Mathematical Programming: Turing completeness and applications to software analysis

Leo Liberti[1,2], Fabrizio Marinelli[3]

[1] IBM "T.J. Watson" Research Center, Yorktown Heights, USA. Email:leoliberti@us.ibm.com
[2] LIX, Ecole Polytechnique, 91128 Palaiseau, France. Email:liberti@lix.polytechnique.fr
[3] DII, Università Politecnica delle Marche, Ancona, Italy. Email:fabrizio.marinelli@univpm.it

October 7, 2013

Abstract

Mathematical Programming is Turing complete, and can be used as a general-purpose declarative language. We present a new constructive proof of this fact, and showcase its usefulness by discussing an application to finding the hardest input of any given program running on a Minsky Register Machine. We also discuss an application of Mathematical Programming to software verification obtained by relaxing one of the properties of Turing complete languages.

1 Introduction

Since the introduction of Linear Programming (LP), the Operations Research community has been using LP and its various generalizations — e.g. Mixed-Integer Linear Programming (MILP), Nonlinear Programming (NLP), Mixed-Integer Nonlinear Programming (MINLP) and others — as a *language* to define (and solve) all kinds of optimization problems. Some definitions of Mathematical Programming (MP) as a formal language can be found in the literature, e.g. the AMPL grammar [11, Appendix A], the GNU MathProg language [24] (whose grammar is written in open-source pure C), and the GAMS language [4], to name but a few. As with any programming language, it is natural to inquire about the limits of its expressive power: can any algorithm be cast in the MP language?

It is well known that the answer is yes (see Sect. 2 for details). We give a new, constructive proof of this fact by means of a reduction from a certain universal computer (i.e. a computer which can simulate any other computer) to a MINLP. The interesting feature of our proof is that it provides a practical tool for answering an interesting question about programs, e.g., what is the input of given size yielding the longest running time of a given code? Moreover, we look at another application of MP to software verification (already briefly discussed in [23, 14]), which relaxes one of the "pillars" of universal languages, i.e. the juxtaposition of commands [3], to compute program invariants.

The rest of this paper is organized as follows. Preliminary notions are defined and discussed in Sect. 2. We prove in Sect. 3 that MP is Turing complete, show the application to finding the hardest input for a given program in Sect. 4, and finally, in Sect. 5, we discuss another application of MP the problem of proving the absence of certain types of bugs from computer programs.

2 Preliminary notions

A Universal Turing Machine (UTM) is a Turing Machine (TM) which can simulate any other TM on arbitrary input [29, 26]. Marvin Minsky described a UTM, close to today's computers, now called *Minsky's register machine* [19, Ch. 4]. This consists in a countably infinite number of registers, each of which can contain a natural number, a finite set of states, and two types of instructions: (1) add 1 to a

given register and change to a given state; (2) test whether a given register is 0, if so change to a given state, else subtract one and change to a given state. Minsky's Register Machine (MRM) is a model of an extremely simple CPU that can carry out two parametrizable instructions in Random Access Memory (RAM).

2.1 Minsky's Register Machine

The MRM is a UTM with infinitely many registers, each of which can hold an arbitrary natural number, and two types of parametrizable instructions:

1. add 1 to a register then switch to a new state;
2. test whether a register holds a positive number: if so subtract 1 then switch to a new state; else switch to a new (different) state.

States are used to index instructions of the MRM, so that "change to a given state" effectively means, in today's terminology, "jump to a given instruction".

2.1 Definition
A MRM is a quadruplet (R, N, S, c) *where:*

1. $R = (R_1, R_2, \ldots)$ *is an infinite sequence of registers each of which can hold an arbitrary natural number;*
2. $N = \{0, \ldots, n\}$*, where* $n \in \mathbb{N}$*, and* $N_+ = N \smallsetminus \{0\}$ *is the set of states;*
3. $S : N_+ \to \mathbb{N} \times \{0, 1\} \times N \times N$*, is a program, and* $S_i = (j, b, k, \ell)$ *is an instruction of type* $b \in \{0, 1\}$ *for all* $i \in N_+$*, whose meaning is explained below;*
4. $c \in N$ *holds the current instruction index.*

The program S *works as follows. For an instruction* $i \in N_+$*, let* $S_i = (j, b, k, \ell)$*:*

1. *if* $b = 0$ *then* $R_j \leftarrow R_j + 1$ *and* $c \leftarrow k$*;*
2. *if* $b = 1$ *and* $R_j = 0$ *then* $c \leftarrow \ell$*;*
3. *if* $b = 1$ *and* $R_j > 0$ *then* $R_j \leftarrow R_j - 1$ *and* $c \leftarrow k$*.*

If $c = 0$ *the execution stops. We remark that if* $b = 0$ *then* ℓ *is unused.*

2.2 Example
The example of MRM program in Fig. 1 (found in [19]) uses 3 registers and 8 states (including the stop state called S_0*). Its purpose is to add to* R_1 *twice the content found initially in* R_2*, using* R_3 *as temporary storage space.*

2.2 Programming languages and interpreters

Let L be a programming language for the UTM U; as is well known, L need not be aware of the physical characteristics of U. A special software $\mathscr{I}$ (which *is* aware of those characteristics), called *interpreter for* L *on* U, is employed to translate any program P (taking c as input, producing x as output, and written in L), into a sequence of instructions that can be performed by U. In other words, $\mathscr{I}$ takes as input P

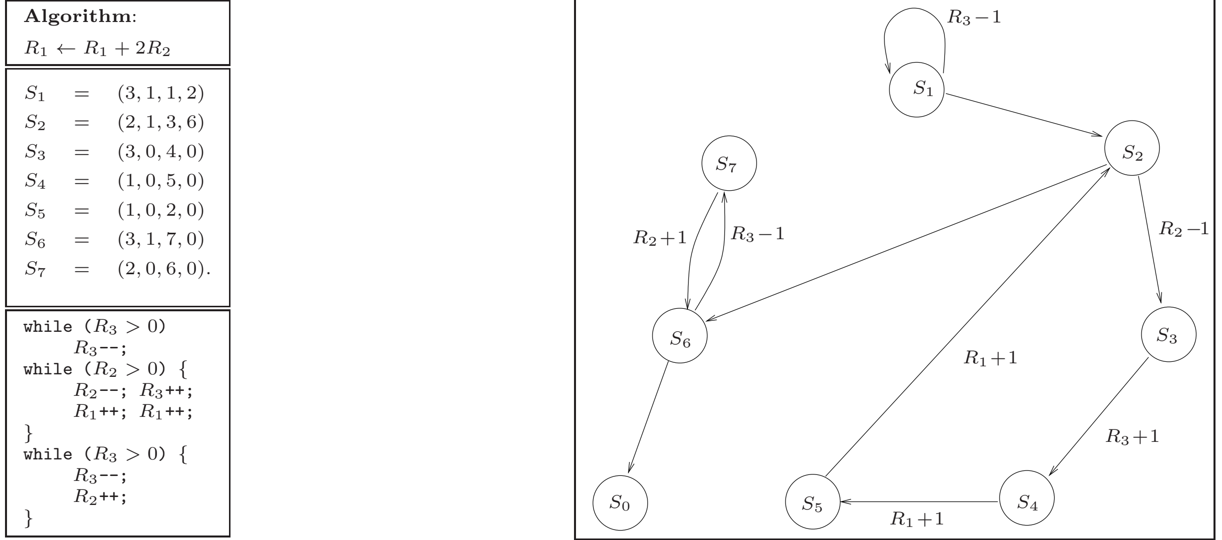

Figure 1: The MRM example from [19]: the meaning (top left), in MRM "language" (middle left), in C-like pseudocode (bottom left), and as a finite state automaton (right).

and c and tells U how to perform the computation whose output is x. The interpreters for imperative programming languages, such as Fortran, C/C++, Java and so on, are usually simple and efficient: each high-level instruction in L is translated into the machine language employed by the CPU being used to perform the computation.

Declarative programming languages, such as MP, Prolog [27] or Constraint Programming (CP) [1], do not prescribe sequences of instructions, but rather define computable functions via a set of conditions, or constraints: for example, a simple linear program $P(c_1, c_2)$ might ask for a vector (x_1, x_2) maximizing the linear form $c_1x_1 + c_2x_2$ subject to the constraints $x_1 + 2x_2 \leq 1$, $x_1 \geq 0$, $x_2 \geq 0$. Interpreters for declarative programming languages are usually rather complicated, and may be very inefficient. Typical conditions expressed in MP, CP or Prolog programs would generally look like "find $x \in X$ such that $f(x) = 0$", where X is a decidable set and f is a computable function. Since the interpreter must be able to interpret any program in the language, it might have to apply brute force to test all x in X for the property $f(x) = 0$. Interpreters for declarative languages are often based on tree-like search methods, such as Branch-and-Bound (BB).

2.3 Turing completeness

If a programming language can be used to program a UTM via an interpreter, then it is *Turing complete*. The fact that MP is Turing complete is a simple corollary to the existence of Universal Diophantine Equations (UDE) [20], insofar as a Diophantine equation is an Integer Polynomial Programming (IPP) feasibility problem, and IPP is a proper subclass of MP. UDEs are polynomial equations $p_i^x(y_1 \ldots, y_n)$, parametrized by x, i, such that

$$\forall i \in \mathbb{N}, x \in X_i \Leftrightarrow \exists y \in \mathbb{Z}^n \ (p_i^x(y) = 0),$$

where $X_1, X_2, \ldots$ is an enumeration of all recursively enumerable sets (it is known that such an enumeration is possible). Efforts to find UDEs of ever smaller sizes (in terms of the number n of variables and the degree d) led to values of n, d as low as $n = 9$ or $d = 4$, but not both: some small known (n, d) pairs are $(58, 4)$, $(38, 8)$, $(32, 12)$, $(29, 16)$ [8]. This also shows that MP is Turing complete even when it is limited to programs having a fixed number of variables. It is known that setting $d = 2$ is not possible,

since quadratic Diophantine equations are decidable [20]; we remark, however, that Integer Quadratic Programming is *not* decidable [18], because of the presence of an objective function.

In Sect. 3 we present a MP that simulates the MRM, which we then use in Sect. 4 to automatically find the most difficult input (of given size) for a given algorithm. Specifically, we construct a pure feasibility IPP of degree 3 which takes as input a program P for the MRM and its input c, and describes a feasible set containing the intermediate and final computation steps carried out by the MRM running P on c. Although the size of our MP grows with the size of P and c, and must actually have infinite size to accommodate programs for which no upper bound on the computation time to termination can be estimated, it can still be useful for practical purposes, as shown in Sect. 4. Specifically, for programs involving values within a given upper bound, our MP can be reformulated (exactly) to an Integer Linear Program (ILP), for which reasonably good off-the-shelf solvers exist.

We also remark that Cook's famous theorem proving that the SATISFIABILITY (SAT) problem is **NP**-hard [9] may also be used to construct a reduction from a UTM to MP (via SAT). From a practical point of view, however, such a reduction presents the following shortcomings (of both practical and theoretical nature).

- Programming a UTM is more difficult than programming the MRM.
- Solving the SAT obtained by Cook's reduction by means of a SAT solver would make it hard or inefficient to add an objective function, which is crucial to our application in Sect. 4.
- A further reduction of the SAT to MP would yield IPPs of any degree ($\wedge$ within literals being represented by products between boolean variables), making them difficult to solve in practice.
- Cook's theorem actually reduces a nondeterministic polynomial time bounded UTM to SAT, whereas in this setting we are interested in deterministic TMs without regard to time constraints. Hence the reduction would require some nontrivial adaptation.

2.4 Software verification

It was shown in [3] that imperative programming languages that are able to express juxtaposition of commands, tests and loops are Turing complete. In the universal MP of Sect. 3, "juxtaposition of commands" refers to the fact that the MP explicitly models every instruction of the program. In consequence of this, every program variable is assigned a unique value at each step; or, in other words, every variable symbol is assigned a sequence of values which is as long as the number of timesteps taken by the program execution. In our application to software verification we relax the command juxtaposition requirement: instead of a sequence, every variable symbol is assigned a set containing at least all of the values in the sequence (and perhaps some more). The order of the timestep (and hence the command juxtaposition) is lost. What we gain are insights about the domain of of the variable, which can be useful to detect bugs.

Static analysis by abstract interpretation [6, 7] aims to find program invariants as over-approximations (also called *abstract semantics*) of the sets of values (also called *concrete semantics*) that the program variables can take at each control point of the program during the whole execution. Abstract semantics are usually restricted to belong to a pre-specified class of sets, e.g. intervals, boxes, spheres, polyhedra and so on. Given one such class $\mathcal{L}$ and the inclusion lattice $(\mathcal{L}, \subseteq)$, the action of the program can be interpreted in the abstract semantics as a function $F : \mathcal{L} \to \mathcal{L}$.

The interpretation of a program as a function on a lattice is best explained using the simple example in Fig. 2. Essentially, the j-th variable symbol occurring in statement i is replaced by the set $X_{ij} \in \mathcal{L}$, which is equated to an expression involving other $X_{k\ell}$'s on the right hand side of the equation; if we let X be the vector of all X_{ij}'s and F be the lattice function representing the expression on the right hand side, then the set of equations has the form:

$$X = F(X), \tag{1}$$

which are fixed point equations, also called *semantic equations* in this context. An over-approximation $X \in \mathcal{L}$ is *invariant* with respect to F if it does not change when F is applied to it, i.e. it satisfies the fixed point equations. In particular, a Least Fixed Point (LFP) X^* of F in $\mathcal{L}$, i.e. a lattice set which is smallest with respect to inclusion, is a smallest invariant of the computer program encoded by F.

If $\mathcal{L}$ is the concrete semantics, then X^* is the set of all values taken by all variables during the program execution. Since finding such a set in general is easily seen to be equivalent to solving the halting problem, X^* is not a decidable set. It turns out, however, that X^* is decidable for certain abstract semantics that still give useful information about the program.

Take for example an array `x` with size 10, indexed by a counter variable `i`. An abstract semantic with an invariant X, whose projection on the `i` coordinate is contained in the interval $[0, 9]$, yields a rigorous proof that no memory error will result from writing or reading out of array bounds. Since these types of runtime errors are typically very difficult to catch, and may yield disastrous as well as unpredictable outcomes, a strategy for proving their absence is valuable. This also explains why large invariants are less interesting: the interval $[-\infty, \infty]$ might be an invariant, but it can only prove the trivial property $\mathtt{i} \in [-\infty, \infty]$.

Two existing solution methods for finding X such that Eq. (1) holds are Kleene's Iteration (KI) [6] and Policy Iteration (PI) [5, 12, 13]. KI is an iterative, possibly nonterminating procedure based on applying F to the largest possible domain in the $\mathcal{L}$ until convergence to a fixed point is attained. PI is a kind of "Newton's method on lattices", which only converges to a guaranteed LFP under some additional conditions on F, namely non-expansiveness, playing the same role as convexity in in a classical setting.

The alternative approach proposed here consists in describing the feasible set defined by (1) using a MP which can be solved in exponential time whenever (1) only involve integer affine arithmetic. Such a MP is of the same type of the UMP of Sect. 3: its solution set is essentially the trace (in the abstract semantics) of an algorithm given in imperative language. A preliminary version of these ideas was given in [22].

3 A universal MP

We now propose a Universal MP (UMP) whose solution set is precisely the output of the MRM together with the set of all values generated during all steps of the computation. The UMP introduces integer decision variables for each register at each time step (or iteration) $t \in \mathbb{N}$ as well as two sets of binary decision variables: one verifies whether a given register contains 0 at a given time step; the other verifies whether c is assigned a given state at a given time step. Intuitively, the proposed UMP works by reformulating the following conditional constraints

$$b = 0 \quad \Rightarrow \quad (R_j = R_j + 1) \wedge (c = k) \tag{2}$$
$$(b = 1) \wedge R_j = 0 \quad \Rightarrow \quad c = \ell \tag{3}$$
$$(b = 1) \wedge R_j > 0 \quad \Rightarrow \quad (R_j = R_j - 1) \wedge (c = k). \tag{4}$$

to integer polynomial equations.

Here follows the definition of the UMP.

- Sets:
  1. $\mathbb{N}$: the set of natural numbers (which includes 0);
  2. $\mathbb{N}_+$: the set of positive natural numbers;
  3. $N = \{0, \ldots, n\}$: set of all states;
  4. $N_+ = N \smallsetminus \{0\}$: set of all non-stop states.

- Parameters:
  - $n \in \mathbb{N}$: the number of states;
  - $h : N \to \mathbb{N}$: register index j targeted by instruction;
  - $b : N \to \{0, 1\}$: instruction type;
  - $k : N \to N$: next state for type 0 instructions, conditional next state otherwise;
  - $\ell : N \to N$: conditional next state for type 1 instructions;
  - $r^0 : \mathbb{N} \to \mathbb{N}$: the initial values held in the registers, i.e. the input;
  - $S^0 \in \mathbb{N}$: the initial state.
- Decision variables:
  1. value held by register j at iteration t, added by 1 (this is to avoid r_{jt} taking 0 as a value, which would invalidate some of the constraints below — the value contained in R_j at iteration t is $r_{jt} - 1$):
  $$\forall j \in \mathbb{N}, t \in \mathbb{N} \quad r_{jt} \in \mathbb{N}_+;$$
  2. test whether register j has value 0 at iteration t:
  $$\forall j \in \mathbb{N}, t \in \mathbb{N} \quad \rho_{jt} = \begin{cases} 1 & \text{if } r_{jt} \geq 2 \\ 0 & \text{if } r_{jt} = 1 \end{cases}$$
  3. test whether the current instruction at iteration t is i:
  $$\forall i \in N, t \in \mathbb{N} \quad x_{it} = \begin{cases} 1 & \text{if } c = i \text{ at iteration } t \\ 0 & \text{otherwise.} \end{cases}$$
- Objective functions: none, this is a pure feasibility problem. Objective functions can be used in possible applications of the UMP, see Sect. 4.
- Constraints:
  - initial register values:
  $$\forall j \in \mathbb{N} \quad r_{j0} = r_j^0; \tag{5}$$
  - initial state:
  $$x_{10} = 1; \tag{6}$$
  - if $c = i$ and $b = 0$, set $R_{h_i} \leftarrow R_{h_i} + 1$:
  $$\forall t \in \mathbb{N}_+, i \in N_+ \quad x_{i,t-1} \, (1 - b_i) \, r_{h_i t} = x_{i,t-1} \, (1 - b_i) \, (r_{h_i,t-1} + 1); \tag{7}$$
  - if $c = i$ and $b = 0$, set $c \leftarrow k_i$:
  $$\forall t \in \mathbb{N}_+, i \in N_+ \quad x_{k_i t} \, x_{i,t-1} \, (1 - b_i) = x_{i,t-1} \, (1 - b_i); \tag{8}$$
  - if $c = i$, $b = 1$ and $R_{h_i} > 0$, set $R_{h_i} \leftarrow R_{h_i} - 1$:
  $$\forall t \in \mathbb{N}_+, i \in N_+ \quad x_{i,t-1} \, b_i \, \rho_{h_i,t-1} \, r_{h_i t} = x_{i,t-1} \, b_i \, \rho_{h_i,t-1} \, (r_{h_i,t-1} - 1); \tag{9}$$
  - if $c = i$, $b = 1$ and $R_{h_i} > 0$, set $c \leftarrow k_i$:
  $$\forall t \in \mathbb{N}_+, i \in N_+ \quad x_{i,t-1} \, b_i \, \rho_{h_i,t-1} \, x_{k_i t} = x_{i,t-1} \, b_i \, \rho_{h_i,t-1}; \tag{10}$$
  - if $c = i$, $b = 1$ and $R_{h_i} = 0$, fix R_{h_i}:
  $$\forall t \in \mathbb{N}_+, i \in N_+ \quad x_{i,t-1} \, b_i \, (1 - \rho_{h_i,t-1}) \, r_{h_i t} = x_{i,t-1} \, b_i \, (1 - \rho_{h_i,t-1}) \, r_{h_i,t-1}; \tag{11}$$

– if $c = i$, $b = 1$ and $R_{h_i} = 0$, set $c \leftarrow \ell_i$:

$$\forall t \in \mathbb{N}_+, i \in N_+ \quad x_{i,t-1}\, b_i\, (1 - \rho_{h_i,t-1})\, x_{\ell_i t} = x_{i,t-1}\, b_i\, (1 - \rho_{h_i,t-1}); \tag{12}$$

– if $c = i$ and $j \neq h_i$, fix R_j:

$$\forall t \in \mathbb{N}_+, i \in N_+, j \in \mathbb{N} \smallsetminus \{h_i\} \quad r_{jt}\, x_{i,t-1} = r_{j,t-1}\, x_{i,t-1}; \tag{13}$$

– if $c = 0$ then stop:

$$\forall t \in \mathbb{N}_+ \quad x_{0t}\, x_{0,t-1} = x_{0,t-1}; \tag{14}$$

– if $c = 0$ then fix R_j:

$$\forall t \in \mathbb{N}_+, j \in \mathbb{N} \quad r_{jt} x_{0,t-1} = r_{j,t-1} x_{0,t-1}; \tag{15}$$

– c can only take one value at any given iteration:

$$\forall t \in \mathbb{N} \quad \sum_{i \in N} x_{it} = 1; \tag{16}$$

– definition of ρ variables in terms of r variables:

$$\forall j \in \mathbb{N}, t \in \mathbb{N} \quad r_{jt} - 1 \geq \rho_{jt} \tag{17}$$

$$\forall j \in \mathbb{N}, t \in \mathbb{N} \quad (r_{jt} - 1)\,(1 - \rho_{jt}) = 0. \tag{18}$$

The fact that the formulation has an infinite number of variables and constraints, albeit countable, prevents it from being solved in practice with standard software. This, however, does not matter when proving Turing completeness: after all even UTMs require an infinite storage tape whereas all existing computers have a finite amount of memory. The crucial fact is that if the MRM terminates on a given input, then the above set of constraints is satisfied by a unique feasible solution (r^*, x^*, ρ^*), which corresponds to the output of the program running on the MRM, as shown in Cor. 3.2. If we give an upper bound on the program running time (we let the index t run over a finite set $T \subseteq \mathbb{N}$) and on the number of registers (we let the index j run over a finite set $M \subseteq \mathbb{N}$), then the above program is a MINLP which can be solved exactly, using for example a spatial Branch-and-Bound (sBB) solver [2], coded so that it uses precise rational arithmetic and configured with an ε tolerance equal to zero.

3.1 Theorem
Every sequence (r, x, ρ) feasible in (5)-(18) has the property that, whenever $t^* = \min\{t \in T \mid x_{0t} = 1\} - 1$ is defined, $R^* = (r_{1t^*} - 1, r_{2t^*} - 1, \ldots)$ is exactly the output R of a MRM with input r^0 which terminates after t^* timesteps.

Proof. We prove this by induction on t. Let (r, x, ρ) be feasible in (5)-(18). When $t = 0$, $R_j = r_{j0} - 1$ for all $j \in \mathbb{N}$ by (5) and the definition of the decision variables r on page 6. Consider a timestep t and suppose $R_j = r_{j,t-1} - 1$ for all j at timestep $t - 1$. Let $S_i = (j, b_i, k, \ell)$ be the MRM instruction applied at timestep $t - 1$. At time t, by (13), for all $j' \neq j$ we have $r_{j't} = r_{j',t-1}$, so that, by the induction hypothesis, $R_{j'} = r_{j't} - 1$. If $j' = j$ and $b_i = 0$, then by (7) we have $r_{jt} = r_{j,t-1} + 1$, and by the induction hypothesis $R_j = r_{jt} - 1$. If $b_i = 1$ and $R_j > 0$ at timestep $t - 1$, by (9) $r_{jt} = r_{j,t-1} - 1$ and by the induction hypothesis $R_j = r_{jt} - 1$ at timestep t; otherwise, if $R_j = 0$ at $t - 1$, then its value is fixed by (11) at time t. Mutual exclusivity of these three choices is enforced by multiplying (7)-(12) by respectively b_i and $\rho_{j,t-1}$, which by (17)-(18) is 1 if and only if $R_j > 0$. The next instruction to be executed is always correctly chosen by (8), (10), (12). Notice (7)-(15) are all multiplied by $x_{i,t-1}$ for $i \neq 0$; by (16), whenever $x_{0t'} = 1$ (i.e., $c = 0$), $x_{it'} = 0$ for all $i \neq 0$ and all t': this implies (7)-(15) are trivially satisfied as $0 = 0$, the only nontrivial constraints being (14)-(18). In particular, by (15) $r_{jt} = r_{j,t-1}$ for all j whenever $c = 0$, which means, by (6), that t^* is by definition the first timestep index when $c = 0$, i.e. R^* contains the output of the MRM. □

3.2 Corollary
If the MRM terminates on a given input, the UMP has a unique solution.

Proof. This follows because the MRM is deterministic. □

3.3 Example
Continuing with Example 2.2, the behaviour of the MRM in Fig. 1 with input $R_1 = R_2 = R_3 = 1$ is given in the table below (t is the iteration counter, c is the current state, R_1, R_2, R_3 are the contents of the three registers).

t	c	R_1	R_2	R_3	Comment	h	b	k	ℓ
0	1	1	1	1	initial state	3	1	1	2
1	1	1	1	**0**	zero R_3	3	1	1	2
2	2	1	1	0	switch to state 2	2	1	3	6
3	3	1	**0**	0	get the value of R_2	3	0	4	0
4	4	1	0	**1**	and copy it in R_3	1	0	5	0
5	5	**2**	0	1	add it to R_1	1	0	2	0
6	2	**3**	0	1	twice	2	1	3	6
7	6	3	0	1	switch to state 6	3	1	7	0
8	7	3	0	**0**	get the value of R_3	2	0	6	0
9	6	3	**1**	0	and restore it to R_2	3	1	7	0
10	0	3	1	0	stop	-	-	-	-

Updated registers are marked in boldface. These values have been found by solving an AMPL [11] implementation of the bounded version of the UMP (5)-(18).

3.1 Reformulation to MILP in the bounded case

If the program running on the MRM is bounded, i.e. there is an upper bound M such that for all $j \in \mathbb{N}$ we have $R_j \leq M$, then (5)-(18) can be reformulated exactly to an Mixed Integer Linear Program (MILP), by applying the INT2BIN exact reformulation [21] to replace the r variables by sets of added binary variables, and then the PRODBIN exact reformulation [21] to replace all bilinear products $\alpha\beta$ of binary variables α, β with continuous added variables $\gamma \in [0, 1]$ and the Fortet constraints [10] $\gamma \leq \alpha$, $\gamma \leq \beta$, $\gamma \geq \alpha + \beta - 1$.

This means that we can solve the (bounded) UMP using a practically efficient MILP solver, such as CPLEX [17].

4 Finding the hardest input

Besides simulating the MRM, our UMP has some interesting possible uses insofar as it can be easily adapted to solve some associated inverse problems, by simply declaring some parameters to be decision variables and fixing some decision variables to appropriate values. For example, in order to bench-test a new algorithm targeting the solution of a given problem, it is useful to find inputs for which the algorithm takes longest. This can be done as follows:

- declare r^0 (the input) to be decision variables;
- adjoin the objective function:

$$\min \sum_{t \in \mathbb{N}} x_{0t}. \tag{19}$$

By setting as few of the x_{0t} to 1 as possible, Eq. 19 essentially tells the UMP to terminate execution as late as possible, whilst being computationally consistent with the r^0 being chosen. In other words, find an input which causes the MRM to continue its computation for as long as possible.

4.1 Exact integer division

We provide a proof of concept example which is simple enough so that the approach works with an off-the-shelf tool (CPLEX [17]), and the corresponding bounded UMP is solved within acceptable computation times. We consider the following algorithm for establishing whether $n \bmod k = 0$, for given $n \geq k$:

1. $n \leftarrow n - k$;
2. if $n = 0$ return YES, else if $n < 0$ return NO, else goto 1,

and ask for what input k the algorithm runs longest, given a fixed n.

The above pseudocode can be implemented in a MRM with 4 registers and 8 states. We limit the analysis of the code to 100 time steps. Register 1 encodes n; registers 2,3 encode k, k' (k' is a complementary storage for k: whenever k changes, k' stores the original value of k); register 4 is the output bit a: 0 if NO, 1 if YES. The actual semantics of the MRM is slightly different from the algorithm above, in that it decreases k incrementally, and n at the same time. The MRM program is as follows, and starts execution with state 1 and k', a initialized to 0.

State	Instruction				Meaning	Comment
0	-	-	-	-	stop	
1	2	1	2	4	if $k > 0$ decrease k and goto *2*, else *4*	start here
2	3	0	3	0	increase k' and goto *3*	invariant: $k + k'$
3	1	1	1	0	if $n > 0$ decrease n and goto *1*, else *0*	$n = 0$ before $k \Rightarrow k \not| n$
4	1	1	5	8	if $n > 0$ decrease n and goto *5*, else *8*	$n, k = 0$: $\Rightarrow k \| n$
5	1	0	6	0	increase n and goto *6*	restore 1 to n
6	3	1	7	1	if $k' > 0$ decrease k' and goto *7*, else *1*	restore k using k'
7	2	0	6	0	increase k and goto *6*	
8	4	0	0	0	increase a and goto *0*	set $a = 1$

We then feed the set of instructions of this MRM to our universal MINLP, modified as above with an objective function, to determine the most computationally expensive input k with n fixed, and find, for $n \in \{5, \ldots, 10\}$, that $k = 1$.

5 Debugging code using MP

In this section we discuss the application of MP to software verification. This MP is obtained by relaxing the Turing completeness requirement about commands juxtaposition [3], as explained in Sect. 2.4.

5.1 Reducing code to a graph

Flowcharts (i.e., directed graph representations of computer programs) are known to be Turing complete [16]. It is not hard to show that Turing completeness is not lost if we require that no flowchart node has more than two incoming arcs. Given such a flowchart (also called a *program graph*) $G = (V, A)$, representing a computer program with n variables $\mathtt{x} = (\mathtt{x}_1, \ldots, \mathtt{x}_n)$, where V is the set of control points of the program and $A = \{a_1, \ldots, a_m\}$ is the set of arcs in the program graph, linking a control point

to the next, we assign a sequence of intervals $X_i = (X_{i1}, \ldots, X_{in})$ to each arc a_i, for all $i \leq m$. For all $i \leq m, j \leq n$, X_{ij} is an over-approximation of the set of values taken by variable $\mathtt{x}_j$ on the arc i over the whole program execution.

Control points in the program are assigned one of the following labels: Entry, eXit, Assignment, Join (i.e. the start of the loop), Test. A sequence of operators $(F_{vj})_{j \leq n}$ is assigned to each control point $v \in V$ according to its label. For every flow arc $a_i = (v, u)$ and $j \leq n$, we state the rules that change X_{ij} according to the program as $X_{ij} = F_{vj}(X)$, where $X = (X_1, \ldots, X_m)$. Since each arc has exactly one head vertex, we can index the operators by arc i instead of control point v, so that we obtain the fixed point equations (1) in the form:

$$\forall i \leq m, j \leq n \quad X_{ij} = F_{ij}(X). \tag{20}$$

Notationwise, we let $F = (F_1 \ldots, F_m)$ where $F_i = (F_{i1}, \ldots, F_{in})$ for all $i \leq m$. An example is given in Fig. 2.

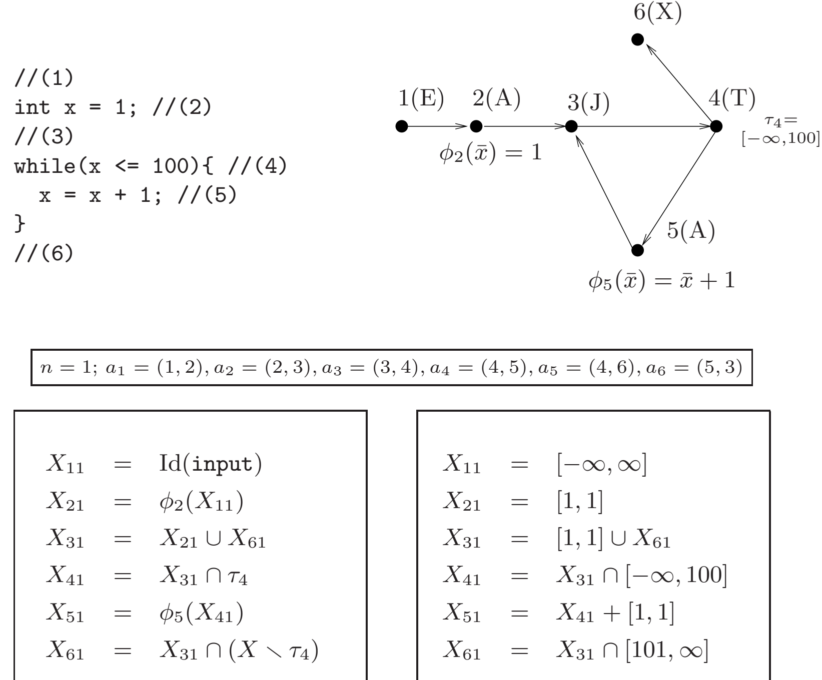

Figure 2: A simple example: program graph (top), semantic equations (bottom left), replacing variable symbols with interval (abstract) semantics (bottom right).

The operator for labels E, X is the identity Id, the operators for label A are the interval arithmetic operators $+, c\times, \uparrow d, \times, 1\div$ (where $+, \times$ are binary operators, $c\times$ is the constant multiplication, $\uparrow d$ is the power to constant, c is a real and d is a positive constant) [15], the operator for label J is the interval union $\cup$ (i.e. the union of two disjoint intervals is the smallest interval containing them) and the operator for label T is the intersection $\cap$.

5.2 An interval MP for computing least fixed points

As over-approximations of the sets of values taken by program variables during execution, we consider the inclusion-wise lattice $(\mathcal{I}, \subseteq)$ of all closed real intervals. For all $i \leq m$, $j \leq n$ we represent the interval X_{ij} by a triplet $(x_{ij}^L, x_{ij}^U, \bar{x}_{ij}) \in \mathbb{R}^2 \times \{0, 1\}$ (subject to $x_{ij}^L \leq x_{ij}^U$) such that $X_{ij} = [x_{ij}^L, x_{ij}^U]$ if and only if $\bar{x}_{ij} = 1$ and $X_{ij} = \emptyset$ otherwise. We also define the following width function:

$$|X_{ij}| = \bar{x}_{ij}(x_{ij}^U - x_{ij}^L) + \log \bar{x}_{ij}, \tag{21}$$

and extend it to $|X| = \sum_{i,j} |X_{ij}|$. This width function is unbounded below if and only if X_{ij} is the empty interval, and is equal to the standard interval width otherwise. Applied to a box X, this width function is unbounded below if and only if X is empty (i.e. at least one of the intervals in the definition of X is empty). Thus, the bottom element $\perp$ of any inclusion sublattice of $\mathcal{I}$ is smallest with respect to the width function $|\cdot|$ restricted to the sublattice.

It is not difficult to establish that all the considered operators are $\subseteq$-monotonic in $\mathcal{I}$. By Tarski's lattice fixed point theorem [28], the LFP of (20) is

$$\mathrm{argmin}\{|X| : X \supseteq F(X)\}. \tag{22}$$

Eq. (22) can be used to construct a mathematical program as follows. For every operator F_{ij} appearing in the computer program, we define the set $\{X \mid X_{ij} \supseteq F_{ij}(X)\}$ in terms of inequality constraints $g^{ij}(x^L, x^U, \bar{x}, y) \leq 0$ involving the decision variables $x^L, x^U, \bar{x}$ and possibly some added binary decision variables y.

The intended semantics of the MP below, which models (22), is as follows:

- it provides the LFP of Eq. (20) as a globally optimal solution if and only if it is a nonempty bounded box;
- it is infeasible or unbounded if only if the LFP of Eq. (20) contains some unbounded intervals and/or is the empty set.

We prove this in Sect. 5.3.15 below.

5.3 Interval operators

As mentioned above, we need to model the operators for labels E, X, A, J, T, i.e. the identity operator, some interval arithmetic operator, interval union and interval intersection. In this section we model the semantics of each operator using MP constraints. We remark that there are many different formulations that model such constraints. In this setting we aim to achieve clarity of exposition, so we give a natural formulation rather than one which is computationally convenient. Computationally more convenient formulations were used to obtain the computational results in [14].

5.3.1 Conditional constraints

To model certain interval operators using MP, we shall need some constraints to hold conditionally to some binary variables taking value zero or one. Suppose $g(x) \leq g_0$ is one such constraint (where x is the vector of all decision variables, g is a function, and g_0 is a constant), and $y \in \{0, 1\}$ is a binary variable. To mean that $g(x) \leq g_0$ holds conditionally to $y = 1$, we simply write $yg(x) \leq yg_0$: if $y = 0$, then the constraint obviously reduces to $0 \leq 0$, which does not change the feasible region of the MP in any way. In the sequel, we use the shorthand notation $y \to g(x) \leq g_0$ to mean $yg(x) \leq yg_0$.

We remark that, if $y \to g(x) \leq g_0$ and $(1 - y) \to g(x) \geq g_0$, then $g(x) \leq g_0$ implies $y = 1$ and $g(x) \geq g_0$ implies $y = 0$ for practical purposes in MP. Thus, we write $y \leftrightarrow g(x) \leq g_0$ to mean:

$$yg(x) \leq yg_0 \quad \wedge \quad (1 - y)g(x) \geq (1 - y)g_0.$$

5.3.2 Interval consistency

For all $i \leq m$, $j \leq n$, X_{ij} is the interval $[x^L_{ij}, x^U_{ij}]$, so the following MP constraints hold:

$$x^L_{ij} \leq x^U_{ij}.$$

5.3.3 Constant

The constant fixed point equation is $X_{ij} = [\beta^L, \beta^U]$ for some $i \le m$ and $j \le n$, where $\beta^L, \beta^U \in \mathbb{R}$; the semantics of this operator must also explicitly account for $X_{ij} = \varnothing$ if and only if $[\beta^L, \beta^U]$ is a non-empty interval. The corresponding MP constraints are:

$$\begin{aligned} \bar{x}_{ij} &= \begin{cases} 1 \text{ if } \beta^L \le \beta^U \\ 0 \text{ otherwise.} \end{cases} \\ \bar{x}_{ij} \to x^L_{ij} &\le \beta^L \\ \bar{x}_{ij} \to x^U_{ij} &\ge \beta^U. \end{aligned}$$

5.3.4 Identity

The identity fixed point equation is $X_{ij} = X_{hj}$ for some $i \ne k \le m$ and $j \le n$; the semantics of this operator must also explicitly account for $X_{ij} = \varnothing$ if and only if $X_{kj} = \varnothing$. The corresponding MP constraints are:

$$\begin{aligned} \bar{x}_{ij} &= \bar{x}_{kj} \\ \bar{x}_{ij} \to x^L_{ij} &\le x^L_{kj} \\ \bar{x}_{ij} \to x^U_{ij} &\ge x^U_{kj}. \end{aligned}$$

5.3.5 Positive constant scaling

The fixed point equation for positive constant scaling is $X_{ij} = \alpha X_{kh}$ for some $i \ne k \le m$, $j, h \le n$ and $\alpha > 0$; the semantics of this operator must also explicitly account for $X_{ij} = \varnothing$ if and only if $X_{kh} = \varnothing$. The corresponding MP constraints are:

$$\begin{aligned} \bar{x}_{ij} &= \bar{x}_{kh} \\ \bar{x}_{ij} \to x^L_{ij} &\le \alpha x^L_{kh} \\ \bar{x}_{ij} \to x^U_{ij} &\ge \alpha x^U_{kh}. \end{aligned}$$

5.3.6 Negative constant scaling

The fixed point equation for negative constant scaling is $X_{ij} = \alpha X_{kh}$ for some $i \ne k \le m$, $j, h \le n$ and $\alpha < 0$; the semantics of this operator must also explicitly account for $X_{ij} = \varnothing$ if and only if $X_{kh} = \varnothing$. The corresponding MP constraints are:

$$\begin{aligned} \bar{x}_{ij} &= \bar{x}_{kh} \\ \bar{x}_{ij} \to x^L_{ij} &\le \alpha x^U_{kh} \\ \bar{x}_{ij} \to x^U_{ij} &\ge \alpha x^L_{kh}. \end{aligned}$$

5.3.7 Positive odd power

The fixed point equation for positive odd power is $X_{ij} = X^d_{kh}$, for some $i \ne k \le m$, $j, h \le n$ and $d \pmod 2 = 1$; the semantics of this operator must also explicitly account for $X_{ij} = \varnothing$ if and only if

$X_{kh} = \varnothing$. The corresponding MP constraints are:

$$\begin{aligned} \bar{x}_{ij} &= \bar{x}_{kh} \\ \bar{x}_{ij} \to x^L_{ij} &\le (x^L_{kh})^d \\ \bar{x}_{ij} \to x^U_{ij} &\ge (x^U_{kh})^d. \end{aligned}$$

5.3.8 Positive even power

The fixed point equation for positive even power is $X_{ij} = X^d_{kh}$, for some $i, k \le m$, $j, h \le n$ and $d \pmod 2 = 0$; the semantics of this operator must also explicitly account for $X_{ij} = \varnothing$ if and only if $X_{kh} = \varnothing$. As usual, this is imposed by

$$\bar{x}_{ij} = \bar{x}_{kh}.$$

Since the even power function is not monotonic, we distinguish two cases:

1. $0 \le x^L_{kh} \le x^U_{kh}$ or $x^L_{kh} \le x^U_{kh} \le 0$, in which case $x^L_{ij} \le \min\{(x^L_{kh})^d, (x^U_{kh})^d\}$ and $x^U_{ij} \ge \max\{(x^L_{kh})^d, (x^U_{kh})^d\}$;
2. $x^L_{kh} \le 0 \le x^U_{kh}$, in which case $x^L_{ij} \le 0$ and $x^U_{ij} \ge \max\{(x^L_{kh})^d, (x^U_{kh})^d\}$.

Since $x^U_{ij} \ge \max\{(x^L_{kh})^d, (x^U_{kh})^d\}$ holds in both cases, we use the constraints:

$$\begin{aligned} \bar{x}_{ij} \to x^U_{ij} &\ge (x^U_{kh})^d \\ \bar{x}_{ij} \to x^U_{ij} &\ge (x^L_{kh})^d. \end{aligned}$$

The first case is equivalent to $x^L_{kh} x^U_{kh} \ge 0$; the second is equivalent to $x^L_{kh} x^U_{kh} \le 0$. We introduce a binary variable $y_{kh} \in \{0, 1\}$ which takes value 0 if $x^L_{kh} x^U_{kh} \le 0$, and enforce this condition by means of the constraint:

$$x^L_{kh} x^U_{kh} y_{kh} \ge 0. \tag{23}$$

Next, we use y_{kh} in the constraints on x^L_{ij}:

$$\begin{aligned} \bar{x}_{ij} \to x^L_{ij} &\le (x^U_{kh})^d y_{kh} \\ \bar{x}_{ij} \to x^L_{ij} &\le (x^L_{kh})^d y_{kh}. \end{aligned}$$

5.3.9 Inverse

The fixed point equation for inverse is $X_{ij} = \frac{1}{X_{kh}}$, for some $i, k \le m$ and $j, h \le n$; the semantics of this operator must also explicitly account for $X_{ij} = \varnothing$ if and only if $X_{kh} = \varnothing$, and $X_{ij} = \top$ (the largest element of the interval lattice) if $0 \in X_{kh}$. As in Sect. 5.3.8, this is equivalent to $x^L_{kh} x^U_{kh} \le 0$, which we model using an added binary variable $y_{kh} \in \{0, 1\}$ subject to (23). We obtain:

$$\begin{aligned} \bar{x}_{ij} &= \bar{x}_{kh} \\ x^L_{kh} x^U_{kh} y_{kh} &\ge 0 \\ \bar{x}_{ij} y_{kh} \to x^L_{ij} &\le \frac{1}{x^U} \\ \bar{x}_{ij} y_{kh} \to x^U_{ij} &\ge \frac{1}{x^L}. \end{aligned}$$

In a MP, however, there is no computationally viable way to constrain an interval to be $\top$. Formally, we can write:

$$\begin{aligned} \bar{x}_{ij}(1 - y_{kh}) \to x^L_{ij} &\le -\infty \\ \bar{x}_{ij}(1 - y_{kh}) \to x^U_{ij} &\ge \infty, \end{aligned}$$

where we take ∞ to be a formal symbol with the meaning of "unbounded"; but in practice we must replace ∞ by a finite (large) number M, as mentioned in Sect. 5.4 below. In the following, we simply assume that there is no inverse operator in the computer code being analyzed; a weaker assumption would be that $0 \notin X_{kh}$, but this is harder to verify aprioristically.

5.3.10 Sum

The fixed point equation for the interval sum operator is $X_{ij} = X_{kh} + X_{\ell f}$ for some distinct $i, k, \ell \leq m$ and $j, h, f \leq n$. If there are empty intervals, the semantics are: $X_{ij} = \varnothing$ if and only if $X_{kh} = \varnothing \vee X_{\ell f} = \varnothing$. The corresponding MP constraints are:

$$\begin{aligned} \bar{x}_{ij} &= \bar{x}_{kh}\bar{x}_{\ell f} \\ \bar{x}_{ij} \to x^L_{ij} &\leq x^L_{kh} + x^L_{\ell f} \\ \bar{x}_{ij} \to x^U_{ij} &\geq x^U_{kh} + x^U_{\ell f}. \end{aligned}$$

5.3.11 Product

The fixed point equation for the interval sum operator is $X_{ij} = X_{kh}X_{\ell f}$ for some distinct $i, k, \ell \leq m$ and $j, h, f \leq n$. If there are empty intervals, the semantics are: $X_{ij} = \varnothing$ if and only if $X_{kj} = \varnothing \vee X_{\ell j} = \varnothing$. The corresponding MP constraints are derived from the interval product $[a, b][c, d] = [\min\{ac, ad, bc, bd\}, \max\{ac, ad, bc, bd\}]$:

$$\begin{aligned} \bar{x}_{ij} &= \bar{x}_{kh}\bar{x}_{\ell f} \\ \bar{x}_{ij} \to x^L_{ij} &\leq x^L_{kh}x^L_{\ell f} \\ \bar{x}_{ij} \to x^L_{ij} &\leq x^L_{kh}x^U_{\ell f} \\ \bar{x}_{ij} \to x^L_{ij} &\leq x^U_{kh}x^L_{\ell f} \\ \bar{x}_{ij} \to x^L_{ij} &\leq x^U_{kh}x^U_{\ell f} \\ \bar{x}_{ij} \to x^U_{ij} &\geq x^L_{kh}x^L_{\ell f} \\ \bar{x}_{ij} \to x^U_{ij} &\geq x^L_{kh}x^U_{\ell f} \\ \bar{x}_{ij} \to x^U_{ij} &\geq x^U_{kh}x^L_{\ell f} \\ \bar{x}_{ij} \to x^U_{ij} &\geq x^U_{kh}x^U_{\ell f}. \end{aligned}$$

5.3.12 Union

The fixed point equation for interval union is $X_{ij} = X_{kh} \cup X_{\ell f}$ for some distinct $i, k, \ell \leq m$ and $j, h, f \leq n$. If there are empty intervals, the semantics are: $X_{ij} = \varnothing$ if and only if $X_{kj} = \varnothing \wedge X_{\ell j} = \varnothing$.

Notice that the union of two intervals is the smallest interval containing both; since the objective function aims to reduce the sum of all interval widths, we need our constraints to only enforce containment, i.e. $x^L_{ij} \leq \min(x^L_{kh}, x^L_{\ell f})$ and $x^U_{ij} \geq \max(x^U_{kh}, x^U_{\ell f})$, which are easily seen to be equivalent to:

$$\begin{aligned} (1 - \bar{x}_{ij}) &= (1 - \bar{x}_{kh})(1 - \bar{x}_{\ell f}) \\ \bar{x}_{ij} \to x^L_{ij} &\leq x^L_{kh} \\ \bar{x}_{ij} \to x^L_{ij} &\leq x^L_{\ell f} \\ \bar{x}_{ij} \to x^U_{ij} &\geq x^U_{kh} \\ \bar{x}_{ij} \to x^U_{ij} &\geq x^U_{\ell f}. \end{aligned}$$

5.3.13 Intersection

The fixed point equations for a program node labeled by $\underline{\mathrm{T}}$ with a test condition $\mathbf{x}_h \leq \mathbf{x}_f$ is $X_{ij} = X_{kh} \cap X_{\ell f}$ for some distinct $i,k,\ell \leq m$ and $j,h,f \leq n$. This test condition is well known to be enough to make the language Turing complete.

If there are empty intervals, the semantics are: $X_{ij} = \varnothing$ if and only if $X_{kj} = \varnothing \vee X_{\ell j} = \varnothing$. Moreover, and differently from above, X_{ij} could be empty even though X_{kh} and $X_{\ell f}$ are non-empty but disjoint: specifically, this happens if $x^U_{kh} < x^L_{\ell f}$ or $x^L_{kh} > x^U_{\ell f}$.

Notice that MP constraints can never be strict inequalities; one way around the issue is to write the inequality $a < b$ as $a = b - e^{-t}$, where t is a continuous unconstrained decision variable. We remark that this solves the issue from a theoretical point of view, but is not a computationally convenient expedient, due to its non-convexity. In practice, we would just pick a small enough number $\epsilon > 0$ and write $a < b$ as $a \leq b - \epsilon$.

Two intervals $X_{kh} = [x^L_{kh}, x^U_{kh}]$ and $X_{\ell f} = [x^L_{\ell f}, x^U_{\ell f}]$ could be in exactly one of six configurations (shown in Fig. 3):

1. non-overlapping left: $x^L_{kh} \leq x^U_{kh} < x^L_{\ell f} \leq x^U_{\ell f}$, in which case $X_{ij} = \varnothing$;
2. non-overlapping right: $x^L_{\ell f} \leq x^U_{\ell f} < x^L_{kh} \leq x^U_{kh}$, in which case $X_j = \varnothing$;
3. overlapping left: $x^L_{kh} \leq x^L_{\ell f} \leq x^U_{\ell f} \leq x^U_{kh}$, in which case $X_{ij} = X_{\ell f}$;
4. overlapping right:$x^L_{\ell f} \leq x^L_{kh} \leq x^U_{kh} \leq x^U_{\ell f}$, in which case $X_{ij} = X_{kh}$;
5. covering top: $x^L_{kh} \leq x^L_{\ell f} \leq x^U_{kh} \leq x^U_{\ell f}$, in which case $X_{ij} = [x^L_{\ell f}, x^U_{kh}]$;
6. covering bottom: $x^L_{\ell f} \leq x^L_{kh} \leq x^U_{\ell f} \leq x^U_{kh}$, in which case $X_{ij} = [x^L_{kh}, x^U_{\ell f}]$.

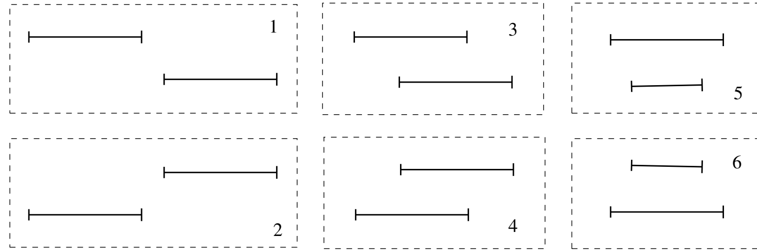

Figure 3: There are six possible interval pair configurations.

The two configurations leading to $X_{ij} = \varnothing$ correspond to strict inequalities $x^U_{kh} < x^L_{\ell f}$ or $x^U_{\ell f} < x^L_{kh}$. We introduce two binary variables $z^1_{ij}, z^2_{ij} \in \{0,1\}$ with the meaning:

$$z^1_{ij} = \begin{cases} 1 \text{ if } x^U_{kh} < x^L_{\ell f} \\ 0 \text{ otherwise} \end{cases}$$

$$z^2_{ij} = \begin{cases} 1 \text{ if } x^U_{\ell f} < x^L_{kh} \\ 0 \text{ otherwise.} \end{cases}$$

The four configurations leading to $X_{ij} \neq \varnothing$ are described by:

$$x^L_{ij} \leq \max(x^L_{kh}, x^L_{\ell f}) \quad (24)$$

$$x^U_{ij} \geq \min(x^U_{kh}, x^U_{\ell f}). \quad (25)$$

Unlike the case of the union operator (Sect. 5.3.12), we cannot simply stack constraints to model Eq. (24)-(25): we need to introduce two binary variables $y^1_{ij}, y^2_{ij} \in \{0,1\}$ with the following meaning:

$$\begin{aligned} y^1_{ij} &= \begin{cases} 1 \text{ if } \max(x^L_{kh}, x^L_{\ell f}) = x^L_{kh} \\ 0 \text{ otherwise} \end{cases} \\ y^2_{ij} &= \begin{cases} 1 \text{ if } \min(x^U_{kh}, x^U_{\ell f}) = x^U_{kh} \\ 0 \text{ otherwise.} \end{cases} \end{aligned}$$

Finally, we need to state that only one of the six configurations can happen at any one time (Eq. (33)). Specifically: Case 1 $\Leftrightarrow z^1_{ij} = 1$, Case 2 $\Leftrightarrow z^2_{ij} = 1$, Case 3 $\Leftrightarrow (1 - y^1_{ij})y^2_{ij} = 1$, Case 4 $\Leftrightarrow y^1_{ij}(1 - y^2_{ij}) = 1$, Case 5 $\Leftrightarrow (1 - y^1_{ij})(1 - y^2_{ij}) = 1$, Case 6 $\Leftrightarrow y^1_{ij} y^2_{ij} = 1$.

Thus, the intersection operator is thus modelled by the following constraints and additional continuous unconstrained variables t^1_{ij}, t^2_{ij}:

$$\begin{aligned} \bar{x}_{ij} &= \bar{x}_{kh}\bar{x}_{\ell f}(1 - z^1_{ij})(1 - z^2_{ij}) && (26) \\ z^1_{ij} \leftrightarrow x^U_{kh} &= x^L_{\ell f} - e^{-t^1_{ij}} && (27) \\ z^2_{ij} \leftrightarrow x^U_{\ell f} &= x^L_{kh} - e^{-t^2_{ij}} && (28) \\ y^1_{ij} \leftrightarrow x^L_{kh} &\geq x^L_{\ell f} && (29) \\ y^2_{ij} \leftrightarrow x^U_{kh} &\leq x^U_{\ell f} && (30) \\ \bar{x}_{ij} \rightarrow x^L_{ij} &\leq y^1_{ij} x^L_{kh} + (1 - y^1_{ij}) x^L_{\ell f} && (31) \\ \bar{x}_{ij} \rightarrow x^U_{ij} &\geq y^2_{ij} x^U_{kh} + (1 - y^2_{ij}) x^U_{\ell f} && (32) \\ 1 &= z^1_{ij} + z^2_{ij} + y^1_{ij} y^2_{ij} + (1 - y^1_{ij}) y^2_{ij} + y^1_{ij}(1 - y^2_{ij}) + (1 - y^1_{ij})(1 - y^2_{ij}). && (33) \end{aligned}$$

As stated previously, for practical purposes we remove the negative exponentials in t^1_{ij}, t^2_{ij} by replacing Eq. (27)-(28) with:

$$\begin{aligned} z^1_{ij} \leftrightarrow x^U_{kh} &\leq x^L_{\ell f} - \epsilon && (34) \\ z^2_{ij} \leftrightarrow x^U_{\ell f} &\leq x^L_{kh} - \epsilon. && (35) \end{aligned}$$

We remark that the intersection constraints are simpler (several decision variables are replaced by fixed constants) if one of $\mathbf{x}_h, \mathbf{x}_f$ is a constant interval. This happens e.g. if the test has the form $\mathbf{x}_h \leq \alpha$, where α is a constant.

5.3.14 Objective function

The objective function minimizes the total width, i.e.:

$$\min \sum_{\substack{i \leq m \\ j \leq n}} (\bar{x}_{ij}(x^U_{ij} - x^L_{ij}) + \log \bar{x}_{ij}). \quad (36)$$

Optimizing the objective subject to the constraints generated for each of the fixed point equations yields the (unique) LFP of Eq. (1).

5.3.15 Semantics

Let P be the MP defined in Sect. 5.3.2-5.3.14, and let $X^* = (X^*_{ij} \mid i \leq m \wedge j \leq n)$ be the LFP of (20). We say that P is *feasible* if it has at least one solution satisfying all the constraints, and *infeasible* if it has none; a feasible P is *bounded* if it has at least one optimal solution, and *unbounded* if it has

none. Moreover, X^* is *empty* if at least one X^*_{ij} is the empty interval, and *non-empty* otherwise; X^* is *unbounded* if at least one X^*_{ij} has at least one infinite interval bound, and *bounded* otherwise. P can only be one of: feasible and bounded, feasible and unbounded, or infeasible. X^* can be only one of: non-empty and bounded, non-empty and unbounded, empty and bounded, empty and unbounded. In particular, a box is empty and unbounded if one of the constituting intervals is empty and another is unbounded.

The following proofs are valid for codes not involving the inverse operator (Sect. 5.3.9); more specifically, they hold as long as P is not unbounded because of an operation $1/\mathbf{x}_j$ occurring at line i where the interval X_{ij} assigned to $\mathbf{x}_j$ contains 0.

5.1 Lemma
If X^* is unbounded, then P is infeasible.

Proof. Let $i \le m$ and $j \le n$ such that $X^*_{ij} = [x^L_{ij}, x^U_{ij}]$, where either or both bounds are $\pm\infty$; and suppose for simplicity that $x^U_{ij} = \infty$ (the other cases are similar). By the constraints of P, this can only happen if both the following conditions hold: (a) there is a cycle C in the program graph with a subsequence of $\underline{\mathrm{A}}$ nodes whose combined action on $\mathtt{x}_j$ is a strictly increasing function; and (b) there is no upper bounding $\underline{\mathrm{T}}$ node on $\mathtt{x}_j$ in C. Suppose then that a subset of the semantic equations (20) indexed by nodes in C projects on X_{ij} so that $X_{ij} = F(X_{hj})$ for some appropriate $h < i$ and interval operator F which strictly increases x^U_{ij}; and suppose further that any intersection operator on X_{ij} only involves intersecting with intervals having the form $[a, \infty]$. Let k be the $\underline{\mathrm{J}}$ node at the beginning of the cycle C, and $(p, k), (q, k)$ be the two incoming arcs on k, with $p < k$ being the node prior to entering C and $q > k$ being the last node in C. Then, by Sect. 5.3.12, $x^U_{kj} \ge \max(x^U_{pj}, x^U_{qj})$. By Sect. 5.3.4 and condition (b) above, $x^U_{hj} \ge x^U_{kj}$. By condition (a), $x^U_{ij} > x^U_{hj}$. Again by Sect. 5.3.4 and condition (b), $x^U_{qj} \ge x^U_{ij}$. So we obtain $x^U_{kj} \ge x^U_{qj} \ge x^U_{ij} > x^U_{hj} \ge x^U_{kj}$, which implies that P is infeasible. □

5.2 Lemma
If X^* is bounded, then P is feasible.

Proof. Since X^* is bounded, then X^* is contained in the box $[-M, M]$ for some large enough real constant M. For each $i \le m$, $j \le n$, set $\bar{x}_{ij} = 0$, $x^L_{ij} = -M$ and $x^U_{ij} = M$ if $X^*_{ij} = \varnothing$. Otherwise, set $\bar{x}_{ij} = 1$ and $[x^L_{ij}, x^U_{ij}] = X^*_{ij}$. We claim $(\bar{x}, x)$ is a feasible solution of P. Suppose not: then there must be $i \le m$ and $j \le n$ such that the corresponding constraints of P are not satisfied by $(x, \bar{x})$. Whatever label i takes in $\{\underline{\mathrm{A}}, \underline{\mathrm{T}}, \underline{\mathrm{J}}\}$, the corresponding constraints in P only depend on decision variables indexed by (i, j), (k, h), and possibly (ℓ, f) where $k, \ell \le m$ and $h, f \le n$, depending on the interval operator on the right hand side of the assignment, test, or join semantic equation (20). Now a long but simple case analysis through the constraints in Sect. 5.3.3-5.3.13 shows that supposing some constraint block indexed by (i, j) is not satisfied by $(x, \bar{x})$ would imply that X^* is not a fixed point, against the assumption. This leaves the interval consistency constraints of Sect. 5.3.2, but again supposing these constraints are not satisfied by x would imply that X^* is not a box, again against the assumption. □

5.3 Corollary
If P is infeasible, then X^* is unbounded.

5.4 Lemma
X^* is empty and bounded if and only if P is unbounded.

Proof. ($\Rightarrow$) If $X^* = \varnothing$ then there must be $X^*_{ij} = \varnothing$, so, by the constraints enforced by P on $\bar{x}$, we have $\bar{x}_{ij} = 0$, which causes the objective function to be unbounded. Since all of the other intervals defining the box X^* are bounded, then, by Lemma 5.2, P is feasible, so P must be unbounded.

($\Leftarrow$) Assume P is unbounded; because of the optimization direction (Eq. (36)), this can only happen if there are $i \le m$ and $j \le n$ such that $\bar{x}_{ij} = 0$, i.e. $X^*_{ij} = \varnothing$, which implies $X^* = \varnothing$. □

5.5 Proposition
X^* is non-empty and bounded if and only if P is feasible and bounded. Moreover, X^* is the global optimum of P.

Proof. Assume X^* is non-empty and bounded. By the correctness of the formulation of Sect. 5.3.2-5.3.14, it follows by an easy induction on the size of the program graph that X^* must be the unique global optimum of P, which is therefore feasible and bounded. Conversely, let Y be the global optimum of P: again by the correctness of the constraints of P, Y is a fixed point of Eq. (20). Because the width function $|\cdot|$ is monotonic with the inclusion direction of the interval lattice, $Y = X^*$. □

Putting all of the previous results together, we have:

5.6 Theorem
P is feasible and bounded if and only if X^* is non-empty and bounded; P is feasible and unbounded if and only if X^* is empty and bounded; P is infeasible if and only if X^* is unbounded.

5.4 Solution methods

The above MP is a MINLP which cannot in general be solved exactly. Solutions with an approximation guarantee on the objective value can be obtained using the sBB algorithm [2] — such solutions are practically useful as every feasible solution is an invariant overapproximation of the LFP.

As long as the computer program only uses integer affine arithmetic (i.e. it does not involve powers, inverses and products, and ϵ can be taken to be 1 in Sect. 5.3.13), we obtain a MINLP only involving products of binary variables $\bar{x}$ by unbounded continuous variables x. This MINLP has then an important property: once the binary variables are fixed, the resulting subproblem is an LP. It can thus be solved exactly in exponential time as follows. For all binary vectors α of size $|\bar{x}|$, fix $\bar{x} = \alpha$ and solve the resulting LP; if the LP is feasible and its solution has better objective function value than the incumbent, update the incumbent. Since LPs can be solved in polynomial time, the whole algorithm runs in exponential time in $|\bar{x}|$. This provides a theoretical improvement to the running time of Kleene's iteration on unbounded interval lattices (which, in the worst case, may fail to converge in finite time [25]). In fact, there is a strongly polynomial algorithm for solving this problem [13]; the trade-off is generality: the polynomial time algorithm *relies* on all program variables being integer and all arithmetic being affine; whereas the approach we propose can be easily generalized in many different directions.

In practice, however, it is usually possible to find a constant $M > 0$ such that all program variables only attain values in $[-M, M]$. If this is the case, all continuous decision variables x in the MINLP are bounded in $[-M, M]$; assuming affine (possibly floating point) arithmetic, the MINLP can then be linearized to a MILP using the `ProdBinCont` reformulation [21]. This MILP can be solved exactly using an off-the-shelf Branch-and-Bound solver such as CPLEX [17]. Computational experiments in this sense [14] gave correct solutions in acceptable CPU times on small and medium-sized instances.

6 Conclusion

In this work we provided a new proof that mathematical programming is Turing complete. We then showcased the use of mathematical programming techniques to solve two problems arising in software verification. As a declarative paradigm, MP benefits of several advantages with respect to empirical

analysis or iterative procedures, i.e., flexibility, proof of optimality, suitability for parallel computation. A few issues such as scalability and comparison with other techniques, e.g., SAT-based model checking, are worthy of investigation in a future work.

Acknowledgments

We are extremely grateful to an anonymous referee for helping to make this a better paper. We are also grateful to S. Bosio, E. Goubault, J. Leconte, A. Lodi, D. Monniaux, F. Roda, F. Raimondi, S. Le Roux for useful discussions, during the course of several years. This work was financially supported by the following grants: ANR "ARS", ANR "ASOPT", Digiteo "PASO" and Digiteo "RMNCCO".

References

[1] K. Apt. *Principles of Constraint Programming.* Cambridge University Press, Cambridge, 2003.

[2] P. Belotti, J. Lee, L. Liberti, F. Margot, and A. Wächter. Branching and bounds tightening techniques for non-convex MINLP. *Optimization Methods and Software*, 24(4):597–634, 2009.

[3] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[4] A. Brook, D. Kendrick, and A. Meeraus. GAMS, a user's guide. *ACM SIGNUM Newsletter*, 23(3-4):10–11, 1988.

[5] A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In K. Etessami and S.K. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *LNCS*, pages 462–475. Springer, 2005.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximations of fixed points. *Principles of Programming Languages*, 4:238–252, 1977.

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[8] M. Davis. Diophantine equations & computation: a tutorial. In *International Conference on Unconventional Computation*, Ponta Delgada, 2009. University of Azores.

[9] M. Davis, R. Sigal, and E. Weyuker. *Computability, complexity and languages.* Academic Press, San Diego, 1994.

[10] R. Fortet. Applications de l'algèbre de Boole en recherche opérationelle. *Revue Française de Recherche Opérationelle*, 4:17–26, 1960.

[11] R. Fourer and D. Gay. *The AMPL Book.* Duxbury Press, Pacific Grove, 2002.

[12] S. Gaubert, E. Goubault, A. Taly, and S. Zennou. Static analysis by policy iteration on relational domains. In R. De Nicola, editor, *European Symposium on Programming*, volume 4421 of *LNCS*, pages 237–252. Springer, 2007.

[13] T. Gawlitza and H. Seidl. Precise fixpoint computation through strategy iteration. In R. De Nicola, editor, *European Symposium on Programming*, volume 4421 of *LNCS*, pages 300–315. Springer, 2007.

[14] E. Goubault, S. Le Roux, J. Leconte, L. Liberti, and F. Marinelli. Static analysis by abstract interpretation: a mathematical programming approach. In A. Miné and E. Rodriguez-Carbonell, editors, *Proceeding of the Second International Workshop on Numerical and Symbolic Abstract Domains*, volume 267(1) of *Electronic Notes in Theoretical Computer Science*, pages 73–87. Elsevier, 2010.

[15] E. Hansen. *Global Optimization Using Interval Analysis.* Marcel Dekker, Inc., New York, 1992.

[16] D. Harel, P. Norvig, J. Rood, and T. To. A universal flowcharter. In *2nd Computers in Aerospace Conference*, volume A79-54378/24-59, pages 218–224, New York, 1979. AAIA.

[17] IBM. *ILOG CPLEX 12.2 User's Manual.* IBM, 2010.

[18] R. Jeroslow. There cannot be any algorithm for integer programming with quadratic constraints. *Operations Research*, 21(1):221–224, 1973.

[19] P. Johnstone. *Notes on logic and set theory.* Cambridge University Press, Cambridge, 1987.

[20] J. Jones. Universal diophantine equation. *Journal of Symbolic Logic*, 47(3):549–571, 1982.

[21] L. Liberti, S. Cafieri, and F. Tarissan. Reformulations in mathematical programming: A computational approach. In A. Abraham, A.-E. Hassanien, P. Siarry, and A. Engelbrecht, editors, *Foundations of Computational Intelligence Vol. 3*, number 203 in Studies in Computational Intelligence, pages 153–234. Springer, Berlin, 2009.

[22] L. Liberti, S. Le Roux, J. Leconte, and F. Marinelli. Mathematical programming based debugging. In Mahjoub [23], pages 1311–1318.

[23] R. Mahjoub, editor. *Proceedings of the International Symposium on Combinatorial Optimization*, volume 36 of *Electronic Notes in Discrete Mathematics*, Amsterdam, 2010. Elsevier.

[24] A. Makhorin. *GNU Linear Programming Kit.* Free Software Foundation, `http://www.gnu.org/software/glpk/`, 2003.

[25] D. Monniaux. A minimalistic look at widening operators. *Higher-Order Symbolic Computation*, 22:145–154, 2009.

[26] C. Shannon. A universal Turing machine with two internal states. In C. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 157–165, Princeton, 1956. Princeton University Press.

[27] E. Shapiro. *The art of Prolog.* MIT Press, Boston, MA, 1997.

[28] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[29] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1937.