# $\{\log \}$ User's Manual 

Version 4.9.8 Release 14i

Gianfranco Rossi<br>E-mail: gianfranco.rossi@unipr.it<br>Università di Parma<br>Italy<br>Maximiliano Cristiá<br>E-mail: cristia@cifasis-conicet.gov.ar Universidad Nacional de Rosario and CIFASIS<br>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

1 Introduction ..... 1
2 Using $\{\log \}$ ..... 2
2.1 Loading $\{\log \}$ libraries ..... 3
2.2 Dealing with $\{\log \}$ programs ..... 3
2.3 Asking for help ..... 4
3 Solving formulas with extensional sets ..... 5
3.1 Set operators ..... 6
3.2 Answers to queries and the groundsol predicate ..... 7
3.3 Considerations on set membership and not membership ..... 9
3.4 Introducing formulas ..... 10
3.5 Negation and the let construct ..... 11
3.6 Using $\{\log \}$ as an automated theorem prover ..... 15
3.7 Modes of operation ..... 15
4 Solving formulas with binary relations ..... 16
4.1 Relational operators ..... 18
4.2 Partial functions ..... 20
4.3 Decidable formulas involving binary relations ..... 21
5 Intensional sets ..... 22
5.1 Solving formulas with Restricted Intensional Sets ..... 22
5.1.1 Parameters and the functional section ..... 23
5.1.2 Parameters and control expressions ..... 24
5.1.3 Encoding sets of structured elements ..... 25
5.1.4 Safe patterns ..... 25
5.1.5 Enumerating the elements of a RIS ..... 27
5.1.6 Automated proofs ..... 27
5.2 General intensional set terms ..... 27
6 Quantifiers ..... 29
6.1 foreach and exists ..... 29
6.2 forall ..... 31
6.3 General existential quantifiers ..... 32
7 Solving formulas including integer numbers ..... 33
7.1 CLP(FD) ..... 35
7.1.1 Finite domains ..... 36
7.1.2 Labeling ..... 36
7.2 CLP(Q) ..... 37
7.3 Which integer solver should be used? ..... 39
8 Cardinality constraints ..... 39
8.1 Decidable formulas involving cardinality constraints ..... 40
8.2 The solved form of formulas involving size constraints ..... 41
9 Finite integer intervals ..... 42
9.1 Decidable formulas involving integer intervals ..... 42
9.2 Defining set operators using intervals ..... 43
10 Aggregation functions ..... 43
10.1 Minimum and maximum of a set ..... 43
10.2 Sum of a set ..... 44
10.3 Sum of an array ..... 45
11 Types in $\{\log \}$ ..... 46
11.1 The type system ..... 48
11.1.1 Integers ..... 49
11.1.2 Character strings ..... 49
11.1.3 Basic types ..... 49
11.1.4 Enumerated types ..... 51
11.1.5 Sum types ..... 52
11.1.6 Product types ..... 53
11.1.7 Set types ..... 53
11.2 Types for binary relations and partial functions ..... 54
11.3 Type declarations ..... 55
11.4 Typing user-defined predicates ..... 56
11.5 Typing polymorphic operators ..... 56
11.6 Running formulas in type-checking mode ..... 57
11.7 groundsol when typechecking is active ..... 57
11.8 Proving goals involving finite types ..... 58
11.9 Admissible terms in type-checking mode ..... 59
11.10User commands to work with types ..... 60
12 Specifying and verifying state machines ..... 61
12.1 The specification of the birthday book problem ..... 61
12.2 Parameters, axioms and theorems ..... 65
12.3 Automatic generation of verification conditions ..... 66
12.4 The findh command family ..... 67
12.5 The verification conditions generated by the VCG ..... 69
13 Control predicates ..... 70
13.1 General ..... 70
13.2 Constraint solving ..... 71
13.3 Execution monitoring ..... 72
14 Prolog- $\{\log \}$ communication ..... 72
14.1 From Prolog to $\{\log \}$ ..... 72
14.2 From $\{\log \}$ to Prolog ..... 76

15 The $\{\log \}$ library
A $\{\log \}$ commands 79

## 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 $\{\log \}$.

[^0]
## 2 Using $\{\log \}$

Assume the $\{\log \}$ interpreter (Prolog source code) has been saved into a file named setlog.pl. To start working with the interpreter, invoke Prolog and then load the $\{\log \}$ interpreter, e.g., by using consult/1:

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

The $\{\log \}$ interpreter is loaded into the Prolog program database. While loading, the interpreter tries to consult three additional files from the working directory (see also the setlog_config/1 built-in predicate in Sect. 14.1):

- 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. 11).
- 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. 12).
Once loaded, there are two ways to use $\{\log \}$ : interactively, much as Prolog itself; and as a Prolog predicate. We will illustrate both of them with a simple example. To access the interactive environment execute the following goal:

```
?- setlog.
```

and $\{\log \}$ will show you its prompt:

$$
\{\log \}=>
$$

Now you can give it goals much as in the Prolog environment. For example, you can ask $\{\log \}$ to solve the following formula (as regards terminology, note that saying "executing the goal $G$ " is the same as saying "asking $\{\log \}$ to solve the formula $G^{\prime \prime}$ ):

$$
\{\log \}=>\operatorname{un}(\{1\},\{2\}, A) .
$$

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. 14.1 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:

$$
\{\log \}=>\text { halt. }
$$

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:

$$
\{l o g\}=>\text { consult_lib. }
$$

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:

$$
\{\log \}=>\text { add_lib('setloglibpf.slog'). }
$$

The same can be obtained from the Prolog environment by issuing

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

See Section 15 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:

$$
\text { un12 (A) :- un }(\{1\},\{2\}, A) .
$$

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

$$
\text { \{log\}=> consult('p.pl'). }
$$

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

$$
\{\log \}=>\text { un12 }(\{1,2,3\}) .
$$

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:

$$
\{l o g\}=>~ l i s t i n g . ~
$$

These clauses can be completely removed by executing:

$$
\text { \{log\}=> abolish. }
$$

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:

$$
\{l o g\}=>~ c o n s u l t(' f i l e . s l o g ', m u t e) .
$$

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

$$
\{l o g\}=>~ c o n s u l t(' f i l e . s l o g ', a p p) .
$$

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

[^1]
## 3 Solving formulas with extensional sets

An extensional set is a set whose elements are enumerated. For example, $\{1, a, h e l l o\}$ 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 $\{\mathrm{e}\}$, 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:

$$
\begin{array}{ll}
X=2, & Y=1 \\
X=1, & Y=2
\end{array}
$$

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 $\{\mathrm{x} / \mathrm{A}\}$ 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 $\{1, \mathrm{a} /\{ \}\}$ instead of $\{1 /\{\mathrm{a} /\{ \}\}\}$; and $\{1, \mathrm{a} /\{ \}\}$ can be further simplified as $\{1, \mathrm{a}\}$. 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:

$$
\begin{aligned}
& { }_{1} X=6, A=\{7,8\} \\
& { }_{2} X=6, A=\{6,7,8\} \\
& { }_{3} X=7, A=\{6,8\} \\
& 4 X=7, A=\{6,7,8\} \\
& { }_{5} X=8, A=\{6,7\} \\
& { }_{6} X=8, A=\{6,7,8\}
\end{aligned}
$$

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:

$$
\begin{aligned}
& X=6, \quad Y=7, \quad A=\{8\} \\
& X=6, Y=7, \quad A=\{7,8\} \\
& X=6, Y=6, \quad A=\{6,7,8\}
\end{aligned}
$$

In $\left\{\mathrm{x}_{1}, \ldots, \mathrm{x}_{n} / \mathrm{A}\right\}, \mathrm{x}_{1}, \ldots, \mathrm{x}_{n}$ 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 $\{\mathrm{x} / \mathrm{A}\}$ 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).

### 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:

$$
\{\log \}=>\operatorname{inters}(\{1\},\{2\}, A) .
$$

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:

| Operator | $\{\log \}$ | Meaning |
| :---: | :---: | :---: |
| set | set (A) | $A$ is a set |
| equality | $\mathrm{A}=\mathrm{B}$ | $A=B$ |
| membership | $x$ in $A$ | $x \in A$ |
| union | un( $\mathrm{A}, \mathrm{B}, \mathrm{C}$ ) | $C=A \cup B$ |
| intersection | inters( $\mathrm{A}, \mathrm{B}, \mathrm{C}$ ) | $C=A \cap B$ |
| difference | $\operatorname{diff}(\mathrm{A}, \mathrm{B}, \mathrm{C})$ | $C=A \backslash B$ |
| subset | subset (A, B) | $A \subseteq B$ |
| disjointness | disj(A,B) | $A \\| B$ |
| strict subset | ssubset (A, B) | $A \subset B$ |
| symmetric difference | sdiff(A,B,C) | $C=A \Delta B$ |
| Negations |  |  |
| set | nset (A) | $A$ is not a set |
| equality | $A$ neq $B$ | $A \neq B$ |
| membership | $x \operatorname{nin} A$ | $x \notin A$ |
| union | nun ( $\mathrm{A}, \mathrm{B}, \mathrm{C}$ ) | $C \neq A \cup B$ |
| intersection | ninters( $\mathrm{A}, \mathrm{B}, \mathrm{C}$ ) | $C \neq A \cap B$ |
| difference | ndiff (A, B, C) | $C \neq A \backslash B$ |
| subset | nsubset ( $\mathrm{A}, \mathrm{B}$ ) | $A \nsubseteq B$ |
| disjointness | ndisj(A, B) | $A \nmid B$ |

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

$$
\{\log \}=>\{1\} \text { in }\{2, a,\{1\}\} .
$$

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 11.

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 queries and the groundsol predicate

The first answer to the following query:

$$
\{\log \}=>\operatorname{diff}(\{1\}, A, B) .
$$

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.
- 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 $\_N 2$ such that $A=\left\{1 / \_N 1\right\}, B$ is a subset of $\{1\}, 1$ doesn't belong to $\_N 1, \_N 2$ is the union of $B$ and $\_N 1,1$ doesn't belong to $\_N 2$ 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 $\mathrm{A}=\{1\}$ 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 $\mathrm{A}=\{1\}, \mathrm{B}=\_\mathrm{N} 1=\_\mathrm{N} 2=\{ \}$ is a ground solution of the first answer returned by $\{\log \}$ after executing $\operatorname{diff}(\{1\}, \mathrm{A}, \mathrm{B})$. Whereas the bindings $\mathrm{A}=\{1,2\}, \mathrm{B}=\{ \},{ }_{2} \mathrm{~N} 1=\_\mathrm{N} 2=\{2\}$ 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.

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.

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 $\mathrm{n}\langle n u m b e r\rangle$, 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 11).

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

### 3.3 Considerations on set membership and not membership

It is easy to prove the following:

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

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 $A=\left\{X / \_N 1\right\} \& X$ 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 $X \operatorname{nin} B$. 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 $B=\left\{X / \_N 1\right\}$ \& $X$ 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.

| 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 $\{\log \}$
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:
$B=\left\{j / \_N 1\right\}$,
$\mathrm{A}=\left\{1, \mathrm{j} / \_\mathrm{N} 1\right\}$
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)
$B=\left\{j / \_\right.$N 1$\}$
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 construct

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, $\operatorname{neg}(X$ in $A \& Z \operatorname{nin} C)$ 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}.
```

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" variables occurring in intensional sets-Sect. 5-and restricted quantifiers-Sect. 6). Hence, solving the body of this clause amounts to solve the following formula:

$$
\begin{equation*}
\exists X:(\exists Y: X=\{Y\}) \tag{2}
\end{equation*}
$$

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., $\},\{1,2\}, 1, \ldots$ ). Hence, the clause body of notsingleton_set should express the formula:

$$
\begin{equation*}
\exists X: \neg(\exists Y: X=\{Y\}) \equiv \exists X:(\forall Y: X \neq\{Y\}) \tag{3}
\end{equation*}
$$

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:

$$
\begin{equation*}
\exists X:(\exists Y: X \neq\{Y\}) \tag{4}
\end{equation*}
$$

Note the difference between the formula obtained by using neg (i.e. (4)) and the actual negation of (2) (i.e. (3)). Thus, for example,

$$
\{\log \}=>\text { notsingleton_set(\{\}). }
$$

correctly answers yes, but issuing

$$
\{\log \}=>\text { notsingleton_set }(\{1\}) .
$$

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:

$$
\begin{equation*}
\forall X, Y: X \neq Y \tag{5}
\end{equation*}
$$

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 (3), 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).
```

where size computes the cardinality of its first argument (see Sect. 8). Note that the body of this clause does not contain any new existentially quantified 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. 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 construct. Say you want to compute the result of the following set expression:

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

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

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

which is interpreted as:

$$
\begin{equation*}
A \cup B=U \wedge U \cap C=\operatorname{Res} \tag{6}
\end{equation*}
$$

See that we need to introduce two new variables: $U$ to get a partial result; and Res to get the desired result. Note that $U$ is simply a "name" for $A \cup B$. Besides, note that $U$ is a shared variable
in the sense that it is present in two (or more) constraints. Now, say that you want to state that Res is different from $(A \cup B) \cap C$, i.e.:

$$
\begin{equation*}
A \cup B=U \wedge U \cap C \neq \operatorname{Res} \tag{7}
\end{equation*}
$$

In $\{\log \}$ using neg in a direct way, i.e., by issuing neg(un(A,B,U) \& inters(U,C,Res)), we actually get:

$$
\neg(A \cup B=U \wedge U \cap C=\operatorname{Res}) \equiv A \cup B \neq U \vee U \cap C \neq \operatorname{Res}
$$

which does not correspond to the intended formula (7).
Hence, for these cases (which appear frequently in $\{\log \}$ ) the language provides the let/3 construct. We can use it to capture the fact that we are naming a subexpression as follows:

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

Then, $\{\log \}$ rewrites the above formula as:

```
un(A,B,U),inters(U,C,Res)
```

However, its negation:

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

is rewritten as:

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

which is the intended formula (recall that ninters is the negated version of inters, see Table 1).

The general form of let is as follows:

$$
\text { let(list_of_vars, } \psi, \phi)
$$

where:

- list_of_vars is a list of distinct variables not appearing anywhere else in the formula.
- $\psi$ is a conjunction of functional predicates and equalities of the form $X=t$, where $X$ is a variable and $t$ a term. 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 predicate is functional but on its first argument. The variables appearing in list_of_vars must be the results of the functional predicates and the variables at the right-hand side of the equalities in $\psi$. A variable in list_of_vars can appear as the result of only one functional predicate or at the right-hand side of only one equality ${ }^{3}$.
- $\phi$ is a $\{\log \}$ formula.

[^2]$\{\log \}$ implements two simple rewrite rules for let:

```
let(list_of_vars,}\psi,\phi)\longrightarrow\existslist_of_vars( \psi ^ \phi
neg(let(list_of_vars, }\psi,\phi))\longrightarrow\existslist_of_vars(\psi ^ neg(\phi)
```

The let construct is useful in general when we have to compute the negation of a conjunction where there are shared variables among its constraints. Consider the following $\{\log \}$ formula:

$$
S=\emptyset \wedge X \in S
$$

If we apply neg to it we get:

$$
S \neq \emptyset \vee X \notin S
$$

which is not correct because $S=\emptyset \wedge X \in S \equiv X \in \emptyset$ and so the negation should be just $X \notin \emptyset$. The problem arises from the fact that $S$ is shared among $S=\emptyset$ and $X \in S$. We can solve the problem by using let:

```
let([],S = {},X in S)
```

In which case the negation is:

$$
S=\{ \} \& X \operatorname{nin} S
$$

Negation of user-defined predicates. When using neg, if its argument is a user-defined predicate, then the user must provide its negation by means of another user-defined predicate. If the head symbol of a user-defined predicate is $p$, with arity $n \geq 0$, then its negation is a userdefined predicate named $n \_p$, with arity $n$. 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, by using n_singleton_set, instead of notsingleton_set, as the name of the predicate defined above, then one can apply directly neg to the positive predicate, as for instance:

$$
\{\log \}=>\text { neg(singleton_set }(\{1\})) .
$$

If the solver does not find any clause defining $n \_p$ when calling neg ( $p(t 1, \ldots, t n$ ), it will use $\operatorname{naf}(p(t 1, \ldots, t n)$ in its place and, if $p(t 1, \ldots, t n)$ 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).
```

If implies or nimplies are used with user-defined predicates, consider that you'll have to declare the negation of those user-defined predicates because, as explained above, $\{\log \}$ doesn't automatically compute the negation of user-defined predicates. A common pattern when $\{\log \}$ is used to prove the unsatisfiability is:

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

which $\{\log \}$ rewrites as:

$$
p \wedge \neg q
$$

Then, if $q$ is a user-defined predicate, a predicate named $n \_q$ representing the negation of $q$ must be in scope.
neg is also implicitly used when solving some constraints on Restricted Intensional Sets (see Sect. 5.1).

### 3.6 Using $\{\log \}$ as an automated theorem prover

$\{\log \}$ can also be used to automatically prove theorems about set theory. If you want to prove the following is a theorem:

$$
\begin{equation*}
x \in B \cup C \Rightarrow(x \in B \vee x \in C) \tag{8}
\end{equation*}
$$

then you can ask $\{\log \}$ to try to find a solution for:

$$
\{\log \}=>\operatorname{neg}((\operatorname{un}(B, C, A) \& X \text { in } A) \text { implies }(X \text { in } B \text { or } X \text { in } C)) \text {. }
$$

that is, the negation of your theorem. In this case $\{\log \}$ returns no which means that there is no value (for $\mathrm{X}, \mathrm{A}, \mathrm{B}$ and C ) satisfying the formula. In turn, this implies that the negation of the formula given to $\{\log \}$ (i.e., your theorem) is always satisfiable. And if a formula is always satisfiable, it is a theorem. In this sense, $\{\log \}$ behaves as a decision procedure for the theory of finite, unbounded sets [5].

It is important to observe that not all formulas that can be written in $\{\log \}$ are actually decidable in $\{\log \}$. For example, as shown above, if negation is used in combination with existentially quantified 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, will be pointed out in next sections.

### 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:

$$
\text { \{log\}=> mode(solver). }
$$

or

$$
\text { \{log\}=> mode(prover). }
$$

The prover mode is better when the formula passed to $\{\log \}$ is supposed to be unsatisfiable. In general, in these cases $\{\log \}$ will be more efficient in concluding that the formula is indeed unsatisfiable, than in solver mode. This means that if users want to use $\{\log \}$ as an automated theorem prover (cf. Section 3.6), they should use it in prover mode.

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

For example, executing the following goal

$$
\{\log \}=>~ s s u b s e t(X,\{a, b\}) .
$$

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

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

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

$$
\begin{aligned}
& X=\{a, b\} \\
& X=\{a\} \\
& X=\{b\} \\
& X=\{ \}
\end{aligned}
$$

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

## 4 Solving formulas with binary relations

A binary relation is a set of ordered pairs. If $X$ and $Y$ are two sets then any set $R$ such that $R \subseteq X \times Y$ is a binary relation. Given that binary relations are sets (of ordered pairs) then $\{\log \}$ can be used to work with formulas involving binary relations [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, $\mathrm{cp}(\mathrm{A}, \mathrm{B})$, 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:

$$
\{\log \}=>\operatorname{dom}(R,\{a\}) .
$$

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 $[\mathrm{a}, \mathrm{Y}]$ 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, $[\mathrm{a}, \mathrm{b}]=[\mathrm{c}, \mathrm{d}]$, if and only if $\mathrm{a}=\mathrm{c}$ and $\mathrm{b}=\mathrm{d}$. 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 $(\mathrm{t})$ (resp., npair $(\mathrm{t})$ ) 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 $\operatorname{comp}(\{[a, a]\}, S, S)$. In effect, $\operatorname{comp}(Q, T, U)$ means $U=Q \circ T$, that is $U$ is the result of the relational composition between $Q$ and $T$. Formally:

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

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

Finally, note that the constraint $[\mathrm{a}, \mathrm{Y}]$ 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 $\mathrm{cp}(\mathrm{A}, \mathrm{B})$ 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:

$$
\{\log \}=>\operatorname{dom}(c p(\{a / A\}, B),\{a\}) .
$$

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

$$
\begin{aligned}
& \{\log \}=>\operatorname{dom}(R,\{a / B\}) \&[b, X] \text { in } R . \\
& \{\log \}=>\operatorname{cp}(A, B)=\{ \} \& \operatorname{ran}(R, B) .
\end{aligned}
$$

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:

$$
\{\log \}=>\operatorname{subset}(R, c p(A, B)) \text {. }
$$

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

$$
\{\log \}=>\operatorname{subset}(R, c p(A,\{1,2\})) \& R=\{[X, 1],[Y, 9]\} .
$$

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

$$
\{\log \}=>\operatorname{subset}(R, c p(A,\{1,2\})) \& R=\{[X, 1],[Y, Z]\} .
$$

See more about types in $\{\log \}$ in Section 11.

### 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 nor integer intervals.

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 $\operatorname{dom}(\mathrm{R}, \mathrm{A})$ is $\operatorname{rel}(R) \wedge \operatorname{dom} R=A$. Of course, similar observations apply to the negative versions of these predicates. For example, the precise meaning of $\operatorname{ndom}(\mathrm{R}, \mathrm{A})$ is $\operatorname{nrel}(R) \vee \operatorname{dom} R \neq A$. For example:

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

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.

| Operator | $\{\log \}$ | Meaning |
| :---: | :---: | :---: |
| binary relation | rel(R) | $R$ is a binary relation |
| domain | $\operatorname{dom}(R, A)$ | $\operatorname{dom} R=A$ |
| range | $\operatorname{ran}(\mathrm{R}, \mathrm{A})$ | $\operatorname{ran} R=A$ |
| composition | $\operatorname{comp}(\mathrm{R}, \mathrm{S}, \mathrm{T})$ | $T=R \circ S$ |
| inverse | $\operatorname{inv}(\mathrm{R}, \mathrm{S})$ | $S=R^{-1}$ |
| identity relation | id(A,F) | id $A=F$ |
| domain restriction | $\operatorname{dres}(A, R, S)$ | $S=A \triangleleft R$ |
| domain anti-restriction | $\operatorname{dares}(A, R, S)$ | $S=A \triangleleft R$ |
| range restriction | $\operatorname{rres}(\mathrm{R}, \mathrm{A}, \mathrm{S})$ | $S=R \triangleright A$ |
| range anti-restriction | $\operatorname{rares}(\mathrm{R}, \mathrm{A}, \mathrm{S})$ | $S=R \triangleright A$ |
| 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 | $n \operatorname{dom}(\mathrm{R}, \mathrm{A})$ | $\operatorname{dom} R \neq A$ |
| range | $\operatorname{nran}(\mathrm{R}, \mathrm{A})$ | $\operatorname{ran} R \neq A$ |
| composition | ncomp (R, S, T) | $T \neq R \circ S$ |
| inverse | $\operatorname{ninv}(\mathrm{R}, \mathrm{S})$ | $S \neq R^{-1}$ |
| identity relation | $\operatorname{nid}(\mathrm{A}, \mathrm{F})$ | id $A \neq F$ |
| domain restriction | ndres (A, R, S ) | $S \neq A \triangleleft R$ |
| domain anti-restriction | ndares ( $A, R, S$ ) | $S \neq A \triangleleft R$ |
| range restriction | $\operatorname{nrres}(\mathrm{R}, \mathrm{A}, \mathrm{S})$ | $S \neq R \triangleright A$ |
| range anti-restriction | $\operatorname{nrares}(\mathrm{R}, \mathrm{A}, \mathrm{S})$ | $S \neq R \triangleright A$ |
| overriding | noplus( $\mathrm{R}, \mathrm{S}, \mathrm{T}$ ) | $T \neq R \oplus S$ |
| relational image | $\operatorname{nrimg}(A, R, B)$ | $B \neq R[A]$ |

Table 3: Relational operators available in $\{\log \}$ ( $\mathrm{R}, \mathrm{S}$ 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 | $R \circ S=\{(x, z) \mid \exists y:(x, y) \in R \wedge(y, z) \in S\}$ |
| inverse | $R^{-1}=\{(y, x) \mid(x, y) \in R\}$ |
| identity relation | $\operatorname{id} A=\{(x, x) \mid x \in A\}$ |
| domain restriction | $A \triangleleft R=\{(x, y) \mid(x, y) \in R \wedge x \in A\}$ |
| domain anti-restriction | $A \triangleleft R=\{(x, y) \mid(x, y) \in R \wedge x \notin A\}$ |
| range restriction | $R \triangleright A=\{(x, y) \mid(x, y) \in R \wedge y \in A\}$ |
| range anti-restriction | $R \triangleright A=\{(x, y) \mid(x, y) \in R \wedge y \notin A\}$ |
| overriding | $R \oplus S=(\operatorname{dom} S \triangleleft R) \cup S$ |
| relational image | $R[A]=\operatorname{ran}(A \triangleleft R)$ |

Table 4: Definition of relational operators

| Operator | $\{\log \}$ | Meaning |
| :---: | :---: | :---: |
| partial function | pfun(F) | $F$ verifies (9) |
| function application | $\operatorname{apply}(\mathrm{F}, \mathrm{X}, \mathrm{Y})$ | $F(X)=Y$ |
| domain | $\operatorname{dompf}(\mathrm{F}, \mathrm{A})$ | $\operatorname{dom}(\mathrm{F}, \mathrm{A})$ |
| composition | comppf(F,G,H) | $\operatorname{comp}(\mathrm{F}, \mathrm{G}, \mathrm{H})$ |
| domain restriction | drespf(F,G,H) | dres(F,G, H) |
| Negations |  |  |
| partial function | npfun( F ) | $F$ does not verifies (9) |
| function application | napply (F, X, Y) | $F(X) \neq Y$ |
| domain | ndompf( $\mathrm{F}, \mathrm{A}$ ) | $\operatorname{dom} F \neq A$ |

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

### 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:

$$
\begin{equation*}
\forall x, y_{1}, y_{2}:\left(x, y_{1}\right) \in f \wedge\left(x, y_{2}\right) \in f \Rightarrow y_{1}=y_{2} \tag{9}
\end{equation*}
$$

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:

$$
\{\log \}=>\operatorname{dom}(F,\{a\}) \&[a, Y 1] \text { in } F \&[a, Y 2] \text { in } F \& Y 1 \text { neq } Y 2 .
$$

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 answer no for the following:

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

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 $\operatorname{dompf}(\mathrm{F}, \mathrm{A})$ is $p f u n(F) \wedge \operatorname{dom} F=A$; while the precise meaning of $\operatorname{ndomp} f(\mathrm{~F}, \mathrm{~A})$ is $n p f u n(F) \vee \operatorname{dom} R \neq A$.
$\{\log \}$ provides also a less restrictive form of function application through the predicate applyTo. applyTo( $\mathrm{R}, \mathrm{X}, \mathrm{Y}$ ) is true if R is a binary relation containing exactly one pair whose first
component is X and whose second component is Y . Formally, $\operatorname{apply} T o(R, x, y)$ holds iff:

$$
\begin{equation*}
(x, y) \in R \wedge\left(\forall y_{1}:\left(x, y_{1}\right) \in R \Rightarrow y_{1}=y\right) \tag{10}
\end{equation*}
$$

This means that one can use the 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:

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

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:

$$
\{\log \}=>\operatorname{comp}(R, S,\{[X, Z] / U\}) \& u n(S, T, U) .
$$

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:

$$
\{\log \}=>\operatorname{comp}(\{[\mathrm{X}, \mathrm{Z}] / \mathrm{R}\},\{[\mathrm{Z}, \mathrm{Y}] / \mathrm{S}\}, \mathrm{R}) \& \operatorname{id}(\mathrm{~A}, \mathrm{R}) .
$$

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:

$$
\begin{equation*}
\{x: D \mid F(x) \bullet P(x)\} \tag{11}
\end{equation*}
$$

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:

$$
\begin{equation*}
\{y: \exists x(x \in D \wedge F(x) \wedge y=P(x))\} \tag{12}
\end{equation*}
$$

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 (11) 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):

$$
\begin{aligned}
\{\log \}=> & N>1 \& M D \text { is } N \operatorname{div} 2 \& \\
& \operatorname{ris}(X \text { in } \operatorname{int}(2, M D),[], 0 \text { is } N \bmod X, X)=\{ \}
\end{aligned}
$$

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

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:

$$
C F=\operatorname{ris}(S \text { in } D,[C], C>1,[S, C], \operatorname{size}(S, C))
$$

In effect, $C F$ 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, $C F$ 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 ${ }^{4}$. The semantics of a parameter is an existentially quantified variable. Then, the semantics of CF is given by the following intentional set:

$$
\begin{equation*}
\{y: \exists s(\exists c(s \in D \wedge|s|=c \wedge c>1 \wedge y=(s, c)))\} \tag{13}
\end{equation*}
$$

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]) = {}.
```

[^3]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 $\operatorname{size}(A, N)$ 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:

$$
\operatorname{ris}(\mathrm{X} \text { in } \mathrm{A},[\text { vars }], \phi, p, \psi)=\operatorname{ris}(\mathrm{X} \text { in } \mathrm{A},[], \operatorname{let}([\text { vars }], \psi, \phi), p)
$$

### 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., $[\mathrm{X}],[\mathrm{X},[\mathrm{Y}], \mathrm{Z}]$, etc.);
- $[\mathrm{X} \mid \mathrm{Y}]$, where X and Y are different variables;
- $\{\mathrm{X} / \mathrm{Y}\}$, where X and Y are different variables.

In all cases the variables in the control expression are variables bound to the RIS. ${ }^{5}$ 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) = {}.
```

[^4]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 $\geq$ 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:

$$
\{[5,1]\}=\operatorname{ris}([\mathrm{X}, \mathrm{Y}] \text { in }\{1,[0,3],[5,1]\}, \mathrm{X}>=\mathrm{Y})
$$

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

### 5.1.3 Encoding sets of structured elements

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

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

where X 1 and X 2 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 $[X, f(\ldots, X, \ldots)]$, where $X$ is the control expression and $f$ is any function;
- terms of the form $[f(\ldots, X, \ldots), X]$, 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 coinjectivity 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 $S=\operatorname{ris}(X$ 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 $\{\mathrm{t} 1, \ldots, \mathrm{tn}\}$ with $\mathrm{t} 1, \ldots, \mathrm{tn}$ 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],\ldots.,[100,10000]}
```

Note, however, that if Y is $\mathrm{X} * \mathrm{X}$ 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

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

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 : exists(V,G)}
{X : exists([ }\mp@subsup{V}{1}{},\ldots,\mp@subsup{V}{n}{}],G)
```

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

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

The following are two simple examples using GIS.

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

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

Sample goal:

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

- cross_product $(A, B, C P)$ is true if $C P$ 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 $\mathrm{A}, \mathrm{F}$ )
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. ${ }^{6}$ Same considerations as with RIS apply to the use of the let construct in RQ (see at the end of Sect. 5.1.1).
- exists(X in A,F)
where arguments are as in foreach/2. The semantics of such a constraint is the expected:

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

- 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., $\exists x: x \in A \wedge \neg F$. 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.

Formulas including foreach and exists are safe provided at least one of the following conditions is met:

[^5]- The formula contains only foreach (including the formulas inside them).
- The formula contains only exists (including the formulas inside them).
- The formula contains foreach and exists but all the foreach are 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 $\{\log \}$ formula not containing REQ and RUQ fitting in a decidable fragment.

- The formula contains foreach and exists but doesn't belong to the above class. In this case the condition for decidability is rather complex. The technical details can be found elsewhere [14, Theorem 6]. Here we describe the condition in a rather informal way. Basically, an exists present after a foreach cannot share the same domain variable. For instance, the following two formulas do not verify the above condition:

```
foreach(X in {W / A}, exists(Y in {V / A}, formula))
forall(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 $B$.
Hence, if $\{\log \}$ is called on any of these formulas, it will loop forever.
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 $(S) \Leftrightarrow r e l(S)$ ).

```
    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. A REQ can be the formula of a RUQ but decidability may be compromised. As an example, the formula:

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

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

$$
\begin{aligned}
\{\log \}=> & S 1=\{a, b\} \& S 2=\{Z\} \& \\
& \text { foreach(X in } S 1, \text { foreach(Y in } S 2, X \text { neq } Y)) .
\end{aligned}
$$

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 $R \mathrm{Q},\{\log \}$ 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)
```


## 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)
```

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 $A,\left[V \_1, \ldots, V \_n\right], F$, 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:

$$
\{\log \}=>\text { foreach }(X \text { in } R, X \text { in }\{a, b\}) \text {. }
$$

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:

$$
\begin{aligned}
& R=\{ \} \\
& R=\{a\} \\
& R=\{a, b\} \\
& R=\{b\}
\end{aligned}
$$

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:

$$
\{\log \}=>\text { forall }(X \text { in } R, X \text { in }\{a / S\}) \& a \operatorname{nin} R .
$$

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

$$
\{l o g\}=>\text { not_singleton_set }(\{1\}) \text {. }
$$

$\{\log \}$ 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

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

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, $\mathrm{S}=\{[1,2],[\mathrm{a}, \mathrm{b}]\}$ ).

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, e2 are arithmetic expressions and n is a either a variable or a numeric constant. An arithmetic expression is either a variable or a numeric constant or an arithmetic function (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:

$$
\begin{aligned}
& \{\log \}=>\mathrm{X} \text { is } 3 * 5 . \\
& \mathrm{X}=15 \\
& \{\log \}=>1.5+1>0.7 . \\
& \text { yes }
\end{aligned}
$$

|  | Operator | $\{\log \}$ |
| :--- | :--- | :--- |
| simple equality | n is e 1 | $n=e_{1}$ |
| less or equal | $\mathrm{e} 1=<\mathrm{e} 2$ | $e_{1} \leq e_{2}$ |
| less | $\mathrm{e} 1<\mathrm{e} 2$ | $e_{1}<e_{2}$ |
| greater or equal | $\mathrm{e} 1>=\mathrm{e} 2$ | $e_{1} \geq e_{2}$ |
| greater | $\mathrm{e} 1>\mathrm{e} 2$ | $e_{1}>e_{2}$ |
| equal | $\mathrm{e} 1=:=\mathrm{e} 2 e_{1}=e_{2}$ |  |
| not equal | $\mathrm{e} 1=\backslash=\mathrm{e} 2 e_{1} \neq e_{2}$ |  |

Table 6: Comparison arithmetic operators available in $\{\log \}$

|  | FUnCTION | $\{\log \}$ | MEANING |
| :--- | :---: | :---: | :---: |
| addition | $\mathrm{e} 1+\mathrm{e} 1$ | $e_{1}+e_{1}$ |  |
| substraction | $\mathrm{e} 1-\mathrm{e} 1$ | $e_{1}-e_{2}$ |  |
| product | $\mathrm{e} 1 \% \mathrm{e} 1$ | $e_{1} \times e_{2}$ |  |
| division | $\mathrm{e} 1 / \mathrm{e} 1$ | $e_{1} / e_{2}$ |  |
| integer division | $\mathrm{e} 1 \mathrm{div} \mathrm{e} 1 e_{1} \mathrm{div} e_{2}$ |  |  |
| integer module/remainder | $\mathrm{e} 1 \mathrm{mod} \mathrm{e} 2 e_{1} \mathrm{mod} e_{2}$ |  |  |

Table 7: Arithmetic functions available in $\{\log \}$

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:

$$
\{\log \}=>2+3 \text { in }\{5\} .
$$

$\{\log \}$ 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:

$$
\{\log \}=>X \text { is } 2+3 \& X \text { in }\{5\} .
$$

turns out to be satisfiable and the answer will be $\mathrm{X}=5$. 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:

$$
\begin{aligned}
& \{\log \}=>1.5+X>0.7 \& X \text { is } 2 * 3.0 . \\
& \text { Problem in arithmetic expression } \\
& \text { no }
\end{aligned}
$$

while

$$
\begin{aligned}
& \{\log \}=>X \text { is } 2 * 3.0 \& 1.5+X>0.7 \\
& X=6.0
\end{aligned}
$$

|  | Operator | $\log$ |
| :--- | :--- | :--- |
| integer |  | integer $(\mathrm{t})$ |
| integer |  | $t$ is an integer number |

Table 8: integer and ninteger constraints
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 $(\mathrm{X})$ 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.

### 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 $\operatorname{int}(m, n)$, 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 $\operatorname{int}(1,10)$
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).
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:

$$
\text { \{log\}=> nolabel. }
$$

If we, successively, give the goal

$$
\{\log \}=>34>x+1 \& X \operatorname{in} \operatorname{int}(1,100) .
$$

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:

$$
\{\log \}=>~ l a b e l . ~
$$

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 $34>\mathrm{X}+1 \& \mathrm{X} \operatorname{in} \operatorname{int}(1,100)$ 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 (14) 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 $\operatorname{CLP}(\mathrm{Q})$ 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 $\operatorname{CLP}(\mathrm{Q})$ 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: $34>\mathrm{X}+1, \mathrm{X}>=1, \mathrm{X}=<100$, integer $(\mathrm{X}), \ldots$
meaning that the input formula is satisfiable. ${ }^{7}$ 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 $\operatorname{CLP}(\mathrm{FD})$ 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 $\operatorname{CLP}(\mathrm{Q})$ the correct way is to find the quotient of $x / 2$ :

$$
\{\log \}=>\mathrm{X} \text { is } 2 * \mathrm{Q}
$$

where Q is a new variable. As can be seen, this is a linear constraint and so $\operatorname{CLP}(\mathrm{Q})$ 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 ${ }^{8}$ for further details.

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:

[^6]```
{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. Recall that in this situation the solver mode should be preferred over the prover mode-cf. Section 3.7. Conversely, if you are using $\{\log \}$ as an automated theorem prover, then $\operatorname{CLP}(\mathrm{Q})$ 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:

$$
\{\log \}=>\operatorname{size}(\{1 / \mathrm{R}\}, \mathrm{M}) .
$$

we get as first answer:

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

If the second argument of size is a constant $k$ and the first is a variable then the computed answer depends on the operation mode of the solver (cf. Section 3.7). In solver mode the most general set of $k$ elements is explicitly shown. As an example, the answer to the following goal:

$$
\{\log \}=>\operatorname{size}(A, 3) .
$$

is

```
\(\mathrm{A}=\{\mathrm{X}, \mathrm{Y}, \mathrm{Z}\}\)
Constraint: X neq \(\mathrm{Y}, \mathrm{X}\) neq \(\mathrm{Z}, \mathrm{Y}\) neq Z
```

|  | OPERATOR | $\{\log \}$ |
| :--- | :--- | :--- |
| set cardinality |  | MeANING |
| not set cardinality |  | nsize $(\mathrm{A}, \mathrm{N})$ |

Table 9: The set cardinality operators
because $\{\mathrm{X}, \mathrm{Y}, \mathrm{Z}\}$, with $\mathrm{X}, \mathrm{Y}$ 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 $\left\{1,2,3, \_N 7, \_N 6, \ldots N 5, \ldots N 4, \_N 3, \ldots N 2, \ldots N 1\right\}$, along with the necessary constraints to ensure that elements in the set are all distinct from each other. ${ }^{9}$

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 $\operatorname{CLP}(\mathrm{Q})$ is the active integer solver.

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

[^7]```
{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:

$$
\begin{aligned}
\{\log \}=> & \operatorname{size}(A, M) \& 1=<M \& M=<2 \& \operatorname{size}(B, N) \& 5=<N \& \\
& \operatorname{subset}(C, B) \& \operatorname{size}(C, K) \& 7=<K .
\end{aligned}
$$

is

$$
\begin{aligned}
& \text { true } \\
& \text { Constraint: } \operatorname{size}(A, M), M>=0,1=<M, M=<2, \operatorname{size}(B, N), N>=0, \\
& \\
& 5=<N, \operatorname{subset}(C, B), \operatorname{size}(C, K), K>=0,7=<K
\end{aligned}
$$

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

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

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

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

$$
\begin{aligned}
\{\log \}=> & \text { show_min. } \\
\{\log \}=> & \operatorname{size}(A, M) \& 1=<M \& M=<2 \& \operatorname{size}(B, N) \& 5=<N \& \\
& \operatorname{subset}(C, B) \& \operatorname{size}(C, K) \& 7=<K .
\end{aligned}
$$

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 $M=1, \quad N=7, K=7$ 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. ${ }^{10}$ 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:

$$
\begin{aligned}
& \{\log \}=>\operatorname{un}(\{X, Y\},\{V, W\}, \operatorname{int}(M, 5)) \& M \text { is } X-Y . \\
& X=4, \\
& Y=2, \\
& V=5, \\
& W=3, \\
& M=2
\end{aligned}
$$

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;
- $\mathrm{CLP}(\mathrm{Q})$ is the active integer solver,
then $\{\log \}$ will always compute the right answer even when set arguments are terms of the form $\operatorname{int}(m, n)$ (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));
- intervals can also occur as the set part of extensional sets, e.g., $\{-1 / \operatorname{int}(1, N)\}$.

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.

[^8]
### 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).
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 ${ }^{11}$ within its decision procedures, and some others can be defined outside of them.

### 10.1 Minimum and maximum of a set

$\{\log \}$ provides predicates $\operatorname{smin}(S, \min )$ and $\operatorname{smax}(S, \max )$ 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 $\{\log \}$

[^9]as long as the second argument participates only in linear integer arithmetic (and the active integer solver is $\operatorname{CLP}(\mathrm{Q})$ ).

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}=> \operatorname{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 $\operatorname{smin}$ 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 ${ }^{12}$ :

$$
\{\log \}=>\operatorname{sum}(\{\mathrm{X} 1, \mathrm{X} 2\}, 2) \& \text { labeling }(\mathrm{X} 1)
$$

we get the three solutions:

[^10]```
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 $\operatorname{int}(m, n)$ for some constants $m$ and $n$, and force labeling for at least some of them (remember that, if $\operatorname{CLP}(\mathrm{FD})$ is activated, then labeling is performed automatically at the end of the computation). For example, $\{\log \}$ gives the correct answer in the following case:

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

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

### 10.3 Sum of an array

The contents of this section and its implementation in $\{\log \}$ is 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 the decidable fragment if the cardinality of arrays, and anything related to them, is never taken. For example, the following is a formula laying outside the decidable fragment:

```
\(\operatorname{arr}(A, N) \& \operatorname{ran}(A, R) \& \operatorname{size}(R, K) \& K<N\)
```

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 N , the following predicate computes the sum of the first K components of A .

```
arrsum(A,N,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]},4,2,S).
S = 8
```

The sum is performed by computing the partial sums in each component of Tr . In effect, the following is true of $\operatorname{arrsum}(A, N, K, S u m)$ :

$$
\begin{aligned}
& \operatorname{Tr}(1)=A(1) \\
& \operatorname{Tr}(2)=\operatorname{Tr}(1)+A(2) \\
& \operatorname{Tr}(3)=\operatorname{Tr}(2)+A(3) \\
& \cdots \cdots \cdots \cdots \cdots \cdots \cdots \\
& \operatorname{Tr}(k)=\operatorname{Tr}(k-1)+A(k)
\end{aligned}
$$

In a sense, Tr records the state trace of the standard imperative algorithm computing the sum of an array. Consequently, arrsum/4 would be an abstract or logical representation of that algorithm.

As can be seen, arrsum/4 never takes the cardinality of anything related to an array. So, according to our first studies, arrsum/4 belongs to the decidable fragment.

## 11 Types in $\{\log \}$

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

$$
\{a, 1,\{2\},[5, \text { "messi" }]\}
$$

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:

$$
\{\log \}=>X \text { Y. }
$$

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

$$
\{\log \}=>X>Y \& u n(X,\{1,2,3\}, Z) .
$$

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:

$$
\{\log \}=>\operatorname{id}(\{X / A\}, R) \& i d(R, A) .
$$

makes $\{\log \}$ to enter an infinite loop that the user can interrupt by typing Ctrl+c. Due to the first constraint, $[\mathrm{X}, \mathrm{X}]$ belongs to R and by the second constraint $[\mathrm{X}, \mathrm{X}],[\mathrm{X}, \mathrm{X}]$ ] 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:

$$
\{1 \mathrm{log}\}=>\text { type_check. }
$$

and is deactivated with:

$$
\text { \{log\}=> notype_check. }
$$

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,set([t,t]))
```

Each dec constraint states that a variable is of a certain type. For instance, $\operatorname{dec}(R, \operatorname{set}([t, t]))$ states that $R$ is a set of ordered pairs whose components are both of type $t$. If this formula is run while the 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),set([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:

$$
\{\log \}=>\operatorname{un}(\{a, 1,\{2\},[5, " m e s s i "]\},\{X\},\{Y / R\}) .
$$

However, $\{\log \}$ provides a type system where an encoding of the set $\{a, 1,\{2\},[5$, "messi" $]\}$ can be correctly typed-see "Towards untyped sets" in Section 11.1.7.

In the following section the type system and other related features are introduced. In Section 12 the reader will find a complete example of a typed $\{\log \}$ program.

### 11.1 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 $\operatorname{dec}(V, t), V$ 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:

$$
\operatorname{dec}\left(\left[V_{i}, \ldots, V_{n}\right], \mathrm{t}\right) \Leftrightarrow \operatorname{dec}\left(V_{i}, \mathrm{t}\right) \wedge \cdots \wedge \operatorname{dec}\left(V_{n}, \mathrm{t}\right)
$$

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

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

### 11.1.1 Integers

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

$$
\{\log \}=>\operatorname{dec}(X, i n t) \& X>Y \& \operatorname{dec}(Y, i n t) .
$$

Numbers are automatically typed as expected. Then a formula such as $\mathrm{X}>10$ can be typed as follows:

$$
\{\log \}=>\operatorname{dec}(X, \text { int }) \& X>10 .
$$

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

### 11.1.2 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.

### 11.1.3 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 $\mathrm{t}:\langle$ atom $\rangle$, 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

$$
\begin{aligned}
& \{\log \}=>\operatorname{dec}(N, \text { name }) \& N=\text { name:john. } \\
& N=\text { name:john }
\end{aligned}
$$

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 11.9 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 ${ }^{13}$ :

$$
\begin{aligned}
& \{\log \}=>\text { idef_type(city,str). } \\
& \{\log \}=>\text { idef_type(name,str). }
\end{aligned}
$$

[^11]make city and name the same type, str. This might be confusing:
\[

$$
\begin{aligned}
& \{\log \}=>\operatorname{dec}(N, \text { name }) \& \operatorname{dec}(C, \text { city }) \& N=\text { "Leo" \& C = "Leo". } \\
& N=\text { Leo, } \\
& C=\text { Leo }
\end{aligned}
$$
\]

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
```


### 11.1.4 Enumerated types

enum $\left(\left[e_{1}, \ldots, e_{n}\right]\right)$ 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 11.10 thus removing all the enumeration types being used.

Note that enum([yes, no ]) is a different type than enum([no, yes]). Actually, the typechecker 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 11.3 to learn how to give a name to enumerated types so you do not have to repeat the enumeration in each type constraint.

### 11.1.5 Sum types

Enumerated types are a particular class of sum types. Sum types implement the widely known notion of variant ${ }^{14}$ (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)]))])
```

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 $\operatorname{pair}([i, j])$ where $i$ and $j$ are of type int and $[i, j]$ is of a product type-see Section 11.1.6. 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 ( $\left[e_{1}, \ldots, e_{n}\right]$ ) is rewritten as sum ( $\left[e_{1}, \ldots, e_{n}\right]$ ). 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 11.10.

These restrictions imply that if you want to introduce, for example, two or more option types ${ }^{15}$ you must use different constructors. In other words you can't use none and some for all the option types. For example:

```
dec(X,\operatorname{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]
### 11.1.6 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:

```
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:

$$
\mathrm{B}=[\mathrm{A}, \mathrm{blue}]
$$

will type-check if dec (A , city) is in context. Conversely,
B is $\mathrm{X}+7$
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]]

### 11.1.7 Set types

If $t$ is a type then $\operatorname{set}(t)$ 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 $\left\{e_{1}, \ldots, e_{n} / S\right\}$ 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:

$$
\{a, 1,\{2\},[5, " m e s s i "]\}
$$

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

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


### 11.2 Types for binary relations and partial functions

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

```
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:

$$
\operatorname{rel}(t, u)==\operatorname{set}([t, u])
$$

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 $\operatorname{rel}(R)$ (cf. Table 3) and $\operatorname{dec}(R, r e l(t, u))$ are the following:

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

Note that a $c p$ term is a set of ordered pairs. Then the type of $\operatorname{cp}(A, B)$ is rel $(u, t)$ 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:

$$
\{\log \}=>\operatorname{dec}(F, r e l(t, u)) \& p f u n(F) \& \ldots
$$

Furthermore, $\mathrm{pfun}(\mathrm{F})$ 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,\operatorname{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.

### 11.3 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([01,02],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([01,02],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:

$$
\operatorname{dec}(V, \mathrm{t}) \Leftrightarrow \operatorname{dec}(V, \text { texpr })
$$

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

See Section 11.10 to learn how type declarations can be consulted and managed.

### 11.4 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,\operatorname{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 11.10 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.

### 11.5 Typing polymorphic operators

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 $\mathrm{T}, \mathrm{U}$ and V are type variables. That is, $\mathrm{T}, \mathrm{U}$ and V are not types but they can be bound to types-recall that types and type constructors always begin with a lowercase letter.

You can define your own polymorphic operators and type them. For example, if you want to define the predicate un 3 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 $p(\ldots)$ 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.

### 11.6 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:

$$
\{\log \}=>\operatorname{dec}([W, X, Y, Z], \operatorname{set}(t)) \& u n 3(W, X, Y, Z) .
$$

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:

$$
\{\log \}=>\operatorname{dec}([X, Y, Z], \operatorname{set}(t)) \& u n 3(\{t: a, t: a a\}, X, Y, Z) .
$$

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

$$
\begin{aligned}
\{\log \}=> & \operatorname{dec}([X, Y, Z], r e l(t, u)) \& \operatorname{dec}(I, t) \& \\
& \operatorname{un} 3(\{[I, u: a b c]\}, X, Y, Z) .
\end{aligned}
$$

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).
```


## 11.7 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 $\mathrm{n}\langle$ number $\rangle$, 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, $t: n 0$ is a first component because the elements of type $t$ are of the form $t:\langle a t o m\rangle$, as explained in Section 11.1.3. In turn, the second components are of the form $\mathrm{n}\langle$ number $\rangle$ because they are strings.

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

- int $\rightarrow 0,1,-1,2,-2, \ldots$
- $\operatorname{str} \rightarrow \mathrm{n} 0, \mathrm{n} 1, \mathrm{n} 2, \ldots$
- basic type $t \rightarrow t: n 0, t: n 1, t: n 2, \ldots$
- sum types $\rightarrow$ 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 $\rightarrow$ ordered pairs where the first components and the second components are constants of the corresponding types.
- set type $\rightarrow$ the constants are singleton sets where the elements are of the corresponding type.


### 11.8 Proving goals involving finite types

Clearly, the following goal is unsatisfiable:

$$
\{\log \}=>\operatorname{dec}(Z, \text { enum }([t, f])) \& Z \text { neq } t \& Z \text { neq } f .
$$

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 $\mathrm{X} 1, \mathrm{X} 2$ and X 3 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 using 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 $[\operatorname{enum}([t, f])$, 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 ( $[\mathrm{t}, \mathrm{f}]$ ) is a finite type.
4. $T$ is the set type of a finite type.

### 11.9 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:

$$
\begin{aligned}
& \{\log \}=>[a \mid[]]=1 \\
& \text { type error: '[a]' doesn't fit in the sum type }
\end{aligned}
$$

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.

### 11.10 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 11.3, 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 ( $\mathrm{t}, \mathrm{E}$ ) 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
```


## 12 Specifying and verifying state machines

The contents of this section and its implementation in $\{\log \}$ is still experimental
$\{\log \}$ 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]. Furthermore, $\{\log \}$ provides declarations that are used to automatically generate verification conditions on those state machines. Later, users can use $\{\log \}$ itself to automatically prove or disprove these verification conditions. We will show how to do that by encoding in $\{\log \}$ the classic birthday book problem used by Spivey to illustrate the Z formal notation.

The features described in this section can only be used if the file setlog_vcg.pl is present in the working directory.

### 12.1 The specification of the birthday book problem

In $\{\log \}$ a state machine is composed of:

1. Zero or more parameters
2. One or more state variables
3. Zero or more axioms
4. Zero or more state invariants
5. Possibly 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. Theorems 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 zero or more invariant declarations but any sensible state machine will have at least one of them. 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.

After introducing an invariant the user has to introduce its negation as well. So the above predicates are completed as follows.

```
invariant(birthdayBookInv).
birthdayBookInv(Known,Birthday) :- dom(Birthday,Known).
n_birthdayBookInv(Known,Birthday) :- neg(dom(Birthday,Known)).
invariant(pfunInv).
pfunInv(Birthday) :- pfun(Birthday).
n_pfunInv(Birthday) :- neg(pfun(Birthday)).
```

Recall Section 3.5 to know how to declare the negation of a user-defined predicate and the limitations of the neg connective. In particular, recall that $\{\log \}$ doesn't automatically compute the negation of these predicates.

Declaring that a predicate is an invariant doesn't mean that the state machine will indeed verify 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 12.3.

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).
dec_p_type(n_birthdayBookInv(kn,bb)).
n_birthdayBookInv(Known,Birthday) :- neg(dom(Birthday,Known)).
invariant(pfunInv).
dec_p_type(pfunInv(bb)).
pfunInv(Birthday) :- pfun(Birthday).
dec_p_type(n_pfunInv(bb)).
n_pfunInv(Birthday) :- neg(pfun(Birthday)).
```

After introducing invariants we can 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.

There can be zero or one initial declaration in any given state machine but most of them will have one. The initial declaration must come after the last invariant (if any) and before the first operation. The predicate defining the initial states of the system must depend on at least one state variable, and may 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 (' ${ }^{\prime}$ '). 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 12.3.

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_i,Date_i,Known_,Birthday_) :- ...
```

Then Known_ and Birthday_ are part 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 addBirthday0k:

```
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 addBirthday0k 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 12.3.

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.

### 12.2 Parameters, axioms and theorems

In this section we will explain how parameters, axioms and theorems can be used to specify state machines.

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 that 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_ as an argument will make $\{\log \}$ to issue an error.

Theorems can be declared after an axiom, invariant or operation. These are used to state properties that can be deduced from axioms, invariants or operations.

For example, if we have:

```
variables([F,G,H]).
invariant(inv1).
inv1(F) :- pfun(F).
invariant(inv2).
inv2(G) :- pfun(G).
```

Then, we can declare the following theorem:

```
theorem(thrm1).
thrm1(F,G,H) :- neg(inv1(F) & inv2(G) & comp(F,G,H) implies pfun(H)).
```

Note that thrm1 is stated as an unsatisfiability condition because, as we will see, $\{\log \}$ will include a verification condition ensuring that this is really a theorem.

### 12.3 Automatic generation of verification conditions

Once we have defined a state machine as described above, we can use the Verification Condition Generator (VCG). 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:

$$
\begin{aligned}
& \{\log \}=>~ c o n s u l t(' b b . s l o g ') . \\
& \{\log \}=>~ v c g(' b b . s l o g ') . ~
\end{aligned}
$$

In this way $\{\log \}$ generates a file named $\mathrm{bb}-\mathrm{vc}$. 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).

### 12.4 The findh command family

$\{\log \}$ was unable to discharge addBirthday_pi_pfunInv because of the way it generates the verification conditions. This point is treated more deeply in Section 12.5. For the moment, it suffice to say that $\{\log \}$ generates each verification condition with the minimum number of hypothesis. In consequence, $\{\log \}$ is sometimes unable to discharge a verification condition because there are some missing hypothesis. Missing hypothesis can always be found among the axioms and invariants declared in the specification.

The findh family of commands helps users to find missing hypothesis in verification conditions whose proofs failed (as with addBirthday_pi_pfunInv). The simplest way of finding missing hypothesis (although in some cases it will take a long time), is by running the findh command:

$$
\begin{aligned}
& \text { \{log\}=> findh. } \\
& \text { Missing hypotheses for addBirthday_pi_pfunInv: [[birthdayBookInv]] }
\end{aligned}
$$

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 hypothesis.

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 hypothesis 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,ole, axinvs) where vc is the identifier of a verification condition (e.g. addBirthday_pi_pfunInv), $n$ is a positive natural number, ole 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 ( $\mathrm{vc}, \mathrm{n}$ ) 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.

### 12.5 The 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. 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 $F$.
3. The initial state satisfies each and every invariant (_sat_). As an example we have:
```
birthdayBookInit_sat_birthdayBookInv :-
    birthdayBookInit(Known,Birthday) &
    birthdayBookInv(Known,Birthday).
```

If there's no initial state, then the proof obligation checks that each invariant is satisfiable.
4. 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_i,Date_o).
```

5. Invariance lemmas: each operation preservers each and every invariant (_pi_). For example:
```
addBirthday_pi_birthdayBookInv :-
    neg(
        % here conjoin other invariants as hypothesis if necessary
        birthdayBookInv(Known,Birthday) &
        addBirthday(Known,Birthday,Name_i,Date_i,Known_,Birthday_) implies
        birthdayBookInv(Known_,Birthday_)
    ).
```

6. Theorems: user-defined theorems are included as verification conditions. Since $\{\log \}$ will try to prove their unsatisfiability, theorems must encode unsatisfiable formulas. Hence, if you want $p$ to be a theorem, then the body of the corresponding clause should be neg ( $p$ ).
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 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.

### 13.1 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:

$$
\{\log \}=>\operatorname{diff}(\{1,2,3\},\{2\}, S) .
$$

makes $\{\log \}$ to answer

$$
S=\left\{1,3 / \_N 1\right\} \text { Constraint: subset(_N1,\{1,2,3\}), set(_N1), } 2 \text { nin _N1 }
$$

while by issuing

$$
\{\log \}=>\text { solve(diff(\{1,2,3\},\{2\},S)). }
$$

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.

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 $\mathrm{X}=\mathrm{a}$ and $\mathrm{X}=\mathrm{b}$.
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.

### 13.2 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:
$-S=c f i r s t:$ 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 ${ }^{16}$. 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:

$$
\{\log \}=>\text { write(b) \& write }(X) \& n l \& X \text { in }\{a\} .
$$

we get as answer

$$
\begin{aligned}
& \mathrm{ba} \\
& \mathrm{X}=\mathrm{a}
\end{aligned}
$$

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.

### 13.3 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 .


## 14 Prolog- $\{\log \}$ communication

### 14.1 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 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 Sect. 13). A complete list of the available execution options can be found in Appendix A. As an example, the following goal:

[^13]```
?- setlog(X in int (1,5),1000,C,R,
    [int_solver(clpfd),nolabel]).
```

requires the goal X in $\operatorname{int}(1,5)$ 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 ${ }_{1}, \ldots$, OptList $\left.\left._{n}\right]\right)$ : same 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}\mp@subsup{}{i}{}
```

terminates with a timeout, then the call:

```
setlog(Goal,Timeout,OutConstrList,Res,OptsLstti+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 $\operatorname{goal} \operatorname{dom}(R, S) \& \operatorname{ran}(R,\{1 / S\})$ to be executed, the first time, with a 2000 millisecond timeout and no execution mode options enforced; since this call terminates with a timeout, then the goal dom $(R, S) \& \operatorname{ran}(R,\{1 / S\})$ 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 $\mathrm{T}^{*} n$, where $n$ is the number of OptsList occurring in the try list.

- 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,\infty,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,_)
```

- 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: to enter/re-enter the $\{\log \}$ interactive environment.
- 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:

$$
\begin{aligned}
& A=1, V=E, E=\{1,2\}, X=\{ \}, \\
& C=[], R=\text { success. }
\end{aligned}
$$

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:

$$
\begin{aligned}
& \text { ?- } \mathrm{G}=\mathrm{X}=1 \text { \& un }(\{1\},\{2\}, \mathrm{V}) \& \mathrm{un}(\mathrm{~V}, \mathrm{Q}, \mathrm{E}) \mathrm{"}, \\
& \text { term_string(_, } \mathrm{G},[\text { variable_names(VN)]), } \\
& \text { setlog_str(G,VN, } 10000, \mathrm{C}, \mathrm{R},[\text { groundsol]). }
\end{aligned}
$$

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, m,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, \infty,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. 13.


### 14.2 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:

- $n l / 0$
- ground/1
- var/1
- nonvar/1
- name/2
- functor/3
- arg/3
- $=. . / 2$
- $==/ 2$
- $\quad \backslash==/ 2$
- $@</ 2$
- $\quad$ a>/2
- $\quad @=</ 2$
- $\quad$ a $>=/ 2$


## 15 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 ( $\mathrm{PS}=2^{\mathrm{S}}$ )
- 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 ( $\mathrm{S}, \mathrm{L}, \mathrm{NewL}$ ): 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 ( $\{4,2\},[\mathrm{a}, \mathrm{h}, \mathrm{g}, \mathrm{m}, \mathrm{t}, \mathrm{r}], \mathrm{L}$ ) returns $L=[h, m]$ )
- drop ( $\mathrm{N}, \mathrm{L}, \mathrm{NewL}$ ): NewL is L with its first N elements removed
- filter ( $\mathrm{L}, \mathrm{S}, \mathrm{NewL}$ ): 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 ( $[\mathrm{a}, \mathrm{h}, \mathrm{g}, \mathrm{m}, \mathrm{t}, \mathrm{r}],\{\mathrm{m}, \mathrm{h}, \mathrm{s}\}, \mathrm{L}$ ) returns $\mathrm{L}=[\mathrm{h}, \mathrm{m}]$ )
- prefix $(P, L)$ : list $P$ is a prefix of list $L$
- sublist ( $\mathrm{Sb}, \mathrm{L}$ ): list Sb is a sublist of list L
- take ( $\mathrm{N}, \mathrm{L}, \mathrm{NewL}$ ): 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. $8^{\text {th }}$ 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. 540556.
[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 set-theoretic decision procedure for quantifier-free, decidable languages extended with restricted quantifiers. CoRR, abs/2208.03518, 2022. Under consideration in Journal of Automated Reasoning.

## A $\{\log \}$ commands

- comp_elim/nocomp_elim (default: comp_elim): to activate/deactivate complete rewriting of comp constraints
- fix_size/nofix_size (default: nofix_size): to activate/deactivate fixed set cardinality mode for size constraint solving
- int_solver(S), S = clpq | clpfd (default: clpq): to select the integer constraint solver
- irules/noirules (default: irules): to activate/deactivate inference rules
- label/nolabel (default: label): to activate/deactivate labeling on integer variables
- mode(M), $M=$ prover | solver (default: prover): to change the solver operation mode
- neq_elim/noneq_elim (default: neq_elim): to activate/deactivate neq elimination
- ran_elim/noran_elim (default: ran_elim)): to activate/deactivate complete rewriting of ran constraints
- strategy(Str), Str = cfirst | ordered | cfirst(list_of_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 $\{\log \}$ typechecking (see Sect. 11).
- groundsol/nogroundsol (default: nogroundsol): to activate/deactivate the computation of ground solutions (see Sect. 3.2).


[^0]:    ${ }^{1}$ 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.

[^1]:    ${ }^{2}$ 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.

[^2]:    ${ }^{3}$ The current version of $\{\log \}$ doesn't implement all these restrictions, so users are responsible of enforcing them.

[^3]:    ${ }^{4}$ 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.

[^4]:    ${ }^{5}$ The variables in control expressions are subjected to the same locality restriction of control variables and parameters.

[^5]:    ${ }^{6}$ 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.

[^6]:    ${ }^{7}$ The constraints not shown in the computed answer are (negligible) integer constraints over fresh internal variables.
    ${ }^{8}$ https://www.swi-prolog.org/pldoc/doc_for?object=bb_inf/4

[^7]:    ${ }^{9}$ As a more practical solution, in the current version, if $k$ is less or equal to a given threshold (now fixed to 6), then solving the constraint size $(A, k)$ causes the set $A$ to be anyway generated, disregarding the solver execution mode.

[^8]:    ${ }^{10}$ From now on, we will say integer interval or just interval meaning finite integer intervals.

[^9]:    ${ }^{11}$ Wikipedia.org: Aggregate function

[^10]:    ${ }^{12}$ Here we assume CLP $(Q)$ is the active integer solver; if $C L P(F D)$ is active, labeling is unnecessary as is performed automatically.

[^11]:    ${ }^{13}$ idef_type is explained in Section 11.3.

[^12]:    ${ }^{14}$ Wikipedia.org: Tagged union
    ${ }^{15}$ Wikipedia.org: Option type

[^13]:    ${ }^{16}$ With the exception of equalities which are in any case considered before any other non-constraint predicate.

