{*log*}: Programming and Automated Proof in Set Theory

Maximiliano Cristiá* Gianfranco Rossi†

{*log*} ('setlog')[1] is a programming language and satisfiability solver based on set theory and set relation algebra, implementing the concept of *formulas as programs*—as an alternative approach to *formulas as types* or *proofs as programs*. As a programming language it is at the intersection of Constraint Logic Programming, set programming and declarative programming. As a satisfiability solver {*log*} implements several decision procedures for the theory of finite sets and finite set relation algebra. In {*log*} programmers can write abstract programs using all the power of set theory and binary relations. These programs can resemble B or Z specifications. Then, the correctness of {*log*} programs is more evident than if they were written in conventional programming languages. Furthermore, {*log*} programs are also set formulas. Hence, programmers can use {*log*} again to *automatically* prove their programs verify non trivial properties. We call the capacity of {*log*} code to behave as a formula and as a program, the *formula-program duality*; in {*log*} one can write *forgrams*[2] instead of plain programs.

The following is a {*log*} forgram computing the minimum (M) of a set (S):

$$\mathsf{min}(S, M) \text{ :- } M \text{ in } S \;\; \& \;\; \mathsf{foreach}(X \text{ in } S, M \leq X).$$

where in corresponds to $\in$ and $\mathsf{foreach}(X \text{ in } S, M \leq X)$ to $\forall X(X \in S \implies M \leq X)$; both in and foreach are constraints. As can be seen, the forgram follows the mathematical definition of minimum of a set. We can use min as a program to compute the minimum of a given set:

$$\mathsf{min}(\{12, 3, 6, 8\}, M)$$

in which case {*log*} answers $M = 3$. {*log*} can also perform symbolic executions of min:

$$\mathsf{min}(\{12, 3, Y, 8\}, M) \qquad [Y \text{ is a variable}] \tag{1}$$

in which case {*log*} (interactively) returns two solutions: $M = 3, 3 \leq Y$, and $M = Y, Y \leq 3$. As this example shows, in {*log*} a query (or goal) is executed against a set of stored clauses, much as in Constraint Logic Programming (CLP) languages. In CLP terminology, the set of stored clauses is called *program*. Hence, min is a program as it is a stored clause. In this sense, {*log*} has an operational semantics similar to that of CLP languages [13].

min is also a formula or specification. Then, we can use {*log*} to check some of its properties. As {*log*} is a satisfiability solver, if we want to check $p \implies q$ we have to call {*log*} on $\neg(p \implies q)$: if the answer is false, it means that $p \implies q$ is a theorem; otherwise, it is not and {*log*} will deliver a counterexample. For instance:

$$\mathsf{neg}(\mathsf{min}(S, M) \;\; \& \;\; \mathsf{min}(S, N) \text{ implies } M = N) \tag{2}$$

returns false, whereas:

$$\mathsf{neg}(\mathsf{min}(S, M) \;\; \& \;\; \mathsf{min}(S, N) \text{ implies } M < N) \tag{3}$$

*Universidad Nacional de Rosario and CIFASIS, Argentina — cristia@cifasis-conicet.gov.ar
†Università di Parma, Italy — gianfranco.rossi@unipr.it

[1] https://www.clpset.unipr.it/setlog.Home.html

[2] Forgram is a portmanteau word resulting from the combination of *for*mula and pro*gram*.

returns a counterexample: $S = \{M \sqcup N_1\} \wedge M = N \wedge \mathsf{foreach}(X \text{ in } N_1, M \leq X)$, where $\{M \sqcup N_1\}$ means $\{M\} \cup N_1$ and N_1 is a new set variable. Note that the counterexample can be satisfied by substituting N_1 with the empty set, in which case we get: $S = \{M\}, M = N$.

{*log*} implements a decision procedure for the theory of *hereditarily finite sets* ($\mathcal{HFS}$), i.e., finitely nested sets that are finite at each level of nesting [13]; a decision procedure for a very expressive fragment of the theory of finite set relation algebras [4]; a decision procedure for $\mathcal{HFS}$ extended with restricted intensional sets [6]; a decision procedure for $\mathcal{HFS}$ extended with cardinality constraints and linear integer arithmetic ($\mathcal{L}_{|\cdot|}$) [10], which uses SWI-Prolog's CLP(Q) to provide a decision procedure for the theory of linear integer arithmetic [15]; a decision procedure for $\mathcal{L}_{|\cdot|}$ extended with integer intervals [9]; and a decision procedure for restricted quantifiers [11]. In this way, the expressiveness of the {*log*} language is equivalent to the class of finite set relation algebras [2]. On top of that, {*log*} adds direct access to integer algebra, restricted quantifiers and integer intervals which are not (directly) present in relation algebra. In this sense, the language is similar in expressiveness to specification languages such as B and Z [1, 16].

For example, the language of {*log*} allows to express the notion of *array* as a forgram:

$$\mathsf{arr}(A, N) \text{ :- } 0 < N \;\; \& \;\; \mathsf{pfun}(A) \;\; \& \;\; \mathsf{dom}(A, \mathsf{int}(1, N)).$$

where $\mathsf{arr}(A, N)$ states that A is an array of length N; $\mathsf{pfun}(A)$ is a constraint forcing A to be a function; and $\mathsf{dom}(A, \mathsf{int}(1, N))$ is a constraint stating that the integer interval $\mathsf{int}(1, N)$ is the domain of A. The argument to pfun is a set. Then, in {*log*} functions are sets of ordered pairs which implies that equality on functions is *extensional*, and not intensional as in functional programming. In turn arr and foreach play an important role when modeling imperative programs as forgrams. For instance, the forgram below computes the minimum (M) of an array (A of length N) by stating conditions over the state trace (T) of the canonical implementation of minimum of an array:[3]

$\mathsf{minarr}(A, N, M)$:-	[$\mathsf{arr}(A, N)$ is assumed]
$\mathsf{arr}(T, N)$ &	[a trace of length N is necessary to compute the minimum of A]
$\mathsf{get}(A, 1, X_1)$ &	[{*log*} implementation of the get operation over arrays]
$T = \{[1, X_1], [N, M] \sqcup T_1\}$ &	[set the first and last states of the trace]
$\mathsf{foreach}(I \text{ in } \mathsf{int}(2, N),$	[{*log*} specification of the loop to search for the minimum]
Let $[M_i, X_i]$	[two existential variables (per iteration) are introduced]
Be $\mathsf{get}(T, I-1, M_i)$ & $\mathsf{get}(A, I, X_i)$	[M_i and X_i are bound to specific values]
In If $X_i < M_i$ Then $[I, X_i]$ in T Else $[I, M_i] \in T$	[the minimum is updated or not]
).	

That is, minarr is a logic representation of the imperative canonical algorithm computing the minimum of an array. Note how each state of the trace is specified in the If statement: if the I-th element of A is less than the accumulated minimum (i.e. M_i) then the new state in T must be $[I, X_i]$; otherwise, the minimum is not updated ($[I, M_i] \in T$). In this way, the structure of minarr somewhat resembles that of the corresponding imperative program but {*log*} processes it as any other forgram.

The decision procedures mentioned above are integrated into a single solver, implemented in Prolog, which constitutes the core of {*log*}. Several in-depth empirical evaluations provide evidence that {*log*} is able to solve non-trivial problems [4, 6, 12]; in particular as an automated verifier of security properties [5, 8, 3] and for Z and B specifications [8, 7]. In all these empirical evaluations {*log*} is mainly used as an automated verifier. As such, its performance is acceptable in terms of the number of verification conditions that it is able to discharge in a reasonable time. For example, concerning the decision procedure for $\mathcal{L}_{|\cdot|}$, {*log*} performs no worse than state-of-the-art SMT solvers and better than special-purpose algorithms [10, Section 7.3].

[3] This is *not* {*log*} syntax; the forgram is written in {*log*} with a less readable syntax. Adding this syntactic sugar to {*log*} is in our TODO list. The essence and structure of the real forgram are basically the same.

Internally, {*log*} is a rewriting system composed of more than a hundred of rewrite rules. These rules implement low-level properties of set theory. One of the cornerstones of the system is the implementation of *set unification* [14] which determines when two finite sets are equal:

$$\{a \sqcup A\} = \{b \sqcup B\} \rightarrow$$
$$(a = b \wedge A = B) \vee (a = b \wedge \{a \sqcup A\} = B) \vee (a = b \wedge A = \{b \sqcup B\}) \vee (A = \{b \sqcup N\} \wedge B = \{a \sqcup N\})$$

where N is a new set variable. Clearly, this rule involves exploration of four disjunctive alternatives which, in general, implies a heavy use of backtracking. Set unification is necessary to express and to (automatically) reason about specifications common in B and Z such as:

$$A' = A \cup \{x\} \tag{Z}$$
$$A := A \cup \{x\} \tag{B}$$

which in {*log*} are written as $A\_ = \{x \sqcup A\}$, where $A\_$ is interpreted as the value of A in the next state.

However, many times such "deep" equality testing is not necessary. For example, checking whether or not all the elements of a set verify a given property does not require, in general, set equality. This leads us to other of the main rewrite rules implemented in {*log*}[4]:

$$x \text{ in } \{y \sqcup A\} \rightarrow x = y \vee x \text{ in } A \tag{4}$$
$$\{x \sqcup A\} \subseteq B \rightarrow x \in B \wedge A \subseteq B \tag{5}$$

As can be seen, rules such as these make a lighter use of backtracking and, more importantly, they tend to be closer to the optimal computational complexity required for those operations. Testing if an element belongs to a set requires a linear search on the set; whereas testing if all the elements of a set belong to another set, requires a quadratic search. This is so regardless of sets being basic types or implemented by means of lists or multisets. Nevertheless, rule (4) could have been defined to make it more efficient by stopping the search once $x = y$ succeeds:

$$x \text{ in } \{y \sqcup A\} \rightarrow x = y! \vee x \text{ in } A \tag{6}$$

where $G!$ is the $\{log\}$ notation to state that we are interested in just the first solution of goal G. However, as y may be a variable and A may contain other variables as elements, this turns the solver incomplete making it unsuitable as an automated theorem prover. {*log*} would no longer be able to return a finite representation of all the possible solutions of the input formula. In this way, {*log*} is able to compute at a symbolic level not present in, for instance, functional programming. This capacity, on the other hand, pays the price of reduced efficiency.

When {*log*} terminates on some formula the result is a simplified, equivalent formula. An example is the answer to formula (2). When the returned formula is not false, it is a disjunction of formulas in *solved form*—e.g., the two solutions of (1). A formula in solved form is guaranteed to be satisfiable—e.g., the solution shown for (3). The disjunction of formulas in solved form is equivalent to the input formula.

The rewriting system is integrated with syntactic unification and CLP(Q) thus generating a truly constraint solving algorithm. As an example, at some point, (1) is rewritten into:

$$M \text{ in } \{12, 3, Y, 8\} \;\&\; M \leq 12 \;\&\; M \leq 3 \;\&\; M \leq Y \;\&\; M \leq 8$$

Then, rule (4) is applied making M to be bound to each element in $\{12, 3, Y, 8\}$, thus creating a series of purely integer formulas that are solved by CLP(Q). For example:

$$M = 12 \rightarrow 12 \leq 12 \;\&\; 12 \leq 3 \;\&\; 12 \leq Y \;\&\; 12 \leq 8 \rightarrow \text{false} \qquad [12 \leq 3]$$
$$M = Y \rightarrow Y \leq 12 \;\&\; Y \leq 3 \;\&\; Y \leq Y \;\&\; Y \leq 8 \rightarrow Y \leq 3$$

[4]Rules for special cases are not shown for brevity.

References

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] Hajnal Andréka, Steven R Givant, and István Németi. *Decision problems for equational theories of relation algebras*, volume 604. American Mathematical Soc., 1997.

[3] Maximiliano Cristiá, Guido De Luca, and Carlos Daniel Luna. An automatically verified prototype of the android permissions system. *CoRR*, abs/2209.10278, 2022. Under consideration in Journal of Automated Reasoning.

[4] Maximiliano Cristiá and Gianfranco Rossi. Solving quantifier-free first-order constraints over finite sets and binary relations. *J. Autom. Reason.*, 64(2):295–330, 2020.

[5] Maximiliano Cristiá and Gianfranco Rossi. Automated proof of Bell-LaPadula security properties. *J. Autom. Reason.*, 65(4):463–478, 2021.

[6] Maximiliano Cristiá and Gianfranco Rossi. Automated reasoning with restricted intensional sets. *J. Autom. Reason.*, 65(6):809–890, 2021.

[7] Maximiliano Cristiá and Gianfranco Rossi. An automatically verified prototype of a landing gear system. *CoRR*, abs/2112.15147, 2021.

[8] Maximiliano Cristiá and Gianfranco Rossi. An automatically verified prototype of the Tokeneer ID station specification. *J. Autom. Reason.*, 65(8):1125–1151, 2021.

[9] Maximiliano Cristiá and Gianfranco Rossi. A decision procedure for a theory of finite sets with finite integer intervals. *CoRR*, abs/2105.03005, 2021.

[10] Maximiliano Cristiá and Gianfranco Rossi. Integrating cardinality constraints into constraint logic programming with sets. *Theory and Practice of Logic Programming*, pages 1–33, 2021.

[11] 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.

[12] Maximiliano Cristiá, Gianfranco Rossi, and Claudia S. Frydman. $\{log\}$ as a test case generator for the Test Template Framework. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *SEFM*, volume 8137 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2013.

[13] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.*, 22(5):861–931, 2000.

[14] Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Set unification. *Theory Pract. Log. Program.*, 6(6):645–701, 2006.

[15] Christian Holzbaur, Francisco Menezes, and Pedro Barahona. Defeasibility in CLP(Q) through generalized slack variables. In Eugene C. Freuder, editor, *CP*, volume 1118 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 1996.

[16] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.