} neq {} & disj(Known,{Name}) &
```

```
     Birthday={_313,_314/_315} & _313 neq _314 &
     dec([_313,_314],[name,date]) & dec(_315,set([name,date])).
```

Observe that the constraints in the first row are those of addBirthday_sp_14 (which in turn include the constraint of addBirthday_dnf_1); the constraints in the second row are those of SC; and the third row consists of variable declarations for the *fresh variables* (e.g. _313). That is, the predicate of addBirthday_sc_143 is the result of conjoining the predicate of the parent node (addBirthday_sp_14) with the predicate generated by SC.

As can be seen after the application of SP and SC, each test specification in the testing tree conjoins the predicate of its parent node with those of the new testing tactic. Consequently, the deeper the testing tree the longer (more complex and more restrictive) the predicates of the leaves. In turn, the more restrictive the predicate of a test specification, the more specific (or accurate or *surgical*) the test case that will be generated from it. Hence, a deeper testing tree means, in general, a better testing coverage.

The final step of the TTF is to generate a test case for each *leaf* of the testing tree. To this aim, {*log*} provides the gentc command[31].

```
{log}=> gentc.
{log}=> writett.

addBirthday_vis
  addBirthday_dnf_1
    addBirthday_sp_12
      addBirthday_sc_121 -> addBirthday_tc_121
      addBirthday_sc_122 -> addBirthday_tc_122
      addBirthday_sc_123 -> addBirthday_tc_123
    addBirthday_sp_14
      addBirthday_sc_141 -> addBirthday_tc_141
      addBirthday_sc_142 -> addBirthday_tc_142
      addBirthday_sc_143 -> addBirthday_tc_143
  addBirthday_dnf_2 -> addBirthday_tc_2
```

Each node labeled with _tc_ contains a test case. Test cases can be explored with the writetc/1 command.

```
{log}=> writetc(addBirthday_tc_123).

addBirthday_tc_123 is:
Name = name:n2,
Known = {},
Birthday = {[name:n0,date:n0],[name:n1,date:n1]}
```

The command writetc/0 writes all the test cases with the above format, and exporttt exports the test cases as well as the test specifications from which they were generated to a file named *a*_*p*-tt.slog, where *a* is the argument passed in to ttf/1 and *p* is the operation passed in to applydnf/1.

[31] gentc stands for *generate test cases*.

If we want to generate test cases for, say, the `remind` operation we first run `ttf/1` again; then `applydnf(remind(Today))` followed by other testing tactics (interleaved with `prunett`) until we consider that we have a reasonable coverage; and finally `gentc` is called. Note that `Today` is the only input variable as `Cards` is supposed to be the result of the operation.

If `gentc` is run over a tree containing some unsatisfiable leaves (because at some point we forgot to run `prunett`), no test cases will be generated for these leaves. In that case run `prunett` to prune those leaves.

Table 11 summarizes all the testing tactics and the corresponding commands provided by {*log*}. Table 12 summarizes all the user commands provided by the TTF except those that apply testing tactics. All commands raise informative exceptions whenever some condition is not met.

14 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.

14.1 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; moreover, `G` is always executed in solver mode (cf. Section 3.7). For example:

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

  makes {*log*} to answer

  ```
  S = {1,3/_N1} Constraint: subset(_N1,{1,2,3}), set(_N1), 2 nin _N1
  ```

  while by issuing

  ```
  {log}=> solve(diff({1,2,3},{2},S)).
  ```

  you get the answer

  ```
  S = {1,3}
  ```

  As another example (comparing `call` and `solve`):

  ```
  {log}=> mode(solver).
  {log}=> call(inters(X,{a,b},Z)) & write(Z).
  {log}=> solve(inters(X,{a,b},Z)) & write(Z).
  ```

  generate the same answers, but the call to `write(Z)` in the second goal prints the internal representation of an uninitialized variable (e.g., `_45554`) since the constraint `inters` is automatically delayed if the third argument and one of the first two are uninitialized variables, while `write(Z)` in the third goal prints `{a,b}`, i.e., the first solution for `Z`.
- `G!`: to make execution of goal `G` deterministic.

Tactic	Command	Description
DNF[a]	`applydnf(`$p(i_1,\ldots,i_n)$`)`	p is the head of an operation; i_k is a variable name used as argument in p's definition. Each i_k is supposed to be an input to p as determined by the user. DNF writes the precondition of the operation into DNF and generates a test specification for each term of the DNF. DNF must be the *first* tactic applied to generate test cases for any operation. All other tactics can be applied only after DNF has been applied, for a given operation.
SP[b]	`applysp(`l`,`$c(p_1,\ldots,p_n)$`)`	$c(p_1,\ldots,p_n)$ is a constraint appearing in the operation; each p_k must be exactly as written in the operation, even with the same variable names. SP defines partitions for mathematical operators—see Figure 2 and the `ttf_sp.pl` file. SP generates a test specification for each conjunction defined in the partition for c. SP defines partitions for: union, intersection, difference, domain and range (anti-)restriction and overriding.
ST[c]	`applyst(`l`,`v`)`	v is of a sum type (Sect. 12.1.5). ST generates a test specification of the form $v = c$ for each constructor c of v's type. If c is of arity $n > 0$ then a term of the form $c(p_1,\ldots,p_n)$, with p_i fresh variable.
II[d]	`applyii(`l`,`v`,`s`)`	v is an integer variable; s is a non-empty list of integer numbers. II sorts s. If $\langle n_1,\ldots,n_k\rangle$ is the result of sorting s, II generates the following test specifications: $v < n_1$; $v = n_1$; $n_1 < v < n_2$; $\ldots$; $n_{k-1} < v < n_k$; $v = n_k$; $n_k < v$.
SC[e]	`applysc(`l`,`v`)`	v is of a set type. SC generates the following test specifications: $v = \emptyset$; $v = \{x\}$; $v = \{x,y/w\} \land x \neq y$, with x, y and w fresh variables.
EX[f]	`applyex(`l`,`e`)`	$e = \mathit{exists}(x \in D, \phi)$ is a REC appearing in the operation; x, D and ϕ must be exactly as written in the operation, even with the same variable names. EX generates these test specifications: $D = \{x\}$; $D = \{x,y/w\} \land x \neq y$; $D = \{x,y/w\} \land x \neq y \land \neg\,\phi(x)$, with x, y and w fresh variables.

[a] Disjunctive Normal Form
[b] Standard Partition
[c] Sum types
[d] Integer Intervals
[e] Set Cardinality
[f] Restricted Existential Quantifier

Table 11: Testing tactics implemented in $\{log\}$. In commands: l is a label of the testing tree; v is an input or before-state variable appearing in the operation's head. All tactics but DNF, partition all the leaves of the subtree whose root node is l.

Command	Description
`ttf(`*a*`)`	*a* is a Prolog atom. This command initializes the TTF. *a* is used to name the file created by `exporttt`. Once test cases for an operation have been generated, `ttf/1` must be called again to apply the TTF to the next operation.
`writett`	Writes the current testing tree to the standard output.
`prunett(`*t*`,`*opt*`)`	*t* is a timeout in milliseconds; *opt* is any term accepted by `setlog/5` in its fifth argument as documented in Sect. 15.1—see also Sect. 11 and 13.3—, i.e. *opt* represents execution options. This command iterates over all the leaves of a testing tree and asks {*log*} whether or not they are unsatisfiable. If {*log*} finds an unsatisfiable test specification, it is eliminated from the testing tree. *t* and *opt* are passed in to {*log*} as in `setlog/5`. This version of `prunett` should be used if `prunett/0` times out on one ore more leaves. Any form of `prunett` can be run even after any version of `gentc` has been executed.
`prunett(`*opt*`)`	It's implemented as `prunett(60000,`*opt*`)`.
`prunett`	It's implemented as `prunett(60000,[])`.
`gentc(`*t*`,`*opt*`)`	*t* and *opt* are as in `prunett/2`. This command attempts to generate a test case for each leaf in the testing tree by calling {*log*} in `groundsol` mode to solve the leaf's predicate. That is, the solution found by {*log*} is the test case. *t* and *opt* are passed in to {*log*} as in `setlog/5`. This version of `gentc` should be used if `gentc/0` times out on one ore more leaves.
`gentc`	It's implemented as `gentc(60000,[type_check])`.
`writetc(`*l*`)`	*l* is a label of a test case in the testing tree. Writes the corresponding test case to the standard output.
`writetc`	Writes all the test cases in the testing tree as `write/0`.
`exporttt`	Writes all the test cases and the corresponding test specifications to a file named *a*_*p*`-tt.slog`, where *a* is the argument passed in to `ttf/1` and *p* is the operation passed in to `applydnf/1`.

Table 12: User commands to run the TTF (except `apply*`)

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.

14.2 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 predicate selection strategy:
  - *S* = `cfirst`: select constraints first
  - *S* = `ordered`: select all predicates in the order they occur
  - *S* = `cfirst(`*list_of_preds*`)`: select predicates in *list_of_preds* just after constraints.

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

Comments about `strategy(`*S*`)`. Predicates in a goal are executed left-to-right in the order they appear in the goal, except that 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 predicates, `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 predicates are executed in the exact order they occur in the goal[32]. 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 predicates. For example:

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

we get as answer

```

ba
X = a
```

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

Note that `call(G)` and `solve(G)` are dealt with as non-constraint predicates, even when `G` is a constraint. Then `call` and `solve` can be used to influence the execution order of predicates in the formula. For example in:

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

`G1` is definitely executed *before* `G2`.

14.3 Execution monitoring

- `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 {*log*} goal `G` (using .

15 Prolog-{*log*} communication

15.1 From Prolog to {*log*}

Main predicates

- `setlog(+Goal,+Timeout,-OutConstrList,-Res,+OptList)`: to execute a {*log*} goal `Goal` with an output constraint list `OutConstrList` and a timeout `Timeout` (an integer constant specifying an amount of time in milliseconds), with a (possibly empty) list `OptList` of execution options. The `Res` argument is used to specify how execution of `Goal` terminates: `success`, if `Goal` terminates successfully within the time `Timeout`; `time_out`, if `Goal` does

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

not terminate within the time `Timeout`; `maybe`, if `Goal` terminates successfully within the time `Timeout` but the computed result is not guaranteed to be reliable.

The possible execution options are those specified by the built-in control predicates mentioned in the previous sections (in particular in sections 11 and 14). A complete list of the available execution options can be found in Appendix A. 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 timeout, using CLP(FD) as the integer solver but disabling the final automatic labeling step (see Sect. 7.1.1).

Options `type_check` and `groundsol` have no effect with `setlog/5`. If they are needed, then use predicates `setlog_str` or `setlog_tc`, as described below.

- `setlog(+Goal,+Timeout,-OutConstrList,-Res,+try([OptList`$_1$`,...,OptList`$_n$`])`: as `setlog/5` in the previous item, but possibly attempting goal `Goal` as many times as the number of `OptList`$_i$ occurring in the argument of term `try`. Precisely, for each $i \in 1..n-1$, if the call:

  `setlog(Goal,Timeout,OutConstrList,OptsLst`$_i$`)`

  terminates with a timeout, then the call:

  `setlog(Goal,Timeout,OutConstrList,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 timeout and no execution options enforced; since this call terminates with a timeout, then the goal `dom(R,S) & ran(R,{1/S})` is executed again, with a `2000` millisecond timeout 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 `OptsList` occurring in the `try` list.

  Also, note that when using any of the execution options `noran_elim`, `nocomp_elim`, `noneq_elim`, the solver is no longer guaranteed to be complete; 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 and the answer we get may be unreliable. In the latter case, the `Res` argument of the `setlog/5` predicate will be bound to the constant `maybe`.

- `setlog(+Goal,+Timeout,-OutConstrList,-Res,try(prover_all))`: is implemented as the following call to the `setlog/5` predicate.

  `setlog(+Goal,+Timeout,-OutConstrList,-Res,try([`$s_1,\ldots,s_n$`]))`

where $\{s_1,\ldots,s_n\} = \mathbb{P}($`prover_all_strategies`$)$. That is, all the possible combinations of the current value of `prover_all_strategies` are attempted, including passing in no options (see also Sect. 11.3).

- `setlog(+Goal,+Timeout,-OutConstrList,-Res,try(prover_all_single))`: is implemented as the following call to the `setlog/5` predicate.

  setlog(+Goal,+Timeout,-OutConstrList,-Res,try([[s_1],...,[s_n]]))

  where [$s_1,\ldots,s_n$] is the current list of options returned by `prover_all_strategies`.

- `setlog(+Goal,+Timeout,-OutConstrList,-Res,+tryp([OptLst`$_1$`,...,OptLst`$_n$`])`: the first four arguments work as in `setlog/5`; each `OptLst`$_i$ is as above. In this case {*log*} creates n (operating system) threads each of which executes `Goal` with one of `OptLst`$_i$ as the active execution options. The execution of the command terminates as soon as one thread terminates. This command is incompatible with non-determinism. Then, if the goal is satisfiable only the first solution is returned.

- `setlog(+Goal,+Timeout,-OutConstrList,-Res,tryp(prover_all))`: is implemented as `setlog/5` with option `try(prover_all)` (see above) but execution is parallelized as in the previous case.

- `setlog(+Goal,+Timeout,-OutConstrList,-Res,tryp(prover_all_single))`: is implemented as `setlog/5` with option `try(prover_all_single)` (see above) but execution is parallelized as in the previous case.

- `setlog(+Goal,+Timeout,-OutConstrList,-Res)`: 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]]).
```

- `setlog(+Goal,-OutConstrList,-Res)`: is implemented as the following call to the `setlog/5` predicate.

  setlog(Goal,∞,OutConstrList,Res,[])

  That is, `setlog/3` is equivalent to a call to `setlog/5` with an infinite timeout and an empty list of execution options.

- `setlog(G,OutCLst)`: is implemented as the following call to the `setlog/3` predicate.

```
  setlog(Goal,OutConstrList,_)
```

- `setlog(G)`: is implemented as the following call to the `setlog/2` predicate.

```
  setlog(Goal,_)
```

- `setlog`: to enter/re-enter the {*log*} interactive environment.

- `rsetlog(+Goal,+Timeout,-OutConstrList,-Res,+OptList)`: 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`, `time_out` or `maybe` as with `setlog/5`.

- `setlog_str(+GoalString,+PrologVars,+TimeOut,-OutConstrList,-Res,+OptList)`: `GoalString` is a {*log*} goal encoded as a string; `PrologVars` is a list of terms of the form `Name = Var`, where `Name` is an atom describing a variable name in `GoalString` and `Var` is a variable; and `OutConstraintList`, `Res` and `OptList` are as in `setlog/5`. `GoalString` is turned into a term but, in some way, if `X` is a variable in `GoalString` and there's an equality `'X' = A` in `PrologVars` then `X` is substituted by `A` in `GoalString` before the execution of the goal is started. After this, the goal is executed as in `setlog/5`.

  For example:

  ```
  ?- setlog_str(
      "X = 1 & un({1},{2},V) & un(V,Q,E)",
      ['X' = A,'V' = V,'Q' = X,'E' = E],
      10000,C,R,[groundsol]).
  ```

  produces as a result:

  ```
  A = 1, V = E, E = {1, 2}, X = {},
  C = [], R = success.
  ```

  The example is clear on the effect of the `PrologVars` parameter. Besides, as can be seen, `setlog_str/6` is able to correctly execute when `groundsol` (and `type_check`, although not shown) is passed as an option.

  If the user doesn't want to write down the list for `PrologVars`, (s)he can use Prolog's `term_string/3`:

  ```
  ?- G = "X = 1 & un({1},{2},V) & un(V,Q,E)",
    term_string(_,G,[variable_names(VN)]),
    setlog_str(G,VN,10000,C,R,[groundsol]).
  ```

  In this case, if the user needs to reuse some variable in `GoalString`, then (s)he can use some member predicate as follows:

  ```
  ?- G = "X = 1 & un({1},{2},V) & un(V,Q,E)",
    term_string(_,G,[variable_names(VN)]),
    setlog_str(G,VN,10000,C,R,[groundsol]),
    member('V' = V,VN),
    write('\nThe result of the first union is: '), write(V), nl.
  ```

- `setlog_str(+GoalString,-EqsString,+TimeOut,-ConstrString,-Res,+OptList)`: same as `setlog_str/6` above but in this case the predicate doesn't receive `PrologVars` but instead it will return in `EqsString` a (possibly empty) list of strings each of which is of the form `Var=term` where `Var` is one of the free variables of `GoalString`. These equalities are the equalities that Prolog would normally output as part of any predicate answer. This means that no binding of terms to variables will be returned except for those in `EqsString`. In the same sense, `ConstrString` is a list of strings each of which is one of the constraints that {*log*} would normally return as part of its answer.

  For example:

```
?- setlog_str(
    "X = 1 & un({1},{2},V) & un(V,Q,E)",
    Eqs,10000,C,R,[groundsol]).
```

produces as a result:

```
Eqs = ["'X'=1", "'V'={1,2}", "'Q'={}", "'E'={1,2}"],
C = [], R = success.
```

- `setlog_str(+GoalString,+PrologVars,-OutConstrList,-Res)`
  `setlog_str(+GoalString,-EqsString,-ConstrString,-Res)`: are implemented as the following call to the `setlog_str/6` predicate.

  ```
  setlog_str(GoalString,Vars_or_Eqs,∞,Constr,Res,[])
  ```

- `setlog_str(+GoalString,+PrologVars,-OutConstrList)`
  `setlog_str(+GoalString,-EqsString,-ConstrString)`: are implemented as the following call to the `setlog_str/4` predicate.

  ```
  setlog_str(GoalString,Vars_or_Eqs,Constr,_)
  ```

- `setlog_str(+GoalString,+PrologVars)`
  `setlog_str(+GoalString,-EqsString)`: are implemented as the following call to the `setlog_str/4` predicate.

  ```
  setlog_str(GoalString,Vars_or_Eqs,_,_)
  ```

- `rsetlog_str(+GoalString,+PrologVars,+TimeOut,-OutConstrList,-Res,+OptList)`
  `rsetlog_str(+GoalString,-EqsString,+TimeOut,-ConstrString,-Res,+OptList)`: same as `setlog_str/6` but with "reification" on `Res`; see `rsetlog/5` for more details.
- `setlog_tc(+GoalString,+PrologVars,-OutConstrList)`
  `setlog_tc(+GoalString,-EqsString,-ConstrString)`: are implemented as the following call to the `setlog_str/6` predicate.

  ```
  setlog_str(GoalString,Vars_or_Eqs,∞,Constr,_,[type_check])
  ```

Other predicates. Besides the already mentioned predicates `setlog_consult/1` and `consult_lib/0` (see Sect. 2), {*log*} provides a few other built-in predicates that can be called directly from the Prolog environment:

- `setlog_clause(Cl)`: to dynamically 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. 14.

15.2 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`

16 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

- `binters(S,R)`: generalized intersection: `R` is the intersection of all elements of the set of sets `S`
- `bun(S,R)`: generalized union: `R` is the union of all elements of the set of sets `S`
- `dint_to_set(A,B,S)`: same as `int_to_set/2` but delayed if interval limits are unknown
- `eq(T1,T2)`: syntactic unification between terms `T1` and `T2`
- `int_to_set(I,S)`: `S` is the set of all elements of the interval `I`
- `list_to_set(L,S)`: `S` is the set of all elements of the list `L`
- `powerset(S,PS)`: powerset ($\mathtt{PS} = 2^{\mathtt{S}}$)
- `set_to_list(S,L)`: `S` is a set and `L` is a list containing all and only the elements of `S`, without repetitions (all possible permutations of `L`).

Dealing with lists

- `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]`)
- `drop(N,L,NewL)`: `NewL` is `L` with its first `N` elements removed
- `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]`)
- `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`.

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] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[3] 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.

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

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

[6] S. Schneider. *The B-method: An Introduction*. Cornerstones of computing. Palgrave, 2001.

[7] 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.

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

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

[10] 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

[11] 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.

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

[13] Cristiá, M., Rossi, G.: Automated proof of Bell-LaPadula security properties. J. Autom. Reason. **65**(4) (2021) 463–478

[14] Maximiliano Cristiá and Gianfranco Rossi. A practical decision procedure for quantifier-free, decidable languages extended with restricted quantifiers. J. Autom. Reason., **68**(4): 23, 2024. DOI 10.1007/S10817-024-09713-6. URL https://doi.org/10.1007/s10817-024-09713-6.

A {*log*} commands and execution options

The following commands can also be used as execution options.

- `comp_elim/nocomp_elim` (default: `comp_elim`): to activate/deactivate complete rewriting of `comp` constraints; using `nocomp_elim` may make the solver incomplete.
- `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` | `prover([opt_1,...,opt_n])`: to change the solver operation mode. In turn, `opt_i` is a prover option (see below).
- `neq_elim/noneq_elim` (default: `neq_elim`): to activate/deactivate `neq` elimination; using `noneq_elim` may make the solver incomplete.
- `ran_elim/noran_elim` (default: `ran_elim`)): to activate/deactivate complete rewriting of `ran` constraints; using `noran_elim` may make the solver incomplete.
- `strategy(Str), Str = cfirst | ordered | cfirst(list_of_preds)`: to change goal predicate selection strategy to `Str`
- `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 (see Sect. 12).
- `groundsol/nogroundsol` (default: `nogroundsol`): to activate/deactivate the computation of ground solutions (see Sect. 3.2).

The prover options are those listed in Table 10. Any prover option is also an execution option.