Rend. Istit. Mat. Univ. Trieste
Vol. 53 (2021), Art. No. 23, pp. 1–24 (electronic)
DOI: 10.13137/2464-8728/33309

{*log*}: Set formulas as programs

Maximiliano Cristiá and Gianfranco Rossi

Abstract. {*log*} *is a programming language at the intersection of Constraint Logic Programming, set programming and declarative programming. But* {*log*} *is also a satisfiability solver for a theory of finite sets and finite binary relations. With* {*log*} *programmers can write abstract programs using all the power of set theory and binary relations. These programs are not very efficient but they are very close to specifications. Then, their correctness is more evident. Furthermore,* {*log*} *programs are also set formulas. Hence, programmers can use* {*log*} *again to automatically prove their programs verify non trivial properties. In this paper we show this development methodology by means of several examples.*

Keywords: set theory, declarative programming, set programming, formal verification, {*log*}.
MS Classification 2020: 68N17, 68V15, 03B70, 03E20, 03E75.

1. Introduction

In 1999 Apt and Bezem [2] proposed a programming paradigm based on the concept of *formulas as programs* as an alternative approach to *formulas as types* or *proofs as programs* [3, 7, 33]. In the latter approach, formal proofs of properties about the correct behavior of a program contain a Lambda calculus term (i.e., a program) which is a correct implementation of that behavior. Hence, by proving properties concerning the behavior of a program one gets in addition correct programs for free. In the 'formulas as programs' approach, the formula (specification) is itself a program. No formal proof is needed to get a program, but the specification might not verify some desired properties making the program faulty. Clearly, both approaches have their advantages and disadvantages. For example, in the 'proofs as programs' approach one have the first version of the program after performing a formal proof of some property and extracting the Lambda term (which is not always easy), but this first version is correct by construction. On the other hand, in the 'formulas as programs' approach one quickly have a first version of the program but it can be wrong, although it can be improved by experimenting with it.

Set theory is deemed as a good vehicle to concisely and accurately describe

algorithms and software systems. Formal specification languages such as Z [32], B [29], TLA+ [6] and VDM [26] support this claim. In this paper we show how set-based specifications can be made to fit in the 'formulas as programs' paradigm.

{*log*} is a constraint logic programming (CLP) language which provides the fundamental forms of set designation, along with a number of basic operations for manipulating them, as first-class entities. Various new features have been added to the core part of the language since the initial development of {*log*} [20][1]. Among them, basic facilities for representing and manipulating integer expressions (integrating the CLP(FD) and CLP(Q) solvers), binary relations, partial functions, Cartesian products and restricted intensional sets. But {*log*} is also a *satisfiability solver*. The CLP language and the satisfiability solver are two sides of the same and only system. That is, {*log*} is not the integration of a CLP interpreter with a satisfiability solver; instead, it is based on mathematical and computational models that produce such a tool.

This means that a piece of {*log*} code is both *a program and a formula*. We call this the *program-formula duality*. In this context, 'formula' refers to a *set formula*. That is, a Boolean combination of set constraints. For example, if `p & q` is a {*log*} formula, where & represents logical conjunction, then `q & p` is semantically equivalent to the former meaning that a programmer will get the same from both of them. Therefore, when {*log*} programmers write code they are writing both a program and a formula. In other words, they are writing a program *as a* formula. When seen as a program, programmers can execute it; when seen as a formula, they can *automatically* prove properties true of it. Hence, a {*log*} programmer writes some code and execute it to see how it works. If everything goes right, (s)he can use {*log*} again to automatically prove properties of that program. All with the same and only formal text and with the same and only tool. Once a {*log*} program is shown to verify some property, we can be sure that all of its executions are correct with respect to that property.

However, {*log*} has some limitations. {*log*} programs perform poorly compared with logic, functional or imperative programs. We see {*log*} programs as *functional prototypes* or *executable specifications*. Not every property true of a {*log*} program can be automatically proved with {*log*}. Further, proving some properties may take too much computing time making the process unpractical. The capacity of {*log*} in automatically proving properties depends on the program-formula fitting inside of the decision procedures implemented by the

[1]The first version of {*log*} came to light in the early 90's with the fundamental contribution of Eugenio Omodeo. His enthusiasm and incomparable competence for computable set theory was an essential stimulus in the decision to pursue the idea of combining sets and logic programming, which was already hypothesized in the far-sighted paper by Ron Sigal [31]. Thus, with the substantial help of two young students from the University of Udine, Agostino Dovier and Enrico Pontelli, the first version of {*log*} took shape in 1990.

tool.

In this paper we will show this program-formula duality through some revealing examples. {*log*} can be downloaded here: `http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html`.

2. {*log*}

{*log*} is a publicly available satisfiability solver and a set-based, constraint-based programming language implemented in Prolog [28].

{*log*} implements a decision procedure for the theory of *hereditarily finite sets* ($\mathcal{SET}$), i.e., finitely nested sets that are finite at each level of nesting [21]; a decision procedure for a very expressive fragment of the theory of finite set relation algebras ($\mathcal{BR}$) [11, 12]; a decision procedure for the theory of finite sets with restricted intensional sets ($\mathcal{RIS}$) [10, 14]; a decision procedure for the theory of hereditarily finite sets extended with cardinality constraints ($\mathcal{L}_{|\cdot|}$) [17]; a decision procedure for the latter extended with integer intervals ($\mathcal{L}_{[]}$) [16]; and uses Prolog's CLP(Q) to provide a decision procedure for the theory of integer linear arithmetic [24]. All these procedures are integrated into a single solver, implemented in Prolog, which constitutes the core part of the {*log*} tool. Several in-depth empirical evaluations provide evidence that {*log*} is able to solve non-trivial problems [10, 11, 12, 19]; in particular as an automated verifier of security properties [13, 15].

Figure 1 schematically describes the stack of the first-order theories supported by {*log*}. The fact that a theory T is over a theory S means that T extends S. E.g., CARD extends both LIA and SET. Figure 2 shortly describes the considered theories, showing for each of them the main constant, function and predicate symbols. The precise definition of the first-order logic languages on which the theories are based on are given in Appendix A.

The integrated constraint language offered by {*log*} is a quantifier-free first-order predicate language with terms of two sorts: terms designating sets and terms designating ur-elements. Terms of either sort are allowed to enter in the formation of *set terms* (in this sense, the designated sets are hybrid), no nesting restrictions being enforced (in particular, membership chains of any finite length can be modeled).

Set terms in {*log*} can be of the following forms:

- A variable is a set term; variable names start with an uppercase letter.
- {} is the term interpreted as the empty set.
- $\{x/A\}$ is called *extensional set* and is interpreted as $\{x\} \cup A$; A must be a set term, x can be any term accepted by {*log*} (basically, any Prolog

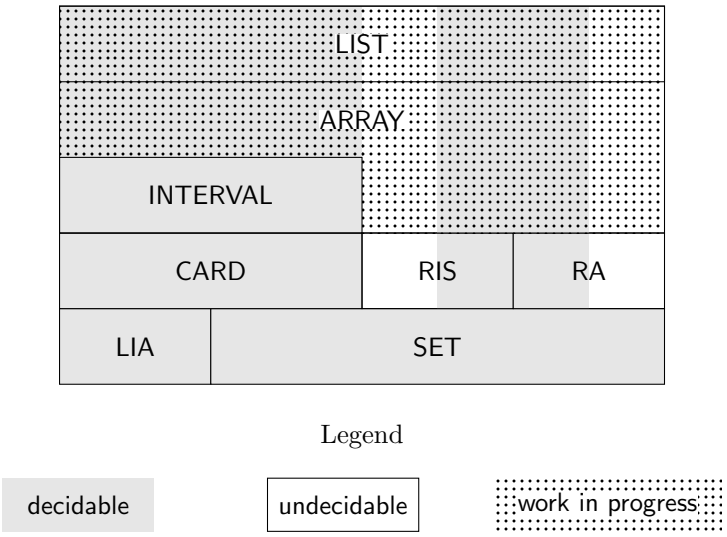

Figure 1: The stack of theories dealt with by $\{log\}$

uninterpreted term, integers, ordered pairs, other set terms, etc.).[2]

As a notational convention, set terms of the form $\{t_1/\{t_2\,/\cdots\{t_n/t\}\cdots\}\}$ are abbreviated as $\{t_1, t_2, \ldots, t_n/t\}$, while $\{t_1/\{t_2\,/\,\cdots\{t_n/\{\}\}\cdots\}\}$ is abbreviated as $\{t_1, t_2, \ldots, t_n\}$.

- $\mathsf{ris}(X \mathsf{\ in\ } A, \phi)$ is called *restricted intensional set* (RIS) and is interpreted as $\{x : x \in A \wedge \phi\}$ where ϕ is any $\{log\}$ formula, A must be a set term, and X is a bound variable local to the RIS. Actually, RIS have a more complex and expressive structure [10, 14].
- $\mathsf{cp}(A, B)$ is interpreted as $A \times B$, i.e., the Cartesian product between A and B.
- $\mathsf{int}(A, B)$ is interpreted as $\{x \in \mathbb{Z} \mid A \leq x \leq B\}$.

Set terms can be combined in several ways: binary relations are hereditarily finite sets whose elements are ordered pairs and so set operators can take binary relations as arguments; RIS and integer intervals can be passed as arguments to set operators and freely combined with extensional sets. $\{log\}$ is an untyped

[2]Note that $\{\_/\_\}$ is the concrete syntax for the (abstract) set term $\{\_ \sqcup \_\}$ of Figure 2.

- LIA: Linear Integer Arithmetic (i.e., the theory that allows inequalities over sums of constant multiples of variables).
  Symbols: $\langle \mathsf{Z}, +, *, -, =, \leq \rangle$, where Z is the set of integer constants.
  Decidable, see for instance [5].

- SET: Hereditarily finite hybrid untyped extensional sets.
  Symbols: $\langle \mathsf{U}, \{\}, \{\cdot \sqcup \cdot\}, =, \neq, \in, \notin, \cup, \| \rangle$, where U is the set of urelements, i.e., non-set objects that are used as set elements, and $\{\cdot \sqcup \cdot\}$ is a binary function symbol which serves as the extensional set constructor.
  Decidable, see [21].

- CARD: Hereditarily finite hybrid untyped extensional sets with cardinality.
  Symbols: $\langle \mathsf{Z}, +, *, -, =, \leq \rangle$, $\langle \mathsf{U}, \{\}, \{\cdot \sqcup \cdot\}, =, \neq, \in, \notin, \cup, \|, |\cdot| \rangle$.
  Decidable, see [34].

- RIS: Hereditarily finite hybrid untyped extensional and intensional sets.
  Symbols: $\langle \mathsf{U}, \{\}, \{\cdot \sqcup \cdot\}, \{\cdot \mid \cdot \bullet \cdot\}, =, \neq, \in, \notin, \cup, \| \rangle$, where $\{\cdot \mid \cdot \bullet \cdot\}$ is a ternary function symbol which serves as the intensional set constructor.
  Decidable fragment, see [14].

- RA: Finite set relation algebras over discrete universe.
  Symbols: $\langle \mathsf{U}, \{\}, \{\cdot \sqcup \cdot\}, (\cdot,\cdot), =, \neq, \in, \notin, \cup, \|, \mathrm{id}, {}_{\mathfrak{g}}, {}^{\smile} \rangle$
  Decidable fragment, see [12].

- INTERVAL: Hereditarily finite hybrid untyped extensional sets and integer intervals with cardinality.
  Symbols: $\langle \mathsf{Z}, +, *, -, =, \leq \rangle$, $\langle \mathsf{U}, \{\}, \{\cdot \sqcup \cdot\}, [\cdot,\cdot], =, \neq, \in, \notin, \cup, \|, |\cdot| \rangle$
  Decidable, see [16].

- ARRAY: Arrays encoded as binary relations.
  Symbols: $\langle \mathsf{Z}, +, *, -, =, \leq \rangle$, $\langle \mathsf{U}, \{\}, \{\cdot \sqcup \cdot\}, \{\cdot \mid \cdot \bullet \cdot\}, [\cdot,\cdot], (\cdot,\cdot), =, \neq, \in, \notin, \cup, \|, |\cdot|, \mathrm{id}, {}_{\mathfrak{g}}, {}^{\smile} \rangle$

- LIST: Lists encoded as binary relations.
  Symbols: $\langle \mathsf{Z}, +, *, -, =, \leq \rangle$, $\langle \mathsf{U}, \{\}, \{\cdot \sqcup \cdot\}, \{\cdot \mid \cdot \bullet \cdot\}, [\cdot,\cdot], (\cdot,\cdot), =, \neq, \in, \notin, \cup, \|, |\cdot|, \mathrm{id}, {}_{\mathfrak{g}}, {}^{\smile} \rangle$

Figure 2: The theories dealt with by {*log*}

formalism; variables are not declared; typing information can be encoded by means of constraints.[3]

Set operators are encoded as atomic predicates, and are dealt with as constraints. For example: $\mathsf{un}(A, B, C)$ is a constraint interpreted as $C = A \cup B$. $\{log\}$ implements a wide range of set and relational operators covering most of those used in Z. For instance, in is a constraint interpreted as set membership (i.e., $\in$); $=$ is set equality; $\mathsf{pfun}(F)$ constrains F to be a (partial) function; $\mathsf{dom}(F, D)$ corresponds to $\operatorname{dom} F = D$; $\mathsf{subset}(A, B)$ corresponds to $A \subseteq B$; $\mathsf{comp}(R, S, T)$ is interpreted as $T = R \fatsemi S$ (i.e., relational composition); and $\mathsf{apply}(F, X, Y)$ is equivalent to $\mathsf{pfun}(F)$ & $[X, Y]\,\mathsf{in}\,F$.

A number of other set, relational and integer operators (in the form of predicates) are defined as $\{log\}$ formulas, thus making it simpler for the user to write complex formulas. Dovier et al. [21] proved that the collection of predicate symbols $\{=, \neq, \in, \notin, \cup, \|\}$ is sufficient to define constraints implementing the set operators $\cap$, $\subseteq$ and $\setminus$. This result has been extended to binary relations [12] by showing that adding to the previous collection the predicate symbols $\{\mathrm{id}, \fatsemi, ^\smile\}$ is sufficient to define constraints for most of the classical relational operators, such as dom, ran, $\lhd$, $\rhd$, etc.. Similarly, $\{=, \neq, \leq\}$ is sufficient to define $<$, $>$ and $\geq$. We call predicates defined in this way, *derived constraints*.

Remark 2.1. Establishing which predicates can be expressed as derived constraints and which, on the contrary, cannot is a critical issue. Primitive constraints are processed by possibly recursive *ad hoc* rewriting procedures, that allow one to implement a form of universal quantification which is not provided by the language. Conversely, derived constraints are processed by simply replacing them by quantifier-free first order formulas.

Choices about primitive vs. derived constraints can be different. For example, in [9] we use dom and ran in place of inv (relational converse, i.e. $^\smile$) and id (identity relation). However, since the inv predicate for binary relations appears not to be definable in terms of the other primitive predicates, inv has been included as a primitive constraint in later work [12], to enlarge the expressiveness of the constraint language. At the same time, dom and ran are moved out of the primitive constraints, since they turn out to be definable in terms of id, comp and inv, thus reducing the number of primitive constraints.

Proving that the selected collection of primitive constraints is the minimal one, as well as comparing one choice to another in terms of, e.g., expressive power, completeness, effectiveness, and efficiency, is a challenging issue for future work.

Negation in $\{log\}$ is introduced by means of so-called *negated constraints*.

---

[3]Recently, a type system and a type checker have been added to the base language for those users who feel more comfortable with typed formalisms.

For example nun(A, B, C) is interpreted as $C \neq A \cup B$ and nin corresponds to $\notin$ – in general, a constraint beginning with 'n' identifies a negated constraint. Most of these constraints are defined as derived constraints in terms of the existing primitive constraints; thus their introduction does not really require extending the constraint language. For formulas to fit inside the decision procedures implemented in {*log*}, users must only use this form of negation.

Formulas in {*log*} are built in the usual way by using conjunctions (&) and disjunctions (or) of atomic constraints.

EXAMPLE 2.2. The following are two simple formulas accepted by {*log*}:

```
a in A & a nin B & un(A,B,C) & C = {X / D}.

un(A,B,C) & N + K > 5 & size(C,N) & B neq {}.
```

As concerns constraint solving, the {*log*} solver repeatedly applies specialized rewriting procedures to its input formula Φ and returns either $false$ or a formula in a simplified form which is guaranteed to be satisfiable with respect to the intended interpretation. Each rewriting procedure applies a few nondeterministic rewrite rules which reduce the syntactic complexity of primitive constraints of one kind. At the core of these procedures is set unification [22]. The execution of the solver is iterated until a fixpoint is reached, i.e., the formula is irreducible.

The disjunction of formulas returned by the solver represent all the concrete (or ground) *solutions* of the input formula. Any returned formula is divided into two parts: the first part is a (possibly empty) list of equalities of the form $X = t$, where X is a variable occurring in the input formula and t is a term; and the second part is a (possibly empty) list of primitive constraints.

3. Uses of {*log*}

In this section we show examples on how {*log*} can be used as a programming language (3.1) and as an automated theorem prover (3.2).

3.1. {*log*} as a programming language

{*log*} is primarily a programming language, at the intersection of declarative programming, set programming [30] and constraint programming. Specifically, {*log*} is an instance of the general CLP scheme. As such, {*log*} programs are structured as a finite collection of *clauses*, whose bodies can contain both atomic constraints and user-defined predicates. The following examples show the *formula-program duality* of {*log*} code along with the notion of clause.

EXAMPLE 3.1. If we want a program that updates function F in X with value Y provided X belongs to the domain of F and get an error otherwise, the {*log*}

code can be the following:

$$\begin{array}{l}\mathsf{update}(F, X, Y, F\_, Error) \text{ :-}\\ \quad F = \{[X, V]/F1\} \ \ \& \ \ [X, V] \ \mathsf{nin} \ F1 \ \ \& \ \ F\_ = \{[X, Y]/F1\} \ \ \&\\ \quad Error = ok\\ \quad \mathsf{or}\\ \quad \mathsf{dom}(F, D) \ \ \& \ \ X \ \mathsf{nin} \ D \ \ \& \ \ Error = err.\end{array}$$

That is, update returns the modified F in $F\_$ and the error code in *Error* – think of $F\_$ as the value of F in the next state. As & and or are logical connectives and = is logical equality, the order of the 'instructions' is irrelevant w.r.t. the functional result – although it can have an impact on the performance. Variable $F1$ is an existentially quantified variable representing the 'rest' of F. If the ordered pair $[X, V]$ does not belong to F then the unification between F and $\{[X, V]/F1\}$ will fail thus making update to execute the other branch.

Now we can call update by providing inputs and waiting for outputs:

$$\mathsf{update}(\{[setlog, 5], [hello, earth], [tokeneer, model]\}, hello, world, G, E).$$

returns:

$$G = \{[hello, world], [setlog, 5], [tokeneer, model]\}, E = ok$$

As a programming language, $\{log\}$ can be used to implement set-based specifications (e.g., Z specifications). As a matter of fact, many of such specifications can be easily translated into $\{log\}$ (see [18]). This means that $\{log\}$ can serve as a programming language in which a *prototype* of a set-based specification can be easily implemented. In a sense, the $\{log\}$ implementation of a set-based specification can be seen as an *executable specification*.

REMARK 3.2. A $\{log\}$ implementation of a set-based specification is easy to get but usually it will not meet the typical performance requirements demanded by users. Hence, we see a $\{log\}$ implementation of a set-based specification more as a *prototype* than as a final program. On the other hand, given the similarities between a specification and the corresponding $\{log\}$ program, it's reasonable to think that the prototype is a *correct* implementation of the specification[4].

For example, in a set-based specification a table (in a database) with a primary key is usually modeled as a partial function, $t : X \nrightarrow Y$. Furthermore, one may specify the update of row r with data d by means of the *oplus* or *override* ($\oplus$) operator: $t' = t \oplus \{r \mapsto d\}$. All this can be easily and naturally translated into $\{log\}$. t is translated as variable T constrained to verify $\mathsf{pfun}(T)$ and the update specification is translated as $\mathsf{oplus}(T, \{[R, D]\}, T\_)$.

[4]In fact, the translation process can be automated in many cases.

However, the oplus constraint will perform poorly compared to the `update` command of SQL, given that oplus's implementation comprises the possibility to operate in a purely logical manner with it (e.g., it allows to compute $\mathsf{oplus}(T, \{[a, D]\}, \{[a, 1], [b, 3]\})$ while `update` does not).

Then, we can use these prototypes to make an early validation of the requirements. Validating user requirements by means of prototypes entails executing the prototypes together with the users so they can agree or disagree with the behavior of the prototypes. This early validation will detect many errors, ambiguities and incompleteness present in the requirements and possible misunderstandings or misinterpretations generated by the software engineers. Without this validation many of these issues would be detected in later stages of the project thus increasing the project costs. Think that if one of these issues is detected once the product has been delivered it means to correct the requirements document, the specification, the design, the implementation, the user documentation, etc.

3.2. {*log*} as an automated theorem prover

{*log*} is also a *satisfiability solver*. This means that {*log*} is a program that can decide if formulas of some theory are *satisfiable* or not. In this case the theory is the theory of finite sets and binary relations, combined with linear integer arithmetic.

Being a satisfiability solver, {*log*} can be used as an automated theorem prover. To prove that formula ϕ is a theorem, {*log*} has to be called to prove that $\neg\, \phi$ is unsatisfiable.

EXAMPLE 3.3. We can prove that set union is commutative by asking {*log*} to prove the following is unsatisfiable:

$$\mathsf{un}(A, B, C) \;\&\; \mathsf{un}(B, A, D) \;\&\; C \;\mathsf{neq}\; D.$$

As there are no sets satisfying this formula {*log*} answers no. Note that the formula can also be written with the nun constraint: $\mathsf{un}(A, B, C) \;\&\; \mathsf{nun}(B, A, C)$.

Evaluating properties with {*log*} helps to run correct simulations by checking that the starting state is correctly defined. It also helps to *test* whether or not certain properties are true of the specification or not. However, by exploiting the ability to use {*log*} as a theorem prover, we can *prove* that these properties are true of the specification.

For instance, since update in Example 3.1 is also a formula we can *automatically* prove properties true of it.

EXAMPLE 3.4. If *Error* is equal to *err* then X does not belong to the domain of F. In order to prove this property we need to call {*log*} on its negation:

$$\mathsf{update}(F, X, Y, F\_, err) \;\&\; \mathsf{dom}(F, D) \;\&\; X \;\mathsf{in}\; D.$$

Then, $\{log\}$ answers no because the formula is unsatisfiable. Further, we can prove that $\mathsf{dom}(F, D)\ \&\ X \mathsf{\ nin\ } D$ is equivalent to $\mathsf{comp}(\{[X, X]\}, F, \{\})$, which allows us to refine update into a version not computing the domain of F. In fact, $\mathsf{comp}(\{[X, X]\}, F, \{\})$ is just a linear iteration over all the elements of F. Then, we need to discharge the following proof obligation:

$$\mathsf{dom}(F, D)\ \&\ X \mathsf{\ nin\ } D \Leftrightarrow \mathsf{comp}(\{[X, X]\}, F, \{\})$$

by proving that its negation is unsatisfiable:

$$\mathsf{dom}(F, D)\ \&\ X \mathsf{\ nin\ } D\ \&\ \mathsf{ncomp}(\{[X, X]\}, F, \{\}) \qquad (\Rightarrow)$$
$$\mathsf{comp}(\{[X, X]\}, F, \{\})\ \&\ \mathsf{dom}(F, D)\ \&\ X \mathsf{\ in\ } D \qquad (\Leftarrow)$$

Hence, now we can write update as follows:

$$\begin{array}{l}\mathsf{update}(F, X, Y, F\_, Error) \text{ :-} \\ \quad F = \{[X, V]/F1\}\ \&\ [X, V] \mathsf{\ nin\ } F1\ \&\ F\_ = \{[X, Y]/F1\}\ \& \\ \quad Error = ok \\ \quad \mathsf{or} \\ \quad \mathsf{comp}(\{[X, X]\}, F, \{\})\ \&\ Error = err.\end{array}$$

Furthermore, $\{log\}$ can be used to automatically discharge verification conditions in the form of invariants. Precisely, in order to prove that an operation T preserves the state invariant I we have to discharge the following proof obligation:

$$I \wedge T \Rightarrow I' \qquad (1)$$

If we want to use $\{log\}$ to discharge (1) we have to ask $\{log\}$ to check if the negation of (1) is *unsatisfiable*. In fact, we need to execute the following $\{log\}$ *program*:

$$I \wedge T \wedge \neg I' \qquad (2)$$

because $\neg\ (I \wedge T \Rightarrow I') \equiv \neg\ (\neg\ (I \wedge T) \vee I') \equiv I \wedge T \wedge \neg\ I'$.

Example 3.5. An invariant property of update is that F is a function. Formally, we can prove the following:

$$\mathsf{pfun}(F)\ \&\ \mathsf{update}(F, X, Y, F\_, E) \Rightarrow \mathsf{pfun}(F\_)$$

as always by proving that its negation is unsatisfiable:

$$\mathsf{pfun}(F)\ \&\ \mathsf{update}(F, X, Y, F\_, E)\ \&\ \mathsf{npfun}(F\_) \qquad (3)$$

As these examples show, {*log*} is a programming and proof platform exploiting the program-formula duality within the theory of finite sets and binary relations.

In particular, many Z specifications can be easily translated into {*log*} (see the on-line document [18]). This means that {*log*} can serve as a *programming language* in which *prototypes* of those specifications can be immediately implemented. Then, {*log*} itself can be used to automatically prove that the specifications preserve some state invariants.

4. Dealing with Binary Relations and Partial Functions

The relational fragment of {*log*} is at least as expressive as the class of full set relation algebras on finite sets [11, 12]. In spite of the inherent undecidability of this class of relation algebras, {*log*} is able to automatically reason about practical problems expressed in relational terms.

Example 4.1. The overriding operator present in the Z formal notation is defined as follows:

$$R \oplus S = ((\mathrm{dom}\, S) \lhd R) \cup S$$

where R and S are binary relations and $\lhd$ is domain anti-restriction. Given that the operation that updates a table can be modeled as an overriding operation, $\oplus$ is frequently used in Z specifications.

Overriding is available in {*log*} in the form of the (derived) constraint oplus:

$$\mathsf{oplus}(R, S, T) \Leftrightarrow T = R \oplus S$$

Hence, we can specify in {*log*} the update operation of Example 3.1 as follows:

$$\begin{array}{l}\mathsf{updateOplus}(F, X, Y, F\_, \mathit{Error}) \text{ :-}\\ \quad \mathsf{oplus}(F, \{[X, Y]\}, F\_) \ \ \&\ \ \mathit{Error} = \mathit{ok}\\ \quad \mathsf{or}\\ \quad \mathsf{dom}(F, D) \ \ \&\ \ X \ \mathsf{nin}\ D \ \ \&\ \ \mathit{Error} = \mathit{err}.\end{array}$$

Then we can use {*log*} to prove that update refines updateOplus:

$$\mathsf{update}(F, X, Y, G) \Rightarrow \mathsf{updateOplus}(F, X, Y, G)$$

by proving the negation to be unsatisfiable. In this way we get a more efficient code given that oplus is too powerful when one only wants to update a single point in the relation.

Remark 4.2. Logical negation can be avoided in {*log*} as long as we work with primitive constraints, since for each of them {*log*} implements also its negation.

On the other hand, if the formula to be negated is a compound formula (i.e., a formula formed by conjunction and disjunction of atomic predicates, such as, for instance, updateOplus in the above example), then we must distribute "by hand" the negation all the way down to the atoms at which point we use the negations of the primitive constraints.

Automating the generation of such kind of negated formulas is one of the improvements that are planned as future work.

The decidable fragment of the relational fragment of $\{log\}$ is still very expressive. In fact, for a formula to be outside the decision procedure it must contain an atom such as $\mathsf{comp}(R, S, \{X/R\})$ or $\mathsf{comp}(S, R, \{X/R\})$ or a subformula that in some way hides such atoms, i.e., it must contain a relational composition where one of the operands shares a variable with the result of the composition.[5] For example, if $\mathsf{dom}(\{X/R\}, A)$ & $\mathsf{ran}(R, A)$ is present in the formula, chances are that it will lay outside the decision procedure, since dom and ran constraints are rewritten to formulas based on comp. When a formula lays outside the decision procedure $\{log\}$ will enter an infinite loop. This means that $\{log\}$ gives correct answers, but it might not give an answer.

The absence of comp constraints of the special form mentioned above is only a sufficient condition for termination of $\{log\}$. In fact, not all formulas containing such constraints go into an infinite loop. For example, the formula $\mathsf{comp}(\{[X, Z]/R\}, \{[Z, Y]/S\}, R) \wedge \mathsf{id}(A, R)$, where the first and the third operands share the same variable R, terminates returning a finite number of solutions. Further investigation on the kind of formulas that makes $\{log\}$ to enter an infinite loop is left for future work. For now, we can observe that these patterns seldom occur in practice. Indeed, an extensive empirical evaluation of a $\{log\}$ shows that the solver is able to automatically prove hundreds of theorems of set theory and relation algebra on finite sets, and to automatically find solutions to systems of constraints of the same theories, as well [12].

5. Dealing with Set Cardinalities

Some times it is necessary to reason about the size of data structures and not only about their contents. For example, within the algebra of finite sets one can partition a given set into two disjoint subsets: $C = A \cup B \wedge A \cap B = \emptyset$. But there is no way to state that A and B must be of the same cardinality. In practice, these constraints might appear, for instance, when part of a given data structure must be put into a cache when its size reaches certain threshold. Specifically, cardinality constraints appear in the verification of some distributed algorithms [1, 4] and are at the base of the notions of integer interval, arrays and lists.

---

[5] The technical details are more complex but this is the essence of the problem [14].

{*log*} implements a decision procedure for the algebra of finite sets with cardinality [17]. In this regard {*log*} combines the rewrite rules of the CLP(SET) scheme with a decision algorithm for formulas including cardinality developed by C. Zarba [34]. Zarba proves that a theory of finite sets equipped with the classic set theoretic operators, including cardinality, combined with linear integer constraints is decidable. The {*log*} decision procedure first uses all the power of {*log*} to produce a simplified, equivalent formula that can be passed to Zarba's algorithm which makes a final judgment about its satisfiability, in case it contains cardinality constraints. At implementation level Zarba's algorithm is implemented by integrating the Prolog Boolean SAT solver developed by Howe and King [25] with SWI-Prolog's implementation of the CLP(Q) system [23]. As a result the implementation integrates three Prolog-based systems: Howe and King's SAT solver, CLP(Q) and {*log*}.

Hence, {*log*} can be used to automatically prove verification conditions based on the cardinality operator.

EXAMPLE 5.1. {*log*} has been tested against +250 verification conditions arising during the analysis of distributed algorithms [27]. For instance, it can automatically discharge the following proof obligation.

$$
\begin{array}{l}
\mathsf{size}(U, N) \ \&\ N > 0 \ \&\ N > 3 * T \ \& \\
\mathsf{subset}(F, U) \ \&\ \mathsf{size}(F, M) \ \&\ M \leq T \ \& \\
\mathsf{subset}(Cgs, U) \ \&\ \mathsf{size}(Cgs, K) \ \&\ 2 * K \geq N - T + 1 \ \& \\
\mathsf{subset}(Bgr, U) \ \&\ \mathsf{size}(Bgr, J) \ \&\ 2 * J \geq N + 3 * T + 1 \ \& \\
\mathsf{inters}(Cgs, Bgr, L) \ \&\ \mathsf{size}(L, 0)
\end{array}
$$

As a consequence of the fact that the new decision procedure is still based on set unification, it can deal with set of sets nested at any level. For example, the decision procedure is able to give all the possible solutions for a goal such as $\mathsf{size}(\{\{X\}, \{Y, Z\}\}, N)$, where X, Y, Z and N are variables.

The formulas returned by {*log*} represent all the concrete (or ground) *solutions* of the input formula. If these formulas do not contain any size or integer constraints, then a concrete solution for such formulas is obtained using the empty set for all set variables occurring in them (with the exception of the variables X in atoms of the form $X = t$). Unfortunately, this is no longer true when considering also the size and integer constraints. For example the answer to the following formula:

$$
\begin{array}{l}
\mathsf{size}(A, M) \ \&\ 1 \leq M \ \&\ M \leq 2 \ \&\ \mathsf{size}(B, N) \ \&\ 5 \leq N \ \& \\
\mathsf{subset}(C, B) \ \&\ \mathsf{size}(C, K) \ \&\ 7 \leq K.
\end{array}
$$

is

$$\begin{aligned}&\mathsf{true}\\ &\mathsf{Constraint}: \mathsf{size}(A, M), M \geq 0, 1 \leq M, M \leq 2, \mathsf{size}(B, N), N \geq 0,\\ &\qquad 5 \leq N, \mathsf{subset}(C, B), \mathsf{size}(C, K), K \geq 0, 7 \leq K\end{aligned}$$

That is, $\{log\}$ returns 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.

For some applications such as model-based testing [19] determining the satisfiability of a formula is not enough. A more or less concrete solution is needed. For this reason $\{log\}$ provides a way in which the solver returns formulas for which is always easy to find a solution. We call such a solution a *minimal solution* because the cardinalities of all the set variables in size constraints are the smallest as to satisfy the formula. When $\{log\}$ is executed in the minimal solution mode, the answer to the above goal is a more concrete solution:

$$\begin{aligned}&A = \{\_N8\}, \quad M = 1,\\ &B = \{\_N7, \_N6, \_N5, \_N4, \_N3, \_N2, \_N1\}, \quad N = 7,\\ &C = \{\_N7, \_N6, \_N5, \_N4, \_N3, \_N2, \_N1\}, \quad K = 7\\ &\mathsf{Constraint}: \_N7 \ \mathsf{neq}\ \_N6, \_N7 \ \mathsf{neq}\ \_N5, ..., \_N3 \ \mathsf{neq}\ \_N1, \_N2 \ \mathsf{neq}\ \_N1\end{aligned}$$

This formula is a finite representation of a subset of the possible solutions for the input formula from which it is trivial to get concrete solutions.

6. Restricted Intensional Sets

Intensional sets are widely recognized as a key feature to describe complex problems, possibly leading to more readable and compact programs than those based on conventional data abstractions. As a matter of fact, various specification or modeling languages provide intensional sets as first-class entities.

$\{log\}$ provides a narrower form of intensional sets, called Restricted Intensional Sets (RIS), that are similar to the set comprehensions available in the formal specification language Z. The basic form of a RIS term is:

$$\mathsf{ris}(X \ \mathsf{in}\ A, \phi, u)$$

where A is a set, ϕ is a $\{log\}$ formula, and u is a term containing X. The intuitive semantics of this RIS is "the set of instances of the term $u(X)$ such that X belongs to A and ϕ holds for X", i.e., $\{y : \exists\, x(x \in A \wedge \phi \wedge y = u(x)\}$.

RIS have the restriction that A must be a *finite set*. This fact, along with a few restrictions on variables occurring in ϕ and u, guarantees that the RIS is a finite set, given that it is at most as large as A. It is important to note that, although RIS are guaranteed to denote finite sets, nonetheless, RIS may be not completely specified. In particular, as the domain can be a variable or a partially specified set, RIS are finite but *unbounded*.

{*log*} formulas containing RIS remain decidable if the formulas inside them are decidable[6].

The next example shows the classes of problems RIS are meant to solve.

EXAMPLE 6.1. First, we can use {*log*} with RIS as a *programming language*. We can think in a program outputting the even numbers (E) present in a set of numbers (S) that is the input to the program[7]:

$$E = \mathsf{ris}(X \ \mathsf{in}\ S, 0 \ \mathsf{is}\ x \bmod 2, X) \tag{4}$$

The RIS term can be written more compactly as $\mathsf{ris}(X \ \mathsf{in}\ S, 0 \ \mathsf{is}\ x \bmod 2)$, since in this case its third argument coincides with its control variable (i.e., the first argument). Then if S is bound to $[-2, 2]$, {*log*} will answer $E = \{-2, 0, 2\}$.

Second, we can use {*log*} with RIS as a *solver* for set formulas. For instance, we want {*log*} to find the most general solution for the following formula:

$$\mathsf{ris}(X \ \mathsf{in}\ A, 0 \ \mathsf{is}\ X \bmod 2) = \{-2, 0, 2\}$$

Note that, in a sense, we are asking {*log*} to find the input values that make program (4) to return a given output. In this case the answer will be:

$$S = \{-2, 0, 2/N\}, \mathsf{ris}(X \ \mathsf{in}\ N, 0 \ \mathsf{is}\ X \bmod 2\} = \{\}$$

where N is a new variable (implicitly existentially quantified). Substituting N by {} yields a ground solution (i.e., $S = \{-2, 0, 2\}$).

Third, we can use {*log*} with RIS to *prove* properties of {*log*} formulas. For instance, to prove that $\mathsf{un}(\{x : S \mid x < m\}, \{x : S \mid x > m\}, B) \Rightarrow m \notin B$, we can prove the following {*log*} formula:

$$\mathsf{un}(\mathsf{ris}(X \ \mathsf{in}\ S, X < M), \mathsf{ris}(X \ \mathsf{in}\ S, X > M), B) \wedge M \ \mathsf{in}\ B$$

which is found to be unsatisfiable.

As LIA is decidable, RIS are a convenient mechanism to model and reason about programs dealing with integers.

---

[6] Among others, more technical, restrictions [14].

[7] {*log*} supports standard Prolog arithmetic operators. In particular, it supports the is operator which should be used to force the evaluation of arithmetic expressions.

EXAMPLE 6.2. RIS can be used to get the subset of a set verifying some LIA formula, which cannot be done rather efficiently in a pure algebraic fragment of set theory.

$$\mathsf{ris}(X \text{ in } S, \mathsf{integer}(X)) \qquad [S \cap \mathbb{Z}]$$
$$\mathsf{ris}(X \text{ in } S, M \leq X \ \ \& \ \ X \leq N) \qquad [S \cap [M, N]]$$
$$\mathsf{ris}(X \text{ in } S, 0 \leq X) \qquad [S \cap \mathbb{N}]$$
$$\mathsf{ris}(X \text{ in } S, 0 \leq 3 * X + 2 * Y)$$

The same can be done with binary relations. As an example, $\mathsf{ris}([X, Y] \text{ in } R, X \leq Y)$ represents the binary relation $R \cap (\_ \leq \_)$.

REMARK 6.3. The language of RIS, called $\mathcal{L}_{\mathcal{RIS}}$, is parametric with respect to any first-order theory $\mathcal{X}$ providing at least equality and a decision procedure for $\mathcal{X}$-formulas. In practice, however, many interesting theories are undecidable and only semi-decision procedures exist for them. This is the case, for instance, for the theory RA of sets and binary relations implemented by $\{log\}$. Hence, the condition on the availability of a decision procedure for *all* $\mathcal{X}$-formulas can be often relaxed. Instead, the existence of some algorithm capable of deciding the satisfiability of a significant fragment of $\mathcal{X}$-formulas can be assumed. If such an algorithm exists and the user writes formulas inside the corresponding fragment, all the theoretical results for RIS still apply.

7. Universal Quantification in $\{log\}$

Formulas that $\{log\}$ can deal with are quantifier-free first-order formulas over finite sets and integer linear arithmetic.

However, $\{log\}$ provides also some form of universal quantification by means of RIS. In effect, the introduction of RIS in $\{log\}$ allows for the definition of *restricted universal quantifiers* (RUQ). In general, if A is a set, then a RUQ is a formula of the following form:

$$\forall\, x \in A : \phi$$

It is easy to prove the following:

$$(\forall\, x \in A : \phi) \Leftrightarrow A \subseteq \{x : x \in A \wedge \phi\} \tag{5}$$

Given that $\{x : x \in A \wedge \phi\}$ is the interpretation of $\mathsf{ris}(X \text{ in } A, \phi)$, the r.h.s. of (5) can be expressed as the $\{log\}$ formula:

$$\mathsf{subset}(A, \mathsf{ris}(X \text{ in } A, \phi))$$

for which $\{log\}$ provides the derived constraint foreach thus making RUQ easier to write:

$$\mathsf{foreach}(X \text{ in } A, \phi) \mathrel{\widehat{=}} \mathsf{subset}(A, \mathsf{ris}(X \text{ in } A, \phi)). \tag{6}$$

There is also a more powerful form of foreach:

$$\mathsf{foreach}(X \text{ in } A, \mathsf{V}, \phi, \psi) \mathrel{\widehat{=}} \mathsf{subset}(A, \mathsf{ris}(X \text{ in } A, \mathsf{V}, \phi, \psi)). \tag{7}$$

where V is a vector of existentially quantified variables inside the foreach and ψ is a so-called *functional predicate*. A predicate $\psi(x_1, \ldots, x_{n+1})$ is a functional predicate iff for any given $x_1, \ldots, x_n$ there exists at most one x_{n+1} making ψ true. Functional predicates enjoy a nice property concerning their negation [14], which considerably extends the class of decidable formulas including foreach constraints.

EXAMPLE 7.1. We use foreach to encode and automatically reason about important security properties [13]. The Bell-LaPadula (BLP) security model proposes two security properties for secure operating systems. The simplest is called security condition of which we show a simplified version:

$$\forall(s, o, x) \in b : x = \mathsf{r} \Rightarrow f_2(o) \leq f_1(s) \wedge f_4(o) \subseteq f_3(s) \tag{8}$$

where b is a ternary relation and f_i are functions. In {*log*} we can encode a ternary relation with a binary relation where the second components are ordered pairs. Then $(s, o, x) \in b$ becomes $[S, [O, X]]$ in B. Therefore, (8) is encoded as follows:

$$\begin{array}{l}
\mathsf{seccond}(F1, F2, F3, F4, B) \text{ :-} \\
\quad \mathsf{foreach}([S, [O, X]] \text{ in } B, [No, Co, Ns, Cs], \\
\qquad No \leq Ns \ \ \& \ \ \mathsf{subset}(Co, Cs), \\
\qquad \mathsf{apply}(F1, S, Ns) \ \ \& \ \ \mathsf{apply}(F2, O, No) \ \ \& \\
\qquad \mathsf{apply}(F3, S, Cs) \ \ \& \ \ \mathsf{apply}(F4, O, Co))
\end{array}$$

where apply is a derived constraint which is defined as

$$\mathsf{apply}(F, X, Y) \mathrel{\widehat{=}} [X, Y] \text{ in } F \ \ \& \ \ \mathsf{pfun}(F)\,.$$

Note that all the apply constraints are placed in the last argument given that they are functional predicates.

REMARK 7.2. Example 7.1 brings in the point of when a binary relation can be *applied* to an element of its domain. Usually the condition for function application is, precisely, for the binary relation to be a function. However, there is a weaker condition in which the binary relation is *locally* functional; that is, the binary relation is a function in (at least) one point. Therefore, in {*log*} the user can work also with the following derived constraint:

$$\begin{array}{l}
\mathsf{applyTo}(F, X, Y) \mathrel{\widehat{=}} \\
\quad F = \{[X, Y]/G\} \ \ \& \ \ [X, Y] \text{ nin } G \ \ \& \ \ \mathsf{comp}(\{[X, X]\}, G, \{\}).
\end{array}$$

Using apply as in Example 7.1 implies that seccond checks that $F1$-$F4$ are functions every time it is called. In a real implementation this is not the case because the fact that $F1$-$F4$ are functions would be a pre-condition. Hence, a more realistic specification would use applyTo instead of apply. We can use $\{log\}$ to automatically prove that if F is a function then applyTo refines apply

$$\mathsf{pfun}(F) \ \& \ \mathsf{applyTo}(F, X, Y) \Rightarrow \mathsf{apply}(F, X, Y)$$

This allows to formally replace apply by applyTo in seccond. Moreover, this results in a more efficient implementation as applyTo is linear in the size of F while apply is quadratic. As a matter of fact, discharging all the verification conditions of the BLP model using applyTo is almost 10 times faster than when using apply.

The foreach constraint can be used to model and reason about order. In fact if F is a function with domain and range in $\mathbb{Z}$, then we can use $\{log\}$ to define a predicate stating whether or not F is a strictly increasing injective function (siif).

Example 7.3. The following formula captures the notion of strictly increasing function F:

$$\forall(a, c) \in F, \forall(x, y) \in F : a < x \Rightarrow c < y$$

which is immediately rendered in $\{log\}$ by the following predicate:

$$\begin{array}{l} \mathsf{siif}(F) \text{ :-} \\ \quad \mathsf{pfun}(F) \\ \quad \wedge\ \mathsf{foreach}([A, C] \text{ in } F, \\ \qquad \mathsf{foreach}([X, Y] \text{ in } F, A \geq X \text{ or } C < Y))) \end{array}$$

Although foreach can be defined as a derived constraint based on the constraint un, in $\{log\}$ we introduce a set of specialized rewrite rules to process this specific kind of predicates more efficiently [14]. As a matter of fact, the formula $\mathsf{foreach}(x \in \{t \sqcup A\}, \phi(x))$ can be seen as an iterative program whose iteration variable is x, the range of iteration is $\{t \sqcup A\}$, and the body is ϕ. In fact, the rewrite rule for this formula basically iterates over $\{t \sqcup A\}$ and evaluates ϕ for each element in that set. If one of these elements does not satisfy ϕ then the loop terminates immediately, otherwise it continues until the empty set is found or a variable is found.

8. Finite Integer Intervals

The theory INTERVAL (cf. Figures 1 and 2) deals with hereditarily finite hybrid untyped extensional sets and integer intervals with cardinality. The bounds of

integer intervals can be either integer constants or variables ranging over integer numbers, and as such they can be constrained through integer linear arithmetic constraints.

{*log*} provides a decision procedure of the theory INTERVAL. Integer intervals in {*log*} are represented by terms of the form $\mathsf{int}(A, B)$, where A and B are integer constants or variables, which is interpreted as the close interval $[A, B]$. Interval terms can be manipulated as sets through set constraints (e.g., $\mathsf{int}(0, B) = \{1, 2/S\}$ or $\mathsf{inters}(\mathsf{int}(3, N), \mathsf{int}(10, 20), A)$ & $A \ \mathsf{neq}\ \{\}$). Moreover, interval bounds can be manipulated as integers through integer constraints (e.g., $A = \mathsf{int}(M, N)$ & $N >= M + 3$).

The decision procedure for the theory INTERVAL allows {*log*} to be used to program and automatically reason about problems such as the following.

EXAMPLE 8.1. Consider two workers who are assigned two disjoint sets of tasks from a set of N tasks. If A and B are the sets of tasks already performed by each worker, then we can model the problem as follows.

$\mathsf{init}(A, B)$:-
    $A = \{\}$ & $B = \{\}$.
$\mathsf{addToA}(N, A, B, J, A\_, B)$:-
    $(J \ \mathsf{nin}\ \mathsf{int}(1, N)\ \mathsf{or}\ J \ \mathsf{in}\ A\ \mathsf{or}\ J \ \mathsf{in}\ B)$ & $A\_ = A$
    or
    $J \ \mathsf{in}\ \mathsf{int}(1, N)$ & $J \ \mathsf{nin}\ A$ & $J \ \mathsf{nin}\ B$ & $A\_ = \{J/A\}$.
$\mathsf{finish}(N, A, B, \{\}, \{\})$:-
    $\mathsf{un}(A, B, \mathsf{int}(1, N))$ & $\mathsf{write}('\mathsf{Jobdone}')$.
% Invariant1 :$\mathsf{un}(A, B, C)$ & $\mathsf{subset}(C, \mathsf{int}(1, N))$
% Invariant2 :$\mathsf{disj}(A, B)$

For brevity we do not show addToB which would be symmetric to addToA. Then, we can use {*log*} to run some simulations:

$$\mathsf{init}(A, B)\ \&\ \mathsf{addToA}(10, A, B, 3, A1, B1)\ \&\ \mathsf{addToB}(10, A1, B1, 7, A2, B2).$$

Finally we can use {*log*} to automatically prove the indicated invariants.

If A and B must done $N/2$ tasks each, then we can add a size-based precondition to addToA and addToB, which would be still inside the decision procedures implemented by {*log*}.

EXAMPLE 8.2. Assume some objects are identified with numbers from 1 up. We want to write a condition stating that a certain set of these objects, A, contains objects with consecutive numbers. It can be written with an INTERVAL-based formula:

$$A = \mathsf{int}(M, N)\ \&\ 1 \leq M$$

Moreover, if A does not verify that condition we would like to compute the missing objects from it:

$$A = \mathsf{int}(M, N) \ \ \& \ \ 1 \leq M$$
$$\mathsf{or}$$
$$\mathsf{nint}(A) \ \ \& \ \ \mathsf{min}(A, M) \ \ \& \ \ \mathsf{max}(A, N) \ \ \& \ \ \mathsf{un}(A, \mathit{Miss}, \mathsf{int}(M, N))$$

where nint is a predicate stating that A is not an integer interval, and min and max compute the minimum and maximum of A; all of which can be stated as INTERVAL formulas.

The key idea for obtaining a decision procedure for the theory INTERVAL is extending the set unification algorithm of CLP($\mathcal{SET}$) [21] with the following identity:

$$m \leq n \Rightarrow (A = [m, n] \Leftrightarrow A \subseteq [m, n] \wedge |A| = n - m + 1)$$

In fact, it suffices to be able to deal with constraints of the form $A \subseteq [m, n]$ in a decidable manner to have a decision procedure for integer intervals.

Exploiting extended set unification with intervals $\{log\}$ allows, for instance, to reconstruct integer intervals even out of underspecified sets:

$$\mathsf{un}(\{X, Y, Z\}, \{1, 4\}, \mathsf{int}(M, N))$$

of which some solutions are:

$$X = 0, Y = 2, Z = 3, M = 0, N = 4$$
$$X = 2, Y = 3, Z = 5, M = 1, N = 5$$
$$X = 2, Y = 3, Z = 1, M = 1, N = 4$$

Remark 8.3. In some cases there are a few different ways of writing the same term or formula. For instance, the RIS term $\mathsf{ris}(X \ \mathsf{in} \ S, M \leq X \ \ \& \ \ X \leq N)$ of Example 6.2 can be written as the INTERVAL formula $\mathsf{inters}(S, \mathsf{int}(M, N), A)$, in which case we have $A = \mathsf{ris}(X \ \mathsf{in} \ S, M \leq X \ \ \& \ \ X \leq N)$. Which is the best language construct to express programs and properties depends on, some times, contradictory concepts such as efficiency and readability. At least $\{log\}$ provides a way to go from one construct to another. That is, if a user writes a formula containing the RIS in question, (s)he can substitute the RIS by A if $\mathsf{inters}(S, \mathsf{int}(M, N), A)$ is conjoined to the formula, after using $\{log\}$ to prove the substitution is correct.

It is worth noting that a combination between the subset relation and integer intervals is the key to encode forms of universal quantification in $\{log\}$ by means of a quantifier-free formula, allowing us to preserve decidability (and thus full automation in proofs). The following example illustrates this idea.

EXAMPLE 8.4. If $X, Y \in D$, Y is the successor of X (in D) if the following holds:

$$\neg\ \exists\, Z \in D : X < Z \wedge Z < Y$$

which is equivalent to:

$$\forall\, Z \in D : Z \leq X \vee Y \leq Z \tag{9}$$

In this case we need to quantify over integer numbers. A way to get rid of this universal quantifier (hence, obtaining a quantifier-free {*log*} formula) is to use a combination between the subset relation and integer intervals as follows:

$$\begin{array}{l}\mathsf{succ}(D, X, Y)\ \text{:-}\\ \quad D = \{X, Y/E\}\ \ \&\ \ \mathsf{un}(Inf, Sup, E)\ \ \&\ \ \mathsf{disj}(Inf, Sup)\ \ \&\\ \quad M\ \mathsf{is}\ X - 1\ \ \&\ \ \mathsf{subset}(Inf, \mathsf{int}(\_, M))\ \ \&\\ \quad N\ \mathsf{is}\ Y + 1\ \ \&\ \ \mathsf{subset}(Sup, \mathsf{int}(N, \_)).\end{array}$$

To confirm that succ is indeed an encoding of (9) we can execute some tests:

$$\begin{array}{ll}\mathsf{succ}(\{4,7,1,8,-3\}, 1, Y) \rightarrow Y = 4 & \\ \mathsf{succ}(\{4,7,1,8,-3\}, 2, Y) \rightarrow \mathsf{no} & [\text{by } 2 \notin D]\\ \mathsf{succ}(\{4,7,1,8,-3\}, 8, Y) \rightarrow \mathsf{no} & [\text{by } max(D) = 8]\\ \mathsf{succ}(\{4,7,1,8,-3\}, 4, Y) \rightarrow Y = 7 & \\ \mathsf{succ}(\{4,7,1,8,-3\}, X, 7) \rightarrow X = 4 & (\dagger)\\ \mathsf{succ}(\{4,7,1,8,-3\}, X, Y) \rightarrow X = 4, Y = 7;\ X = 7, Y = 8;\ \ldots & (\ddagger)\end{array}$$

Note that (†) shows that {*log*} does not really distinguish between inputs and outputs; and (‡) shows that {*log*} is able to return all solutions one after the other. Furthermore, to collect stronger evidences that succ is correct we can use {*log*} to automatically prove some properties true of it:

$$\mathsf{succ}(D, X, Y) \Rightarrow (\forall\, Z \in D : Z \leq X \vee Y \leq Z)$$

whose negation is:

$$\mathsf{succ}(D, X, Y)\ \ \&\ \ Z\ \mathsf{in}\ D\ \ \&\ \ X < Z\ \ \&\ \ Z < Y$$

to which {*log*} answers no. And further we can prove:

$$\mathsf{succ}(D, X, Y)\ \ \&\ \ \mathsf{succ}(D, Y, Z) \Rightarrow X < Z$$

whose negation is:

$$\mathsf{succ}(D, X, Y)\ \ \&\ \ \mathsf{succ}(D, Y, Z)\ \ \&\ \ Z \leq X$$

Finally, integer intervals are a key component in the definition of arrays and list as sets; hence, to implement the theories ARRAY and LIST in $\{log\}$ (cf. Figure 1). In particular, if $\mathsf{array}(A, n)$ is a predicate stating that A is an array of length n whose components take values on some universe U, then it can be defined as $array(A, n) \Leftrightarrow A : [1, n] \to \mathsf{U}$, i.e., as a partial function between the integer interval $[1, n]$ and U. Since $\{log\}$ supports a broad class of set relation algebras, including partial functions and the domain operator, then it would be possible to use $\{log\}$ to automatically reason about broad classes of programs with arrays. Lists could be introduced in a similar way.

However, supporting arrays and lists in $\{log\}$ is a line of future research.

9. Concluding Remarks

The CLP language $\{log\}$ provides decision procedures for expressive classes of extensional and intensional hereditarily finite hybrid sets, including binary relations, integer intervals and Cartesian products, extended with cardinality constraints and integer constraints for integer linear arithmetic.

In this paper we have shown how $\{log\}$, with its decision procedures, can be exploited: (i) as a *programming language*, in which a *prototype* of a set-based specification can be immediately implemented; (ii) as a *satisfiability solver* for formulas of the different theories, in particular for formulas representing the implementation of a set-based specification for which $\{log\}$ can be used to *prove* that certain properties are true of the specification. In this paper we have provided evidence for this claim by showing a number of simple working examples written in $\{log\}$.

Besides the possible future work pointed out throughout the paper, we are investigating the possibility to add interactive theorem proving capabilities to $\{log\}$ [8] in order to make it capable of proving properties outside of the implemented decision procedures.

References

[1] F. Alberti, S. Ghilardi, and E. Pagani, *Cardinality constraints for arrays (decidability results and applications)*, Formal Methods Syst. Des. **51** (2017), no. 3, 545–574.

[2] K. R. Apt and M. Bezem, *Formulas as programs*, The Logic Programming Paradigm, Springer, 1999, pp. 75–107.

[3] J. L. Bates and R. L. Constable, *Proofs as programs*, ACM Trans. Program. Lang. Syst. **7** (1985), no. 1, 113–136.

[4] I. Berkovits, M. Lazic, G. Losa, O. Padon, and S. Shoham, *Verification of threshold-based distributed algorithms by decomposition to decidable logics*, Computer Aided Verification, CAV (I. Dillig and S. Tasiran, eds.), Lecture Notes Comput. Sci., vol. 11562, Springer, 2019, pp. 245–266.

[5] M. Bromberger, T. Sturm, and C. Weidenbach, *A complete and terminating approach to linear integer solving*, J. Symb. Comput. **100** (2020), 102–136.
[6] M. J. Butler, A. Raschke, T. S. Hoang, and K. Reichl, *Abstract state machines, alloy, B, TLA, VDM, and Z*, Lecture Notes in Comput. Sci., vol. 10817, Springer, 2018.
[7] T. Coquand and G. P. Huet, *The calculus of constructions*, Inform. and Comput. **76** (1988), no. 2/3, 95–120.
[8] M. Cristiá, R. D. Katz, and G. Rossi, *Proof automation in the theory of finite sets and finite set relation algebra*, Comput. J. (2021), bxab030.
[9] M. Cristiá and G. Rossi, *A decision procedure for sets, binary relations and partial functions*, Computer Aided Verification, CAV 2016 (S. Chaudhuri and A. Farzan, eds.), Lecture Notes in Comput. Sci., vol. 9779, Springer, 2016, pp. 179–198.
[10] M. Cristiá and G. Rossi, *A decision procedure for restricted intensional sets*, Automated Deduction - CADE 26 (L. de Moura, ed.), Lecture Notes in Comput. Sci., vol. 10395, Springer, 2017, pp. 185–201.
[11] M. Cristiá and G. Rossi, *A set solver for finite set relation algebra*, Relational and Algebraic Methods in Computer Sciencs, RAMiCS 2018 (J. Desharnais, W. Guttmann, and S. Joosten, eds.), Lecture Notes in Comput. Sci., vol. 11194, Springer, 2018, pp. 333–349.
[12] M. Cristiá and G. Rossi, *Solving quantifier-free first-order constraints over finite sets and binary relations*, J. Autom. Reason. **64** (2020), no. 2, 295–330.
[13] M. Cristiá and G. Rossi, *Automated proof of Bell-LaPadula security properties*, J. Autom. Reason. **65** (2021), no. 4, 463–478.
[14] M. Cristiá and G. Rossi, *Automated reasoning with restricted intensional sets*, J. Autom. Reason. **65** (2021), no. 6, 809–890.
[15] M. Cristiá and G. Rossi, *An automatically verified prototype of the Tokeneer ID station specification*, J. Autom. Reason. **65** (2021), no. 8, 1125–1151.
[16] M. Cristiá and G. Rossi, *A decision procedure for a theory of finite sets with finite integer intervals*, (2021), CoRR abs/2105.03005, under consideration in Theoretical Computer Science.
[17] M. Cristiá and G. Rossi, *Integrating cardinality constraints into constraint logic programming with sets*, Theory Pract. Log. Program. (2021), 1–33.
[18] M. Cristiá and G. Rossi, {*log*}*: Applications to software specification, prototyping and verification*, (2021), CoRR abs/2103.14933.
[19] M. Cristiá, G. Rossi, and C. S. Frydman, {*log*} *as a test case generator for the Test Template Framework*, Software Engineering and Formal Methods, SEFM 2013 (R. M. Hierons, M. G. Merayo, and M. Bravetti, eds.), Lecture Notes in Comput. Sci., vol. 8137, Springer, 2013, pp. 229–243.
[20] A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi, *A language for programming in logic with finite sets*, J. Log. Program. **28** (1996), no. 1, 1–44.
[21] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi, *Sets and constraint logic programming*, ACM Trans. Program. Lang. Syst. **22** (2000), no. 5, 861–931.
[22] A. Dovier, E. Pontelli, and G. Rossi, *Set unification*, Theory Pract. Log. Program. **6** (2006), no. 6, 645–701.
[23] C. Holzbaur, *OFAI CLP(Q,R) manual*, Tech. report, edition 1.3.3. Technical

Report TR-95-09, Austrian Research Institute for Artificial Intelligence, 1995.
[24] C. Holzbaur, F. Menezes, and P. Barahona, *Defeasibility in CLP(Q) through generalized slack variables*, Principles and Practice of Constraint Programming, CP96 (E. C. Freuder, ed.), Lecture Notes in Comput. Sci., vol. 1118, Springer, 1996, pp. 209–223.
[25] J. M. Howe and A. King, *A pearl on SAT and SMT solving in Prolog*, Theoret. Comput. Sci. **435** (2012), 43–55.
[26] C. B. Jones, *Systematic software development using VDM*, 2nd ed., Prentice Hall Ser. Comput. Sci., Prentice Hall, 1991.
[27] R. Piskac, *Efficient automated reasoning about sets and multisets with cardinality constraints*, Automated Reasoning, IJCAR 2020 (N. Peltier and V. Sofronie-Stokkermans, eds.), Lecture Notes in Comput. Sci., vol. 12166, Springer, 2020, pp. 3–10.
[28] G. Rossi, {*log*}, `http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html`, 2008, Last access 2021.
[29] S. Schneider, *The B-method: An introduction*, Cornerstones of computing, Palgrave, 2001.
[30] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg, *Programming with sets - an introduction to SETL*, Texts Monogr. Comput. Sci., Springer, 1986.
[31] R. Sigal, *Desiderata for logic programming with sets*, 4th National Conference on Logic Programming GULP89 (Bologna) (P. Mello, ed.), June 1989.
[32] J. M. Spivey, *The Z notation: a reference manual*, Prentice Hall, 1992.
[33] S. Thompson, *Type theory and functional programming*, Internat. Comput. Sci. Ser., Addison-Wesley, 1991.
[34] C. G. Zarba, *Combining sets with integers*, Frontiers of Combining Systems, FroCoS 2002 (Armando A., ed.), Lecture Notes in Comput. Sci., vol. 2309, Springer, 2002, pp. 103–116.

Authors' addresses:

Maximiliano Cristiá
Departamento de Ciencias de la Computación
Universidad Nacional de Rosario
Pellegrini 250, (2000) Rosario, Argentina
E-mail: `cristia@cifasis-conicet.gov.ar`

Gianfranco Rossi
Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Università di Parma
Parco Area delle Scienze, 53/A, 43100 Parma, Italia
E-mail: `gianfranco.rossi@unipr.it`

Received May 10, 2021
Revised September 6, 2021
Accepted September 7, 2021