# Automatic modular abstractions for template numerical constraints

David Monniaux

# AUTOMATIC MODULAR ABSTRACTIONS FOR TEMPLATE NUMERICAL CONSTRAINTS

DAVID MONNIAUX

Abstract. We propose a method for automatically generating abstract transformers for static analysis by abstract interpretation. The method focuses on linear constraints on programs operating on rational, real or floating-point variables and containing linear assignments and tests. Given the specification of an abstract domain, and a program block, our method automatically outputs an implementation of the corresponding abstract transformer. It is thus a form of program transformation.

In addition to loop-free code, the same method also applies for obtaining least fixed points as functions of the precondition, which permits the analysis of loops and recursive functions.

The motivation of our work is data-flow synchronous programming languages, used for building control-command embedded systems, but it also applies to imperative and functional programming.

Our algorithms are based on new quantifier elimination and symbolic manipulation techniques over linear arithmetic formulas. We also give less general results for nonlinear constraints and nonlinear program constructs.

## 1. Introduction

Program analysis consists in, given a program, obtaining properties about its possible executions. Example of interesting properties include: "the program always terminates"; "the program never executes a division by zero"; "the program always outputs a well-formed XML document"; "variable `x` always lies between 1 and 3". Abstract interpretation (§1.1) is a general framework in which to design sound static analyzes — analyzes that produce correct results by construction. Most such analyzes are neither *modular* nor *optimal*: they generally need the program to be analyzed all at once, as opposed to being able to analyze parts of it separately, and they may err on the pessimistic side — for instance, provide variation bounds on variables that are not tight. In this article, we describe methods that are optimal in some sense, and that allow parts of a program to be analyzed separately from their execution environment. Our methods are based on quantifier elimination (§1.2), an algorithmic property of certain logics.

*This article is an expanded version of two conference articles [70, 71].*

1.1. **Abstract interpretation.** It is well-known that, in the general case, fully automatic program analysis is impossible for any nontrivial property.[1] Thus, all analysis methods must have at least one of the following characteristics:

[1]This result, formally given within the framework of recursive function theory, is known as Rice's theorem. [82, p. 34][80, corollary B] It is obtained by generalization from Turing's halting theorem. Interpreted upon program semantics, the theorem states that the only properties of the denotational semantics of programs that can be algorithmically decided on the source code are the trivial properties: uniformly "true" or uniformly "false".

```
int x, y;
boolean a;
if (x < y) a = false;
```

(a) Original program

```
boolean a;
if (choice()) a = false;
```

(b) Boolean program

FIGURE 1. Transformation of a program into a Boolean program by erasing the numeric part and replacing tests over numerical quantities by nondeterministic choice (`choice()` nondeterministically returns `true` or `false`).

- They may bound the memory size of the studied program, which then becomes a finite automaton, on which most properties are decidable. *Explicit-state model-checking* works by enumerating all reachable states (which are in finite number) while *implicit state model-checking* represents sets of states using clever data structures. [22]
- They may restrict the programming language used, making it not Turing-complete, so that properties become decidable. For instance, reachability in pushdown automata is decidable even though their memory size is unbounded. [13]
- They may restrict the class of properties expressed to properties of bounded executions; e.g., "within the first 10000 steps of execution, there is no division by zero", as in *bounded model checking*. [9]
- They may be *unsound* as proof methods: they may fail to detect that the desired property is violated. Typically, *bug-finding* and testing programs are in that category, because they may fail to detect a bug. Some such analysis techniques are not based on program semantics, but rather on finding patterns in the program syntax. [34]
- They may be *incomplete* as proof methods: they may fail to prove that a certain property holds, and report spurious violations. Methods based on abstraction fall in that category.

*Abstraction* by over-approximation consists in replacing the original problem, undecidable or very difficult to decide, by a simpler "abstract" problem whose behaviours are guaranteed to include all behaviours of the original problem.

An example of an abstraction is to erase from the program all constructs dealing with numerical and pointer types (or replacing them with nondeterministic choices, if their value is used within a test), keeping only Boolean types (Fig. 1). Obviously, the behaviours of the resulting program encompass all the behaviours of the original program, plus maybe some extra ones. Further abstraction can be applied to this Boolean program: for instance, the "3-value logic" abstraction [79] which maps any input or output variable to an abstract parameter taking its value in a 3-element set: "is 0", "is 1", "can be 0 or 1";[2] for practical purposes it may be easier to encode these values using a couple of Booleans, respectively meaning "can be 0" and "can be 1", thus the abstract values $(1, 0)$, $(0, 1)$ and $(1, 1)$. The abstract value $(0, 0)$ obtained for any variable at a program point means that this program point is unreachable. Given a vector of input abstract parameters, one for each input

[2]For brevity, we identify "false" with 0 and "true" with 1.

```
double x, y, z;
/* x lies in [m_x, M_x] */
y = x;
z = x-y;
```

(a) Original program

$$m'_y = m_x;$$
$$M'_y = M_y;$$
$$m'_z = m_x - M'_y;$$
$$M'_z = M_x - m'_y;$$

(b) Abstract transformer program

FIGURE 2. The transformer for the interval domain obtained by composition of locally optimal abstract transformers is imprecise. For each statement (on the left) we use a corresponding optimal transformer (on the right), but the composition of these transformers is not optimal. For the sake of simplicity, all variables are considered to be real numbers.

variable of the program, the *forward abstract transfer function* gives a correct vector of output abstract parameters, one for each output variable of the program. Quite obviously, in the absence of loops, it is possible to obtain a suitable forward abstract transfer function by applying simple logical rules to the Boolean function defined by the Boolean program. An effective implementation of the forward abstract transfer function can thus be obtained by a transformation of the source program.

For programs operating over numerical quantities, a common abstraction is *intervals*. [29, 30] To each input $x$, one associates an interval $[m_x, M_x]$, to each output $x'$ an interval $[m'_x, M'_x]$. How can one compute the $(m'_v, M'_v)_{v \in V}$ bounds from the $(m_v, M_v)_{v \in V}$? The most common method is *interval arithmetic*: to each elementary arithmetic operation, one attaches an abstract counterpart that gives bounds on the output of the operation given bounds on the inputs. For instance, if one knows $[m_a, M_a]$ and $[m_b, M_b]$, and $c = a + b$, the one computes $[m_c, M_c]$ as follows: $m_c = m_a + m_b$ and $M_c = M_a + M_b$. If a program point can be reached by several paths (e.g. at the end of an if-then-else construct), then a suitable interval $[m_x, M_x]$ can be obtained by a *join* of all the intervals for $x$ at the end of these paths. Again, for a loop-free program, one can obtain a suitable effective forward abstract transfer function by a program transformation of the source code.

The abstract transfer function defined by interval arithmetic is always correct, but is not necessarily the most precise. For instance, on example in Fig. 2(a), the best abstract transfer function maps any input ranges to $m'_z = M'_x = 0$, since the output $z$ is always zero, while the one obtained by applying interval arithmetic to all program statements (Fig. 2(b)) yields, in general, larger intervals. The weakness of the interval domain on this example is evidently due to the fact that it does not keep track of relationships between variables (here, that $x = y$). *Relational abstract domains* such as the *octagons* [64, 66, 67] or *convex polyhedra* [4, 32, 53] address this issue. Yet, neither octagons nor polyhedra provide analyses that are guaranteed to give optimal results.

Consider the following program:

```
int x;
if (x > 0) x= 1; else x= -1;
if (x == 0) x= 2;
```

Obviously, the second test can never be taken, since `x` can only be $\pm 1$; however an interval, octagon or polyhedral analysis will conclude, after joining the informations

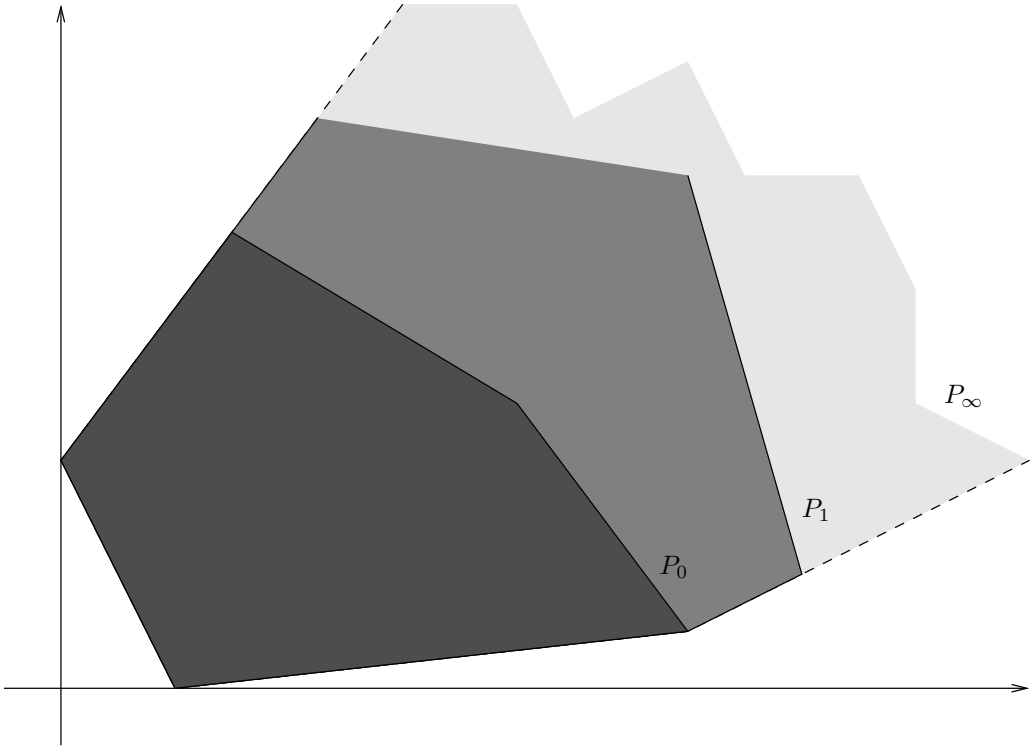


FIGURE 3. The standard widening on convex polyhedra [32, 53], here demonstrated on polyhedra in dimension 2 (polygons). The widening operator observes the sets of reachable states $P_0$ and $P_1$ at two consecutive iterations, and keeps only the constraints (polyhedral faces, here polygon edges) that are stable across iterations. The resulting $P_\infty$ polyhedron is then proposed as an invariant.

for both branches of the first test, that x lies in $[-1, 1]$ and will not be able to exclude the case $x = 0$.

This is the first problem that this article addresses: how to obtain, in general, effective, optimal abstract transfer functions for certain classes of loop-free programs and numerical constraints.[3]

We have so far left out programs containing loops; when programs contain loops or recursive functions, a central problem of program analysis is to find *inductive invariants*. In the case of Boolean programs, given constraints on the input parameters, the set of reachable states can be computed exactly by model-checking algorithms; yet, these algorithms do not give a closed-form representation of the abstract transfer function mapping input parameters to output parameters for the

[3]Our analysis in that respect produces the same result as projecting over the output constraints the results of an input-output relational analysis over the disjunctive completion of the convex polyhedra; that is, computing the set of possible couples $(\sigma, \sigma')$ of input and output states as an union of convex polyhedra, then projecting it over the template. The quantifier elimination procedure can however avoid enumerating all the polyhedra in this disjunction, for instance by performing projections at the same time. [69]

3-value abstraction. In the case of numerical abstractions such as the intervals, octagons or polyhedra, the most common way to find invariants is through the use of a *widening operator*. [30] Intuitively, widening operators observe the sets of reachable states after $N$ and $N+1$ loop iterations and extrapolate them to a "candidate invariant" (see Fig. 3 for an example: the standard widening operator on convex polyhedra).

Let $u_0$ be the set of initial states of a loop, and let $\rightarrow_\tau$ be transition relation for this loop ($\sigma \rightarrow_\tau \sigma'$ means that $\sigma'$ is reachable in one loop step from $\sigma$). The set of reachable states at the head of the loop is the least fixed point of $f : u \mapsto u \cup \{\sigma' \mid \exists \sigma \in u \wedge \sigma \rightarrow_\tau \sigma'\}$, which is obtained as the limit of the ascending sequence defined by $u_{n+1} = f(u_n)$. By abstract interpretation, we replace this sequence by an abstract sequence $u_n^\sharp$ defined by $u_{n+1}^\sharp = f^\sharp(u_n^\sharp)$, such that for any $n$, $u_n^\sharp$ is an abstraction of $u_n$. If this sequence is stationary, that is, $u_{N+1}^\sharp = u_N^\sharp$ for some $N$, then $u_N^\sharp$ is an abstraction of the least fixed point of $f$ and thus of the least invariant of the transition relation $\tau$ containing $u_0$.

Again, widening operators provide correct results, but these results can be grossly over-approximated. Much of the literature on applied analysis by abstract interpretation discusses workarounds that give precise results in certain cases: *narrowing* iterations [29, 30], widening "up to" [52, §3.2], "delayed" or with "thresholds" [11], etc. Yet, such workarounds typically fail on other examples than the ones they were designed to address.

This is the second problem that this article addresses: how to obtain, in general, optimal invariants for certain classes of programs and numerical constraints. Furthermore, our methods provide these invariants as functions of the parameters of the precondition of the loop, thus, again, they provide effective, optimal abstract transfer functions for loop constructs.

1.2. **Quantifier elimination.** Consider a set $A$ of atomic formulas. The set $U(A)$ of *quantifier-free formulas* is the set of formulas constructed from $A$ using operators $\wedge$, $\vee$ and $\neg$; the set $Q(A)$ of *quantified formulas* is the set of formulas constructed from $A$ using the above operators and the $\exists$ and $\forall$ quantifiers. Such formulas are thus trees whose leaves are the atomic formulas. A *literal* is an atomic formula or the negation thereof. The set of *free variables* $FV(F)$ of a formula $F$ is defined as usual. A quantifier-free formula without variables is said to be *ground*. A formula without free variables is said to be *closed*; the *existential closure* of a formula $F$ is $F$ with existential quantifiers for all free variables prepended; the *universal closure* is the same with universal quantifiers. A quantifier-free formula is said to be in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions, that is, is of the form $(l_{1,1} \wedge \cdots \vee l_{1,n_1}) \vee \cdots \vee (l_{m,1} \wedge \cdots \vee l_{m,n_m})$, and is said to be in *conjunctive normal form* (CNF) if it is a conjunction of disjunctions. Any quantifier-free formula can be converted into CNF or DNF by application of the distributivity laws $(A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C)$ and $(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C)$, though better algorithms exist, such as ALL-SAT modulo theory [69].

1.2.1. *Booleans.* If $A$ is a set $V$ of variables, then $U(A)$ is the set of propositional formulas over these variables. A *valuation* $m$ for a formula $F$ is a mapping from $FV(F)$ to $\{\text{false}, \text{true}\}$; it is said to be a *model* for $F$, noted $m \models F$, if $F$ evaluates to true after its variables have been replaced by Boolean values according to $m$. We then say that we are considering the *theory* of Booleans. Two formulas $F$

and $G$ are said to be *equivalent*, noted $F \equiv G$, if they have the same models. A formula is said to be *satisfiable* if it admits at least one model. Deciding whether a propositional formula is satisfiable is known as the SAT-problem, the canonical NP-complete problem; although it is conjectured that there exists no polynomial algorithm for this problem, there exist many implementations capable of deciding many large problems in practice. [57, ch. 2] $Q(A)$ is the set of quantified Boolean formulas. Deciding the satisfiability of such formulas is the canonical PSPACE-complete problem, known as QBF; [45, §7.4] note that QBF is included in all the quantified theories that we shall see further, which indicates that they are at least as hard to decide!

The theory of Booleans admits algorithmic *quantifier elimination*: there exists an algorithm that takes as an input any formula in $Q(A)$ and outputs an equivalent quantifier-free formula. One naive method to achieve this is to proceed by induction over the structure of the formula and convert any $\forall x\ F$ formula to $F[\text{true}/x] \wedge F[\text{false}/x]$ and any $\exists x\ F$ to $F[\text{true}/x] \vee F[\text{false}/x]$. Such a method, replacing $\forall x\ F$ by a finite conjunction of formulas of the form $F[x_i/x]$ and $\exists x\ F$ by a finite disjunction of formulas of the form $F[x_i/x]$, is known as a *substitution method*. It is obvious that substitution methods are possible for quantification over any finite domain, but we shall now see that they are also possible for certain logics over infinite domains.

1.2.2. *Linear real inequalities.* Consider now $A$ the set of linear inequalities with integer or rational coefficients over a set of variables $V$. By elementary calculus, such inequalities can be equivalently written in the following forms: $l(v_1, v_2, \dots) \geq C$ or $l(v_1, v_2, \dots) > C$, with $l$ a linear expression with integer coefficients over $V$ and $C$ a constant. Let us first consider the theory of *linear real arithmetic* (LRA): models of a formula $F$ are mappings from $F$ to the real field $\mathbb{R}$, and notions of equivalence and satisfiability follow. Note that satisfiability and equivalence are not affected by taking models to be mappings from $F$ to the rational field $\mathbb{Q}$. Deciding whether a LRA formula is satisfiable is, again, a NP-complete problem known as *satisfiability modulo theory* (SMT) of real linear arithmetic. Again, practical implementations, known as SMT-solvers , are capable of dealing with rather large formulas; examples include Yices [41] and Z3 [37].[4]

The theory of linear real arithmetic admits quantifier elimination. For instance, the quantified formula $\forall x\ (x \geq y \implies x \geq 3)$ is equivalent to the quantifier-free formula $y \geq 3$. Quantified linear real arithmetic formulas are thus decidable: by quantifier elimination, one can convert the existential closure of the formula to an equivalent ground formula, the truth of which is trivially decidable by evaluation. The decision problem for quantified formulas over rational linear inequalities requires at least exponential time, thus quantifier elimination is at least exponential. [15, §7.4][44, Th. 3] Weispfenning [94] discusses complexity issues in more detail.

Again, most quantifier elimination algorithm proceed by induction over the structure of the formula, and thus begin by eliminating the innermost quantifiers, progressively replacing branches of the formula containing quantifiers by quantifier-free equivalent branches. By application of the equivalence $\forall x\ F \equiv \neg\exists x\ \neg F$, one can reduce the problem to eliminating existential quantifiers only. Consider now

[4]The yearly SMT-COMP competition has SMT-solvers compete on a large set of benchmarks.

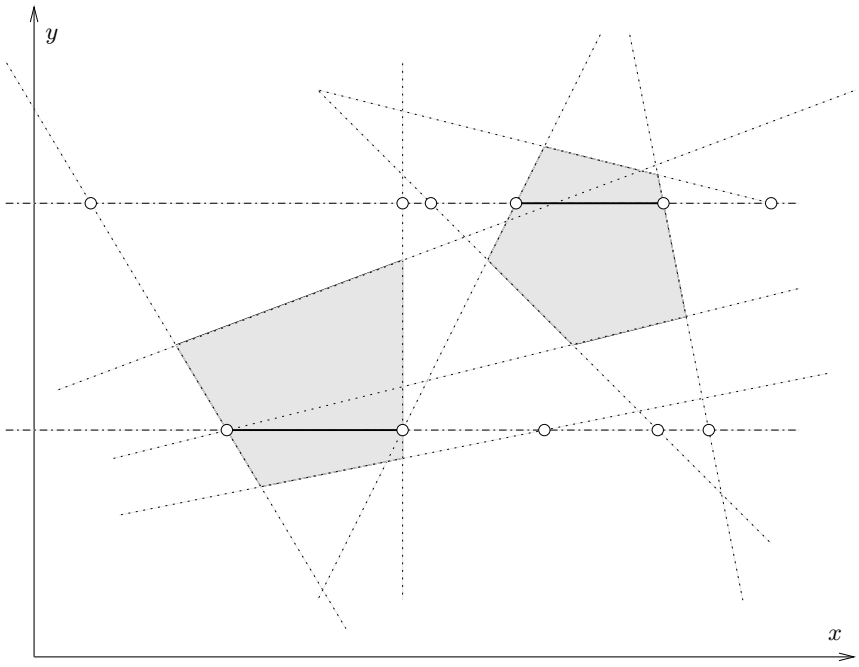


FIGURE 4. The gray zone $S$ is the set of $(x, y)$ solutions of formula $F$, whose atoms are the linear inequalities corresponding to the lines $\Delta$ drawn with dashes. For fixed $y = y_0$, the set of $x$ such that $F(x, y)$ is true is made of intervals whose ends lie within the set $I$ of intersections of the $y = y_0$ line with the lines in $\Delta$, drawn with a small circle. $y = y_0$ therefore has an intersection with $S$ if and only if a point in $I$, or an interval with both ends in $I \cup \{-\infty, +\infty\}$, lies within $S$. This condition can be tested using $x \to \pm\infty$ and all midpoints to intervals with both ends in $I$, as per Ferrante and Rackoff [43], or, in addition to $I \cup \{-\infty\}$, for any element of $I$ a point infinitesimally close to the right of it, as per Loos and Weispfenning [60]. Both methods exploit the fact that the coordinates of all points from $I$ (intersection of $y = y_0$ and a line from $\Delta$) can be expressed as affine linear functions of $y_0$.

the problem of eliminating the existential quantifiers from $\exists\ x_1 \ldots x_n\ F$ where $F$ is quantifier-free. We can first convert into DNF: $\exists x_1 \ldots x_n\ (C_1 \vee \cdots \vee C_m)$ where the $C_i$ are conjunctions, then to the equivalent formula $(\exists\ x_1 \ldots x_n\ C_1) \vee \ldots (\exists\ x_1 \ldots x_n\ C_m)$. We thus have reduced the quantifier elimination problem for general formula to the problem of quantifier elimination for conjunctions of linear inequalities. Remark that, geometrically, this quantifier elimination amounts to computing the projection of a convex polyhedron along the dimensions associated with the variables $x_1, \ldots, x_n$, with the original polyhedron and its projection being defined by systems of linear inequalities. The Fourier-Motzkin elimination procedure [57, §5.4] converts $\exists x_1 \ldots x_n\ C$ into an equivalent conjunction.

There exist more efficient quantifier elimination procedures than conversion to DNF followed by Fourier-Motzkin elimination. Ferrante and Rackoff [43] proposed a substitution method [15, §7.3.1][75, §4.2][94]: a formula of the form $\exists x\ F$ where $F$ is quantifier-free is replaced by an equivalent disjunction $F[e_1/x] \vee \cdots \vee F[e_n/x]$, where the $e_i$ are affine linear expressions built from the free variables of $\exists x\ F$. Note the similarity to the naive elimination procedure we described for Boolean variables: even though the existential quantifier ranges over an infinite set of values, it is in fact only necessary to test the formula $F$ at a finite number of points (see Fig. 4). The drawback of this algorithm is that $n$ is proportional to the square of the number of occurences of $x$ in the formula; thus, the size of the formula can be cubed for each quantifier eliminated. Loos and Weispfenning [60] proposed a *virtual substitution* algorithm[5] [75, §4.4] that works along the same general ideas but for which $n$ is proportional to the number of occurences of $x$ in the formula. Our benchmarks show that Loos and Weispfenning's algorithm is much more efficient than Ferrante and Rackoff's, despite the latter method being better known. Another class of algorithms improve on the conversion to DNF then projection algorithm, by combining both phases: we proposed an eager algorithm based on this idea [69] and are working on a lazy version.

1.2.3. *Presburger arithmetic.* With the same language of formulas, consider now models over the integers, thus the theory of *linear integer arithmetic* (LIA), also known as Presburger arithmetic. We immediately notice that linear inequalities are insufficient for quantifier elimination — we also need congruence constraints: $\exists k\ x = 2k$ simplifies to $x \equiv 0 \pmod 2$.

Decision of formulas in Presburger arithmetic is doubly exponential, and thus quantifier elimination is very expensive in the worst case [44]. Presburger [77] provided a quantifier elimination procedure, but its complexity was impractical; Cooper [26] proposed a better algorithm [15, §7.2]; Pugh [78] proposed the "Omega test" [57, §5.5]. Practical complexities are still high in practice .

Cooper and Pugh's procedures are very geometrical in nature. Integers, however, can also be seen as words over the $\{0, 1\}$ alphabet, and sets of integers can thus be recognized by finite automata [76, §8]. Addition is encoded as a 3-track automaton recognizing that the number on the third track truly is the sum of the numbers on the first two tracks; equivalently, this encodes subtraction. Existential quantifier elimination just removes some of the tracks, making transitions depending on bits read on that track nondetermistic. Negation is complementation (which can be costly, thus explaining the high cost of quantifier alternation). Multiplication by powers of two are also easily encoded, and multiplications by arbitrary constants can be encoded by a combination of additions and multiplications by powers of two. The same idea can be extended to real numbers written as their binary expansion, using automata on infinite words.

This leads to an interesting arithmetic theory, with two sorts of variables: reals (or rationals) and integers. This could be used to model computer programs, with

[5]This method replaces $x$ by a formula that does not evaluate to a rational number, but to a sum of a rational number and optionally an infinitesimal $\varepsilon$, taken to be a number greater than 0 but less than any positive real; the infinitesimals are then erased by application of the rules governing comparisons. In practical implementations, one does both substitution and erasure of infinitesimals in one single pass, and infinitesimals never actually appear in formulas; thus the phrase *virtual substitution*.

integers for integer variables and reals for floating-point variables (if necessary by using the semantic transformations described in § 3.4). Boigelot et al. [12] described a restricted class of $\omega$-automata sufficient for quantifier elimination. Becker et al. [8] implemented the LIRA tool based on such ideas. Unfortunately, this approach suffers from two major drawbacks: the practical performances are very bad for purely real problems [69], and it is impossible to recover an arithmetical formula from almost all these automata. We therefore did not pursue this direction.

1.2.4. *Nonlinear real arithmetic.* What happens if we do not limit ourselves to linear arithmetic, but also allow polynomials? Over the integers, the resulting theory is known as Peano arithmetic. It is well known that there can exist no decision procedure for quantified Peano arithmetic formulas;[6] thus, since the ground formulas for this theory are trivially decidable, there can exist no quantifier elimination algorithm.

The situation is wholly different over the real numbers. The satisfiability or equivalence of polynomial formulas does not change whether the models are taken over the real numbers, the real algebraic numbers, or, for the matter, any *real closed field*; this theory is thus known as the theory of real closed fields, or *elementary algebra*. Tarski [90, 91] and Seidenberg [86] showed that this theory admits quantifier elimination, but their algorithms had impractically high complexity. Collins [23] introduced a better algorithm based on *cyclindrical algebraic decomposition*. For instance, quantifier elimination on $\exists x\ ax^2 + bx + c = 0$ by cylindrical algebraic decomposition yields

(1)
$$\left(c < 0 \wedge \left(\left(b < 0 \wedge a \geq \frac{b^2}{4c}\right) \vee\ (b = 0 \wedge a > 0) \vee \left(b > 0 \wedge a \geq \frac{b^2}{4\ c}\right)\right)\right) \vee c = 0 \vee$$
$$\left(c > 0 \wedge \left(\left(b < 0 \wedge\ a \leq \frac{b^2}{4c}\right) \vee (b = 0 \wedge a < 0) \vee \left(b > 0 \wedge a \leq\ \frac{b^2}{4c}\right)\right)\right)$$

Note the cylindrical decomposition: first, there is a case disjunction according to the values of $c$, then, for each disjunct for $c$, a case disjunction for the value of $b$; more generally, cylindrical algebraic decomposition builds a tree of case disjunctions over a sequence of variables $v_1, v_2, \ldots$ , where the guard expressions defining the cases for $v_i$ can only refer to $v_1, \ldots, v_i$. This decomposition only depends on the polynomials inside the formula and not on its Boolean structure, and computing it may be very costly even if the final result is simple. This is the intuition why despite various improvements [7, 20] the practical complexity of quantifier elimination algorithms for the theory of real closed fields remain high. The theoretical space complexity is doubly exponential [17, 36].

Minimal extensions to this formula language may lead to undecidability. This is for instance the case when one adds trigonometric functions: it is possible to define $\pi$ as the least positive zero of the sine, then define the set of integers as

[6]One does not need the full language of quantified Peano formulas for the problem to become undecidable; a single unbounded quantifier block is sufficient. The decision problem for $\Sigma_1^0$ formulas, that is, formulas of the form $\exists n_1 \ldots n_k\ \phi$ where $\phi$ only has bounded quantifiers, is undecidable. Following Cook [25] [95, ch. 7], the termination property of a program containing a single loop and operating over unbounded integers (a model equivalent to Turing machines) can be algorithmically encoded as a $\Sigma_1^0$ formula, thus deciding the truth of such formulas would entail deciding Turing's halting problem.

the numbers $k$ such that $\sin(k\pi) = 0$, and thus one can encode Peano arithmetic formulas into that language [2]. Also, naive restrictions of the language do not lead to lower complexity. For instance, limiting the degree of the polynomials to two does not make the problem simpler, since formulas with polynomials of arbitrary degrees can be encoded as formulas with polynomials of degree at most two, simply by introducing new variables standing for subterms of the original polynomials. For instance, $\exists x\ ax^3 + bx^2 + cx + d = 0$ can be encoded, using Horner's form for the polynomial, as $\exists x \exists y \exists z\ z = ax + b \wedge y = zx + c \wedge yx + d = 0$. Certain stronger restrictions may however work; for instance, if the variables to be eliminated occur only linearly, then one can adapt the substitution methods described in §1.2.2.

## 2. Optimal Abstraction over Template Linear Constraint Domains

When applying abstract interpretations over domains of linear constraints, such as octagons [64, 66, 67], one generally applies a widening operator, which may lead to imprecisions. In some cases, *acceleration* techniques leading to precise results can be applied [48, 49]: instead of attempting to extrapolate a sequence of iterates to its limit, as does widening, the exact limit is computed. In this section, we describe a class of constraint domains and programs for which abstract transfer functions of loop-free codes and of loops can be exactly computed; thus the *optimality*. Furthermore, the analysis outputs these functions in closed form (as explicit expressions combining linear expressions and functional if-then-else constructs), so the result of the analysis of a program fragment can be stored away for future use; thus the *modularity*. Our algorithms are based on quantifier elimination over the theory of real linear arithmetic (§1.2).

### 2.1. Template Linear Constraint Domains.

Let $F$ be a formula over linear inequalities. We call $F$ a domain definition formula if the free variables of $F$ split into $n$ *parameters* $p_1, \ldots, p_n$ and $m$ *state variables* $s_1, \ldots, s_m$. We note $\gamma_F : \mathbb{Q}^n \to \mathcal{P}(\mathbb{Q}^m)$ defined by $\gamma_F(\vec{p}) = \{\vec{s} \in \mathbb{Q}^m \mid (\vec{p}, \vec{s}) \models F\}$. As an example, the interval abstract domain for 3 program variables $s_1, s_2, s_3$ uses 6 parameters $m_1, M_1, m_2, M_2, m_3, M_3$; the formula is $m_1 \leq s_1 \leq M_1 \wedge m_2 \leq s_2 \leq M_2 \wedge m_3 \leq s_3 \leq M_3$.

In this section, we focus on the case where $F$ is a conjunction $L_1(s_1, \ldots, s_m) \leq p_1 \wedge \cdots \wedge L_n(s_1, \ldots, s_m) \leq p_n$ of linear inequalities whose left-hand side is fixed and the right-hand sides are parameters. Such conjunctions define the class of *template linear constraint domains* [24]. Particular examples of abstract domains in this class are:

- the intervals (for any variable $s$, consider the linear forms $s$ and $-s$);
- the difference bound matrices (for any variables $s_1$ and $s_2$, consider the linear form $s_1 - s_2$);
- the octagon abstract domain (for any variables $s_1$ and $s_2$, distinct or not, consider the linear forms $\pm s_1 \pm s_2$) [64]
- the octahedra (for any tuple of variables $s_1, \ldots, s_n$, consider the linear forms $\pm s_1 \cdots \pm s_n$). [21]

Remark that $\gamma_F$ is in general not injective, and thus one should distinguish the *syntax* of the values of the abstract domain (the vector of parameters $\vec{p}$) and their *semantics* $\gamma_F(\vec{p})$. As an example, if one takes $F$ to be $s_1 \leq p_1 \wedge s_2 \leq p_2 \wedge s_1 + s_2 \leq p_3$, then both $(p_1, p_2, p_3) = (1, 1, 2)$ and $(1, 1, 3)$ define the same set for state variables

$s_1$ and $s_2$. If $\vec{u} \leq \vec{v}$ coordinate-wise, then $\gamma_F(\vec{u}) \subseteq \gamma_F(\vec{v})$, but the converse is not true due to the non-uniqueness of the syntactic form.

Take any nonempty set of states $W \subseteq \mathbb{Q}^m$. Take for all $i = 1, \dots, m$: $p_i = \sup_{\vec{s} \in W} L_i(\vec{s})$. Clearly, $W \subseteq \gamma_F(p_1, \dots, p_m)$, and in fact $\vec{p}$ is such that $\gamma_F(\vec{p})$ is the least solution to this inclusion. $p_i$ belongs in general to $\mathbb{R} \cup \{+\infty\}$, not necessarily to $\mathbb{Q} \cup \{+\infty\}$. (for instance, if $W = \{s_1 \mid s_1^2 \leq 2\}$ and $L_1 = s_1$, then $p_1 = \sqrt{2}$). We have therefore defined an $\alpha_F : \mathcal{P}(\mathbb{R}^m) \to \{\bot\} \cup (\mathbb{R} \cup \{+\infty\})^n$, and $(\alpha_F, \gamma_F)$ form a *Galois connection*: $\alpha_F$ maps any set to its best upper-approximation. The fixed points of $\alpha_F \circ \gamma_F$ are the *normal forms*. For instance, $s_1 \leq 1 \wedge s_2 \leq 1 \wedge s_1 + s_2 \leq 2$ is in normal form, while $s_1 \leq 1 \wedge s_2 \leq 1 \wedge s_1 + s_2 \leq 3$ is not.

2.2. **Optimal Abstract Transformers for Program Semantics.** We shall consider the input-output relationships of programs with rational or real variables. We first narrow the problem to programs without loops and consider programs consisting in linear arithmetic assignments, linear tests, and sequences. Noting $a, b, \dots$ the values of program variables $\mathtt{a}, \mathtt{b} \dots$ at the beginning of execution and $a', b', \dots$ the output values, the semantics of a program $P$ is defined as a formula $[\![P]\!]$ such that $(a, b, \dots, a', b', \dots) \models P$ if and only if the memory state $(a', b', \dots)$ can be reached at the end of an execution starting in memory state $(a, b, \dots)$:

- **Arithmetic:** $[\![a := L(a, b, \dots) + K]\!]_F \triangleq a' = L(a, b, \dots) + K \wedge b' = b \wedge c' = c \wedge \dots$ where $K$ is a real constant and $L$ is a linear form, and $b, c, d \dots$ are all the variables except $a$;
- **Tests:** $[\![\mathtt{if}\ c\ \mathtt{then}\ p_1\ \mathtt{else}\ p_2]\!] \triangleq (c \wedge [\![p_1]\!]_F) \vee (\neg c \wedge [\![p_2]\!]_F)$;
- **Non deterministic choice:** $[\![a := \mathtt{random}]\!] \triangleq b' = b \wedge c' = c \wedge \dots$, for all variables except $a$;
- **Failure:** $[\![\mathtt{fail}]\!] \triangleq \mathsf{false}$;
- **Skip:** $[\![\mathtt{skip}]\!] \triangleq a' = a \wedge b' = b \wedge c' = c \wedge \dots$
- **Sequence:** $[\![P_1; P_2]\!]_F \triangleq \exists a'', b'', \dots\ f_1 \wedge f_2$ where $f_1$ is $[\![P_1]\!]_F$ where $a'$ has been replaced by $a''$, $b'$ by $b''$ etc., $f_2$ is $[\![P_2]\!]_F$ where $a$ has been replaced by $a''$, $b$ by $b''$ etc.

In addition to linear inequalities and conjunctions, such formulas contain disjunctions (due to tests and multiple branches) and existential quantifiers (due to sequential composition).

Note that so far, we have represented the concrete denotational semantics *exactly*. This representation of the transition relation using existentially quantified formulas is evidently as expressive as a representation by a disjunction of convex polyhedra (the latter can be obtained from the former by quantifier elimination and conversion to disjunctive normal form), but is more compact in general. This is why we defer quantifier elimination to the point where we compute the abstract transfer relation.

Consider now a domain definition formula $F \triangleq L_1(s_1, s_2, \dots) \leq p_1 \wedge \dots \wedge L_n(s_1, s_2, \dots) \leq p_n$ on the program inputs, with parameters $\vec{p}$ and free variables $\vec{s}$, and another $F' \triangleq L'_1(s'_1, s'_2, \dots) \leq p'_1 \wedge \dots \wedge L'_n(s'_1, s'_2, \dots) \leq p'_n$ on the program outputs, with parameters $\vec{p'}$ and free variables $\vec{s'}$. Sound forward program analysis consists in deriving a *safe post-condition* from a precondition: starting from any state verifying the precondition, one should end up in the post-condition. Using our notations, the *soundness condition* is written

$$\forall \vec{s}, \vec{s'}\ F \wedge [\![P]\!] \implies F' \tag{2}$$

The free variables of this relation are $\vec{p}$ and $\vec{p'}$: the formula links the value of the parameters of the input constraints to admissible values of the parameters for the output constraints. Note that this soundness condition can be written as a universally quantified formula, with no quantifier alternation. Alternatively, it can be written as a conjunction of correctness conditions for each output constraint parameter: $C'_i \triangleq \forall \vec{s}, \vec{s'}\ F \wedge [\![P]\!] \implies L'_i(\vec{s'}) \leq p'_i$.

Let us take a simple example: if $P$ is the program instruction $z := x + y$, $F \triangleq x \leq p_1 \wedge y \leq p_2$, $F' \triangleq z \leq p'_1$, then $[\![P]\!] \triangleq z' = x + y$, and the soundness condition is $\forall x, y, z\ (x \leq p_1 \wedge y \leq p_2 \wedge z = x + y \implies z \leq p'_1)$. Remark that this soundness condition is equivalent to a formula without quantifiers $p'_1 \geq p_1 + p_2$, which may be obtained through quantifier elimination. Remark also that while any value for $p'_1$ fulfilling this condition is *sound* (for instance, $p'_1 = 1000$ for $p_1 = p_2 = 1$), only one value is *optimal* ($p'_1 = 2$ for $p_1 = p_2 = 1$). An optimal value for the output parameter $p'_i$ is defined by $O'_i \triangleq C'_i \wedge \forall q'_i\ (C'_i[q'_i/p'_i] \implies p'_i \leq q'_i)$. Again, quantifier elimination can be applied; on our simple example, it yields $p'_1 = p_1 + p_2$.

If there are $n$ input constraint parameters $p_1, \ldots, p_n$, then the optimal value for each output constraint parameter $p'_i$ is defined by a formula $O'_i$ with $n + 1$ free variables $p_1, \ldots, p_n, p'_i$. This formula defines a *partial function* from $\mathbb{Q}^n$ to $\mathbb{Q}$, in the mathematical sense: for each choice of $p_1, \ldots, p_n$, there exist at most a single $p'_i$. The values of $p_1, \ldots, p_n$ for which there exists a corresponding $p'_i$ make up the *domain of validity* of the abstract transfer function. Indeed, this function is in general not defined everywhere; consider for instance the program:

```
if (x >= 10) { y = nondeterministic_choice_in_all_reals; } else { y = 0; }
```

If $F = x \leq p_1$ and $F' = y \leq p'_1$, then $O'_1 \equiv p_1 < 10 \wedge p'_1 = 0$, and the function is defined only for $p_1 < 10$.

At this point, we have a characterization of the optimal abstract transformer corresponding to a program fragment $P$ and the input and output domain definition formulas as $n$ formulas (where $n$ is the number of output parameters) $O'_i$ each defining a function (in the mathematical sense) mapping the input parameters $\vec{p}$ to the output parameter $p'_i$.

Another example: the absolute value function $y := |x|$, again with the interval abstract domain. The semantics of the operation is $(x \geq 0 \wedge y = x) \vee (x < 0 \wedge y = -x)$; the precondition is $x \in [x_{\min}, x_{\max}]$ and the post-condition is $y \in [y_{\min}, y_{\max}]$. Acceptable values for $(y_{\min}, y_{\max})$ are characterized by formula

$$G \triangleq \forall x\ x_{\min} \leq x \leq x_{\max} \implies y_{\min} \leq |x| \leq y_{\max} \tag{3}$$

The optimal value for $y_{\max}$ is defined by $G \wedge \forall y'_{\max}\ G[y'_{\max}/y_{\max}] \implies y_{\max} \leq y'_{\max}$. Quantifier elimination over this last formula gives as characterization for the least, optimal, value for $y_{\max}$:

$$(x_{\min} + x_{\max} \geq 0 \wedge y_{\max} = x_{\max}) \vee (x_{\min} + x_{\max} < 0 \wedge y_{\max} = -x_{\min}). \tag{4}$$

We shall see in the next sub-section that such a formula can be automatically compiled into code such as:

```
if (xmin + xmax >= 0) {
  ymax = xmax;
} else {
  ymax = -xmin;
}
```

2.3. **Generation of the Implementation of the Abstract Domain.** Consider formula 4, defining an abstract transfer function. On this disjunctive normal form we see that the function we have defined is *piecewise linear*: several regions of the range of the input parameters are distinguished (here, $x_{\min} + x_{\max} < 0$ and $x_{\min} + x_{\max} \geq 0$), and on each of these regions, the output parameter is a linear function of the input parameters. Given a disjunct (such as $y_{\max} = -x_{\min} \wedge x_{\min} + x_{\max} < 0$), the domain of validity of the disjunct can be obtained by existential quantifier elimination over the result variable (here $\exists y_{\max}\ (y_{\max} = -x_{\min} \wedge x_{\min} + x_{\max} < 0)$). The union of the domains of validity of the disjuncts is the domain of validity of the full formula. The domains of validity of distinct disjuncts can overlap, but in this case, since $O'_i$ defines a function in the mathematical sense, the functions defined by such disjuncts coincide on their overlapping domains of validity.

This suggests a first algorithm for conversion to an executable form:

(1) Put $O'_i$ into quantifier-free, disjunctive normal form $C_1 \wedge \cdots \wedge C_n$.
(2) For each disjunct $C_i$, obtain the validity domain $V_i$ as a conjunction of linear inequalities and solve for $p'_i$ (obtain $p'_i$ as a linear function $v_i$ of the $p_1, \ldots, p_n$).
(3) Output the result as a cascade of if-then-else and assignments, as in the example at the end of Sec. 2.2.

**Algorithm 1** ToITEtree$(F, z, T)$: turn a formula defining $z$ as a function of the other free variables of $F$ into a tree of if-then-else constructs, assuming that $T$ holds.

$D(= C_1 \wedge \cdots \wedge C_n) \leftarrow$ QElimDNFModulo$(\{\}, F, T)$
**for all** $C_i \in D$ **do**
  $P_i \leftarrow$ QElimDNFModulo$(z, F, T)$
**end for**
$P \leftarrow$ Predicates$(P_1, \ldots, P_n)$
**if** $P = \emptyset$ **then**
**Ensure:** $\exists z\ F$ is always true
  $O \leftarrow$ Solve$(D, z)$
**else**
  $K \leftarrow$ Choose$(P)$
  $O \leftarrow$ IfThenElse$(K,$ ToITEtree$(F, z, T \wedge K),$ ToITEtree$(F, z, T \wedge \neg K))$
**end if**

An if-then-else cascade may be inefficient, since identical conditions may have to be tested several times. We could of course factor out all conditions and assign them to Boolean variables, but then, some of the tests performed may actually not be needed. We therefore propose an algorithm for building an if-then-else *tree*. The idea of the algorithm is as follows:

- Each path in the if-then-else tree corresponds to a conjunction $C$ of conditions (if one goes through the "if" branch of `if (a)` and the "else" branch of `if (b)`, then the path corresponds to $a \wedge \neg b$).
- The formula $O'_i$ is simplified relatively to $C$, a process that prunes out conditions that are always or never satisfied when $C$ holds.

- If the path is deep enough, then the simplified formula becomes a conjunction. One then solves this conjunction to obtain the computed variable (here, $y_{\max}$) as a function.

Our algorithm ToITEtree$(F, z, T)$ (Alg. 1) uses a function QElimDNFModulo $(\vec{v}, F, T)$ that, given a possibly empty vector of variables $\vec{v}$, a formula $F$ and a formula $T$, outputs a quantifier-free formula $F'$ in disjunctive normal form such that $F' \equiv_T \exists \vec{v}\ F$ and no useless predicates are used. Predicates$(F)$ returns the set of atomic predicates of $F$. Solve$(D, z)$ solves a minimal disjunction $D$ of inequalities for variable $z$, assuming that there is at most one solution for $z$ for each choice of the other variables; one simple way to do that is to look for any constraint of the form $z \geq L$ or $z \leq L$ and output $z = L$. Choose$(P)$ chooses any predicate in $P$ (one good heuristic seems to be to choose the most frequent in $P_1, \ldots, P_n$).

Let us take, as a simple example, formula 4. We wish to obtain $y_{\max}$ as a function of $x_{\min}$ and $x_{\max}$, so in the algorithm ToITEtree we set $z \triangleq y_{\max}$. $C_1$ is the first disjunct $x_{\min} + x_{\max} \geq 0 \wedge y_{\max} = x_{\max}$, $C_2$ is the second disjunct $x_{\min} + x_{\max} < 0 \wedge y_{\max} = -x_{\min}$. We project $C_1$ and $C_2$ parallel to $y_{\max}$, obtaining respectively $P_1 = (x_{\min} + x_{\max} \geq 0)$ and $P_2 = (x_{\min} + x_{\max} < 0)$. We choose $K$ to be the predicate $x_{\min} + x_{\max} \geq 0$ (in this case, the choice does not matter, since $P_1$ and $P_2$ are the negation of each other).

- The first recursive call to ToITEtree is made in the context of $T \triangleq (x_{\min} + x_{\max} \geq 0)$. Obviously, $F \wedge T \equiv (y_{\max} = x_{\max}) \wedge T$ and thus $(\exists y_{\max} F) \wedge T \equiv T$.

  QElimDNFModulo$(y_{\max}, F, T)$ will then simply output the formula "true". It then suffices to solve for $y_{\max}$ in $y_{\max} = x_{\max}$. This yields the formula for computing the correct value of $y_{\max}$ in the cases where $x_{\min} + x_{\max} \geq 0$.
- The second recursive call is made in the context of $T \triangleq (x_{\min} + x_{\max} < 0$. The result is $y_{\max} = -x_{\min}$, the formula for computing the correct value of $y_{\max}$ in the cases where $x_{\min} + x_{\max} < 0$.

These two results are then reassembled into an if-then-else statement, yielding the program at the end of §2.2.

The algorithm terminates because paths of depth $d$ in the tree of recursive calls correspond to truth assignments to $d$ atomic predicates among those found in the domains of validity of the elements of the disjunctive normal form of $F$. Since there is only a finite number of such predicates, $d$ cannot exceed that number. A single predicate cannot be assigned truth values twice along the same path because the simplification process in QElimDNFModulo erases this predicate from the formula.

2.4. **Least Inductive Invariants.** We have so far considered programs without loops. We shall now see that not only can we compute the optimal abstract postcondition of a block as a simple, executable function of the parameters of the precondition, but we can also compute the parameters of the least inductive invariant of a program block that is of the form specified by the abstract domain.[7] Beware

[7]In order to specify the least invariant, we would have to quantify over all sets of states, then filter those which are inductive invariants. This is second-order quantification, which we cannot handle. By restricting ourselves to invariants of a certain shape, we replace it by first order quantification.

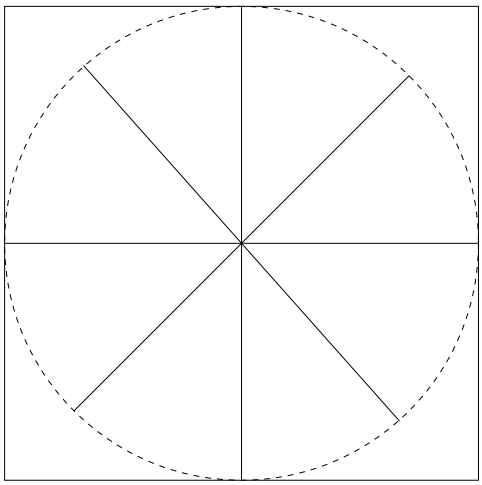

FIGURE 5. The least fixed point representable in the domain $(\mathrm{lfp}(\alpha \circ f \circ \gamma))$ is not necessarily the least approximation of the least fixed point $(\alpha(\mathrm{lfp}\, f))$ inside the abstract domain. For instance, if we take a program initialized by $x \in [-1, 1]$ and $y = 0$, and at each iteration, we rotate the point by $45°$, the least invariant is an 8-point star, and the best approximation inside the abstract domain of intervals is the square $[-1, 1]^2$. However, this square is not an inductive invariant: no rectangle (product of intervals) is stable under the iterations, thus there is no abstract inductive invariant within the interval domain.

that this least inductive invariant found in the abstract domain is in general different from the least element of the abstract domain that includes the least inductive invariant of the system (Fig. 5).

2.4.1. *Stability Inequalities.* Consider a program fragment: `while (c) { p; }`. We have domain definition formulas $F \triangleq L_1(s_1, \ldots, s_m) \leq p_1 \wedge \cdots \wedge L_n(s_1, \ldots, s_m) \leq p_n$ for the precondition of the program fragment , and $F' \triangleq L'_1(s_1, \ldots, s_m) \leq p'_1 \wedge \cdots \wedge L'_n(s_1, \ldots, s_m) \leq p'_n$ for the invariant.

Define $G = [\![c]\!] \wedge [\![p]\!]$. $G$ is a formula whose free variables are $s_1, \ldots, s_m, s'_1, \ldots, s'_m$ such that $(s_1, \ldots, s_m, s'_1, \ldots, s'_m) \models G$ if and only if the state $(s'_1, \ldots, s'_m)$ can be reached from the state $(s_1, \ldots, s_m)$ in exactly one iteration of the loop. A set $W \subseteq \mathbb{Q}^m$ is said to be an *inductive invariant* for the head of the loop if $\forall \vec{s} \in W, \forall \vec{s'}\ (\vec{s}, \vec{s'}) \models G \implies \vec{s'} \in W$. We seek inductive invariants of the shape defined by $F'$, thus solutions for $\vec{p'}$ of the *stability condition*:

$$\forall \vec{s}, \vec{s'}\ F' \wedge G \implies F'[\vec{s'}/\vec{s}]. \tag{5}$$

Not only do we want an inductive invariant, but we also want the initial states of the program to be included in it. The condition then becomes

$$H \triangleq (\forall \vec{s}, F \implies F') \wedge (\forall \vec{s}, \vec{s'}\ F' \wedge G \implies F'[\vec{s'}/\vec{s}]) \tag{6}$$

This formula links the values of the input constraint parameters $p_1, \ldots, p_n$ to acceptable values of the invariant constraint parameters $p'_1, \ldots, p'_n$. In the same way

that our soundness or correctness condition on abstract transformers allowed any sound post-condition, whether optimal or not, this formula allows any inductive invariant of the required shape as long as it contains the precondition, not just the least one.

The intersection of sets defined by $\vec{p'}_1$ and $\vec{p'}_2$ is defined by $\min(\vec{p'}_1, \vec{p'}_2)$. More generally, the intersection of a family of sets, unbounded yet closed under intersection, defined by $\vec{p'} \in Z$ is defined by $\min\{p' \mid p' \in Z\}$. We take for $Z$ the set of acceptable parameters $\vec{p'}$ such that $\vec{p'}$ defines an inductive invariant and $\forall \vec{s}, F \implies F'$; that is, we consider only inductive invariants that contain the set $I = \{\vec{s} \mid F\}$ of precondition states.

We deduce that $p'_i$ is uniquely defined by: $p'_i = \min(\exists p'_1, \ldots, p'_{i-1}, p'_{i+1}, \ldots, p'_n\ H)$ which can be rewritten as

$$(\exists p'_1, \ldots, p'_{i-1}, p'_{i+1}, \ldots, p'_n\ H) \wedge (\forall \vec{q'}\ H[\vec{q'}/\vec{p'}] \implies p'_i \leq q'_i) \tag{7}$$

The free variables of this formula are $p_1, \ldots, p_n, p'_i$. This formula defines a function (in the mathematical sense) defining $p'_i$ from $p_1, \ldots, p_n$. As before, this function can be compiled to an executable version using cascades or trees of tests.

2.4.2. *Simple Loop Example.* To show how the method operates in practice, let us consider first a very simple example (`something_happens` is a nondeterministic choice):

```
int i=0;
while (i <= n) {
  if (something_happens) {
    i=i+1;
    if (i == n) {
      i=0;
    }
  }
}
```

Let us abstract `i` at the head of the loop using an interval $[i_{\min}, i_{\max}]$. For simplicity, we consider the case where the loop is at least entered once, and thus $i = 0$ belongs to the invariant. For better precision, we model each comparison $x \neq y$ over the integers as $x >= y + 1 \vee x <= y - 1$; similar transformations apply for other operators. The formula expressing that such an interval is an inductive invariant is:

$$\begin{aligned} i_{\min} \leq 0 \wedge 0 \leq i_{\max} \wedge \forall i \forall i'\ ((i_{\min} \leq i \wedge i \leq i_{\max} \wedge \\ (((i+1 \leq n-1 \vee i+1 \geq n+1) \wedge i' = i+1) \vee \\ (i+1 = n+1 \wedge i' = 0) \vee i' = i)) \implies (i_{\min} \leq i' \wedge i' \leq i_{\max})) \end{aligned} \tag{8}$$

Quantifier elimination produces:

$$\begin{aligned} (i_{\min} \leq 0 \wedge i_{\max} \geq 0 \wedge i_{\max} < n \wedge -i_{\min} + n - 2 < 0) \vee \\ (i_{\min} \leq 0 \wedge i_{\max} \geq 0 \wedge i_{\max} - n + 1 \geq 0 \wedge i_{\max} < n) \end{aligned} \tag{9}$$

The formulas defining optimal $i_{\min}$ and $i_{\max}$ are:

$$i_{\min} \geq 0 \wedge i_{\min} \leq 0 \wedge n > 0 \tag{10}$$

$$(i_{\max} = 0 \wedge \wedge n > 0 \wedge n < 2) \vee (i_{\max} = n - 1 \wedge i_{\max} \geq 1) \tag{11}$$

We note that this invariant is only valid for $n > 0$, which is unsurprising given that we specifically looked for invariants containing the precondition $i = 0$. The output abstract transfer function is therefore:

```
if (n <= 0) {
  fail();
} else {
  iMin = 0;
  if (n < 2) {
    iMax = 0;
  } else /* n >= 2 */
    iMax = n-1;
  }
}
```

The case disjunction `n < 2` looks unnecessary, but is a side effect of the use of rational numbers to model a problem over the integers. The resulting abstract transfer function is optimal, but on such a simple case, one could have obtained the same using polyhedra [32] or octagons [64].

Let us now consider the same program, simply replacing `n` by the constant `20`. All implementations of intervals (and thus of octagons and polyhedra, since we only have one variable), will overshoot the $i_{\max} = 19$ target when using the traditional widening and narrowing strategies: they will compute `i` $\in [0, 0]$, then $\in [0, 1]$, $\in [0, 2]$ and widen to $[0, +\infty[$, and narrowing will not reduce the interval. Even if we replaced `i == 20` by `i >= 20`, narrowing would still fail to reduce the interval due to the nondeterministic choice since the concrete transfer function $f$, mapping sets of states at the head of the loop to sets of states at the next iteration, is expansive: for all set of states $W$, $W \subseteq f(W)$. This is a well-known weakness of the widening/narrowing approach, and the workaround is a *syntactic* trick known as *widening up to* or *widening with thresholds*: for all variables, the constants to which it is compared are gathered and used as widening steps [11, Sec. 7.1.2]. This syntactic approach fails if tests are more indirect, whereas our semantic approach is not affected.

2.4.3. *Synchronous Data Flow Example: Rate Limiter.* To go back to the original problem of floating-point data in data-flow languages, let us consider the following library block: a *rate limiter*. When compiled into C, such a block in inserted in a reactive loop, as shown below, where `assume(c)` stands for `if (c) {} else {fail();}`:

```
while (true) {
  ...
  e1 = random(); assume(e1 >= e1min && e1 <= e1max);
  e2 = random(); assume(e2 >= e2min && e2 <= e2max);
  e3 = random(); assume(e3 >= e3min && e3 <= e3max);
  olds1 = s1;
  if (random) {
    s1 = e3;
  } else {
    if (e1 - olds1 < -e2) {
      s1 = olds1 - e2;
    }
```

```
    if (e1 - olds1 > e2) {
      s1 = olds1 + e2;
    }
  }
  ...
}
```

We are interested in the input-output behavior of that block: obtain bounds on the output `s1` of the system as functions of bounds on the inputs (`e1`, `e2`, `e3`). Note that in this case, `s1`, `e1`, `e2`, `e3` are *streams*, not single scalars. One difficulty is that the `s1` output is memorized, so as to be used as an input to the next computation step. The semantics of such a block is therefore expressed as a fixed point.

We wish to know the least inductive invariant of the form $s_{1\min} \leq s_1 \leq s_{1\max}$ under the assumption that $e_{1\min} \leq e_{1\max} \wedge e_{2\min} \leq e_{2\max} \wedge e_{3\min} \leq e_{3\max}$. The stability condition yields, after quantifier elimination and projection on $s_{1\max}$ the condition $s_{1\max} \geq e_{1\max} \wedge s_{1\max} \geq e_{3\max}$. Minimization then yields an expression that can be compiled to an if-then-else tree:

```
if (e1max > e3max) {
  s1max = e1max;
} else {
  s1max = e3max;
}
```

This result, automatically obtained, coincides with the intuition that a rate limiter (at least, one implemented with exact arithmetic) should not change the range of the signal that it processes. This program fragment has a rather more complex behavior if all variables and operations are IEEE-754 floating-point, since rounding errors introduce slight differences of regimes between ranges of inputs (Sec. 3.4, 2.5). Rounding errors in the program to be analyzed introduce difficulties for analyzes using widenings, since invariant candidates are likely to be "almost stable", but not truly stable, because of these errors. Again, there exist workarounds so that widening-based approaches can still operate [11, Sec. 7.1.4].

2.5. **Implementations and Experiments.** We have implemented the techniques of Sec. 2 in quantifier elimination packages, including Mathematica[8] and Reduce 3.8[9] + Redlog[10] in addition to our own package, Mjollnir [69].[11] We ignore which exact techniques are implemented within Mathematica. [12] Redlog appears to implement some virtual substitution method [40, 94].

As test cases, we took a library of operators for synchronous programming, having streams of floating-point values as input and outputs. These operators are

[8]Mathematica is a commercial computer algebra package available under an unfree license from Wolfram Research [96].

[9]Reduce is a computer algebra package from Anthony C. Hearn, now available under a modified BSD licence.

[10]Redlog is an extension to Reduce for working over quantified formulas.

[11]Mjollnir is available under a free license from the author's home page. In addition to the author's own quantifier elimination techniques, it implements Ferrante and Rackoff and Loos and Weispfenning's.

[12]Loos and Weispfenning's quantifier elimination procedure is used by Mathematica to perform simplifications over linear inequalities [96, §A.9.5], but we are unsure whether this is the algorithm called by the `Reduce` function.

written in a restricted subset of C and take as much as 20 lines. A front-end based on CIL [73] converts them into formulas, then these formulas are processed and the corresponding abstract transfer functions are pretty-printed. Since for our application, it is important to bound numerical quantities, we chose the interval domain.

For instance, the rate limiter presented in Sec. 2.4.3 was extracted from that library. Since this operator includes a memory (a variable whose value is retained from a call to the operator to the next one), its data-flow semantics is expressed using a fixed-point. When considered with real variables, the resulting expanded formula was approximately 1000 characters long, and with floating point variables approximately 8000 characters long. Despite the length of these formulas, they can be processed by Mjollnir in a matter of seconds. The result can then be saved once and for all.

Analyzers such as Astrée [10, 11, 33] must have special knowledge about such operators, otherwise the analysis results are too coarse (for instance, the intervals do not get stabilized at all). The Astrée development team therefore had to provide specialized, hand-written analyzes for certain operators. In contrast, all linear floating-point operators in the library were analyzed within a fraction of a second using the method in the present article, assuming that floating-point values in the source code were real numbers. If one considered instead the abstraction of floating-point computations using real numbers from Sec. 3.4, computation times did not exceed 17 seconds per operator; the formulas produced are considerably more complex than in the real case. Note that this computation is done once and for all for each operator; a static analyzer can therefore cache this information for further use and need not recompute abstractions for library functions or operators unless these functions are updated.

Our analyzer front-end currently cannot deal with non-numerical operations and data structures (pointers, records, and arrays). It is therefore not yet capable of directly dealing with the real control-command programs that e.g. Astrée accepts, which do not consist purely of numerical operators. We plan to integrate our analysis method into a more generic analyzer. Alternatively, we plan to adapt a front-end for synchronous programming languages such as Simulink, a tool widely used by control/command engineers.

The correctness of the methods described in this article does not rely on any particularity of the quantifier elimination procedure used, provided one also has symbolic computation procedures for e.g. putting formulas in disjunctive normal form and simplifying them. The difference between the various quantifier elimination and simplification procedures is efficiency; experiments showed that ours was vastly more efficient than the others tested for this kind of application. For instance, our implementation of our quantifier elimination algorithm [69] was able to complete the analysis of the rate limiter of Sec. 2.4.3, implemented over the reals, in 1.4 s, and in 17 s with the same example over floating-point numbers, while Redlog took 182 s for the former and could not finish the latter, and Mathematica could analyze neither (out-of-memory). On other examples, our quantifier elimination procedure is faster than the other ones, or can complete eliminations that the others cannot.

## 3. Extensions to the Admissible Domains and Operations

The class of domains and program constructs of the preceding section may seem too limited. We shall see here a few extensions.

3.1. **Infinities.** Consider the interval abstract domain, defined by $x \leq p_2 \wedge -x \leq p_1$. The techniques explained in Sec. 2.1 allow only finite bounds. Yet, it makes sense that $p_1$ and $p_2$ could be equal to $+\infty$ so as to represent infinite intervals. This can be easily achieved by a minor alteration to our definitions. Each parameter $p_i$ is replaced by two parameters $p_i^b$ and $p_i^\infty$. $p_i^\infty$ is constrained to be in $\{0,1\}$ (if the quantifier elimination procedure in use allows Boolean variables, then $p_i^\infty$ can be taken as a Boolean variable); $p_i^\infty = 0$ means that $p_i$ is finite and equal to $p_i^b$, $p_i^\infty = 1$ means $p_i = +\infty$. $L_i \leq p_i$ becomes $(p_i^\infty > 0) \vee (L_i \leq p_i^b)$, $L_i < p_i$ becomes $(p_i^\infty > 0) \vee (L_i < p_i^b)$. After this rewriting, all formulas are formulas of the theory of linear inequalities without infinities and are amenable to the appropriate algorithms.

3.2. **Non-Convex Domains.** Section 2.1 constrains formulas to be conjunctions of inequalities of the form $L_i \leq p_i$. What happens if we consider formulas that may contain disjunctions?

The template linear constraint domains of section 2.1 have a very important property: they are closed under (infinite) intersection; that is, if we have a family $\vec{p} \in W$, then there exist $p_0$ such that $\bigcap_{\vec{p} \in W} \gamma_F(\vec{p}) = \gamma_F(\vec{p}_0)$ (besides, $p_0 = \inf\{\vec{p} \mid \vec{p} \in W\}$). This is what enables us to request the *least* element that contains the exact post-condition, or the least inductive invariant in the domain: we take the intersection of all acceptable elements.

Yet, if we allow non-convex domains, there does not necessarily exist a least element $\gamma_F(\vec{p})$ such that $S \subseteq \gamma_F(\vec{p})$. Consider for instance $S = \{0,1,2\}$ and $F$ representing unions of two intervals $((-x \leq p_1 \wedge x \leq p_2) \vee (-x \leq p_3 \wedge x \leq p_4)) \wedge p_2 \leq p_3$. There are two, incomparable, minimal elements of the form $\gamma_F(\vec{p})$: $p_1 = p_2 = 0 \wedge p_3 = -1 \wedge p_4 = 2$ and $p_1 = 0 \wedge p_2 = 1 \wedge p_3 = -2 \wedge p_4 = 2$.

We consider formulas $F$ built out of linear inequalities $L_i(s_1, \ldots, s_n) \leq p_i$ as atoms, conjunctions, and disjunctions. By induction on the structure of $F$, we can show that $\gamma_F : (\mathbb{R} \cup \{-\infty\})^n \to \mathcal{P}(\mathbb{R}^n)$ is inf-continuous; that is, for any descending chain $(\vec{p}_i)_{i \in I}$ such that $\lim_i \vec{p}_i = \vec{p}_\infty$, then $\gamma_F(\vec{p}_i)$ is decreasing and $\bigcap_{i \in I} \gamma_F(\vec{p}_i) = \gamma_F(\vec{p}_\infty)$. The property is trivial for atomic formulas, and is conserved by greatest lower bounds ($\wedge$) as well as binary least upper bounds ($\vee$).

Let us consider a set $S \subseteq \mathcal{P}(\mathbb{R}^n)$, stable under arbitrary intersection (or at least, greatest lower bounds of descending chains). $S$ can be for instance the set of invariants of a relation, or the set of over-approximations of a set $W$. $\gamma_F^{-1}(S)$ is the set of suitable domain parameters; for instance, it is the set of parameters representing inductive invariants of the shape specified by $F$, or the set of representable over-approximations of $W$. $\gamma_F^{-1}(S)$ is stable under greatest lower bounds of descending chains: take a descending chain $(\vec{p}_i)_{i \in I}$, then $\gamma_F(\lim_i \vec{p}_i) = \cap_i \gamma_F(\vec{p}_i) \in S$ by inf-continuity and stability of $S$. By Zorn's lemma, $\gamma_F^{-1}(S)$ has at least one minimal element.

Let $P[\vec{p}]$ be a formula representing $\gamma_F^{-1}(S)$ (Sec. 2.1 proposes formulas defining safe post-conditions and inductive invariants). The formula $G[\vec{p}] \triangleq P[\vec{p}] \wedge \forall \vec{p'}\ P[\vec{p'}] \wedge \vec{p'} \leq \vec{p} \implies \vec{p} \leq \vec{p'}$ defines the minimal elements of $\gamma^{-1}(S)$.

For instance, consider $\vec{p} = (a, b, c, d)$, $F \triangleq (-x \leq a \wedge x \leq b) \vee (-x \leq c \wedge x \leq d)$, representing unions of two intervals $[-a, b] \cup [-c, d]$. We want upper-approximations of the set $\{0, 1, 3\}$; that is $P[\vec{p}] \triangleq \forall x\ (x = 0 \vee x = 1 \vee x = 3 \implies F[\vec{p}, x])$. We add the constraint that $-a \leq b \wedge b \leq -c \wedge -c \leq d$, so as not to obtain the same solutions twice (by exchange of $(a, b)$ and $(c, d)$) or solutions with empty intervals. By quantifier elimination over $G$, we obtain $(a = 0 \wedge b = 1 \wedge c = -3 \wedge d = 3) \vee (a = 0 \wedge b = 0 \wedge c = -1 \wedge d = 3)$, that is, either $[0, 0] \cup [1, 3]$ or $[0, 1] \cup [3, 3]$.

3.3. **Domain Partitioning.** Non-convex domains, in general, are not stable under intersections and thus "best abstraction" problems admit multiple solutions as minimal elements of the set of correct abstractions. There are, however, non-convex abstract domains that are stable under intersection and thus admit least elements as well as the template linear constraint domains of Sec. 2.1: those defined by partitioning of the state space. Consider pairwise disjoint subsets $(C_i)_{i \in I}$ of the state space $\mathbb{Q}^m$, and abstract domains stable under intersection $(S_i)_{i \in I}$, $S_i \subseteq \mathcal{P}(C_i)$. Elements of the partitioned abstract domain are unions $\bigcup_{i \in I} s_i$ where $s_i \in S_i$. If $(\bigcup_i s_{i,j}])_{j \in J}$ is a family of elements of the domain, then $\bigcap_{j \in J} \left(\bigcup_{i \in I} s_{i,j}]\right) = \bigcup_{i \in I} \bigcap_{j \in J} s_{i,j}$; that is, intersections are taken separately in each $C_i$.

Take a family $(F_i[\vec{p}])_{i \in I}$ of formulas defining template linear constraint domains (conjunctions of linear inequalities $L_i(s_1, \ldots, s_n) \leq p_i$) and a family $(C_i)_{i \in I}$ of formulas such that for all $i$ and $i'$, $C_i \wedge C_{i'}$ is equivalent to $\mathsf{false}$ and $C_1 \vee \cdots \vee C_l$ is equivalent to $\mathsf{true}$. $F = (C_1 \wedge F_1) \vee \cdots \vee (C_l \wedge F_l)$ then defines an an abstract domain such that $\gamma_F$ is a inf-morphism. All the techniques of Sec. 2.1 then apply.

3.4. **Floating-Point Computations.** Real-life programs do not operate on real numbers; they operate on fixed-point or floating-point numbers. Floating point operations have few of the good algebraic properties of real operations; yet, they constitute approximations of these real operations, and the *rounding error* introduced can be bounded.

In IEEE floating-point [54], each atomic operation (noting $\oplus$, $\ominus$, $\otimes$, $\oslash$, $\sqrt{}_f$ for operations so as to distinguish them from the operations $+$, $-$, $\times$, $/$, $\sqrt{}$ over the reals) is mathematically defined as the image of the exact operation over the reals by a rounding function.[13] This rounding function, depending on user choice, maps each real $x$ to the nearest floating-point value $r_n(x)$ (*round to nearest mode*, with some resolution mechanism for non representable values exactly in the middle of two floating-point values), $r_{-\infty}(x)$ the greatest floating-point value less or equal to $x$ (*round toward* $-\infty$), $r_{+\infty}(x)$ the least floating-point value greater or equal to $x$ (*round toward* $+\infty$), $r_0(x)$ the floating-point value of the same sign as $x$ but whose magnitude is the greatest floating-point value less or equal to $|x|$ (*round toward* 0). If $x$ is too large to be representable, $r(x) = \pm\infty$ depending on the size of $x$

The semantics of the rounding operation cannot be exactly represented inside the theory of linear inequalities.[14] As a consequence, we are forced to use an axiomatic

[13]We leave aside the peculiarities of some implementations, such as those of most C compilers over the 32-bit Intel platform where there are "extended precisions" types used for some temporary variables and expressions can undergo double rounding. [72]

[14]To be pedantic, since IEEE floating-point formats are of a finite size, the rounding operation could be exactly represented by enumeration of all possible cases; this would anyway be impossible in practice due to the enormous size of such an enumeration.

over-approximation of that semantics: a formula linking a real number $x$ to its rounded version $r(x)$.

Miné [65] uses an inequality $|r(x) - x| \leq \varepsilon_{\text{rel}} \cdot |x| + \varepsilon_{\text{abs}}$, where $\varepsilon_{\text{rel}}$ is a *relative error* and $\varepsilon_{\text{abs}}$ is an *absolute error*, leaving aside the problem of overflows. The relative error is due to rounding at the last binary digit of the significand, while the *absolute error* is due to the fact that the range of exponents is finite and thus that there exists a least positive floating-point number and some nonzero values get rounded to zero instead of incurring a relative error.

Because our language for axioms is richer than the interval linear forms used by Miné, we can express more precise properties of floating-point rounding. We recall briefly the characteristics of IEEE-754 floating-point numbers. Nonzero floating point numbers are represented as follows: $x = \pm s.m$ where $1 \leq m < 2$ is the *mantissa* or *significand*, which has a fixed number $p$ of bits, and $s = 2^e$ the *scaling factor* ($E_{\min} \leq e \leq E_{\max}$ is the *exponent*). The difference introduced by changing the last binary digit of the mantissa is $\pm s.\varepsilon_{\text{last}}$ where $\varepsilon_{\text{last}} = 2^{-(p-1)}$: the *unit in the last place* or *ulp*. Such a decomposition is unique for a given number if we impose that the leftmost digit of the mantissa is 1 — this is called a *normalized representation*. Except in the case of numbers of very small magnitude, IEEE-754 always works with normalized representations. There exists a least positive normalized number $m_{\text{normal}}$ and a least positive denormalized number $m_{\text{denormal}}$, and the denormals are the multiples of $m_{\text{denormal}}$ less than $m_{\text{normal}}$. All representable numbers are multiples of $m_{\text{denormal}}$.

Consider for instance floating-point addition or subtraction $x = \pm a \pm b$. Suppose that $0 \leq x \leq m_{\text{normal}}$. $a$ and $b$ are multiples of $m_{\text{denormal}}$ and thus $a - b$ is exactly represented as a denormalized number; therefore $r(x) = x$. If $x > m_{\text{normal}}$, then $|r(x) - x| \leq \varepsilon_{\text{rel}}.x$. The cases for $x \leq 0$ are symmetrical. We can therefore characterize $r(x) - x$ using linear inequalities through case analysis over $x$: $\mathit{Round}_+(a \oplus b, a + b)$ (respectively, $\mathit{Round}_+(a \ominus b, a - b)$) holds, where

$$\mathit{Round}_+(r, x) \triangleq (x \leq m_{\text{normal}} \wedge r = x) \vee (x > m_{\text{normal}} \wedge -\varepsilon_{\text{rel}}.x \leq r - x \leq \varepsilon_{\text{rel}}.x \tag{12}$$

$$\begin{aligned} \mathit{Round}(r, x) \triangleq (x = 0 \wedge r = 0) \vee (x > 0 \wedge r \geq 0 \wedge \mathit{Round}_+(r, x)) \vee \\ (x < 0 \wedge r \leq 0 \wedge \mathit{Round}_+(-r, -x)) \end{aligned} \tag{13}$$

To each floating-point expression $e$, we associated a "rounded-off" variable $r_e$, the value of which we constrain using $\mathit{Round}(r_e, e)$ or $\mathit{Round}_+(r_e, e)$. For instance, a expression $e = a \oplus b$ is replaced by a variable $r_e$, and the constraint $\mathit{Round}_+(r_e, a+b)$ is added to the semantics. In the case of a compound expression $e = ab + c$, we introduce $e_1 = ab$, and we obtain $\mathit{Round}_+(r_e, r_{e_1} + c) \wedge \mathit{Round}(r_{e_1}, ab)$. If we know that the compiler uses a fused multiply-add operator, we can use $\mathit{Round}(r_e, ab + c)$ instead.

3.5. **Integers.** We have mentioned in § 1.2.3 that Presburger arithmetic admits quantifier elimination. Conceivably, we could handle integer programs using quantifier elimination techniques sur as Cooper's. Problems over the integers are however more difficult than problems over the reals, which are already costly, and we did not follow this direction.

Instead, we used a *relaxation* approach: all integers are treated as reals; strict inequalities $a < b$ where both sides are integers are recoded $a \leq b - 1$. In some

cases, such as the McCarthy 91 function example from §4.2, it is necessary to constraint the reasoning procedures so that they consider that the negation of $a \leq b$ is $a \geq b+1$. We hope that improvements of quantifier elimination algorithms will be able to allow a more elegant approach.

Another issue is that in many programming languages, integers are bounded and that arithmetic operations are actually performed modulo $2^n$ (with $n$ typically 8, 16, 32 or 64). The problem then lies within an enormous, but finite, state space. Clever techniques for reasoning about *bit-vector arithmetic* are being investigated by the SMT-solving community. Again, we hope that future work will provide good quantifier elimination techniques for this arithmetic, or combinations thereof with the linear theory of reals.

## 4. Complex control flow

We have so far assumed no procedure call, and at most one single loop. We shall see here how to deal with arbitrary control flow graphs and call graph structures.

4.1. **Loop Nests.** In Sec. 2.4, we have explained how to abstract a single fixed point. The method can be applied to multiple nested fixed points by replacing the inner fixed point by its abstraction. For instance, assume the rate limiter of Sec. 2.4.3 is placed inside a larger loop. One may replace it by its abstraction:

```
if (e1max > e3max) {
  s1max = e1max;
} else {
  s1max = e3max;
}
assume(s1 <= s1max);
/* and similar for s1min */
```

Alternatively, we can extend our framework to an arbitrary control flow graph with nested loops, the semantics of which is expressed as a single fixed point. We may use the same method as proposed by Gulwani et al. [51, §2] and other authors. First, a *cut set* of program locations is identified; any cycle in the control flow graph must go through at least one program point in the cut set. In widening-based fixed point approximations, one classically applies widening at each point in the cut set. A simple method for choosing a cut set is to include all targets of back edges in a depth-first traversal of the control-flow graph, starting from the start node; in the case of structured program, this amounts to choosing the head node of each loop. This is not necessarily the best choice with respect to precision, though [51, §2.3]; Bourdoncle [14, Sec. 3.6] discusses methods for choosing such as cut-set.

To each point in the cut set we associate an element in the abstract domain, parameterized by a number of variables. The values of these variables for all points in the cut-set defines an invariant candidate. Since paths between elements of the cut sets cannot contain a cycle, their denotational semantics can be expressed simply by an existentially quantified formula. Possible paths between each source and destination elements in the cut-set defined a stability condition (Formula 5). The conjunction of all these stability conditions defines acceptable inductive invariants. As above, the least inductive invariant is obtained by writing a minimization formula (Sec. 2.4).

Let us take a simple example:

```
i=0;
while(true) { /* A */
  if (choice()) {
    j=0;
    while(j < i) { /* B */
      /* something */
      j=j+1;
    }
    i=i+1;
    if (i==20) {
      i=0;
    }
  } else {
    /* something */
  }
}
```

We choose program points $A$ and $B$ as cut-set. At program point A, we look for an invariant of the form $I_A(i,j) \triangleq i_{\min,A} \leq i \leq i_{\max,A}$, and at program point B, for an invariant of the form $I_B(i,j) \triangleq i_{\min,B} \leq i \leq i_{\max,B} \wedge j_{\min} \leq j \leq j_{\max} \wedge \delta_{\min} \leq i - j \leq \delta_{\max}$ (a *difference-bound* invariant). The (somewhat edited for brevity) stability formula is written:

$$
\begin{aligned}
(14)\quad & \forall j\ I_A(0,j) \wedge \forall i \forall j\ ((I_B(i,j) \wedge j \geq i \wedge (i+1 \leq 19 \vee \\
& \quad i+1 = 20 \vee i+1 \geq 21)) \Rightarrow \mathrm{If}[i+1=20, I_A(0,j), I_A(i+1,j)]) \wedge \\
& \qquad \forall i \forall j\ (I_A(i,j) \Rightarrow I_B(i,0)) \wedge \forall i \forall j\ ((I_B(i,j) \wedge j < i) \\
& \qquad\qquad \Rightarrow I_B(i,j+1))
\end{aligned}
$$

Replacing $I_A$ and $I_B$ into this formula, then applying quantifier elimination, we obtain a formula defining all acceptable tuples $(i_{\min,A}, i_{\max,A}, i_{\min,B}, i_{\max,B}, j_{\min}, j_{\max}, \delta_{\min}, \delta_{\max})$. Optimal values are then obtained by further quantifier elimination: $i_{\min,A} = i_{\min,B} = j_{\min} = 0$, $i_{\max,A} = i_{\max,B} = 19$, $j_{\max} = 20$, $\delta_{\min} = 1$, $\delta_{\max} = 19$.

The same example can be solved by replacing 20 by another variable `n` as in Sec. 2.4.2.

4.2. **Procedures and Recursive Procedures.** We have so far considered abstractions of program blocks with respect to sets of program states. A program block is considered as a transformer from a state of input program states to the corresponding set of output program states. The analysis outputs a sound and optimal (in a certain way) abstract transformer, mapping an abstract set of input states to an abstract set of output states.

Assuming there are no recursive procedures, procedure calls can be easily dealt with. We can simply inline the procedure at the point of call, as done in e.g. Astrée [10, 11, 33]. Because inlining the concrete procedure may lead to code blowup, we may also inline its abstraction, considered as a nondeterministic program. Consider a complex procedure `P` with input variable `x` and output variable `x`. We abstract the procedure automatically with respect to the interval domain for the postcondition $(m_z \leq z \leq M_z)$; suppose we obtain $M_z := 1000; m_z := x$

then we can replace the function call by `z <= 1000 && z >= x`. This is a form of *modular interprocedural analysis*: considering the call graph, we can abstract the leaf procedures, then those calling the leaf procedures and so on. This method is however insufficient for dealing with recursive procedures.

In order to analyze recursive procedures, we need to abstract not sets of states, but sets of pairs of states, expressing the input-output relationships of procedures. In the case of recursive procedures, these relationships are the least solution of a system of equations.

To take a concrete example, let us consider McCarthy's famous "91 function" [61, 62], which, non-obviously, returns 91 for all inputs less than 101:

```
int M(int n) {
  if (n > 100) {
    return n-10;
  } else {
    return M(M(n+11));
  }
}
```

The concrete semantics of that function is a relationship $R$ between its input $n$ and its output $r$. It is the least solution of

$$R \supseteq \{(n,r) \in \mathbb{Z}^2 \mid (n > 100 \wedge r = n - 10) \vee (n \leq 100 \wedge \exists n_2 \in \mathbb{Z}(n+11, n_2) \in R \wedge (n_2, r) \in R)\} \tag{15}$$

We look for a inductive invariant of the form $I \triangleq ((n \geq A) \wedge (r - n \geq \delta) \wedge (r - n \leq \Delta)) \vee ((n \leq B) \wedge (r = C))$, a non-convex domain (Sec. 3.2). By replacing $R$ by $I$ into inclusion 15, and by universal quantification over $n, r, n_2$, we obtain the set of admissible parameters for invariants of this shape. By quantifier elimination, we obtain $(C = 91) \wedge (\delta = \Delta = -10) \wedge (A = 101) \wedge (B = 100)$ within a fraction of a second using Mjollnir (see Sec. 2.5).

In this case, there is a single acceptable inductive invariant of the suggested shape. In general, there may be parameters to optimize, as explained in Sec. 2.4. The result of this analysis is therefore a map from parameters defining sets of states to parameters defining sets of pairs of states (the abstraction of a transition relation). This abstract transition relation (a subset of $X \times Y$ where $X$ and $Y$ are the input and output state sets) can be transformed into an abstract transformer in $X^\sharp \to Y^\sharp$ as explained in Sec. 2.2. Such an interprocedural analysis may also be used to enhance the analysis of loops [63].

## 5. Optimal abstraction over polynomial constraint domains

We now consider the abstraction of program states (in $\mathbb{R}^V$) using domains defined by polynomial constraints, a natural extension of those seen previously (§2.1); the orthogonal extensions from § 3 also apply. Instead of quantifier elimination in linear real arithmetic, we shall use quantifier elimination in the theory of real closed fields. One difference, though, is that we will not be able to produce nice, closed form formulas.

### 5.1. Generalization.

We generalize the constructs of §2, except those of § 2.3, to formulas over polynomial inequalities.

The same results hold:

- For any loop-free program code, and any template polynomial abstract domain with parameters $p_1, \ldots, p_n$, there is a family of formulas $F_1, \ldots, F_n$ that uniquely defines the optimal parameters $p'_1, \ldots, p'_n$ of the postcondition with respect those $p_1, \ldots, p_n$ in the precondition (the free variables of $F_i$ are among $p_1, \ldots, p_n, p'_i$).
- For any loop, and any template polynomial abstract domain with parameters $p_1, \ldots, p_n$, there is a family of formulas $F_1, \ldots, F_n$ that uniquely defines the optimal parameters $p'_1, \ldots, p'_n$ of the least inductive invariant for that loop, with respect those $p_1, \ldots, p_n$ in the precondition (the free variables of $F_i$ are among $p_1, \ldots, p_n, p'_i$).

The main obstacle is the high cost of quantifier elimination in the theory of real closed fields. The other crucial difference is that it is in general impossible to move from such a formulas to a formula computing $p'_i$ from $p_1, \ldots, p_n$, as we did in § 2.3. By performing the cylindrical algebraic decomposition with the variable $p_1, \ldots, p_n$ first, we could obtain the a tree structure with case disjunctions, as the output of Algorithm 1. But at the leaves, we would obtain formulas defining $p'_i$ as a specific root of a polynomial in the variable $p'_i$, with coefficients themselves polynomials in $p_1, \ldots, p_n$. The Abel-Ruffini theorem, from Galois theory, states that for polynomials in one variable of degrees higher or equal to 5, there is in general no way to express the value of the roots using only arithmetic operations ($+$, $-$, $\times$, $/$) and radicals ($\sqrt[n]{\ }$). This implies that, in general, there is no way to convert the leaves of the case disjunction tree into formulas giving the $p'_i$ as expressions built from $p_1, \ldots, p_n$, arithmetic operations ($+$, $-$, $\times$, $/$) and radicals ($\sqrt[n]{\ }$).

Let us now assume that there are no precondition parameters $p_1, \ldots, p_n$ or, equivalently, that we know exactly their value. Would it be at least possible to compute the values of the $p'_i$? Each of these numbers is defined as the unique solution of a conjunction of polynomial equalities and inequalities. We shall now see that it is possible to compute these numbers to arbitrary precision from such a representation.

5.2. **Computable reals.** Our abstract domains will "compute" reals in an indirect way: instead of computing the value of a real number (which is impossible to do exactly in most cases), the abstract domain will define it as the unique solution of a quantifier-free formula with one variable; for instance, $\sqrt{2}$ would be defined as the unique $x$ such that $x^2 = 2 \wedge x > 0$. In this section, we show that given such a characterization, one can algorithmically bound the real number with arbitrary precision; that is, given $\varepsilon \in \mathbb{Q}$, $\varepsilon > 0$, obtain $m, M \in \mathbb{Q}$ such that $m \leq x \leq M$ and $M - m \leq \varepsilon$. More generally, we shall show that for any quantifier-free formula in the theory of real closed fields with one free variable, we can obtain a finite description of its domain of validity, such that all numbers used inside the description are computable with arbitrary precision.[15]

We define computable reals through *approximation functions*: instead of a real $r$, which cannot be represented directly in a machine, we shall consider a computable function $\tilde{r}$ taking a positive rational number $\varepsilon$ as a parameter and outputting a

[15]This is a generalization of a result of Turing, that real algebraic numbers are computable [92, §1.vi].

couple $(m, M)$ of rational numbers such that $M - m \leq \varepsilon$ and $m \leq r \leq M$, called an *$\varepsilon$-approximation.*[16]

We shall give our algorithms in a "literate programming" or "proof-carrying code" fashion, mixing each algorithm with a proof of its correctness. For the sake of simplicity, we preferred to give all algorithms "from scratch" instead of relying on advanced techniques.[17]

Let $a$ and $b$ be computable reals given by approximation functions $\tilde{a}$ and $\tilde{b}$. It is straightforward to compare these two numbers, provided that we know that they are different.

**Algorithm 5.1** (Compare — Compare two computable reals known to be different)**.** If $a \neq b$, then there is an algorithm that decides whether $a < b$ or $a > b$ given approximation functions $\tilde{a}$ and $\tilde{b}$: start with $\varepsilon = 1$; compare the intervals $\tilde{a}(\varepsilon)$ and $\tilde{b}(\varepsilon)$; if they do not overlap, the case is settled, otherwise divide $\varepsilon$ by 10 and try again. The algorithm will terminate at the latest when $\varepsilon < |b - a|/2$.

This algorithm loops forever if $a = b$. Throughout the rest of this section, we shall take precautions so that we never use Compare on operands that could be equal.[18] We then define elementary arithmetic operators over approximation functions:

**Algorithm 5.2** (Plus — Add two computable reals)**.** $a + b$ is also a computable real: for $\varepsilon > 0$, compute $\varepsilon/2$-approximations $[m_a, M_a]$ of $a$, $[m_b, M_b]$ of $b$, and output $[m_a + m_b, M_a + M_b]$ as a $\varepsilon$-approximation of $a + b$.

A similar algorithm works for $a - b$, defining Minus.

**Algorithm 5.3** (Mult — Multiply two computable reals)**.**

- compute a $1/2$-approximation $[m_a, M_a]$ of $a$; if $0 \in [m_a, M_a]$, then write $a \cdot b = (a+1) \cdot b - b$ and the problem is reduced to the case where $a$ cannot be zero;
- decide whether $a < 0$ or $a > 0$ by testing whether $m_a < 0$: if $a < 0$, write $a \cdot b = -((-a) \cdot b)$ and the problem is reduced to the case where $a > 0$; we also have computed a upper bound $L_a \in \mathbb{Q}$ of $a$;
- do similarly with $b$ and the problem is reduced to the case where $b > 0$; we also have computed a upper bound $L_b \in \mathbb{Q}$ of $b$;
- compute a $\varepsilon/(2L_b)$-approximation $[m_a, M'_a]$ of $a$ and a $\varepsilon/(2L_a)$-approximation $[m_b, M'_b]$ of $b$; let $M_A = \min(M'_A, L_a)$ and $M_b = \min(M'_b, L_b)$ and output $[m_a m_b, M_a M_b]$; this is a $\varepsilon$-approximation of $a.b$ since $M_a M_b - m_a m_b = M_a(M_b - m_b) + m_b(M_a - m_a) \leq \varepsilon$.

Now for three algorithms that will be later used as subroutines:

[16] Turing's original characterization of the class of computable reals [92] [93, Def. 4.1.12] used machines that enumerated the decimals of the number. The class of computable reals defined in this fashion is identical to ours, but there are drawbacks to this representation: it may be necessary in order to compute the $n$-th digit of a result to go arbitrarily far in the representation of the operands. We thus rather use a representation very close to that of Weihrauch [93, §1.3.2]

[17] Alternatively, the same result may be reached using published algorithms [7, Alg. 10.4 to 10.17] for isolating roots of polynomials, pairs of polynomials or finding the sign of a polynomial at the roots of another, together with a dichotomy solving method.

[18] This is actually an essential restriction of any representation of computable reals. [93, Th. 4.1.16]

**Algorithm 5.4** (DecideSign — Decide the sign of $P(x)$ if $P(x) \neq 0$)**.** It follows that if $P \in \mathbb{Q}[X]$, and $r$ is a real given by $\tilde{r}$ such that $P(r) \neq 0$, then we can decide whether $P(r) < 0$ or $P(r) > 0$ using Plus and Mult over the polynomial structure, then Compare.

**Algorithm 5.5** (FindRoot — Find the unique root of $P$ in an interval $[r_1, r_2]$ of monotonicity)**.** Let $r_1 < r_2$, given by $\tilde{r}_1$ and $\tilde{r}_2$, and $P$ a polynomial such that $P$ is strictly increasing over $[r_1, r_2]$, $P(r_1) < 0$ and $P(r_2) > 0$. Let $\varepsilon > 0$. Compute $[m_1, M_1]$ a $\varepsilon$-approximation of $r_1$ and $[m_2, M_2]$ an $\varepsilon$-approximation of $r_2$. If $P(M_1) \geq 0$, then $[m_1, M_1]$ is an $\varepsilon$-approximation of $r_0$. If $P(m_2) \leq 0$, then $[m_2, M_2]$ is a $\varepsilon$-approximation of $r_0$. We thus suppose $P(M_1) < 0$ and $P(m_2) > 0$ and apply a dichotomy algorithm between the two, until we reach the desired precision.

**Algorithm 5.6** (FindRootInf — Find the unique root of $P$ in an interval $(-\infty, r_2]$ of monotonicity[19])**.** If we know that $P$ is strictly increasing on $(-\infty, r_2]$, $P(r_2) > 0$, noting $r$ the root of $P$ such that $r < r_2$, then, similarly, let $\varepsilon > 0$; compute $[m_2, M_2]$ a $\varepsilon$-approximation of $r_2$; if $P(m_2) \leq 0$ then $[m_2, M_2]$ is a $\varepsilon$-approximation of $r$. If $P(m_2) > 0$ then take $k \in \mathbb{N}$, $-k < m_2$, $k$ increasing until $P(-k) < 0$; then apply the dichotomy algorithm between $-k$ and $m_2$.

Let us recall a familiar result, which we shall use with $K = \mathbb{Q}$ and $K' = \mathbb{R}$:

**Lemma 1.** *Let $K$ be a field and $K'$ an extension of $K$. If $\xi \in K'$ is a common root of nonzero polynomials $P$ and $Q$ from $K[X]$, then it is a root of their greatest common divisor* $\gcd(P, Q)$ *in $K[X]$. Thus, co-prime polynomials have no common root.*

*Proof.* $K[X]$ is a principal ring [59, Ch. 4, Th. 1.2], there exist polynomials $A$ and $B$ in $K[X]$ such that $\gcd(P, Q) = A.P + B.Q$. The result follows by applying both members of the equation to $\xi$. □

Several of our algorithms operate on *sign diagrams*. A sign diagram for a nonzero polynomial $P \in \mathbb{Q}[X]$ is a sequence $-, \tilde{r}_1, +, \tilde{r}_2, -, \tilde{r}_3, +, \ldots, \tilde{r}_n, +$, where the $\tilde{r}_i$ are approximation functions for the roots of $P$. Such a diagram means that the polynomial function $P(x)$ is negative for large negative $x$, then passes a root $r_1$ that can be approximated to arbitrary precision by $\tilde{r}_1$, then becomes positive, etc.

Sign diagrams for polynomials of degrees 0 and 1 are straightforward to compute, as are the first and final signs of the diagram for any polynomial, which are obtained from the parity of the degree of the polynomial and the sign of the leading coefficient. Given the diagram of $P$ and a nonnegative exponent $e$, it is straightforward to compute the diagram for $P^e$; and given the diagram for $P$ and a coefficient $a \in \mathbb{Q}$, it is also straightforward to compute the diagram for $aP$.

Given two polynomials $P$ and $Q$ with no common roots, one obtains the sign diagram for $P.Q$ through a simple sorted list merging procedure using Compare. This algorithm, however, does not apply in case $P$ and $Q$ have common roots. We use the fact (Lem. 1) that the common roots of $P$ and $Q$ are the roots of the greatest common divisor $\gcd(x, y)$ of these polynomials to work around this difficulty. $\gcd(x, y)$ can be computed using Euclid's algorithm.

[19] We note open intervals $(a, b)$, closed intervals $[a, b]$.

**Algorithm 5.7** (SignDiagram — Compute the sign diagram of a polynomial)**.** We shall now show how to compute the sign diagram of a polynomial $P$ by induction on the degree $n$ of $P$. We have already noted that it is trivial to compute diagrams for polynomials of degrees 0 and 1. We now shall suppose that we can compute the sign diagrams of polynomials of degree less than $n$, and show that we can compute the sign diagram of a polynomial of degree $n$. First, define a subroutine:

**Algorithm 5.8** (SignDiagramProduct — )**.** Take as input a list $(P_1, e_1), \ldots, (P_m, e_m)$ of couples each formed of a polynomial of degree less than $n$ and a positive exponent, output the sign diagram of the product $P_1^{e_1} \times \cdots \times P_m^{e_m}$. We proceed by induction on the sum of the degrees of $P_1, \ldots, P_m$. If this sum is 0 or 1, then the case is trivial.

- Check whether there exist $P_i$ and $P_j$ ($i \neq j$) not co-prime; if so, compute $Q_i = P_i / \gcd(P_i, P_j)$ and $Q_j = P_j / \gcd(P_i, P_j)$, then replace $(P_i, e_i)$ and $(P_j, e_j)$ by $(Q_i, e_i)$, $(Q_j, e_j)$, $(\gcd(P_i, P_j), e_i + e_j)$ in the list. The sum of the degrees has decreased by the degree of $\gcd(P_i, P_j)$, but the product $P_1^{e_1} \times \cdots \times P_m^{e_m}$ has stayed the same, and thus we can solve the problem through a recursive call.
- Otherwise, the $P_i$ are pairwise co-prime. Since they all have degree less than $n$, we can obtain their sign diagrams. We then apply the exponent algorithm, then the algorithm for the sign diagrams of a product of polynomials with no common roots.

Consider now a polynomial $P$ of degree $n$.

- If $P$ and its derivative $P'$ are not co-prime, then let $Q = P / \gcd(P, P')$. $Q$ and $\gcd(P, P')$ will have degree at most $n - 1$, so we can invoke SignDiagramProduct and obtain the sign diagram of their product $P$.
- If they are co-prime: $P$ only has single roots. Compute the sign diagram of $P'$, which gives us intervals of monotonicity for $P$. Then, compute the sign diagram of $P$ as follows:
  - The leftmost sign is deduced from the leading coefficient and parity of the degree of $P$. Without loss of generality, we shall suppose it is positive.
  - Compute the sign of $P(r_1)$ (using DecideSign) where $r_1$ is the first root in the sign diagram of $P'$; this is possible because $r_1$ is not a root of $P$. If it is negative, search for a root of $P$ to the left of $r_1$ using FindRootInf.
  - For each subsequent root $r_k$ of $P'$, compute the sign of $P(r_k)$ (using DecideSign), and if it is different from the sign of $P(r_{k-1})$, search for a root of $P$ in $[r_{k-1}, r_k]$ using FindRoot.

For a system $S$ of polynomial equalities or inequalities over a real variable $x$, we call *validity diagram* a sequence $b_0, r_1(B_1), b_1, r_2(B_2), \ldots, r_m(B_m)$ where $r_1, \ldots, r_m$ are given by approximation functions $\tilde{r}_1, \ldots, \tilde{r}_m$, and the $b_i$ and $B_i$ are booleans; $b_0$ says whether $S$ is always or never satisfied over $(-\infty, r_1)$, $B_1$ whether $S$ is satisfied at $r_1$, $b_1$ whether $S$ is always or never satisfied over $(r_1, r_2)$ and so on.

**Algorithm 5.9** (Domain — Domain of validity of a quantifier-free formula with one free variable)**.** Consider now a quantifier-free formula $F$ with one free variable,

made up of of polynomial equalities and inequalities $P_i \bowtie 0$. Similarly as in SignDiagramProduct, take greatest common divisors until obtaining a base $B_k$ of pairwise co-prime polynomials such that for all $i$, $P_i$ can be written $P_i = B_1^{e_1} \times \cdots \times B_m^{e_m}$. Compute the sign diagrams of all $B_k$. The validity diagram of $F$ can be computed from the $B_k$ using, as previously, a variant of the merging of sorted lists and the fact the $B_k$, pairwise, have no common roots.

By preprocessing formulas through quantifier elimination, we can algorithmically approximate to arbitrary precision any (algebraic) real defined by a formula in the theory of real closed fields.

**Corollary 2.** *If $F$ is a formula of the theory of real closed fields with one free variable, such that $F$ defines a single real, then this real is algebraic and can be algorithmically approximated to arbitrary precision.*

## 6. Related work

There is a sizeable amount of literature concerning relational numerical abstract domains; that is, domains that express constraints between numerical variables. Convex polyhedra were proposed in the 1970s [32, 53], and there have been since then many improvements to the technique; a bibliography was gathered by Bagnara et al. [4]. Algorithms on polyhedra are costly and thus a variety of domains intermediate between simple interval analysis and convex polyhedra were proposed [21, 64, 85]. All these domains compute invariants using a *widening* operator [30, 31, 32], as described in §1.1. There is, however, no guarantee that the resulting invariant is the best representable in the abstract domain, even with the use of *narrowing* iterations; this is one difference with our proposal, which computes the best representable inductive invariant.

Another difference is that these domains are designed to work with numerical values for the input constraints, thus the computation must be done for every value of the input constraints parameters. Using simple program transformations, they may also apply to symbolic input constraints (constraint parameters being taken as extra variables), but in general this will lead to bad results; for instance, the input-output relationship for the rate limiter of Sec. 2.4.3 is not convex, while numerical abstract domains in the literature are convex. In comparison the algorithm in this article can be run once to obtain a *formula* that gives the best invariant depending on the input constraints, allowing *modular* analysis.

Several methods have been proposed to synthesize invariants without using widening operators [24, 28, 84]. In common with us, they express as constraints the conditions under which some parametric invariant shape truly is an invariant, then they use some resolution or simplification technique over those constraints. Again, these methods are designed for solving the problem for one given set of constraints on the inputs, as opposed to finding a relation between the output or fixed-point constraints and the input constraints. In some cases, the invariant may also not be minimal.

Bagnara et al. [5, 6] proposed improvements over the "classical" widenings on linear constraint domains [53]. Gopan and Reps [50] introduced "lookahead widenings": standard widening-based analysis is applied to a sequence of syntactic restrictions of the original program, which ultimately converges to the whole programs; the idea is to distinguish phases or modes of operation in order to make the widening

more precise. Gonnord [48], Gonnord and Halbwachs [49] have proposed *acceleration* techniques for linear constraints. These do not replace widenings altogether, but they alleviate the need for some of the costly workarounds to the imprecision introduced by widenings, such as delayed widening [11, Sec. 7.1.3]. These address a different problem from ours. On the one hand, neither improved widenings nor acceleration guarantee that the inductive invariant obtained at the end is the least one (indeed, they can yield the top element $\top$). [20] Furthermore, the invariant that these methods obtain is not parametric in the precondition, contrary to the one that our method obtains. On the other hand, improved widenings work regardless of the form of the transition relation, which our method constrains to be piecewise linear. Some of the cited methods operate on general polyhedra, while our method constrains the shape of the polyhedra that are found to a certain template.

Gawlitza and Seidl [47] proposed replacing the usual widening / narrowing iteration techniques by a *policy iteration* (or *strategy iteration*) approach. Their approach converges on a fixed point, but not necessarily the least one. Their idea is to replace computing the least fixed point of a complex abstract operator (the point-wise minimum of a family of simpler operators) by a sequence of least fixed point computations for these simple operators. Their technique anyway needs to compute these latter least fixed points, and it is possible that our method can help in that respect. Adjé et al. [1], Costan et al. [27], Gaubert et al. [46] proposed a different policy iteration approach, by downwards iterations providing successive over-approximations of the least fixed point.

Gulwani et al. [51] have also proposed a method for generating linear invariants over integer variables, using a class of templates. The methods described in the present article can be applied to linear invariants over integer variables in two ways: either by abstracting them using rationals (as in examples in Sec. 2.4.2, 4.1), either by replacing quantifier elimination over rational linear arithmetic by quantifier elimination over linear integer arithmetic, also known as Presburger arithmetic (§3.5). Gulwani et al. instead chose to first consider integer variables as rationals, so as to be able to compute over rational convex polyhedra, then bound variables and constraint parameters so as to model them as finite bit vectors, finally obtaining a problem amenable to SAT solving. Program variables *are* finite bit vectors in most industrial programming languages, and parameters to useful invariants over integer variables are often small, thus their approach seems justified. We do not see, however, how their method could be applied to programs operating over real or floating-point variables, which are the main motivation for the present article.

The idea of producing procedure summaries [88] as formulas mapping input bounds to output bounds is not new. Rugina and Rinard [83], in the context of pointer analysis (with pointers considered as a base plus an integer offset), proposed a reduction to linear programming. This reduction step, while sound, introduces an imprecision that is difficult to measure in advance; our method, in contrast, is guaranteed to be "optimal" in a certain sense. Rugina and Rinard's method, however, allows some nonlinear constructs in the program to be analyzed. Martin et al. [63] proposed applying interprocedural analysis to loops.

[20] There exist *exact acceleration* techniques but these rather apply to discrete automata.

Seidl et al. [87] also produce procedure summaries as numerical constraints. Our procedure summaries are implementations of the corresponding abstract transformer over some abstract domain, while theirs outputs a relationship between input and output concrete values. Their analysis considers a *convex* set of concrete input-output relationships, expressed as a *simplices*, a restricted class of convex polyhedra. This restriction trades precision for speed: the generator and constraint representations of simplices have approximately the same size, while in general polyhedra exponential blowup can occur. Tests by arbitrary linear constraints cannot be adequately represented within this framework. Seidl et al. [87, Sec. 4] propose deferring those constraints using auxiliary variables; this, however, loses some precision. Their analysis and ours are therefore incomparable, since they make different choices between precision and efficiency.

Lal et al. [58] proposed an interprocedural analysis of numerical properties of functions using weighted pushdown automata. The "weights" are taken in a finite height abstract domain, while the domains we consider have infinite height.

In earlier works we have proposed a method for obtaining input-output relationships of digital linear filters with memories, taking into account the effects of floating-point computations [68]. This method computes an exact relationship between bounds on the input and bounds on the output, without the need for an abstract domain for expressing the local invariant; as such, for this class of problems, it is more precise than the method from this article. This technique, however, cannot be easily generalized to cases where the operator block contains tests and other nonlinear constructs; the semantics of nonlinear constructs must be approximated by e.g. interval analysis.

There have been several published approaches to finding nonlinear relationships between program variables. One approach obtains polynomial equalities through computations on ideals using Gröbner bases [81]. This work only deals with equalities (not inequalities), uses a classical approach of computing output constraints from a set of input constraints (instead of finding relationships between the two sets of constraints), and deals with loops using a widening operator. In comparison, our approach abstracts whole program fragments, and is modular — it is possible to "plug" the result of the analysis of a procedure at the location of a procedure call.

Kapur [56] also proposed to use quantifier elimination to obtain invariants: he considers program invariants with parameters, and derives constraints over those parameters from the program. Our work improves on his by noting that least invariants of the chosen shape can be obtained, not just any invariant; that the abstraction can be done modularly and compositionally (a program fragment can be analyzed, and the result of its analysis can be plugged into the analysis of a larger program), or combined into a "conventional" abstract interpretation framework (by using invariants of a shape compatible with that framework), and that the resulting invariants can be "projected" to obtain numerical quantities.

## 7. Conclusion and future prospects

Writing static analyzers by hand has long been found tedious and error-prone. One may of course prove an existing analyzer correct through assisted proof techniques, which removes the possibility of soundness mistakes, at the expense of much increased tediousness. In this article, we proposed instead effective methods to synthesize abstract domains by automatic techniques. The advantages are

twofold: new domains can be created much more easily, since no programming is involved; a single procedure, testable on independent examples, needs be written and possibly formally proved correct. To our knowledge, this is the first effective proposal for generating numerical abstract domains automatically, and one of the few methods for generating numerical summaries. Also, it is also the only method so far for computing summaries of *floating-point* functions.

We have shown that floating-point computations could be safely abstracted using our method. The formulas produced are however fairly complex in this case, and we suspect that further over-approximation could dramatically reduce their size. There is also nowadays significant interest in automatizing, at least partially, the tedious proofs that computer arithmetic experts do and we think that the kind of methods described in this article could help in that respect.

We have so far experimented with small examples, because the original goal of this work was the automatic, on-the-fly, synthesis of abstract transfer functions for small sequences of code that could be more precise than the usual composition of abstract of individual instructions, and less tedious for the analysis designer than the method of pattern-matching the code for "known" operators with known mathematical properties. A further goal is the precise analysis of longer sequences, including integer and Boolean computations. We have shown in Sec. 3.3 how it was possible to partition the state space and abstract each region of the state-space separately; but naive partitioning according to $n$ Booleans leads to $2^n$ regions, which can be unbearably costly and is unneeded in most cases. We think that automatic refinement and partitioning techniques [55] could be developed in that respect.

The main practical application that we envision is to be able to analyze numerical operator blocks from synchronous programming languages such as Simulink,[21] Scicos,[22] Lustre,[23] Scade[24] or Sao,[25] which are widely used for programming control systems [3], particularly in the automative and avionic industries. In order to obtain good analysis precision, such blocks often have to be analyzed as a whole instead of decomposing them into individual components and applying individual transfer functions, as in our rate limiter example. The static analysis tool Astrée [11, 33, 34, 35, 39, 89] outputs few, if any, false alarms on some classes of control programs because it has specific specialized transfer functions for certain operator blocks or coding patterns. Such transfer functions had to be implemented by hand; the techniques described in the present article could have been used to implement some of them automatically and even on-the-fly.

[21]Simulink is a graphical dataflow modeling tool sold as an extension to the Matlab numerical computation package. It allows modeling a physical or electrical environment along the computerized control system. A code generator tool can then provide executable code for the control system for a variety of targets, including generic C. Simulink is available from The Mathworks.

[22]Scicos is a graphical dataflow modeling tool coming with the Scilab numerical computation package, similar in use to Simulink. [18] It is available from INRIA under the GNU General Public License and also has code generation capabilities.

[23]Lustre is a synchronous programming language, from which code can be generated for a variety of platforms [19].

[24]Scade is a graphical synchronous programming language derived from Lustre. It is available from Esterel Technologies. It was used for implementing parts of the Airbus A380 fly-by-wire systems, among others. [39, 89]

[25]Sao is an earlier industrial graphical synchronous programming language, used, for implementing parts of the Airbus A340 fly-by-wire systems [16], among others.

There are two important drawbacks to our method. One is the high cost of quantifier elimination. Despite our work on new algorithms [69], in which we are still making progress, scalability remains an issue. The other one is the necessity to provide templates with a fixed number of parameters; in comparison to polyhedral analysis [32, 53], this means that we have to decide in advance the directions of the faces of the polyhedron, whereas polyhedral analysis discovers them. Possible solutions could include running the polyhedral analysis first, obtaining the directions of the faces in the resulting invariant, then using these directions in the template; the resulting parameters could be tighter than the ones obtained by polyhedral analysis.

## References


[1] Assalé Adjé, Stéphane Gaubert, and Éric Goubault. Computing the smallest fixed point of nonexpansive mappings arising in game theory and static analysis of programs. preprint, arXiv:0806.1160v2, 2008. URL `http://arxiv.org/abs/0806.1160`.

[2] Hirokazu Anai and Volker Weispfenning. Deciding linear-trigonometric problems. In *International symposium on symbolic and algebraic computation (ISSAC)*. ACM, 2000. ISBN 1-58113-218-2. doi: 10.1145/345542.345567.

[3] Karl Johan Åström and Björn Wittenmark. *Computer-controlled systems*. Prentice-Hall, 1997. ISBN 0-13-314899-8.

[4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. *The Parma Polyhedra Library, version 0.9*. URL `http://www.cs.unipr.it/ppl`.

[5] Roberto Bagnara, Patricia M. Hill, Elena Mazzi, and Enea Zaffanella. Widening operators for weakly-relational numeric abstractions. In Chris Hankin and Igor Siveroni, editors, *Static Analysis (SAS)*, volume 3672 of *LNCS*, pages 3–18. Springer, 2005. doi: 10.1007/11547662_3.

[6] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. *Sci. Comput. Program.*, 58(1-2): 28–56, 2005. doi: 10.1016/j.scico.2005.02.003.

[7] Saugata Basu, Richard Pollack, and Marie-Francoise Roy. *Algorithms in real algebraic geometry*. Algorithms and computation in mathematics. Springer, deuxième edition, 2006. ISBN 3-540-33098-4. URL `http://perso.univ-rennes1.fr/marie-francoise.roy/bpr-posted1.html`.

[8] Bernd Becker, Christian Dax, Jochen Eisinger, and Felix Klaedtke. LIRA: handling constraints of linear arithmetics over the integers and the reals. In Werner Damm and Holger Hermanns, editors, *Computer-aided verification (CAV)*, volume 4590 of *LNCS*, pages 307–310. Springer, 2007. ISBN 978-3-540-73367-6. doi: 10.1007/978-3-540-73368-3_36.

[9] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, August 2003. doi: 10.1016/S0065-2458(03)58003-2.

[10] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In Torben Æ. Mogensen, David A. Schmidt, and

I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, number 2566 in LNCS, pages 85–108. Springer, 2002. ISBN 3-540-00326-6. doi: 10.1007/3-540-36377-7_5.

[11] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM, 2003. ISBN 1-58113-662-5. doi: 10.1145/781131.781153.

[12] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper. An effective decision procedure for linear arithmetic over the integers and reals. *ACM Transactions on Computational Logic (TOCL)*, 6(3):614–633, 2005. ISSN 1529-3785. doi: 10.1145/1071596.1071601.

[13] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *Concurrency Theory (CONCUR)*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997. ISBN 3-540-63141-0. doi: 10.1007/3-540-63141-0_10.

[14] François Bourdoncle. *Sémantique des langages impératifs d'ordre supérieur et interprétation abstraite*. PhD thesis, École polytechnique, Palaiseau, 1992.

[15] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, October 2007. ISBN 3-540-74112-7.

[16] Dominique Brière and Pascal Traverse. Airbus A320/A330/A340 electrical flight controls — a family of fault-tolerant systems. In *FTCS-23 (Symposium on Fault-Tolerant Computing)*, pages 616–623. IEEE, June 1993. ISBN 0-8186-3680-7. doi: 10.1109/FTCS.1993.627364.

[17] Christopher W. Brown and James H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *ISSAC (Symposium on Symbolic and algebraic computation)*, pages 54–60. ACM, 2007. ISBN 978-1-59593-743-8. doi: 10.1145/1277548.1277557.

[18] Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and Simulation in Scilab/Scicos*. Springer, 2006. ISBN 0-387-27802-8.

[19] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE: a declarative language for real-time programming. In *POPL (Symposium on Principles of programming languages)*, pages 178–188. ACM, 1987. ISBN 0-89791-215-2. doi: 10.1145/41625.41641.

[20] Bob F. Caviness and Jeremy R Johnson, editors. *Quantifier elimination and cylindrical algebraic decomposition*. Springer, 1998. ISBN 3-211-82794-3.

[21] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. In Roberto Giacobazzi, editor, *Static Analysis (SAS)*, number 3148 in LNCS. Springer, 2004. ISBN 3-540-22791-1. doi: 10.1007/b99688.

[22] Edmund M. Clarke, Jr, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999. ISBN 0-262-03270-8.

[23] George Collins. Quantifier elimination for real closed fields by cylindric algebraic decomposition. In *Automata theory and formal languages (2nd GI conference)*, LNCS, pages 134–183. Springer, 1975. ISBN 0-387-07407-4. reprinted as [20].

[24] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification (CAV)*, number 2725 in LNCS, pages 420–433. Springer, 2003. ISBN 3-540-40524-0. doi: 10.1007/b11831.

[25] Steven A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978. ISSN 0097-5397. doi: 10.1137/0207005.

[26] D. C. Cooper. Theorem proving in arithmetic without multiplication. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 7*, pages 91—100. Edinburgh University Press, 1972. ISBN 0-85224-234-4.

[27] Alexandru Costan, Stephane Gaubert, Éric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In Etessami and Rajamani [42], pages 462–475. ISBN 3-540-27231-3. doi: 10.1007/11513988_46.

[28] Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In Radhia Cousot, editor, *Verification, Model Checking and Abstract Interpretation (VMCAI)*, number 3385 in LNCS, pages 1–24. Springer, 2005. ISBN 3-540-24297-X. doi: 10.1007/b105073.

[29] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. State doctorate thesis, Université scientifique et médicale de Grenoble and Institut National Polytechnique de Grenoble, 1978. URL `http://tel.archives-ouvertes.fr/tel-00288657/en/`.

[30] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming (1976)*, pages 106–130, Paris, 1977. Dunod. ISBN 2-04-005185-6. Also known as *Actes du deuxième colloque international sur la programmation*.

[31] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. of Logic and Computation*, pages 511–547, August 1992. ISSN 0955-792X. doi: 10.1093/logcom/2.4.511.

[32] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–96. ACM, 1978. doi: 10.1145/512760.512770.

[33] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In Shmuel "Mooly" Sagiv, editor, *Programming Languages and Systems (ESOP)*, number 3444 in LNCS, pages 21–30. Springer, 2005. ISBN 3-540-25435-8. doi: 10.1007/b107380.

[34] Patrick Cousot, Radhia Cousot, Jerome Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of static analyzers: A comparison with ASTRÉE. In *Theoretical Aspects of Software Engineering (TASE)*. IEEE, 2007. ISBN 0-7695-2856-2.

[35] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science — ASIAN 2006*, volume 4435 of *LNCS*, pages 272–300. Springer, 2008. ISBN 978-3-540-77504-1. doi: 10.1007/978-3-540-77505-8_23. URL

http://www.di.ens.fr/~cousot/COUSOTpapers/ASIAN06.shtml.

[36] James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1–2):29–35, April 1988. doi: 10.1016/S0747-7171(88)80004-X.

[37] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. ISBN 3-540-78799-2. doi: 10.1007/978-3-540-78800-3_24.

[38] Rocco de Nicola, editor. *Programming Languages and Systems (ESOP)*, volume 4421 of *LNCS*, 2007. Springer. ISBN 978-3-540-71316-6.

[39] David Delmas and Jean Souyris. Astrée: From research to industry. In Nielson and Filé [74], pages 437–451. ISBN 978-3-540-74060-5. doi: 10.1007/978-3-540-74061-2_27.

[40] Andreas Dolzmann, Andreas Seidl, and Thomas Sturm. *Redlog User Manual*, 3.1 edition, 2006. for Redlog version 3.06 and Reduce version 3.8.

[41] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Computer-aided verification (CAV)*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006. ISBN 3-540-37406-X. doi: 10.1007/11817963_11.

[42] Kousha Etessami and Sriram K. Rajamani, editors. *Computer Aided Verification (CAV)*, number 4590 in LNCS, 2005. Springer. ISBN 3-540-27231-3. doi: 10.1007/b138445.

[43] Jeanne Ferrante and Charles Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. on Computing*, 4(1):69–76, March 1975. ISSN 0097-5397. doi: 10.1137/0204006.

[44] Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of Presburger arithmetic. In Richard Karp, editor, *Complexity of Computation*, number 7 in SIAM–AMS proceedings, pages 27–42. American Mathematical Society, 1974. ISBN 0-8218-1327-7.

[45] Michael R. Garey and David S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W.H. Freeman, 1985. ISBN 0-7167-1045-5.

[46] Stéphane Gaubert, Éric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In de Nicola [38], pages 237–252. ISBN 978-3-540-71316-6.

[47] Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In de Nicola [38], pages 300–315. ISBN 978-3-540-71316-6. doi: 10.1007/978-3-540-71316-6_21.

[48] Laure Gonnord. *Accelération abstraite pour l'amélioration de la précision en analyse des relations linéaires*. PhD thesis, Université Joseph Fourier, October 2007. URL http://tel.archives-ouvertes.fr/tel-00196899/en/.

[49] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In Kwangkeun Yi, editor, *Static analysis (SAS)*, volume 4134 of *LNCS*, pages 144–160. Springer, 2006. ISBN 3-540-37756-5. doi: 10.1007/11823230_10.

[50] Denis Gopan and Thomas W. Reps. Lookahead widening. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 452–466. Springer, 2006. ISBN 3-540-37406-X. doi: 10.1007/11817963_41.

[51] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Programming Language Design and Implementation (PLDI)*. ACM, 2008. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375616.

[52] Nicolas Halbwachs. Delay analysis in synchronous programs. In Costas Courcoubetis, editor, *Computer Aided Verification (CAV)*, volume 697 of *LNCS*, pages 333–346. Springer, 1993. ISBN 3-540-56922-7. doi: 10.1007/3-540-56922-7_28.

[53] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. State doctorate thesis, Université scientifique et médicale de Grenoble and Institut National Polytechnique de Grenoble, 1979. URL `http://tel.archives-ouvertes.fr/tel-00288805/en/`.

[54] *IEEE standard for Binary floating-point arithmetic for microprocessor systems*. IEEE, 1985. ANSI/IEEE Std 754-1985.

[55] Bertrand Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23 (1):5–37, July 2003. ISSN 0925-9856. doi: 10.1023/A:1024480913162.

[56] Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In *Applications of Computer Algebra (ACA)*, 2004.

[57] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2008. ISBN 978-3-540-74104-6.

[58] Akash Lal, Gogul Balakrishnan, and Thomas Reps. Extended weighted pushdown systems. In Etessami and Rajamani [42], pages 343–357. ISBN 3-540-27231-3. doi: 10.1007/11817963_32.

[59] Serge Lang. *Algebra*. Addison-Wesley, troisième edition, 1993. ISBN 0201555409.

[60] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, 1993. Special issue on computational quantifier elimination.

[61] Zohar Manna and John McCarthy. Properties of programs and partial function logic. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, 5, pages 27–38. Edinburgh University Press, 1969. ISBN 0-85224-176-3.

[62] Zohar Manna and Amir Pnueli. Formalization of properties of functional programs. *J. ACM*, 17(3):555–569, 1970. doi: 10.1145/321592.321606.

[63] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In Kai Koskimies, editor, *Compiler Construction (CC)*, volume 1383 of *LNCS*, pages 80–94. Springer, 1998. ISBN 3-540-64304-4. doi: 10.1007/BFb0026424.

[64] Antoine Miné. The octagon abstract domain. In Axel Simon, Andy King, and Jacob M. Howe, editors, *WCRE (Analysis, Slicing, and Transformation)*, pages 310–319. IEEE, 2001. doi: 10.1109/WCRE.2001.957836.

[65] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In David Schmidt, editor, *Programming Languages and Systems (ESOP)*, number 2986 in LNCS, pages 3–17. Springer, 2004. ISBN 3-540-21313-9. doi: 10.1007/b96702.

[66] Antoine Miné. *Domaines numériques abstraits faiblement relationnels*. PhD thesis, École polytechnique, 2004.

[67] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. doi: 10.1007/s10990-006-8609-1.

[68] David Monniaux. Compositional analysis of floating-point linear numerical filters. In Etessami and Rajamani [42], pages 199–212. ISBN 3-540-27231-3. doi: 10.1007/b138445.

[69] David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 5330 of *LNAI*, pages 243–257. Springer, 2008. ISBN 978-3-540-89438-4. doi: 10.1007/978-3-540-89439-1_18.

[70] David Monniaux. Automatic modular abstractions for linear constraints. In Benjamin C. Pierce, editor, *Symposium on Principles of programming languages (POPL)*. ACM, 2009. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480899.

[71] David Monniaux. Optimal abstraction on real-valued programs. In Nielson and Filé [74], pages 104–120. ISBN 978-3-540-74060-5. doi: 10.1007/978-3-540-74061-2_7.

[72] David Monniaux. The pitfalls of verifying floating-point computations. *Transactions on programming languages and systems (TOPLAS)*, 30(3): 12, May 2008. ISSN 0164-0925. doi: 10.1145/1353445.1353446. URL `http://hal.archives-ouvertes.fr/hal-00128124/en/`.

[73] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, *Compiler Construction (CC)*, volume 2304 of *LNCS*, pages 209–265. Springer, 2002. ISBN 3-540-43369-4. doi: 10.1007/3-540-45937-5_16.

[74] Hanne Riis Nielson and Gilberto Filé, editors. *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *LNCS*, 2007. Springer. ISBN 978-3-540-74060-5.

[75] Tobias Nipkow. Linear quantifier elimination. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated reasoning (IJCAR)*, volume 5195 of *LNCS*, pages 18–33. Springer, 2008. ISBN 978-3-540-71069-1. doi: 10.1007/978-3-540-71070-7_3.

[76] Dominique Perrin. Finite automata. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, chapter 1. MIT Press, 1990. ISBN 0444880747.

[77] Mojżesz Presburger. Über die Vollstandigkeit eines Gewissen Systems der Arithmetik Ganzer Zahlen, in Welchem die Addition als Einzige Operation Hervortritt. In *Comptes-rendus du premier congrès des mathématiciens des pays slaves*, pages 92–101, Warsaw, 1929.

[78] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pages 4–13. ACM, 1991. ISBN 0-89791-459-7. doi: 10.1145/125826.125848.

[79] Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Static program analysis via 3-valued logic. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 2004. ISBN 3-540-22342-8. doi: 10.1007/b98490.

[80] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, March 1953.

[81] Enric Rodríguez-Carbonell and Deepak Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *SAS (Static*

*Analysis)*, number 3148 in LNCS. Springer, 2004.

[82] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987. ISBN 0-262-68052-1.

[83] Radu Rugina and Martin Rinard. Symbolic bounds analysis for pointers, array indices, and accessed memory regions. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 27(2):185–235, 2005. doi: 10.1145/349299.349325.

[84] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Constraint-based linear-relations analysis. In *SAS*, number 3148 in LNCS, pages 53–68. Springer, 2004.

[85] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, volume 3385 of *LNCS*, pages 21–47. Springer Verlag, 2005.

[86] Abraham Seidenberg. A new decision method for elementary algebra. *Annals of Mathematics*, 60(2):365–374, September 1954. URL `http://www.jstor.org/stable/1969640`.

[87] Helmut Seidl, Andrea Flexeder, and Michael Petter. Interprocedurally analysing linear inequality relations. In de Nicola [38], pages 284–299. ISBN 978-3-540-71316-6. doi: 10.1007/978-3-540-71316-6_20.

[88] Micha Sharir and Amir Pnueli. Two approaches to inter-procedural data-flow analysis. In Neil Jones and Steven Muchnik, editors, *Program Flow Analysis: Theory and Application*. Prentice-Hall, 1981.

[89] Jean Souyris and David Delmas. Experimental assessment of Astrée on safety-critical avionics software. In Francesca Saglietti and Norbert Oster, editors, *SAFECOMP*, volume 4680 of *LNCS*, pages 479–490. Springer, 2007. ISBN 978-3-540-75100-7. doi: 10.1007/978-3-540-75101-4_45.

[90] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951. reprinted as [91].

[91] Alfred Tarski. A decision method for elementary algebra and geometry. Technical Report 109, The RAND Corporation, 1957. Second edition (original 1948, revised 1951), reprint of [90].

[92] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, series 2*, 42: 230–265, 1936.

[93] Klaus Weihrauch. *Computable analysis: an introduction*. Texts in Theoretical Computer Science. Springer, 2000.

[94] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1–2):3–27, February 1988. ISSN 0747-7171. doi: 10.1016/S0747-7171(88)80003-8.

[95] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. MIT Press, 1993. ISBN 0-262-23169-7.

[96] Stephen Wolfram. *The* Mathematica *Book*. Wolfram Research, 2005. Manual coming with version 5.2 of *Mathematica*.