Code verification by static analysis: a mathematical programming approach

Jeremy Leconte*, Stephane Le Roux*, Leo Liberti[1]*, Fabrizio Marinelli†

* *LIX, École Polytechnique, F-91128 Palaiseau, France*
Email:jeremyleconte@hotmail.com, {leroux,liberti}@lix.polytechnique.fr

† *DIIGA, Università Politecnica delle Marche, Ancona, Italy*
Email:marinelli@diiga.univpm.it

2nd May 2009

Abstract

Automatic verification of computer code is of paramount importance in embedded systems supplying essential services. One of the most important verification techniques is static code analysis by abstract interpretation: the concrete semantics of a programming language (i.e. values χ that variable symbols `x` appearing in a program can take during its execution) are replaced by abstract semantics (for example the assignment of convex over-approximations $X \supseteq \chi$ to `x`). Using abstract semantics, we represent the effect of the program on X by a function $F(X)$. All sets X satisfying the condition $F(X) \subseteq X$ are such that all values outside X are never assigned to `x` during program execution. We are particularly interested in finding the *smallest* such X, which in itself satisfies the fixpoint equation $X = F(X)$: this allows (static) detection of several types of errors, such as overflow-type bugs. Traditionally, the equations $X = F(X)$ are solved computationally using Kleene's or Policy Iteration algorithms: these methods can only guarantee convergence to the smallest fixpoint X under additional (often stringent) conditions. We propose a mathematical program whose constraints define the same space as $X \supseteq F(X)$ and whose objective function minimizes the size of X, whenever X is an array of intervals. This yields a Mixed-Integer Linear Program for code based on integer arithmetic, and a Mixed-Integer Nonlinear Program otherwise. These programs can then be solved to guaranteed global optimality by means of general purpose Branch-and-Bound type algorithms.

1 Introduction

In this paper we propose a mathematical program whose optimal solution encodes a sequence of intervals X such that: (a) the relation $x \in X$ holds for all values x taken by the variables `x` of a given computer program during its execution, and (b) for all other sequences of intervals $Y \subsetneq X$, the relation $x \in Y$ might become false during the execution of the program. An X obeying (a) is called a *program invariant* (because the relation $x \in X$ remains true during the program execution); among the program invariants, the smallest ones (i.e. those obeying (b)) are most interesting. Invariants are used to verify given properties of computer programs, such as for example "the variable $\mathtt{x}_i$ never exceeds the bounds $[0, 10]$": if we are able to show that the smallest invariant for $\mathtt{x}_i$ is $[1, 5]$ then we are sure that the property is verified. This should also explain why large invariants are less interesting: the interval $[-\infty, \infty]$ might be an invariant, but it can only prove the trivial property $\mathtt{x}_i \in [-\infty, \infty]$. The verification of such properties is very useful in finding bugs before programs are actually run, and therefore is extremely important in safety-critical applications. The discipline that studies program invariants is called Static Analysis (SA).

We recall that the *syntax* of a computer language over an alphabet is a set of rules establishing whether a given string of characters in the alphabet belongs to the language or not. Its *semantics* is an assignment of mathematical entities (for example, sets) to the variable symbols of strings in the language. Syntax

[1]Corresponding author.

and semantics of a language are usually defined conjunctively by means of *formal grammars* (for example using the software tools LEX and YACC [17]). SA by Abstract Interpretation (AI) [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. We usually restrict abstract semantics to belong to a pre-specified class of sets (called *abstract domain*): e.g. intervals, octagons, spheres, polyhedra, etc. Abstract domains may be relational (e.g. polyhedra, linear equations, linear congruences, octagons), where each lattice element is parametrized according to several variable symbols, or non-relational (e.g. intervals, translated integer ideals such as $a\mathbb{Z} + b$). Whenever the application requires finding the abstract semantics explicitly, domains which are amenable to a finite numeric representation are used. For example, closed real intervals can be represented by triplets (lower and upper bounds, and a binary value stating whether the interval is empty or not), octagons can be represented by a set of eight planes, and so on. Abstract domains are made into lattices (under setwise inclusion) via the definition of appropriate union and intersection operators. Given such a lattice $(\mathcal{L}, \subseteq)$, the action of the program on the abstract semantics can be seen as a function F mapping $\mathcal{L}$ to itself. Thus some $X \in \mathcal{L}$ is invariant with respect to the program action if it obeys the fixpoint equations for F:

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

usually called *semantic equations*. A solution of (1) is a *fixpoint* of F. In particular, the least fixpoint of F with respect to set inclusion is the smallest invariant of the computer program encoded in F.

The best known techniques for finding a solution to (1) are Kleene's Iteration (KI) and Policy Iteration (PI). KI [6, 8] is an iterative procedure based on applying F to the largest lattice element until convergence to a fixpoint is attained. The main disadvantage of KI is that in general there is no finite time bound for its termination, even for programs only involving integer affine arithmetic. A significant body of research on KI exists, however, implying bounded termination for many special cases (for example if the program variables can only attain values within given bounds known *a priori*), excellent implementations and good scalability with respect to program size. PI [4, 12, 13] is a sort of Newton's method for lattices which only converges to a guaranteed least fixpoint in bounded time under some additional conditions on F — namely non-expansiveness — playing the same role as convexity in the traditional Newton's method. The main disadvantage of PI is that if the program is not non-expansive, there is no guarantee about the minimality of the found fixpoint. Both techniques can be adapted to be used with most of the abstract domains aforementioned: it suffices to re-define the lattice's union and intersection operators. The alternative approach proposed in this paper finds a guaranteed least fixpoint in (exponentially) bounded time, at least for programs involving integer arithmetic. Although its applicability is currently limited to the abstract domain of intervals, we hope to extend this approach to other domains. We use the Mathematical Programming (MP) language in order to describe the smallest interval feasible in (1) and employ a Branch-and-Bound algorithm to solve this MP to optimality. Depending on the operations carried out in the computer program being analyzed, our MP may be a Mixed-Integer Linear Program (MILP) or a Mixed-Integer Nonlinear Program (MINLP). Typical techniques for finding optimal solutions of MILPs and MINLPs are based on the Branch-and-Bound (BB) algorithm [1]. In the case of MILPs, the BB algorithm yields finite convergence to an exact optimum [27]. In the case of MINLPs, in general, the spatial Branch-and-Bound (sBB) algorithm yields finite convergence only to an ϵ-approximate optimum (for a given $\epsilon > 0$) [26]. As long as the computer program is limited to integer affine arithmetic, we get a MILP that we can solve exactly. If the computer program has integer non-affine arithmetic, we get a MINLP that we can reformulate exactly to a MILP [19, 20]. If the computer program has floating point affine arithmetic, we can derive a MILP whose optimal solution in principle exactly encodes the least fixpoint of (1), but whose numeric parameters are so badly scaled that only a good approximation can be provided. In the remaining case (floating-point non-affine arithmetic) we can only provide an approximation. In this paper we only explicitly discuss the case of integer affine arithmetics.

We remark that optimization techniques were previously employed in software verification [5, 22, 23] but in different contexts. In [5], the analysis addresses semi-algebraic programs of the form `while` B `do` C `od` where B is a boolean condition and C is an imperative command and aims to establish loop termination and (not necessarily smallest) invariants, which are found using Lagrangian relaxation semidefinite programming. In [22], a program is seen as a dynamical system: its termination and boundedness prop-

erties are inferred from appropriate Lyapunov functions, which are approximated via finite sequences of values found using convex optimization techniques. In [23], mathematical programming type constraints are proposed to model several arithmetic operators; the union and intersection operators, however, necessary to model (1), are not treated.

This paper makes two original contributions. Most importantly, it provides a new method for solving the semantic equations (1) that finds a provably optimal solution in bounded time, at least for programs with integer affine arithmetic, and approximate solutions for floating point and/or non-affine arithmetic. Secondly, our MP is built from the computer program's syntactical elements via a first reformulation to a flowchart-like computational model and a second reformulation to semantic equations. This is in contrast with respect to the more usual, static way of presenting a MP: it is interesting that a static set of declarations (the MP) can capture the semantics of a dynamic computer program cast in an imperative language.

The rest of this paper is organized as follows. In Sect. 2 we introduce a graph-based syntax (close to flowcharts) for computer programs. This syntax allows us to derive semantic equations where the right hand side involves at most one operator. In Sect. 3 we present an MP formulation whose constraints model a relaxation of the semantic equations applied to interval domains, and whose objective function identify the least fixpoint of (1). In Sect. 4 we discuss the solution techniques employed and in Sect. 5 we present our computational results. Sect. 6 concludes the paper.

2 A graph model for the semantic equations

Abstract interpretation provides a way to represent the action of a computer program on the domain X of the program variables as a function F acting on X. After the application of the program, the domain becomes $F(X)$. This means that domains with the property that $X = F(X)$ are invariant with respect to the action of the computer program: these are precisely the fixpoints of F. The main interest here is finding the smallest fixpoints with respect to set inclusion, as they represent the minimal set of values that the program variables may take after any run of the computer program. The semantic equations (1) provide a representation of the semantics of a given computer program. Deriving (1) explicitly from a piece of code can be done directly by using the formal grammar that generates the language used to write the code (e.g. by means of such well-established tools as LEX and YACC [17]). Since the intended readership of this paper is likely to be more literate on graph theory than formal languages, we are first going to encode the concrete semantics of a program in a flowchart-like graph [3, 15], and then assign an abstract semantics X to the arcs of this graph.

In [6], a computer program with n program variables $\mathtt{x} = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ is represented by a directed graph $G = (V, A)$ (which we call the *program graph*) where V is a set of control points in the program and the arcs describe the flow through such control points. Let $m = |A|$ and $\ell(v) \in \{\mathrm{E}, \mathrm{S}, \mathrm{T}, \mathrm{J}, \mathrm{X}\}$ be the label of the node v, for any $v \in V$. The labels respectively stand for *entry*, *assignment*, *test*, *junction* and *exit* and are defined as follows:

$$\ell(v) = \mathrm{E} \quad \leftrightarrow \quad |\delta^-(v)| = 0 \wedge |\delta^+(v)| = 1 \tag{2}$$

$$\ell(v) = \mathrm{S} \quad \leftrightarrow \quad |\delta^-(v)| = 1 \wedge |\delta^+(v)| = 1 \tag{3}$$

$$\ell(v) = \mathrm{T} \quad \leftrightarrow \quad |\delta^-(v)| = 1 \wedge |\delta^+(v)| = 2 \tag{4}$$

$$\ell(v) = \mathrm{J} \quad \leftrightarrow \quad |\delta^-(v)| > 1 \wedge |\delta^+(v)| = 1$$
$$\wedge \forall u \in \delta^-(v)\ (\ell(u) \neq \mathrm{J}) \tag{5}$$

$$\ell(v) = \mathrm{X} \quad \leftrightarrow \quad |\delta^-(v)| = 1 \wedge |\delta^+(v)| = 0, \tag{6}$$

where $\delta^-(v) = \{u \in V \mid (u, v) \in A\}$ and $\delta^+(v) = \{w \in V \mid (v, w) \in A\}$. The full framework of abstract interpretation [6, 7] allows many types of code analyses (flow analysis, performance analysis and so on). We are focusing here on the static determination of the domains of program variables. Because of this

scope limitation, we can drop some of the formalisms of [6] and only consider program variable domains on the arcs A.

Let X be the domain of the program variables, extended with infinity values (e.g. for a program with two 4-bytes non-negative integer variables, $X = \{0, \ldots, 2^{32} - 1, \pm\infty\}^2$). For all $(u, v) \in A$ let $X_{(uv)}$ be the domain of the program variables on the arc (u, v). Depending on the context we would also write X_i to indicate the domain of the program variables on the arc $i \leq m$. To all nodes $u \in V$ with $\ell(u) = \mathrm{S}$ (assignment nodes) we associate a function $\phi_u : X \to X$ such that, if the program variables `x` have values $\bar{x}$ at u and $\delta^+(u) = \{v\}$, then `x` have values $\phi_u(\bar{x})$ at v (we remark that we limit assignments to arithmetic functions, all of them extended to deal with infinity values). For all nodes $u \in V$ with $\ell(u) = \mathrm{T}$ (*test* nodes) we consider a set $\tau_u \subseteq X$ such that, for given values $\bar{x}$ of the program variables `x`, the test is true if $\bar{x} \in \tau_u$ and false otherwise. We can take the following statements as axioms describing the semantics of the program.

1. For a node $v \in V$ such that $\ell(v) = \mathrm{S}$, $\delta^-(v) = \{u\}$ and $\delta^+(v) = \{w\}$ we have $X_{(vw)} = \phi_v(X_{(uv)}) = \bigcup_{\bar{x} \in X_{(uv)}} \phi_u(\bar{x})$.
2. For a node $v \in V$ such that $\ell(v) = \mathrm{T}$, $\delta^-(v) = \{u\}$ and $\delta^+(v) = \{w_1, w_2\}$ we have $X_{(vw_1)} = X_{(uv)} \cap \tau_v$ and $X_{(vw_2)} = X_{uv} \cap (X \setminus \tau_v)$.
3. For every node $v \in V$ with $\ell(v)$ not in $\{\mathrm{S}, \mathrm{T}\}$ we have $\bigcup_{u \in \delta^-(v)} X_{(uv)} = \bigcup_{w \in \delta^+(v)} X_{(vw)}$.

By using (3)-(6) and by Axioms 1-3, for all $(u, v) \in A$ it is possible to write the following equations describing the action of the program on the domain variables on (u, v):

$$X_{(uv)} = f^-_{uv}(\{X_{(tu)} \mid t \in \delta^-(u)\}) \quad (7)$$

$$X_{(uv)} = f^+_{uv}(\{X_{(vw)} \mid w \in \delta^+(v)\}), \quad (8)$$

where f^-_{uv}, f^+_{uv} are functions whose exact form is derived from Axioms 1-3 and involves operators/functions/sets $\cap, \cup, \phi, \tau$. Eq. (7)-(8) provide the explicit form of the semantic equations (1). See Fig. 1 for a simple, worked out example that appears in most of the abstract interpretation literature.

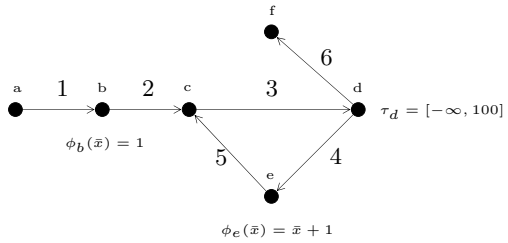
Eq. (5) says that junction nodes cannot have other junction nodes as immediate predecessors. The reason for this choice is that it reduces the number of nodes, as two successive junction nodes $(j_1, j_2) \in A$ can be contracted into a single junction node j without changing the program semantics (Fig. 2). On the other hand, this implies the existence of in-stars $\delta^-(v)$ of various cardinalities. The *contraction operation* mentioned above, however, can be reversed: a single junction node j with $\delta^-(j) = \{v_1, \ldots, v_k\}$ can be replaced by a sequence of $k-1$ junction nodes $\{j_1, \ldots, j_{k-1}\}$ by adding arcs (j_{i-1}, j_i) for all $2 \leq i \leq k-1$, (v_{i+1}, j_i) for all $1 \leq i \leq k-1$, and (v_1, j_1) (Fig. 2). In the sequel, we employ the following, modified version of (5):

$$\forall v \in V \quad \ell(v) = \mathrm{J} \leftrightarrow |\delta^-(v)| = 2 \wedge |\delta^+(v)| = 1, \quad (9)$$

because in the corresponding semantic equations (1) the union operator always has at most two operands. We call a program graph defined as per (2)-(6) a *contracted graph*, whereas the program graph obtained by replacing (5) with (9) an *expanded graph*. It is easy to establish that in a program graph such that (2)-(4), (6), (9) hold, the semantic equations (1) involving the union operator all have the form $X_{(vw)} = X_{(u_1 v)} \cup X_{(u_2 v)}$ for nodes $u_1, u_2, v, w \in V$. Furthermore, an expanded graph $G = (V, A)$ and a contracted graph $G' = (V', A')$ for the same program give rise to sets of semantic equations having essentially the same set of solutions.

Next, we remark that all tests can be written in the form (variable $\in$ interval). Expressions $f_1 \wedge f_2$ appearing in tests can be replaced by sequences of adjacent test nodes representing "`if` f_1 `then if` f_2 `then`...". Expressions $f_1 \vee f_2$ appearing in tests can be rewritten as "`if` f_1 `then` p `endif; if` f_2 `then if` $\neg f_1$ `then` p `endif`", where p is a set of instructions of the program to be executed only if $f_1 \vee f_2$

```
                        //a: l(a) = E
int x = 1;              //b: l(b) = S
                        //c: l(c) = J
while(x <= 100){        //d: l(d) = T
  x = x + 1;            //e: l(e) = S
}
                        //f: l(f) = X
```



$$
\begin{aligned}
X_1 &= \mathrm{Id}(\texttt{input}) \\
X_2 &= \phi_b(X_1) \\
X_3 &= X_2 \cup X_5 \\
X_4 &= X_3 \cap \tau_d \\
X_5 &= \phi_e(X_4) \\
X_6 &= X_3 \cap (X \setminus \tau_d).
\end{aligned}
$$

$$
\begin{aligned}
X_1 &= [-\infty, \infty] \\
X_2 &= [1, 1] \\
X_3 &= [1, 1] \cup X_5 \\
X_4 &= X_3 \cap [-\infty, 100] \\
X_5 &= X_4 + [1, 1] \\
X_6 &= X_3 \cap [101, \infty].
\end{aligned}
$$

Figure 1: The program graph and semantic equations worked out for a simple but classic example.

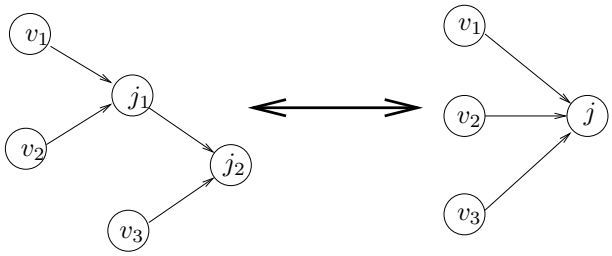


Figure 2: Expansion and contraction of junction nodes.

holds. This program transformation can be applied recursively so that all tests are of the form f or $\neg f$ where f are boolean conditions not including $\wedge$ and $\vee$, i.e. they have the form $(f(\mathtt{x}) \in \tau)$, where $\mathtt{x}$ are program variables, $f : \mathbb{R}^n \to \mathbb{R}$ and $\tau = [\tau^L, \tau^U]$ is an interval. Negated conditions $f(\mathtt{x}) \notin \tau$ can be replaced by pairs $f(\mathtt{x}) \leq \tau^L \vee f(\mathtt{x}) \geq \tau^U$, which we know how to rewrite eliminating the $\vee$ operator. Tests such as $f(\mathtt{x}) \in \tau$ can be rewritten by adding a new program variable $\mathtt{y}_f$ and an assignment node $\mathtt{y}_f = f(\mathtt{x})$ to the program, so that the test condition becomes $\mathtt{y}_f \in \tau$. Finally, we can break up complex assignment nodes $\mathtt{x}_i = f(\mathtt{x})$ (where f is a mathematical expression involving arithmetic operators) into a list of assignment nodes involving at most one operator on the right hand side by means of introducing added program variables and assignment nodes. This is similar to the standard MINLP form described in [24, 18].

This yields a rewriting of a program P into a program P' with program variables $\mathtt{x}, \mathtt{y}$ such that the smallest fixpoint (X^*, Y^*) of the semantic equations for P' has the property that X^* is the smallest fixpoint of the semantic equations for P. The argument (technical but not difficult) is based on using the added assignment nodes in P' in order to eliminate Y from the semantic equations for P' and obtain those for P. The relevance of this rewriting is that we can assume that the semantic equations only involve unary and binary operators on the variable domains. Given a program graph $G = (V, A)$, the

semantic equations (1) are:

$$\forall i \leq m \quad X_i = \begin{cases} F_i(X_{i_1}, X_{i_2}) & F_i \in \mathbb{O}_b \\ F_i(X_{i_1}) & F_i \in \mathbb{O}_u \\ F_i & F_i \in \mathbb{O}_0 \end{cases} \tag{10}$$

where $\mathbb{O}_0$ is the set of 0-ary operators (or constants), $\mathbb{O}_u = \{\mathrm{Id}, c\times, \cap\tau\}$ is the set of unary operators (Id being the identity, $c\times$ the multiplication by a constant, and $\cap\tau$ the intersection with a constant interval) and $\mathbb{O}_b = \{+, \cup\}$ is the set of binary operators. We shall assume that for all considered domains (type of sets assigned to the X symbols) we have $X_i - X_j = X_i + (-1) \times X_j$: this is certainly true for intervals. We note that we need Id because it is the operator assigned to the arc incident to the entry nodes, and if the program graph represents a function in a program, the first arc actually "copies" the values passed as function arguments.

One last important remark: every arc $i \leq m$ of the program graph encodes the changes carried out to *all* program variables $\mathtt{x} = (\mathtt{x}_1, \ldots, \mathtt{x}_n)$. Thus, each X_i is a sequence of sets $(X_{i1}, \ldots, X_{in})$. If the operator $\otimes_i$ on the arc i only changes the value of variable $\mathtt{x}_j$, it is taken for granted that $\otimes_i$ acts as the identity for all other program variables $\mathtt{x}_k$ with $k \neq j$. Hence, X is in fact an $m \times n$ rectangular array where X_{ij} is a set for all $i \leq m, j \leq n$.

3 Mathematical programming formulation

In this section, we assume that the domain type of the abstract semantics is $\mathbb{I}$, the set of all closed intervals in $\mathbb{R} \cup \{\pm\infty\}$. Since with the syntax given in Sect. 2 each semantic equation (10) only involves one operator that only acts on one single program variable, all the other program variables being fixed, it suffices to exhibit MP constraints defining the behaviour of the following operators:

1. *constant*: $Z = [a^L, a^U]$ (where $a^L \leq a^U \in \mathbb{R} \cup \{\pm\infty\}$);
2. *identity*: $Z = \mathrm{Id}(R)$;
3. *sum*: $Z = R + S$;
4. *constant multiplication*: $Z = c \times R$ (where $c \in \mathbb{R}$);
5. *union*: $Z = R \cup S$;
6. *intersection*: $Z = R \cap \tau$ (where $\tau \in \mathbb{I}$),

where $R = [r^L, r^U], S = [s^L, s^U], Z = [z^L, z^U]$ are intervals in $\mathbb{I}$. Recall that $\cup$ is involved in loops, $\cap\tau$ in tests, Id in entry/exit nodes and the other mathematical operators in assignments. The arithmetic operators follow the definitions of interval arithmetic and are naturally extended to deal with the infinity values $\{-\infty, +\infty\}$. Finally, we define $R \cup S$ to be the smallest interval in $\mathbb{I}$ containing both R, S setwise.

We first remark that we can relax set equality with set inclusion provided we minimize with respect to the interval width $|Z| = z^U - z^L$. More precisely, as long as R, S are not empty,

$$\begin{aligned} \forall \otimes \in \mathbb{O}_0 \quad \mathrm{argmin}\{|Z| \mid Z \supseteq \otimes\} &= \{Z \in \mathbb{I} \mid Z = \otimes\} \\ \forall \otimes \in \mathbb{O}_u \quad \mathrm{argmin}\{|Z| \mid Z \supseteq \otimes R\} &= \{Z \in \mathbb{I} \mid Z = \otimes R\} \\ \forall \otimes \in \mathbb{O}_b \quad \mathrm{argmin}\{|Z| \mid Z \supseteq R \otimes S\} &= \{Z \in \mathbb{I} \mid Z = R \otimes S\}. \end{aligned}$$

The provided MP constraints will therefore actually model a *post-fixpoint* $\mathbf{Z} \in \mathbb{I}^m$, i.e. an interval vector $\mathbf{Z}$ such that $\mathbf{Z} \supseteq F(\mathbf{Z})$, equality being enforced by the minimization of the interval widths.

Observe that $\mathbb{I}$ is not closed under $\cap\tau$ since $\emptyset \notin \mathbb{I}$. Indeed the behaviour of the empty set w.r.t. the operators in $\mathbb{O}_u \cup \mathbb{O}_b$ is not the same as other intervals: $\mathrm{Id}\emptyset = c \times \emptyset = \emptyset \cap \tau = \emptyset$, and $\emptyset \cup S = S$, $R \cup \emptyset = R$,

$\emptyset + S = R + \emptyset = \emptyset$. There is no interval in $\mathbb{I}$ with such properties: thus, the empty interval cannot be represented in the form $[z^L, z^U]$. We therefore introduce, for each interval Z, a binary variable $\bar{z}$ that has value 1 if and only if Z is empty:

$$Z = \begin{cases} [z^L, z^U] & \text{if} \quad \bar{z} = 0 \\ \emptyset & \text{if} \quad \bar{z} = 1. \end{cases} \tag{11}$$

Equivalently, each interval $Z \in \mathbb{I}$ is represented by a triplet $(z^L, z^U, \bar{z}) \in \mathbb{R}^2 \times \{0, 1\}$ such that if $\bar{z} = 1$ the MP constraints enforce the empty interval behaviour and by convention, $z^L = z^U = 0$.

In all cases described in this section, the definition of interval must be enforced:

$$z^L \leq z^U, \qquad r^L \leq r^U, \qquad s^L \leq s^U. \tag{12}$$

In the following, we will choose a suitable constant to represent the infinity value, exploiting the fact that practical computers are equivalent to *bounded* Turing machines rather than truly universal Turing machines: commonly available implementations of the floating point number field $\mathbb{FP}$ are such that $\exists M \in \mathbb{R}\ \forall r \in \mathbb{FP}\ (|r| \leq M)$ and $\exists \varepsilon > 0\ \forall r, s \in \mathbb{FP}\ (r \neq s \rightarrow |r - s| > \varepsilon)$. Therefore, all the intervals are considered bounded:

$$-M \leq z^L, z^U \leq M \tag{13}$$

and the assignment $z = M$ is read as $z = \infty$. Notationwise, for a set $\Theta(x_1, \ldots, x_p) \subseteq \mathbb{R}^p$ and a (sub)list of coordinate directions $x' = (x_{i_1}, \ldots, x_{i_q})$, we indicate by $\text{proj}(\Theta, x')$ the projection of Θ on the subspace spanned by unit vectors in the coordinate directions listed in x'. Most of the arguments below follow from interval analysis [14].

3.1 Constant and Identity

A 0-ary operator $\otimes$ is a constant. In the class of intervals, it means a constant interval $\tau = [\tau^L, \tau^U]$ appears in the semantic equations (10). The constraints for $Z \supseteq \tau$ are:

$$z^L \leq \tau^L \tag{14}$$
$$z^U \geq \tau^U \tag{15}$$
$$\bar{z} = 0. \tag{16}$$

The only point worth emphasizing is that in case an interval constant appears explicitly in the semantic equations, we assume the interval to be non-empty (there exist no valid assignment imperative statement in the reference programming language corresponding to the constant assignment $Z = \emptyset$). Thus, there are some intervals Z for which $\bar{z}$ is forced to be zero: without (16), the objective function direction (see (56) below) would force all such variables to be set at 1, which would cause all the relevant constraints to be inactive, which in turn would yield the empty set as the (invalid) least fixpoint of (10).
The unary operator $\text{Id}(R)$ is modeled by slightly modifying constraints (14)-(16) as follows:

$$z^L \leq r^L + 2M\bar{z} \tag{17}$$
$$z^U \geq r^U - 2M\bar{z} \tag{18}$$
$$\bar{z} = \bar{r}. \tag{19}$$

3.2 Sum

In interval arithmetic, the sum of two non-empty intervals R, S is the interval $Z = [r^L + s^L, r^U + s^U]$. In order to extend the semantic of the sum operator to the set of closed intervals in $\mathbb{R} \cup \{\pm\infty\}$, we introduce the following binary variables and constraints:

- $z_+^{LR} = 1$ if and only if $r^L > -\infty$;

- $z_+^{UR} = 1$ if and only if $r^U < +\infty$;
- $z_+^{LS} = 1$ if and only if $s^L > -\infty$;
- $z_+^{US} = 1$ if and only if $s^U < +\infty$;
- $z_+^{L} = 1$ if $r^L = -\infty$ or $s^L = -\infty$;
- $z_+^{U} = 1$ if $r^U = +\infty$ or $s^U = +\infty$.

$$
\begin{aligned}
\epsilon - M(3 - 2z_+^{LR}) &\leq r^L \leq & M(2z_+^{LR} - 1) && (20)\\
M(1 - 2z_+^{UR}) &\leq r^U \leq & M(3 - 2z_+^{UR}) - \epsilon && (21)\\
\epsilon - M(3 - 2z_+^{LS}) &\leq s^L \leq & M(2z_+^{LS} - 1) && (22)\\
M(1 - 2z_+^{US}) &\leq s^U \leq & M(3 - 2z_+^{US}) - \epsilon && (23)\\
2z_+^{L} &\geq & 2 - z_+^{LR} - z_+^{LS} && (24)\\
2z_+^{U} &\geq & 2 - z_+^{UR} - z_+^{US} && (25)\\
z^L &\leq & (r^L + s^L)(1 - z_+^L) - Mz_+^L + 2M\bar{z} && (26)\\
z^U &\geq & (r^U + s^U)(1 - z_+^U) + Mz_+^U - 2M\bar{z} && (27)\\
2\bar{z} &\geq & \bar{r} + \bar{s} && (28)\\
\bar{z} &\leq & \bar{r} + \bar{s}. && (29)
\end{aligned}
$$

3.1 Lemma
Let $\Theta = \{(z^L, z^U, r^L, r^U, s^L, s^U, z_+^{LR}, z_+^{UR}, z_+^{LS}, z_+^{US}, z_+^L, z_+^U, \bar{z}, \bar{r}, \bar{s}) \mid$ *(12), (13), (20)-(29)*$\}$. *Then, if* $R, S \neq \emptyset$, $Z = proj(\Theta, (z^L, z^U)) \supseteq R + S$; *otherwise* $\bar{z} = 1$.

Proof. Assume $R, S \neq \emptyset$ and let $z \in R + S$. $\bar{r} = \bar{s} = 0$ implies by (29) $\bar{z} = 0$. It is easy to see that Constraints (20)-(23) properly set variables z_+^{LR}, z_+^{UR}, z_+^{LS} and z_+^{US}. Constraint (24) enforces $z_+^L = 1$ as soon as z_+^{LR}, z_+^{LS} or both are set to 0. Analogously, Constraints (25) set $z_+^U = 1$ if z_+^{UR}, z_+^{US} or both are set to 0. Constraint (26) sets z^L to a value lower than or equal to either $(r^L + s^L)$, if both r^L and s^L are not infinity, i.e., if $z_+^L = 0$, or $-M$ otherwise. Constraint (27) is the analog of (26) for the variable z^U. Then, by (26) and (27), we have $z^L \leq r^L + s^L$ and $z^U \geq r^U + s^U$, which implies $Z \supseteq R + S$.

If one of both operands R and S are empty, i.e., if either $\bar{r} = 1$ or $\bar{s} = 1$, than (28) implies $\bar{z} = 1$. As a consequence, Constraints (26) and (27) are inactive at all and variables z^L and z^U are free. □

Observe that Constraints (26) and (27) are needed to guarantee model feasibility since they correctly allow the operations $r^U + M = M$ and $r^L - M = -M$. Moreover, it is easy to provide cases having least fixpoints with at least one interval that diverges to infinity.

3.3 Constant multiplication

The modeling of the $c\times$ operator is very similar to that described for the sum. Again, the main issue concerns the extension of the semantic to the closed intervals in $\mathbb{R} \cup \{\pm\infty\}$ but now we need of less variables and constraints since $c\times$ is a unary operator. Assume $c \geq 0$ and consider the following binary variable and constraints:

- $z_\times^L = 1$ if and only if $r^L > -\infty$;
- $z_\times^U = 1$ if and only if $r^U < +\infty$;

$$\begin{aligned}
\epsilon - M(3 - 2z_\times^L) &\leq r^L \leq & M(2z_\times^L - 1) && (30)\\
M(1 - 2z_\times^U) &\leq r^U \leq & M(3 - 2z_\times^U) - \epsilon && (31)\\
z^L &\leq & cr^L(1 - z_\times^L) - Mz_\times^L + 2M\bar{z} && (32)\\
z^U &\geq & cr^U(1 - z_\times^U) + Mz_\times^U - 2M\bar{z} && (33)\\
\bar{z} &= & \bar{r}. && (34)
\end{aligned}$$

The case $c < 0$ can be easily addressed by replacing R with $R' = [r^U, r^L]$.

3.4 Union

In interval analysis, the union of two non-empty intervals R, S is the convex hull of the setwise union $R \cup S$ of the two intervals, i.e. the smallest interval containing both:

$$R \cup S = [\min(r^L, s^L), \max(r^U, s^U)].$$

If $R = \emptyset$, then $R \cup S = S$; if $S = \emptyset$, $R \cup S = R$. If both R, S are empty, $R \cup S = \emptyset$. The following constraints correctly model the $\cup$ operator:

$$\begin{aligned}
z^L &\leq r^L + 2M\bar{z} && (35)\\
z^L &\leq s^L + 2M\bar{z} && (36)\\
z^U &\geq r^U - 2M\bar{z} && (37)\\
z^U &\geq s^U - 2M\bar{z} && (38)\\
\bar{z} &\geq \bar{r} + \bar{s} - 1 && (39)\\
2\bar{z} &\leq \bar{r} + \bar{s}. && (40)
\end{aligned}$$

3.2 Lemma
Let $\Theta = \{(z^L, z^U, r^L, r^U, s^L, s^U, \bar{z}, \bar{r}, \bar{s}) \mid$ (12), (13), (35)-(40)$\}$. Then, if either R or S is not empty (or neither is), $Z = proj(\Theta, (z^L, z^U)) \supseteq R \cup S$; otherwise, if $R = S = \emptyset$, $\bar{z} = 1$.

Proof. By (39) and (40), $\bar{z} = 1$ if and only if $\bar{r} = \bar{s} = 1$, i.e., if and only if R, S are both empty. In that case, constraints (35)-(38) are all inactive. Otherwise, by (35)-(36) and (37)-(38), $\bar{z} = 0$ implies $z^L \leq \min(r^L, s^L)$ and $z^L \geq \max(r^L, s^L)$, so $z \in Z$. □

3.5 Intersection

The intersection $Z = [z^L, z^U]$ of two non-empty intervals $R = [r^L, r^U]$, $\tau = [\tau^L, \tau^U]$ is defined as follows (see Fig. 3):

1. if $r^U < \tau^L$ then $Z = \emptyset$;
2. if $r^L > \tau^U$ then $Z = \emptyset$;
3. if $r^L \leq \tau^L$ and $r^U \leq \tau^U$ and $r^U \geq \tau^L$ then $Z = [\tau^L, r^U]$;
4. if $r^L \leq \tau^L$ and $r^U \geq \tau^U$ then $Z = [\tau^L, \tau^U]$;
5. if $r^L \geq \tau^L$ and $r^U \leq \tau^U$ then $Z = [r^L, r^U]$;
6. if $r^L \geq \tau^L$ and $r^U \geq \tau^U$ and $r^L \leq \tau^U$ then $Z = [r^L, \tau^U]$;

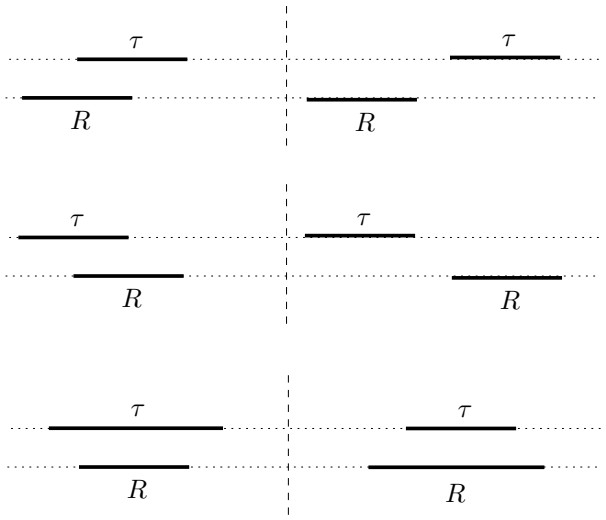


Figure 3: Six possibilities for interval intersection.

Moreover Z is empty if at least one between R and τ is empty.

The intersection $R \cap \tau$ can be modeled by the following binary variables and constraints:

- $z_{\cap}^{UL} = 1$ if and only if $r^U < \tau^L$ (case 1.);
- $z_{\cap}^{LU} = 1$ if and only if $r^L > \tau^U$ (case 2.);
- $z_{\cap}^{L\tau} = 1$ if and only if $z^L = \tau^L$ (cases 3. or 4.);
- $z_{\cap}^{LR} = 1$ if and only if $z^L = r^L$ (cases 5. or 6.);
- $z_{\cap}^{UR} = 1$ if and only if $z^U = r^U$ (cases 3. or 5.);
- $z_{\cap}^{U\tau} = 1$ if and only if $z^U = \tau^U$ (cases 4. or 6.).

$$
\begin{aligned}
(1 - z_{\cap}^{UL})(\tau^L - r^U) &\leq 0 && (41)\\
z_{\cap}^{UL}(\tau^L - r^U - \epsilon) &\geq 0 && (42)\\
(1 - z_{\cap}^{LU})(r^L - \tau^U) &\leq 0 && (43)\\
z_{\cap}^{LU}(r^L - \tau^U - \epsilon) &\geq 0 && (44)\\
\bar{r} + \bar{\tau} &\leq 2\bar{z} && (45)\\
z_{\cap}^{UL} + z_{\cap}^{LU} &\leq 2\bar{z} && (46)\\
z_{\cap}^{UL} + z_{\cap}^{LU} + \bar{r} + \bar{\tau} &\geq \bar{z} && (47)\\
z_{\cap}^{L\tau} + z_{\cap}^{LR} + \bar{z} &= 1 && (48)\\
z_{\cap}^{UR} + z_{\cap}^{U\tau} + \bar{z} &= 1 && (49)\\
z_{\cap}^{L\tau}\left(z^L - \tau^L\right) &= 0 && (50)\\
z_{\cap}^{LR}\left(z^L - r^L\right) &= 0 && (51)\\
z_{\cap}^{U\tau}\left(z^U - \tau^U\right) &= 0 && (52)\\
z_{\cap}^{UR}\left(z^U - r^U\right) &= 0. && (53)
\end{aligned}
$$

Variables $z_{\cap}^{UL}$ and $z_{\cap}^{LU}$ are needed to model the empty intersection. It is immediate to see that (41)-(42) defines $z_{\cap}^{UL}$ and (43)-(44) defines $z_{\cap}^{LU}$. Constraint (45) ensures that Z will set to empty if at least one of the intervals R and τ is empty, whereas constraint (46) models the cases 1. and 2. Finally, constraint (47) enforces Z to be not empty when R or τ are not empty or when cases 1. and 2. do not occur.

Variables $z_{\cap}^{L\tau}$ and $z_{\cap}^{LR}$ and constraints (50) and (51) assign to the lower endpoint of Z one lower endpoint between τ^L and x^L . Similarly, the upper endpoint of Z is modeled by variables $z_{\cap}^{U\tau}$ and $z_{\cap}^{UR}$ and by constraints (52) and (53). Clearly, $z_{\cap}^{L\tau}$ and $z_{\cap}^{LR}$ as well as $z_{\cap}^{U\tau}$ and $z_{\cap}^{UR}$ are mutually exclusive, as imposed by constraints (48) and (49).

Observe that if $\bar{z}$ is set to one then all the variables $z_{\cap}^{L\tau}$, $z_{\cap}^{LR}$, $z_{\cap}^{U\tau}$ and $z_{\cap}^{UR}$ are forced to be zero by Constraints (48) and (49) and, therefore, variables z^L and z^U are free.

3.3 Lemma
Let $\Theta = \{(z^L, z^U, r^L, r^U, \tau^L, \tau^U, z_{\cap}^{LU}, z_{\cap}^{UL}, z_{\cap}^{L\tau}, z_{\cap}^{LR}, z_{\cap}^{U\tau}, z_{\cap}^{UR}, \bar{z}, \bar{r}, \bar{\tau}) \mid$ *(12), (13), (41)-(53)*$\}$. *Then* $Z = proj(\Theta, (z^L, z^U))$ *contains* $R \cap \tau$, *otherwise* $\bar{z} = 1$.

Proof. Suppose $R \neq \emptyset$ and $\tau \neq \emptyset$, i.e., $\bar{r} = \bar{\tau} = 0$ and (45) inactive.

Case 1: by (41)-(42), $r^U < \tau^L$ implies $z_{\cap}^{UL} = 1$ and, by (46), $\bar{z} = 1$. Constraints (48) and (49) then enforce $z_{\cap}^{L\tau} = z_{\cap}^{LR} = z_{\cap}^{U\tau} = z_{\cap}^{UR} = 0$ and therefore (50)-(53) are all satisfied for any value of z^L and z^U. Finally, by (12), $r^L \leq \tau^U$ and hence $z_{\cap}^{LU} = 0$ by (43) and (44).
Case 2: by (43) and (44), $\tau^U < r^L$ implies $z_{\cap}^{LU} = 1$ and, by (46), $\bar{z} = 1$. As in the previous case, constraints (48) and (49) enforce $z_{\cap}^{L\tau} = z_{\cap}^{LR} = z_{\cap}^{U\tau} = z_{\cap}^{UR} = 0$ and therefore (50)-(53) are all satisfied for any value of z^L and z^U. Since $\tau^L \leq r^U$ is implied by (12), $z_{\cap}^{UL} = 0$ is enforced by (41) and (42).
Case 3: $r^U \geq \tau^L$, (41) and (42) imply $z_{\cap}^{UL} = 0$, whereas $z_{\cap}^{LU} = 0$ by (12), (43) and (44). The setting $z_{\cap}^{UL} = z_{\cap}^{LU} = 0$ makes (46) inactive and enforces $\bar{z} = 0$ through (47). By (48), exactly one between $z_{\cap}^{L\tau} = 1$ and $z_{\cap}^{LR} = 1$ must occur. In the former case $z^L = \tau^L$ by (50); in the latter case $z^L = r^L$ by (51). Analogously, By (49), exactly one between $z_{\cap}^{U\tau} = 1$ and $z_{\cap}^{UR} = 1$ must occur. In the former case $z^U = \tau^U$ by (52); in the latter case $z^U = r^U$ by (53). In any case, Z contains $R \cap \tau$.
Case 4 and 5: (12), (41) and (42) enforce $z_{\cap}^{UL} = 0$, whereas (12), (43) and (44) enforce $z_{\cap}^{LU} = 0$. So far the cases boil down to Case 3.
Case 6: $r^L \leq \tau^U$, (43) and (44) imply $z_{\cap}^{LU} = 0$, whereas $z_{\cap}^{UL} = 0$ by (12), (41) and (42). So far the case boils down to Case 3.

Now suppose $R = \emptyset$ or $\tau = \emptyset$, i.e., $\bar{r} = 1$ or $\bar{\tau} = 1$. Constraints (45) and (47) enforce $\bar{z} = 1$ and make (46) inactive. $z_{\cap}^{L\tau} = z_{\cap}^{LR} = z_{\cap}^{U\tau} = z_{\cap}^{UR} = 0$ by constraints (48) and (49) and therefore (50)-(53) are all satisfied for any value of z^L and z^U. □

3.6 Dealing with dead code

We now consider (10) where $X_i = (X_{i1}, \ldots, X_{in})$, with $X_{ij} \in \mathbb{I}(M)$ for all $i \leq m$ and $j \leq n$. The empty interval needs special handling whenever it appears in unreachable segments of code: $X_{ij} = \emptyset$ in fact implies that the j-th variable cannot take any value at control point i or, equivalently, it signals that the control point i is never executed for any given program input. Clearly, if the control point i is unreachable, no variable can ever take a value at i. Thus, $X_{ij} = \emptyset$ implies $X_{ik} = \emptyset$ for any $k \neq j$. This behaviour however is not correctly modeled by the constraints given above, as shown in the following example:

```
int x1,x2;      //(1)
x1 = 1;         //(2)
if (x1 == 0){   //(3)
    x2 = 4;     //(4)
} else {        //(5)
    x2 = 0;     //(6)
}
//(7)
```

$$
\begin{array}{lcl}
(X_{11}, X_{12}) & = & [-\infty, \infty], [-\infty, \infty] \\
(X_{21}, X_{22}) & = & [1,1], [-\infty, \infty] \\
(X_{31}, X_{32}) & = & [0,0] \cap X_{21}, X_{22} \\
(X_{41}, X_{42}) & = & X_{31}, [4,4] \\
(X_{51}, X_{52}) & = & [-\infty, \infty] \cap X_{21}, X_{22} \\
(X_{61}, X_{62}) & = & X_{51}, [0,0] \\
(X_{71}, X_{72}) & = & X_{41} \cup X_{61}, X_{42} \cup X_{62}.
\end{array}
$$

The intersection $X_{31} = [0,0] \cap X_{21}$ results in an empty interval indicating that the code `x2 = 4` is unreachable (the conditional statement `x1 == 0` is in fact always `false`). However, the equation $X_{42} = [4,4]$, being modeled by constraints (14)-(16), results in the non-empty interval $[4,4]$. Therefore, $X_{72} = [0,4]$ instead of $X_{72} = [0,0]$.

Due to unreachability, the interval $X_{ij} = R \otimes S$ should be empty even though the constraints given above for $R \otimes S$ would normally not yield the empty interval. Therefore, when $n > 1$ a further binary variable $\hat{x}$ is required in order to distinguish the case $R \otimes S = \emptyset$ from the case when i is unreachable. Thus, we introduce:

$$
\hat{x}_{ij} = \begin{cases} 1 & \text{if} \quad R \otimes S = \emptyset \\ 0 & \text{otherwise,} \end{cases} \tag{54}
$$

replace $\bar{z}$ with $\hat{z}$ in (16), (19), (28), (29), (34), (39), (40), (45), (46) and (47), and add the following linking constraints:

$$
\left.
\begin{array}{rcl}
\forall i \leq m, j \leq n \qquad \bar{x}_{ij} & \leq & n \left(\hat{x}_{ij} + \sum\limits_{\substack{k=1 \\ k \neq j}}^{n} \bar{x}_{ik} \right) \\
\forall i \leq m, j \leq n \qquad \hat{x}_{ij} & \leq & n \bar{x}_{ij} - \sum\limits_{\substack{k=1 \\ k \neq j}}^{n} \bar{x}_{ik} \\
\forall i \leq m \quad \sum\limits_{k=1}^{n} \bar{x}_{ik} & \leq & n \sum\limits_{k=1}^{n} \hat{x}_{ik}.
\end{array}
\right\} \tag{55}
$$

3.7 Putting it all together

Each interval X_{ij} is given as a triplet $(x^L_{ij}, x^U_{ij}, \bar{x}_{ij})$ such that $X_{ij} = [x^L_{ij}, x^U_{ij}] \neq \emptyset$ if and only if $\bar{x}_{ij} = 0$, and $X_{ij} = \emptyset$ if and only if $\bar{x}_{ij} = 1$. We let $x^L_i = (x^L_{i1}, \ldots, x^L_{in})$, $x^U_i = (x^U_{i1}, \ldots, x^U_{in})$ and $\bar{x}_i = (\bar{x}_{i1} \ldots, \bar{x}_{in})$ for all $i \leq m$, and $x^L = (x^L_1 \ldots, x^L_m)$, $x^U = (x^U_1, \ldots, x^U_m)$, $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_m)$. We introduce further binary variables $z_\otimes$ appearing in the sum (labelled z_+ in Sect. 3.2), constant multiplication (labelled $z_\times$ in Sect. 3.3) and intersection operators (labelled $z_\cap$ in Sect. 3.5), and $\hat{x}$ as explained in Sect. 3.6.

For each $i \leq m$, the i-th semantic equation $X_i = F_i(X)$ is replaced by its post-fixpoint relaxation $X_i \supseteq F_i(X)$: we let $g_i(x^L, x^U, \bar{x}, \hat{x}, z_\otimes) \leq 0$ be the constraints (given in Sect. 3.1-3.5) whose feasible region defines precisely all those intervals X such that $X_i \supseteq F_i(X)$.

3.4 Theorem
The least fixpoint of (10) is the optimal solution of the following mathematical program:

$$
\left.
\begin{array}{rcl}
\min \quad \sum\limits_{i \leq m} \sum\limits_{j \leq n} (x^U_{ij} - x^L_{ij} - \bar{x}_{ij}) & & \\
\forall i \leq m \qquad g_i(x^L, x^U, \bar{x}, \hat{x}, z_\otimes) & \leq & 0 \\
-M \leq x^L \leq x^U & \leq & M \\
\bar{x}, \hat{x}, z_\otimes & \in & \{0,1\}.
\end{array}
\right\} \tag{56}
$$

Proof. First note that $\mathbb{I}(M)^{mn}$ is a complete lattice with respect to inclusion, union (defined over intervals as the smallest interval containing both arguments), and intersection.

Claim 1. For all $i \leq m$, F_i is monotone in the interval lattice.

For all $U, W, Y \in \mathbb{I}(M)$ such that $U \subseteq W$, it is easy to establish that $U \cap Y \subseteq W \cap Y$, $U \cup Y \subseteq W \cup Y$, $U + Y \subseteq W + Y$. For these binary operators, monotonicity in the second argument follows by commutativity. For unary operators, $c \times U \subseteq c \times W$ ($c \in \mathbb{R}$) are also easy to establish.

Since F is monotone, it has a unique least fixpoint by Tarski's fixpoint theorem [25]: there is $X \in \mathbb{I}(M)^{mn}$ such that $F(X) = X$. Furthermore, the least fixpoint coincides with the least post-fixpoint of F [25], i.e. $F(X) \subseteq X$. As shown in Sect. 3.1-3.5, the intervals in $\mathbb{I}(M)^{mn}$ that are feasible with respect to the constraints $g_i(x^L, x^U, \bar{x}, \hat{x}, z_\otimes) \leq 0$ are precisely those for which $F_i(X) \subseteq X_i$ for each $i \leq m$.

Claim 2: the objective function, mapping $\mathbb{I}(M)^{mn}$ to $\mathbb{R}$, is strictly increasing.

For non-empty intervals, this is obvious. Otherwise, notice that whenever $\bar{x}_{ij} = 1$ for some $i \leq m, j \leq n$, all the relevant constraints in $g_i \leq 0$ are inactive, save for (12): the objective function direction then ensures that at the optimum, $x^L_{ij} = x^U_{ij}$. Since $\bar{x}_{ij} = 1$ if and only if $X_{ij} = \emptyset$, the value of the objective function when X_{ij} is empty is lower.

Thus, any globally optimal solution of (56) is the least fixpoint of F. □

The theorem above shows that when the domain type for program variables consists of uniformly bounded closed intervals, the least fixpoint of the semantic equations (1) is characterised as the optimal solution of a special mathematical program that is built component-wise (*i.e.* control point by control point) along the abstract semantics of the program.

3.8 Properties of the mathematical program

If only integer arithmetic is used in the computer program, i.e., when all decision variables x^L, x^U are constrained to be integer, the mathematical model (56) can be always written as a MILP. In fact, in the case of affine arithmetic the operators appearing in (10) are only *constant*, Id, $c\times$, $+$, $\cap\tau$ and $\cup$. The associated constraints (see Sect. 3.1-3.5) are all linear except (26)-(27), (32)-(33), (41)-(44) and (50)-(53), exhibiting products between a binary and a continuous (or integer) decision variable. Due their particular form, they admit a well-known exact linear reformulation (in the sense given in [19, 20]).

In the case of non-affine integer arithmetic, additional operators (such as e.g. *inverse*, *constant power* and *product*) must be considered. The constraints needed to model them — which we do not report here — exhibit nonlinear terms, but all the nonlinearities can be reformulated exactly to linear terms via the introduction of added variables and constraints (in particular, by the reformulations Int2Bin, ProdBin, ProdBinCont, AbsDiff and Step 2, p. 188 of [20]). Finally, we can choose $\varepsilon = 1$, which means that the obtained MILP will be well-scaled and practically solvable.

If floating point affine arithmetic is used in the computer program, we get a MILP where ε must be set to the smallest possible floating point number: although the optimal solution of this MILP in principle encodes the guaranteed least fixpoint of (10), practical solution will not be achievable without a possibly large numerical error. In any case, for a suitable choice of ε we still derive an over-approximation of the least fixpoint, which is a valid invariant. If floating point non-affine arithmetic is used, products, inverses and powers of continuous decision variables cannot be reformulated to linear terms exactly as in the integer case. Although the obtained MINLP is badly scaled, its optimal solution encodes the guaranteed least fixpoint of (10). However, in this case even a good choice of ε would yield a MINLP for which we can only hope to find an ϵ-approximate solution at best.

4 Solution techniques

The model developed in Sect. 3 belongs to the family of MILP formulations. These can be solved by employing an exact BB algorithm such as the one implemented in CPLEX [16] or a heuristic such as local branching [10] or feasibility pump [9] if a guarantee of fixpoint minimality is not needed. More precisely, the qualitative difference between an exact and heuristic algorithm, in this context, simply concerns the minimality of objective function. In other words, both approaches provide valid solutions to the semantic equations (1) as these are the constraints of the model. Only the exact algorithm can provide a guarantee of optimality of the solution with respect to inclusion-wise fixpoint minimization.

Our implementation employs a parser to transform a C program into a set of semantic equations (1). These are then automatically transformed into a mathematical program as per Sect. 3 expressed in the AMPL language [11]. The AMPL interpreter is finally instructed to call the CPLEX 11 solver [16], which solves the problem to optimality. As further detailed in Sect. 5, although the worst-case time complexity of any BB algorithm is exponential, the particular form of the mathematical programming formulations obtained by typical C code snippets includes many equality constraints, which usually speed up the solution process considerably (as they may be used to substitute a decision variable out of the model). We were able to ascertain empirically that the practical performance of the proposed solution method mainly depends on the number of intersections present in the semantic equations, i.e., on the number of test nodes present in the program graph. Therefore the BB performance would seem to be mainly linked to the *cyclomatic number*, the well known McCabe software complexity metric [21], rather than to the actual number of code lines.

Mathematical program (56) resorts to the numerical constant M, which represents a valid upper bound to the finite values that each variable can take in each control point of the program. The value assigned to M could be critical. In fact, on the one hand, underestimated values for M result in a feasible set strictly included into the set of fixpoints of F or even empty, making Thm. 3.4 false. On the other hand, too much large values for M could make the model ill-conditioned and harder to solve. We therefore are interested in computing the smallest value for the upper bound M. Let i be the arc between nodes u and v, and $\bar{X}_i = [\bar{x}_i^L, \bar{x}_i^U]$, $i \leq m$, be a tight over-approximation of X_i, with $\bar{x}_i^L, \bar{x}_i^U \in (\mathbb{R} \cup \{\pm\infty\})^n$. $\bar{X}_i$ can be easily derived on the base of the rules listed in Table 1:

operator	X_i	$\bar{x}_i^L$	$\bar{x}_i^U$
constant	$X_i = [a^L, a^U]$	a^L	a^U
identity	$X_i = \mathrm{Id}(X_j)$	$\bar{x}_j^L$	$\bar{x}_j^U$
arithmetic	$X_i = \phi_u(X_j)$	$\phi_u(\bar{x}_j^L)$	$\phi_u(\bar{x}_j^U)$
intersection	$X_i = X_j \cap \tau$	τ^L	τ^U
union	$X_i = X_j \cup X_k$	$\min\{\bar{x}_j^L, \bar{x}_k^L\}$	$\max\{\bar{x}_j^U, \bar{x}_k^U\}$

Table 1: Rules for deriving $\bar{X}_i$.

4.1 Proposition
The smallest valid upper bound M for program (56) is given by

$$M = \max\{\max_{i \leq m}\{|\bar{x}_i^L|, \bar{x}_i^L \neq -\infty\}, \max_{i \leq m}\{|\bar{x}_i^U|, \bar{x}_i^U \neq +\infty\}\} + \epsilon. \tag{57}$$

Proof. The set V of the nodes of the program graph can be totally ordered so that each cycle $(v_p, \ldots, v_q)$ of G with $p < q$ (representing loops in the program) has $\ell(v_p) = \mathrm{J}$. Therefore, we can assume $j < i$ for all the indices in Table 1 and, except for the union operator, an over-approximation of each interval X_i can be obtained by a forward substitution process. Observe that the operand X_k in Table 1 corresponds to the arc (q, p) of the relevant cycle and, due the particular structure of cycles, $\ell(v_{p+1}) = \mathrm{T}$. Since the intersection is over-approximated by the constant interval τ and $\ell(v_h) \neq \mathrm{J}$, for $p < h \leq q$, all

the intervals within nodes v_{p+1} and v_q can be over-approximated by forward substitution as well, and therefore also X_k. □

Similar arguments can be used for computing the greater value for the lower bound ϵ. However in our computational experience ϵ has been set to 1 since all the instances only use integer variables.

Our implementation is able to deal with many program variables of different type (integer or floating point), arrays, and function calls. In particular, each array is treated as a single summary object. According to this approach, known as array *smashing* [2], the interval that represents the array is shared by all the array elements.

5 Computational results

Computational experience on a set of small instances has been gathered in order to validate the mathematical model. These instances can be downloaded from `http://www.lix.polytechnique.fr/~liberti/verif-instances.zip`. These are simple C programs without pointers or dynamic memory allocation. For each instance, Table 2 lists the number of lines of code, variables, loops, and the maximum level of loop nesting (columns *lines*, *vars*, *loops* and *indent lev.*, respectively).

For each instance, column *simplex* reports the total number of pivot operations needed to solve the linear relaxations associated to the nodes of the branch-and-bound tree, whereas column *BB nodes* shows the size of the branch-and-bound tree.

The computational results were obtained on an Intel Xeon 2.4GHz with 8GB RAM running Linux. Notice that 84% of the instances are solved at root node (i.e. no variable branching is ever required) just by means of preprocessing and/or simplex algorithm; 32% of these are solved by pre-processing only. For all instances requiring full BB, the search tree is very small: 46.71 nodes on average. The mean number of simplex iterations is also very small (79.04), resulting in negligible computational times in practice. This points out that our method can also be used for considerably larger instances than those we tested.

Solutions obtained by CPLEX 11 have been compared to those provided by an implementation of the Policy Iteration (PI) algorithm [4], which only guarantees optimality in the case of non-expansive fixpoint mappings F. This is apparent in Table 2: instances for which our method finds a better fixpoint are emphasized in boldface. As per the usual efficacy/efficiency trade-off, it is also apparent that the simple PI implementation we used is one order of magnitude faster than the state-of-the-art CPLEX solver; so for large-scale programs, obtaining a guaranteed optimal solution might be computationally infeasible. Lastly, our test PI implementation could not tackle static arrays and function calls (hence the '-' sign in the table).

6 Conclusion and extensions

We described a new mathematical programming based method for finding guaranteed least fixpoints of the semantic equations arising from static analysis of computer programs. We developed our modelling procedure using non-relational domains (interval domains for the values of program variables) and presented some promising computational results.

One extremely attractive feature of using mathematical programming as a solution method for the semantic equations (1) is that it allows to seamlessly add arbitrary relations between the program variables. Supposing we know that during the execution of a given program a set of program variables $x \in \mathbb{R}^n$ always attain values within a given polyhedron $\{x \in \mathbb{R}^n \mid Ax \leq b\}$, it suffices to add the constraints linking lower and upper variable bounds according to the linear relations $Ax \leq b$. This has a special relevance

Instance					CPLEX 11				Policy Iteration	
name	lines	vars	loops	indent lev.	simplex	nodes	CPU	width	CPU	width
simple 1	10	1	0	0	0	0	0.006	0	0.000	0
simple 2	8	1	0	0	0	0	0.006	0	0.000	0
simple 3	10	1	0	0	2	0	0.010	0	0.001	0
simple 4	8	1	1	1	22	0	0.012	199	0.000	199
simple 5	8	1	1	1	21	0	0.011	198	0.001	198
simple 6	7	1	1	1	5	0	0.009	2	0.001	2
simple 7	8	1	1	1	7	0	0.009	2	0.001	2
simple 8	11	2	0	0	0	0	0.010	40000	0.000	40000
simple 9	11	2	0	0	0	0	0.006	40000	0.001	40000
simple 10	11	2	0	0	0	0	0.009	60004	0.001	60004
simple 11	12	2	0	0	0	0	0.018	20000	0.002	20000
simple 12	8	2	0	0	0	0	0.011	160000	0.001	160000
simple 13	11	2	1	1	0	0	0.012	20000	0.001	20000
simple 14	11	2	1	1	0	0	0.017	20000	0.001	20000
simple 15	9	2	1	1	0	0	0.008	60000	0.001	60000
simple 16	12	2	1	1	4	0	0.013	39996	0.003	39996
simple 17	9	2	1	1	0	0	0.010	40000	0.001	40000
simple 18	10	2	1	1	13	0	0.012	20000	0.001	20000
simple 19	10	2	1	1	8	0	0.013	20002	0.001	20002
simple 20	11	2	1	1	6	0	0.014	20000	0.000	20000
simple 21	17	2	1	1	41	0	0.015	60699	0.002	60699
simple 22	11	2	1	1	84	8	0.029	20075	0.002	20075
simple 23	13	2	1	1	29	0	0.018	20002	0.003	20002
simple 24	10	2	1	1	93	13	0.030	21080	0.003	21080
simple 25	12	2	2	2	35	0	0.019	**70494**	0.003	110095
simple 26	13	2	2	2	43	0	0.022	61078	0.006	61078
simple 27	19	2	2	2	123	0	0.045	20158	0.004	20158
simple 28	17	3	0	0	0	0	0.006	240000	0.001	240000
simple 29	17	3	0	0	0	0	0.015	60000	0.002	60000
simple 30	11	3	1	1	19	0	0.019	79997	0.003	79997
simple 31	32	3	1	1	2	0	0.042	500002	0.002	500002
simple 32	20	3	1	1	90	6	0.032	300075	0.002	300075
simple 33	12	3	1	1	57	0	0.019	**169199**	0.003	169499
simple 34	18	3	2	2	0	0	0.016	280402	0.002	280402
simple 35	22	3	3	2	138	13	0.036	**62028**	0.005	121308
simple 36	17	3	3	3	48	0	0.030	**220789**	0.006	270289
simple 37	25	3	4	2	8	0	0.041	569353	0.004	569353
simple 38	35	3	6	2	1529	183	0.163	**64501**	0.007	202821
PI comparison 1	13	2	1	1	29	0	0.017	**20398**	0.003	50392
PI comparison 2	14	2	2	2	278	46	0.042	**20084**	0.001	60048
PI comparison 3	13	2	2	2	83	0	0.026	**21374**	0.004	60582
arrays	22	6	2	1	56	0	0.068	**600139**	-	-
functions	62	11	3	1	509	58	0.144	**444430**	-	-
fun+arrays	53	10	2	2	96	0	0.048	**340105**	-	-

Table 2: Computational results.

to relational polyhedral domains, which we plan to investigate in future work.

Acknowledgments

We are profoundly indebted to Prof. Eric Goubault, who initiated us to the mysteries of static analysis by abstract interpretation. Financial support by: Île-de-France research council (post-doctoral fellowship), System@tic consortium ("EDONA" project), ANR 07-JCJC-0151 "Ars", ANR 08-SEGI-023 "Asopt", Digiteo Emergence "Paso" is gratefully acknowledged.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In T. Mogensen, D. Schmidt, and I. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *LNCS*, pages 85–108. Springer-Verlag, Berlin, 2002.

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

[5] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In R. Cousot, editor, *Verification, Model Checking and Abstract Interpretation*, volume 3385 of *LNCS*, pages 17–19, 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] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.

[9] M. Fischetti, F. Glover, and A. Lodi. The feasibility pump. *Mathematical Programming*, 104(1):91–104, 2005.

[10] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–37, 2005.

[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 (ESOP)*, 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 (ESOP)*, volume 4421 of *LNCS*. Springer, 300-315.

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

[15] 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.

[16] ILOG. *ILOG CPLEX 11.0 User's Manual*. ILOG S.A., Gentilly, France, 2008.

[17] R. Levine, T. Mason, and D. Brown. *Lex and Yacc*. O'Reilly, Cambridge, second edition, 1995.

[18] L. Liberti. Writing global optimization software. In L. Liberti and N. Maculan, editors, *Global Optimization: from Theory to Implementation*, pages 211–262. Springer, Berlin, 2006.

[19] L. Liberti. Reformulations in mathematical programming: Definitions and systematics. *RAIRO-RO*, 43(1):55–86, 2009.

[20] 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.

[21] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[22] M. Roozbehani, A. Megretski, and E. Feron. Convex optimization proves software correctness. In *Proceedings of the American Control Conference*, 2005.

[23] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Transactions on Programming Languages and Systems*, 27(2):183–235, 2005.

[24] E.M.B. Smith and C.C. Pantelides. A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex MINLPs. *Computers & Chemical Engineering*, 23:457–478, 1999.

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

[26] H. Tuy. *Convex Analysis and Global Optimization*. Kluwer Academic Publishers, Dordrecht, 1998.

[27] L.A. Wolsey. *Integer Programming*. Wiley, New York, 1998.