A Sound Floating-Point Polyhedra Abstract Domain ⋆

Liqian Chen[1,2], Antoine Miné[3], and Patrick Cousot[1]

[1] École Normale Supérieure, Paris, France
{chen,mine,cousot}@di.ens.fr
[2] National Laboratory for Parallel and Distributed Processing, Changsha, P.R.China
[3] CNRS, École Normale Supérieure, Paris, France

Abstract. The polyhedra abstract domain is one of the most powerful and commonly used numerical abstract domains in the field of static program analysis based on abstract interpretation. In this paper, we present an implementation of the polyhedra domain using floating-point arithmetic without sacrificing soundness. Floating-point arithmetic allows a compact memory representation and an efficient implementation on current hardware, at the cost of some loss of precision due to rounding. Our domain is based on a constraint-only representation and employs sound floating-point variants of Fourier-Motzkin elimination and linear programming. The preliminary experimental results of our prototype are encouraging. To our knowledge, this is the first time that the polyhedra domain is adapted to floating-point arithmetic in a sound way.

1 Introduction

Static analysis is a technique to automatically discover program properties at compile-time. One important application is to prove the absence of run-time errors in a program before actually running it. Since, in general, the exact behavior of a program cannot be computed statically, an analysis needs to use approximation. We only consider analyses that are *sound*, that is, compute an over-approximation of all possible behaviors including all real errors, but may fail to prove the correctness if the approximation is too coarse.

The abstract interpretation framework [6] allows devising static analyses that are sound by construction. A core concept in abstract interpretation is that of *an abstract domain*, that is, a set of computer-representable properties together with operators to model soundly the semantic actions of a program (assignments, tests, control-flow joins, loops, etc.). Specifically, we are interested in *numerical* abstract domains that represent properties of the numerical variables of a program.

Among them, one of the most famous is the *polyhedra* abstract domain introduced in 1978 by Cousot and Halbwachs [7] which can infer linear relationships between variables in a program. It has a wide range of applications in the field of the analysis and

⋆ This work is supported by the INRIA project-team Abstraction common to the CNRS and the École Normale Supérieure. This work is partially supported by the Fund of the China Scholarship Council and National Natural Science Foundation of China under Grant No.60725206.

verification of hardware and software systems. A number of implementations for manipulating polyhedra are currently available. Recent ones include the Parma Polyhedra Library (PPL) [3] and the APRON library [1].

However, most implementations suffer from scalability problems [20]. One reason is the use of arbitrary precision rationals which are slow and may lead to excessively large numbers even when analyzing simple programs involving only small integer values. Alternatively, one can use fast machine integers but then overflows can cause much precision loss. Floating-point numbers are not only fast but they also allow a *gradual* loss of precision. Unfortunately, the pervasive rounding errors make it difficult to guarantee soundness. This is the problem we tackle in this paper.

This paper presents a sound floating-point implementation of the polyhedra abstract domain. Our approach is based on three key points: a constraint-based representation using floating-point coefficients, a sound version of Fourier-Motzkin elimination using floating-point arithmetic, and a rigorous linear programming method proposed in [19]. The preliminary experimental results are promising when analyzing programs involving coefficients of large magnitude, e.g., floating-point programs.

The rest of the paper is organized as follows. Section 2 discusses some related work. In Section 3, we review the design of the polyhedra domain based on a constraint-only representation over the rationals. In Section 4, we adapt this framework to the floating-point world. Section 5 discusses precision and efficiency issues due to rounding and proposes some solutions. Section 6 presents our prototype implementation together with preliminary experimental results. Finally, conclusions as well as suggestions for future work are given in Section 7.

2 Related Work

The Polyhedra Abstract Domain. Common implementations of the polyhedra domain [1, 3] are based on a dual representation [7]. A polyhedron can be described as the conjunction of a finite set of linear constraints. Dually, in the frame representation, it can be represented as a finite collection of generators, that is, vertices or rays. Some domain operations (e.g., meet and test) can be performed more efficiently on the constraint representation, while some others (e.g., projection and join) can be performed more efficiently on the frame representation. Thus, it is often necessary to convert from one representation to the other. The dual conversions are performed using the Chernikova algorithm [13] which can produce an output that is exponential in the size of the input.

Recently, as an alternative to the dual representation, Simon and King [23] have demonstrated that the polyhedra domain can be fully implemented using only constraints, with the aim to remove the complexity bottleneck caused by the frame representation. Our work is based on the same principle.

In order to reduce the complexity, it has also been proposed to abandon general polyhedra in favor of less expressive weakly relational domains which are polyhedra of restricted forms that benefit from specialized algorithms with improved complexity. Examples include the Octagon domain [17], the Two Variables Per Inequality (TVPI) domain [24], and the Template Constraint Matrix (TCM) domain [22].

Linear Programming. Linear Programming (LP) [2] is a method used to find the optimal value of some affine function (so-called objective function) subject to a finite system of linear constraints (defining the so-called feasible space) . It is a well-studied problem for which highly efficient algorithms have already been developed that scale up to hundreds of thousands of variables and constraints. Most state-of-the-art LP solvers use floating-point arithmetic and only give approximate solutions which may not be the actual optimum solution or may even lie outside the feasible space. It would be possible but costly in practice to compute the exact solution using exact arithmetic. Instead, we take advantage of recent progress that has been made on computing *rigorous* bounds for the objective value using floating-point arithmetic [19]. In particular, the ability to infer rigorous bounds provides a basis for soundly implementing some operations in our domain.

Note that using LP in a polyhedral analysis is not new. Sankaranarayanan et al. [22] use it in their TCM domain which is less expressive than general polyhedra. They use approximate floating-point LP but the result is then checked using exact arithmetic. Simon et al. [23] consider general polyhedra but use exact arithmetic. We will consider here general polyhedra and use only floating-point arithmetic.

Static Analysis of Floating-Point Programs. A related problem is that of analyzing programs featuring floating-point computations. In [8], Goubault analyzes the origin of the loss of precision in floating-point programs. ASTRÉE [5] computes the set of reachable values for floating-point variables in order to check for run-time errors. In this paper, we will also apply our abstract domain to the reachability problem in floating-point programs.

Note that the polyhedra abstract domain described in this paper abstracts sets of real numbers. As in ASTRÉE, we rely on the linearization technique of [15] to soundly abstract floating-point computations in the analyzed program into ones over the field of reals. The use of floating-point arithmetic in the abstraction becomes very important for efficiency when analyzing linearized floating-point programs as they involve coefficients of large magnitude.

Although sound implementations of floating-point interval arithmetic have been known for a long time [18], such adaptations to *relational* domains are recent and few [15]. To our knowledge, we are the first to tackle the case of general polyhedra.

3 Rational Polyhedra Domain Based on Constraints

In this section, we describe the design of a polyhedra domain based on a constraint-only representation using rational numbers, most of which has been previously known [23] except for the implementation of the standard widening (Sect. 3.7). Internally, a *rational* polyhedron P is described as an inequality system $Ax \leq b$, where A is a matrix and b is a vector of rational numbers. It represents the set $\gamma(P) = \{x \in \mathbb{Q}^n \mid Ax \leq b\}$ where each point x is a possible environment, i.e., an assignment of rational values to abstract variables. In practice, program variables have to be mapped to these abstract variables by a memory model (see, e.g., [16]). We will now briefly describe the implementation of the common domain operations.

3.1 Redundancy Removal

The constraint representation of a polyhedron is not unique. For efficiency reasons, it is desirable to have as few constraints as possible. An inequality $\varphi \in P$ is said to be redundant when φ can be entailed by the other constraints in P, that is, $P \setminus \{\varphi\} \models \varphi$. Given $\varphi = (\sum_i a_i x_i \leq b)$ in P, we can check whether φ is redundant by solving the LP problem: $\mu = \max \sum_i a_i x_i$ subject to $P \setminus \{\varphi\}$. If $\mu \leq b$, then φ is redundant and can be eliminated from P. This process is repeated until no more inequality can be removed.

3.2 Emptiness Test

A polyhedron is *empty* if and only if its constraint set is infeasible. The feasibility of a constraint system is implicitly checked by LP solvers when computing the maximum (minimum) of an objective function. During program analysis, constraints are often added one by one. Thus, the test for emptiness can also be done incrementally. When adding a new constraint $\sum_i a_i x_i \leq b$ to a nonempty polyhedron P, we solve the LP problem μ= min $\sum_i a_i x_i$ subject to P. If $b < \mu$, the new polyhedron is indeed empty.

3.3 Projection

An important operation on polyhedra is to remove all information pertaining to a variable x_i without affecting the relational information between other variables. To this end, we define the projection operator $\pi(P, x_i) \overset{\text{def}}{=} \{\, x[x_i/y] \mid x \in \gamma(P), y \in \mathbb{Q} \,\}$, where $x[x_i/y]$ denotes the vector x in which the i-th element is replaced with y. It can be computed by eliminating all occurrences of x_i in the constraints defining P, using the classic Fourier-Motzkin algorithm:

$$Fourier(P, x_i) \overset{\text{def}}{=} \left\{ (-a_i^-)c^+ + a_i^+ c^- \,\middle|\, \begin{array}{l} c^+ = (\sum_k a_k^+ x_k \leq b^+) \in P, a_i^+ > 0 \\ c^- = (\sum_k a_k^- x_k \leq b^-) \in P, a_i^- < 0 \end{array} \right\}$$
$$\cup \{\, (\sum_k a_k x_k \leq b) \ \in P \ \mid \ a_i = 0 \,\}.$$

The projection is useful to model the non-deterministic assignment of an unknown value to a variable x_i, namely by defining: $[\![x_i := \text{random}()]\!]^\# (P) \overset{\text{def}}{=} Fourier(P, x_i)$, where $[\![\cdot]\!]^\# (P)$ denotes the effect of a program statement on the polyhedron P.

3.4 Join

To abstract the control-flow join, we need to compute the union of environments of program variables. The smallest polyhedron enclosing this union is the topological closure of the convex hull. To compute it, we use the method proposed in [23]. Given $\gamma(P) = \{x \in \mathbb{Q}^n \mid Ax \leq b\}$ and $\gamma(P') = \{x \in \mathbb{Q}^n \mid A'x \leq b'\}$, the convex hull of P and P' is

$$\gamma(P_H) = \left\{ x \in \mathbb{Q}^n \,\middle|\, \begin{array}{c} x = \sigma_1 z + \sigma_2 z' \wedge \sigma_1 + \sigma_2 = 1 \wedge \sigma_1 \geq 0 \wedge \\ Az \leq b \quad \wedge \quad A'z' \leq b' \ \wedge \sigma_2 \geq 0 \end{array} \right\}$$

where $\sigma_1, \sigma_2 \in \mathbb{Q}$ and $x, z, z' \in \mathbb{Q}^n$. To avoid the non-linear equation $x = \sigma_1 z + \sigma_2 z'$, we introduce $y = \sigma_1 z$ as well as $y' = \sigma_2 z'$ and relax the system into

$$\gamma(P_{CH}) = \left\{ x \in \mathbb{Q}^n \;\middle|\; \begin{array}{l} x = y + y' \;\wedge \sigma_1 + \sigma_2 = 1 \;\wedge \sigma_1 \geq 0 \;\wedge \\ Ay \leq \sigma_1 b \;\wedge\; A'y' \leq \sigma_2 b' \;\wedge \sigma_2 \geq 0 \end{array} \right\} . \tag{1}$$

Projecting out $\sigma_1, \sigma_2, y, y'$ from (1) yields the closure of the convex hull of P and P'.

3.5 Transfer Functions

Test Transfer Function. An affine test with exact rational arithmetic can be converted to the form $\sum_i a_i x_i \leq c$. The result of such a test $[\![\sum_i a_i x_i \leq c]\!]^{\#}(P)$ is simply the polyhedron P with the constraint $\sum_i a_i x_i \leq c$ added. Note that a test may introduce redundancy or make the polyhedron infeasible. More complicated cases, such as tests involving floating-point or non-linear operations, can be soundly abstracted to the form $\sum_i a_i x_i \leq c$ following the method in [15]. In the worst case, we can always ignore the effect of a test, which is sound.

Assignment Transfer Function. The assignment of some expression e to x_j can be modeled using projection, test, and variable renaming as follows:

$$[\![x_j := e]\!]^{\#}(P) \stackrel{\text{def}}{=} (Fourier([\![x'_j - e = 0]\!]^{\#}(P), x_j))[x'_j/x_j] \, .$$

First, a fresh variable x'_j is introduced to hold the value of the expression. Then, we project out x_j by Fourier-Motzkin elimination and the final result system is obtained by renaming x'_j back to x_j. The temporary variable x'_j is necessary for invertible assignments such as $x := x + 1$.

3.6 Inclusion Test

Inclusion test between two polyhedra P_1 and P_2, denoted $P_1 \sqsubseteq P_2$, reduces to the problem of checking whether each inequality in P_2 is entailed by P_1, which can be implemented using LP. For each $\sum_i a_i x_i \leq b$ in P_2, compute $\mu = \max \sum_i a_i x_i$ subject to P_1 . If $\mu > b$, the inclusion does not hold.

3.7 Widening

For loops, widening ensures an efficient analysis by accelerating the fixpoint computation [6]. The first widening operator on polyhedra was proposed in [7] using the dual representation. Its improvement, presented in [9], is now the standard widening:

Definition 1 (Standard widening). *Given two polyhedra $P_1 \sqsubseteq P_2$, represented by sets of linear inequalities, we define*

$$P_1 \nabla P_2 \stackrel{\text{def}}{=} \mathcal{S}_1 \cup \mathcal{S}_2$$

where

$$\begin{aligned} \mathcal{S}_1 &= \{\, \varphi_1 \in P_1 \mid P_2 \models \varphi_1 \,\}, \\ \mathcal{S}_2 &= \{\, \varphi_2 \in P_2 \mid \exists \varphi_1 \in P_1, \gamma(P_1) = \gamma((P_1 \setminus \{\varphi_1\}) \cup \{\varphi_2\}) \,\}. \end{aligned}$$

The key point of the standard widening is to keep not only the inequalities $\mathcal{S}_1$ from P_1 satisfied by P_2, but also the inequalities $\mathcal{S}_2$ from P_2 that are mutually redundant with an inequality of P_1 with respect to P_1. $\mathcal{S}_2$ ensures that the result does not depend on the representation of P_1 and P_2. Note that $\mathcal{S}_1$ can be computed using entailment checks. The following property shows that $\mathcal{S}_2$ also reduces to entailment checks, which shows that the standard widening can be efficiently implemented using LP only.

Property 1. $\forall \varphi_1 \in P_1, \varphi_2 \in P_2,\ \gamma(P_1) = \gamma((P_1 \setminus \{\varphi_1\}) \cup \{\varphi_2\})$ *iff* $P_1 \models \varphi_2$ and $((P_1 \setminus \{\varphi_1\}) \cup \{\varphi_2\}) \models \varphi_1$.

4 Floating-Point Polyhedra Domain

In this section, we present a floating-point implementation of the polyhedra domain. A *floating-point* polyhedron is represented as an inequality system $Ax \leq b$ where coefficients in A and b are now floating-point numbers. Such a system still represents a set of environments with rational-valued variables, namely $\{ x \in \mathbb{Q}^n \mid Ax \leq b\}$ where $Ax \leq b$ is interpreted mathematically (rather than in floating-point semantics).

In order to distinguish floating-point arithmetic operations from exact arithmetic ones, we introduce additional notations. As usual, $\{+, -, \times, /\}$ are used as exact rational arithmetic operations. The corresponding floating-point operations are denoted by $\{\oplus_r, \ominus_r, \otimes_r, \oslash_r\}$, tagged with a rounding mode $r \in \{-\infty, +\infty\}$ ($-\infty$: downward; $+\infty$: upward). The floating-point unary minus $\ominus$ is exact and does not incur rounding. For the sake of convenience, we occasionally use the command *roundup* (respectively *rounddown*) to change the current rounding mode to upward (respectively downward). All the algorithms in this section are implemented in floating-point arithmetic.

4.1 Linearization

We say that a point x satisfies some linear interval inequality $\varphi : \sum_k [a_k, b_k] \times x_k \leq c$, denoted by $x \in \gamma(\varphi)$, when for all k there is some $d_k \in [a_k, b_k]$ such that $\sum_k d_k \times x_k \leq c$ holds. This definition lifts to systems of inequalities straightforwardly and corresponds to the classic notion of *weak solution* in the field of linear interval optimization [21].

A linear inequality in the common sense is simply a linear interval inequality where all the coefficients are singletons (scalars). Our internal representation of a polyhedron supports only linear (non-interval) inequalities, while as we will see in the following sections, some operations of our domain naturally output linear interval inequalities. To convert a linear interval inequality φ to a linear inequality, we adapt the *linearization* technique from [15]. Our linearization operator $\zeta(\varphi, \mathbf{x})$ is defined with respect to a bounding box $\mathbf{x}$ of variables as follows:

Definition 2 (Linearization operator). *Let $\varphi : \sum_k [a_k, b_k] \times x_k \leq c$ be a linear interval inequality and $\mathbf{x} := [\underline{x}, \overline{x}]$ be the bounding box of x.*

$$\zeta(\varphi, \mathbf{x}) \overset{\text{def}}{=} \sum_k d_k \times x_k \leq c \oplus_{+\infty} \bigoplus_k {}_{+\infty} (\max\{b_k \ominus_{+\infty} d_k,\ d_k \ominus_{+\infty} a_k\} \otimes_{+\infty} |x_k|)$$

where d_k can be any floating-point number inside $[a_k, b_k]$ and $|x_k| = \max\{-\underline{x_k}, \overline{x_k}\}$.

In theory, d_k can be any floating-point number in $[a_k, b_k]$. In practice, we often choose the midpoint $d_k = (a_k \oplus_r b_k) \oslash_r 2$ which causes the least loss of precision. More strategies to choose a proper d_k will be discussed in Sect. 5.3.

Example 1. Consider the linear interval inequality $[0,2]x + [1,1]y \le 2$ with respect to the bounding box $x, y \in [-10, 5]$. If we choose the midpoint of $[a_k, b_k]$ as d_k, the linearization result will be $x + y \le 12$ (since $2 \oplus_{+\infty} \max\{2 \ominus_{+\infty} 1, 1 \ominus_{+\infty} 0\} \otimes_{+\infty} 10 \oplus_{+\infty} \max\{1 \ominus_{+\infty} 1, 1 \ominus_{+\infty} 1\} \otimes_{+\infty} 10 = 12$). Note that some loss of precision happens here, e.g., the point (0,12) satisfies the result inequality $x + y \le 12$ but does not satisfy the original interval inequality $[0,2]x + [1,1]y \le 2$.

Theorem 1 (Soundness of the linearization operator). *Given a linear interval inequality φ and a bounding box $\mathbf{x}$, $\zeta(\varphi, \mathbf{x})$ soundly over-approximates φ, that is, any point in $\mathbf{x}$ that also satisfies φ satisfies $\zeta(\varphi, \mathbf{x})$: $\forall x \in \mathbf{x}, x \in \gamma(\varphi) \Rightarrow x \in \gamma(\zeta(\varphi, \mathbf{x}))$.*

Proof. For any linear interval inequality $\varphi : \sum_k [a_k, b_k] \times x_k \le c$,

$$\begin{aligned} & \textstyle\sum_k [a_k, b_k] \times x_k \le c \\ \Longleftrightarrow\ & \textstyle\sum_k (d_k + [a_k - d_k, b_k - d_k]) \times x_k \le c \\ \Longleftrightarrow\ & \textstyle\sum_k d_k \times x_k \le c + \sum_k [d_k - b_k, d_k - a_k] \times x_k \\ \Longrightarrow\ & \textstyle\sum_k d_k \times x_k \le (c \oplus_{+\infty} \bigoplus_k {}_{+\infty}(\max\{b_k \ominus_{+\infty} d_k,\ d_k \ominus_{+\infty} a_k\} \otimes_{+\infty} |x_k|) \end{aligned}$$

□

Note that although the value of the right hand of $\zeta(\varphi, \mathbf{x})$ depends on the evaluation ordering of the summation $\bigoplus_{+\infty}$, the linearization operator is still sound because in fact every ordering gives an upper bound of $c + \sum_k [d_k - b_k, d_k - a_k] \times x_k$ in the real field.

4.2 Floating-Point Fourier-Motzkin Elimination

The key idea in building a sound floating-point Fourier-Motzkin elimination algorithm is to use interval arithmetic with outward rounding (i.e., rounding upper bounds up and lower bounds down). Then, using the linearization operator introduced in Sect. 4.1, the interval coefficients in the result can be linearized to scalars.

Assume we want to eliminate variable x_i from the following two inequalities

$$\begin{cases} a_i^+ x_i + \sum_{k \neq i} a_k^+ \times x_k \le c^+, & \text{where } a_i^+ > 0 \\ a_i^- x_i + \sum_{k \neq i} a_k^- \times x_k \le c^-, & \text{where } a_i^- < 0. \end{cases} \tag{2}$$

After dividing (2) by the absolute value of the coefficient of x_i using interval arithmetic with outward rounding, we get

$$\begin{cases} x_i + \sum_{k \neq i} [a_k^+ \oslash_{-\infty} a_i^+, a_k^+ \oslash_{+\infty} a_i^+] \times x_k \le c^+ \oslash_{+\infty} a_i^+, & \text{where } a_i^+ > 0 \\ -x_i + \sum_{k \neq i} [a_k^- \oslash_{-\infty} (\ominus a_i^-), a_k^- \oslash_{+\infty} (\ominus a_i^-)] \times x_k \le c^- \oslash_{+\infty} (\ominus a_i^-), & \text{where } a_i^- < 0 \end{cases}$$

and by addition

$$\begin{aligned} & \textstyle\sum_{k \neq i} [(a_k^+ \oslash_{-\infty} a_i^+) \oplus_{-\infty} (a_k^- \oslash_{-\infty} (\ominus a_i^-)), (a_k^+ \oslash_{+\infty} a_i^+) \oplus_{+\infty} (a_k^- \oslash_{+\infty} (\ominus a_i^-))] \times x_k \\ & \qquad \le (c^+ \oslash_{+\infty} a_i^+) \oplus_{+\infty} (c^- \oslash_{+\infty} (\ominus a_i^-)). \end{aligned} \tag{3}$$

Then (3) can be abstracted into a linear (non-interval) form by the linearization operator ζ. We denote as $Fourier_f(P, x_i)$ the result system with x_i projected out from P this way.

Theorem 2 (Soundness of the floating-point Fourier-Motzkin elimination). *Given a polyhedron P, a variable x_i and a bounding box $\mathbf{x}$, any point in $\mathbf{x}$ that also satisfies $Fourier(P, x_i)$ satisfies $Fourier_f(P, x_i)$: $\forall x \in \mathbf{x}, x \in \gamma(Fourier(P, x_i)) \Rightarrow x \in \gamma(Fourier_f(P, x_i))$.*

The key point is that the coefficient of the variable to be eliminated can always be reduced exactly to 1 or -1 by division. In some cases, an alternative algorithm can be used. Suppose that $a_i^+ \otimes_{-\infty} (\ominus a_i^-) = a_i^+ \otimes_{+\infty} (\ominus a_i^-)$, i.e., the floating-point multiplication of a_i^+ and $\ominus a_i^-$ is exact. Then, the Fourier-Motzkin elimination can be done in a multiplicative way:

$$\begin{array}{c}\sum_{k\neq i} [(a_k^+ \otimes_{-\infty} (\ominus a_i^-)) \oplus_{-\infty} (a_k^- \otimes_{-\infty} a_i^+),\ (a_k^+ \otimes_{+\infty} (\ominus a_i^-)) \oplus_{+\infty} (a_k^- \otimes_{+\infty} a_i^+)] \times x_k \\ \le (c^+ \otimes_{+\infty} (\ominus a_i^-)) \oplus_{+\infty} (c^- \otimes_{+\infty} a_i^+)\ \ .\end{array} \qquad (4)$$

Note that the condition $a_i^+ \otimes_{-\infty} (\ominus a_i^-) = a_i^+ \otimes_{+\infty} (\ominus a_i^-)$ guarantees that the coefficient of x_i is exactly 0 in (4). When all the coefficients in (2) are small integers, (4) often gives an exact result, which is rarely the case of (3). In practice, the Fourier-Motzkin elimination by multiplication is very useful for producing constraints with regular coefficients, especially for programs with only integer variables.

Example 2. Consider two inequalities $3x + y \le 10$ and $-7x + y \le 10$ with respect to the bounding box $x, y \in (-\infty, 10]$. After eliminating the variable x, (4) will result in $y \le 10$ while (3) will result in $y \le +\infty$.

4.3 Rigorous Linear Programming

The rigorous bounds for the objective function in floating-point linear programming can be derived by a cheap post-processing on the approximate result given by a standard floating-point LP solver [19].

Assume the linear program is given in the form

$$\begin{array}{ll} min & c^T x \\ s.t. & Ax \le b \end{array}$$

the dual of which is

$$\begin{array}{ll} max & b^T y \\ s.t. & A^T y = c,\ y \le 0. \end{array}$$

Suppose that y is an approximate solution of the dual program, then we calculate a rigorous interval $\mathbf{r}$ using interval arithmetic with outward rounding as follows:

$$r := A^T y - c \in \mathbf{r} = [\underline{r}, \overline{r}].$$

Recall that $y \le 0$ and $Ax \le b$, hence $y^T Ax \ge y^T b$. By introducing the interval vector $\mathbf{x} := [\underline{x}, \overline{x}]$, we get

$$c^T x = (A^T y - r)^T x = y^T Ax - r^T x \ge y^T b - r^T x \in y^T b - \mathbf{r}^T \mathbf{x}$$

and

$$\mu := \inf(y^T b - \mathbf{r}^T \mathbf{x}) \qquad (5)$$

is the desired rigorous lower bound for $c^T x$. The value of (5) can be calculated as follows using floating-point arithmetic:

```
rounddown;
$\underline{r} = A^T y - c$;
$t = y^T b$;
roundup;
$\overline{r} = A^T y - c$;
$\mu = \max\{ \underline{r}^T \underline{x},\ \underline{r}^T \overline{x},\ \overline{r}^T \underline{x},\ \overline{r}^T \overline{x} \} - t$;
$\mu = -\mu$;
```

Note that the precision of such a rigorous bound depends on the range of the bounding box **x**. Moreover, finding a rigorous upper bound for the maximum objective function can be reduced to the minimum case as $\max c^T x = -\min (-c)^T x$.

4.4 Soundness of the Floating-Point Polyhedra Domain

Observe that the rational domain of Sect. 3 relies on two primitives: Fourier-Motzkin elimination and linear programming. Substituting the two primitives with the floating-point Fourier-Motzkin elimination algorithm (Sect. 4.2) and rigorous linear programming (Sect. 4.3) yields the floating-point polyhedra domain. Note that both primitives may produce floating-point overflows or the value NaN (Not a Number). In these cases, a sound Fourier-Motzkin elimination can be obtained by discarding the constraint. With respect to the rigorous linear programming, we return $+\infty$ (respectively $-\infty$) as the maximum (respectively minimum) objective value.

The soundness of the whole floating-point polyhedra domain is guaranteed by the soundness of each domain operation, which means that each operation should result in a conservative answer with respect to the exact one. Due to the floating-point Fourier-Motzkin elimination algorithm of Sect. 4.2, the projection operator will always result in a sound over-approximated polyhedron compared to the exact one, which implies the soundness of both the join (convex hull) operator and the assignment transfer operator. The soundness of redundancy removal and the test transfer operator is obvious. For the entailment check of an inequality with respect to a polyhedron by rigorous LP, a positive answer indicates actual entailment while a negative answer is inconclusive. Indeed, if an inequality is entailed but is close to or touches the polyhedron, rigorous LP may give a too conservative objective value and fail to declare the entailment. As a consequence, our inclusion test actually outputs either “true” or “don’t know”. This kind of approximation does not alter the overall soundness of an analysis by abstract interpretation. A similar argument can be given for the incremental emptiness test. Another consequence is that our widening may keep fewer inequalities than an exact implementation would, but this is also sound.

5 Precision and Efficiency Issues

Each operation of the floating-point polyhedra domain outputs over-approximations of those of the rational domain, which indicates that some loss of precision may happen along with each operation. Also, the conservative results returned by rigorous LP cause efficiency degradations since redundancy removal may fail to remove many constraints generated during Fourier-Motzkin elimination, making the system larger and larger as the analysis proceeds. This section addresses these problems from a practical point of view. We propose some tactics to regain some precision and make the domain more efficient while still retaining soundness.

5.1 Bounds Tightening

The bounds of variables play a very important role in our domain, as they determine how much precision is lost in both the linearization and the rigorous LP. The bounds may change along with the operations on the polyhedra, especially when the polyhedron is restricted by adding new constraints. In this case, the bounds must be updated. Bounds tightening can be achieved using different strategies.

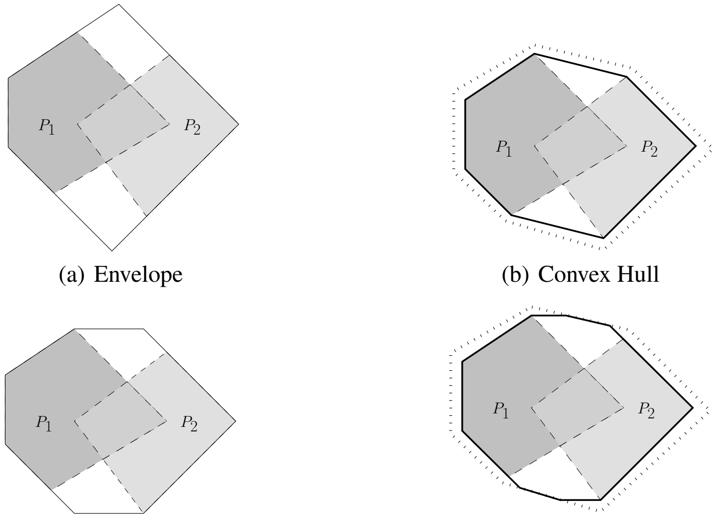
Rigorous Linear Programming. A simple way to tighten the bound information of a polyhedron P is to use the rigorous LP, to calculate max (min) x_k subject to P and get the upper (lower) bound of variable x_k. However, since the result given by rigorous LP itself depends on the range of the bounding box, the bounds found by rigorous LP may be too conservative, especially when the bounds of some variable are very large or even lost after widening. In addition, it is costly to run $2n$ linear programs after every operation on an n-dimensional polyhedron. Thus we need some alternative lightweight methods for bounds tightening.

Bound Propagation. Bound propagation is a kind of constraint propagation widely used in constraint programming. Each inequality in the linear constraints of the polyhedron can be used to tighten the bounds for those variables occurring in it. Given an inequality $\sum_i a_i x_i \leq b$, if $a_i > 0$, a new candidate upper bound v for x_i comes from: $x_i \leq v = (b - \sum_{j \neq i} a_j x_j)/a_i$. In practice, an over-approximation of v can be computed by interval arithmetic with outward rounding. If $a_i < 0$, we find a new candidate lower bound in the same way. If the new bounds are tighter, then x_i's bounds are updated. This process can be repeated with each variable in that inequality and with each inequality in the system.

Combining Strategies. In fact, the above two methods for bounds tightening are complementary with respect to each other. Each of them may find tighter bounds than the other one in some cases.

Example 3. Given $\{-x + 3y \leq 0, x - 6y \leq -3\}$ with the bounds $x, y \in (-\infty, +\infty)$, the bound propagation fails to find any tighter bounds while the rigorous LP will only find the tighter bounds $x \in [3, +\infty), y \in (-\infty, +\infty)$. Then, if we perform bound propagation on $\{-x+3y \leq 0, x-6y \leq -3\}$ with the bounds $x \in [3, +\infty)$ and $y \in (-\infty, +\infty)$, the exact bounds $x \in [3, +\infty)$ and $y \in [1, +\infty)$ can be found.

Therefore, we should combine the above strategies and strike a balance between cost and precision. For example, we can use rigorous LP to tighten only the bounds of those variables that appear with high frequency in the system, and then use bound propagation to tighten the other variables. Note that both rigorous LP and bound propagation are sensitive to the ordering of variables considered. More precision can be achieved, at greater cost, by iterating the process.



(a) Envelope

(b) Convex Hull

(c) Envelope & Bounds

(d) Convex Hull Tightening

Fig. 1. (a) the envelope $env(P_1, P_2)$ (solid lines), (b) the exact convex hull (solid bold lines) and a possible approximate floating-point convex hull (dotted lines), (c) the smallest polyhedron which can be determined by the envelope and bounding box (solid lines), (d) the floating-point convex hull (dotted lines) and the convex hull after tightening by the envelope and bounding box (solid bold lines).

5.2 Convex Hull Tightening

The convex hull computation is the most complicated part of our domain and also where the most severe loss of precision may happen because it is internally implemented via Fourier-Motzkin elimination. In many cases, part of the precision can be recovered by applying certain heuristics, such as using the envelope [4] and bounds information.

Definition 3 (Envelope). *Given two polyhedra P_1 and P_2, represented by sets of linear inequalities, the envelope of P_1 and P_2 is defined as*

$$env(P_1, P_2) \overset{\text{def}}{=} \mathcal{S}_1 \cup \mathcal{S}_2$$

where

$$\mathcal{S}_1 = \{\, \varphi_1 \in P_1 \mid P_2 \models \varphi_1 \,\},$$
$$\mathcal{S}_2 = \{\, \varphi_2 \in P_2 \mid P_1 \models \varphi_2 \,\}.$$

It is easy to see that the envelope is an over-approximation of the convex hull and contains a subset of the constraints defining the convex hull. In other words, all the inequalities in the envelope can be safely added to the final convex hull. Using rigorous LP, most of the envelope constraints can be determined by entailment checking on the arguments, before the convex hull computation.

The bounding box of the convex hull can also be obtained exactly before the convex hull computation as it is the join, in the interval domain, of the bounding box of the arguments.

We add all constraints from the envelope and the bounding box to tighten the floating-point convex hull and retrieve some precision while still retaining soundness, as shown in Fig. 1. This is of practical importance because at the point of widening, such constraints often hold a large percentage of the stable ones.

5.3 Linearization Heuristics

In polyhedral analysis, new relationships between variables are often derived from the convex hull operation. Their actual coefficients depend greatly on the choices made during the linearization step, in particular the choice of $d_k \in [a_k, b_k]$. In Sect. 4.1, we advocated the use of the interval mid-point $d_k = (a_k \oplus_r b_k) \oslash_r 2$, a greedy choice as it minimizes the constant term of the result constraint. However, choosing a more regular value, such as an integer, will improve the efficiency and numerical stability of subsequent computations, such as LP solving. In addition, due to rounding errors, computations that give the same result in exact arithmetic may give different floating-point results in floating-point arithmetic. Thus, it is desirable that the same d_k is chosen when some slight shift occurs on the input interval $[a_k, b_k]$. This is particularly important when looking for stable invariants in loops.

In practice, we use two strategies: rounding the mid-point to the nearest integer and reusing the coefficient already chosen for another variable. Other strategies may be devised. In fact, it is even possible to choose d_k outside $[a_k, b_k]$, by slightly adapting the formula in Def. 2.

5.4 Efficient Redundancy Removal

As mentioned before, the rigorous LP may fail to detect some redundant constraints due to the conservative over-approximation of the objective value, which greatly weakens the tractability of our domain. However, it is worth mentioning that the removal operation is always sound even when some non-redundant constraints are removed, in which case, the result is merely a larger polyhedron. In order to remove as many redundant constraints as possible, we can use less conservative approaches which may remove constraints that are *likely* to be redundant, but may not be. One approach is to employ a standard LP solver instead of a rigorous one. We can even go further by removing inequalities $\varphi = \sum_i a_i x_i \leq b$ in P when max $\sum_i a_i x_i$ subject to $P \setminus \{\varphi\}$ is less than $(1 + \epsilon)b$, for some tolerance $\epsilon > 0$.

In order to remove constraints more efficiently, it is worth using lightweight redundancy removal methods first and resorting to the expensive LP-based method only when necessary. First, we use a syntactic check: given a pair of inequalities $\sum_i a_i x_i \leq b$ and $\sum_i a'_i x_i \leq b'$, if $\forall i. a'_i = a_i$, only the inequality with the smaller constant needs to be kept. Second, we first check an inequality against the bounding box of the polyhedron before the actual polyhedron. Finally, we employ methods proposed in [10, 11] to tackle the combinatorial explosion problem of redundant constraints occurring during sequences of Fourier-Motzkin eliminations (e.g., in the join computations).

```
$Y \leftarrow [-M, M]$;
while random() {
      $X \leftarrow [-128, 128]$;
      $D \leftarrow [1, 16]$;
      $S \leftarrow Y$;
 ①    $R \leftarrow X \ominus_? S$;
      $Y \leftarrow X$;
      if $R \leq \ominus D$ {   $Y \leftarrow S \ominus_? D$   }  else
      if $D \leq R$    {   $Y \leftarrow S \oplus_? D$   }
} ②
```

Fig. 2. Floating-point rate limiter program. Different values of the parameter M give different versions of the program (see Fig. 3). $\odot_?$ denotes single precision floating-point semantics with arbitrary rounding mode ($? \in \{+\infty, -\infty\}$).

6 Implementation and Experimental Results

Our prototype domain, FPPol, is developed using only double precision floating-point numbers. It makes use of GLPK (GNU Linear programming kit) [14] which implements the simplex algorithm for linear programming. FPPol is interfaced to the APRON library [1] which provides a common interface for numerical abstract domains. Our experiments were conducted using the Interproc [12] static analyzer. In order to assess the precision and efficiency of FPPol, we compare the obtained invariants as well as the performance of FPPol with the NewPolka library which is implemented using exact arithmetic in APRON.

We tested FPPol on all examples from Interproc. Most of them are pure integer programs using exact arithmetic, except *numerical* which is a program involving both integer and real variables with floating-point arithmetic. We also analyzed the *ratelimiter* program presented in Fig.2, which is a more challenging example extracted from a real-life system and uses single precision floating-point numbers. In theory, any interval $[-M, M]$, where $M = 128 + \epsilon$ and $\epsilon > \epsilon_0$, is stable at ②, for some very small positive ϵ_0. Because this example requires relational invariants, the non-relational interval domain fails to find any stable interval for Y, while the weakly relational octagon domain, although better, can only find over-approximated stable intervals wherein $M > M_0$ and $M_0 \approx 144.00005$. The smallest stable interval that can be found using the polyhedra domain is the interval $[-M_1, M_1]$ wherein $M_1 \approx 128.000047684$. This example is interesting since abstracting floating-point expressions to linear interval expressions over reals [15] gives rise to rather complex expressions. For example, at ①, the assignment $R \leftarrow X \ominus_? S$ is abstracted into:

$$R \leftarrow [1 - p, 1 + p] \times X - [1 - p, 1 + p] \times S + [-mf, mf]$$

with $p = 2^{-23}$ and $mf = 2^{-149}$ (respectively corresponding to the relative error and the smallest non-zero positive value in the single precision floating-point format). Note that this expression contains numbers of large magnitude, which are costly to represent using exact rationals.

Fig. 3 shows the type of each benchmark program: “int” means the program involves only integer variables with exact arithmetic and “fp” means that the program

Program		Analyzer	FPPol			NewPolka		Result
type	name	#∇delay	#iterations	#lp	$t(ms)$	#iterations	$t(ms)$	Invar.
int	ackerman	1	6	1476	35	6	7	=
int	bubblesort	1	8	675	24	8	8	=
int	fact	1	9	2106	65	9	15	=
int	heapsort	1	4	1968	76	4	15	=
int	maccarthy91	1	4	418	13	4	3	=
int	symmetricalstairs	1	6	480	18	6	6	=
fp	numerical	1	1	250	17	1	31	≈
fp	ratelimiter(M=128)	3	5	1777	125	5	394	≈
fp	ratelimiter(M=128)	4	5	2555	227	6	809	>
fp	ratelimiter(M=128.000047683)	6	9	4522	510	8	1889	≈
fp	ratelimiter(M=128.000047683)	7	8	3688	238	9	2435	>
fp	ratelimiter(M=128.000047684)	1	3	1068	57	3	116	≈

Fig. 3. Experimental results for benchmark examples.

involves real variables with floating-point arithmetic. The column "#∇ delay" specifies the value of the widening delay parameter for Interproc (i.e., the number of loop iterations performed before applying the widening operator). The column "#iterations" gives the number of loop iterations before a fixpoint is reached.

Invariants. The column "Result Invar." compares the invariants obtained. A "=" indicates that FPPol outputs exactly the same invariants as NewPolka. A "≈" means that FPPol finds the same invariants as NewPolka, up to slight variations in coefficients due to rounding. In this case, the polyhedra computed by FPPol are slightly larger than those computed by NewPolka. A ">" denotes that FPPol finds strictly stronger invariants than NewPolka. For the integer programs, all the invariants obtained by FPPol were the same as those produced by NewPolka. Indeed, such programs involve only small integer values. In these cases, we can often use the multiplicative version of the Fourier-Motzkin elimination, which incurs no precision loss.

The *numerical* program involves floating-point arithmetic but without loops, so it provides no challenge. For *ratelimiter*, FPPol can find the invariant $-M_1 \leq x \leq M_1$ where $M_1 \approx 128.000047684$ if the widening delay parameter is set large enough: at least 4 when M=128, 7 when M=128.000047683, 1 when M=128.000047684, whereas NewPolka can only find the invariant when M=128.00004 7684. Interestingly, for *ratelimiter* with $M = 128.000047683$, NewPolka fails to find any invariant at ② even when delaying the widening for 100 iterations (413.2 seconds). In this case, the floating-point over-approximations within FPPol actually accelerate the fixpoint computation even before applying widening and help in reaching a fixpoint faster than when using NewPolka.

Performance. Fig. 3 presents the analysis times in milliseconds when the analyzer runs on a 2.4GHz PC with 2GB of RAM running Fedora Linux. For integer programs, NewPolka outperforms FPPol. Because such programs involve small integer values, the computation in NewPolka is very cheap while FPPol needs a number of expensive

LP queries. However, for the floating-point programs, FPPol greatly outperforms NewPolka. Indeed, after floating-point abstractions, programs involve rational numbers of large magnitude which degrade the performance of NewPolka, while the floating-point number representation avoids such problems in our domain.

LP costs. Fig. 3 shows also statistics on the number of LP queries (#*lp*) in FPPol. In addition, we found that, for floating-point programs, more than 75% of the LP queries are used for redundancy removal, almost 80% of which come from the convex hull computation. The performance of our domain completely relies on the LP solver we use. During our experiments, we found that the time spent in the LP solver frequently takes more than 85% of the total analysis time for floating-point programs and 70% for integer programs. Note that a naive floating-point implementation of polyhedra, without any soundness guarantee, could not bypass these LP computations either. Thus, the soundness guarantee in our domain does not incur much overhead.

Numerical instability. During our experiments on floating-point programs, GLPK often encountered the "numerical instability" problem due to the simultaneous occurrence of tiny and large coefficients. Indeed, during the analysis, tiny floating-point numbers introduced due to rounding are propagated in the whole system and produce large coefficients by division. In our implementation, we solve the problem by shifting the tiny term or huge term into the constant term following the same idea as linearization in Sect. 4.1, e.g, choosing $d_k = 0$. We believe a faster, more robust LP solver with better scalability, such as the CPLEX LP solver, may greatly improve the precision, performance and scalability of our domain.

7 Conclusion

In this paper, we presented a sound implementation of the polyhedra domain using floating-point arithmetic. It is based on a constraint-only representation, together with a sound floating-point Fourier-Motzkin elimination algorithm and rigorous linear programming techniques. Moreover, we proposed advanced tactics to improve the precision and efficiency of our domain, which work well in practice. The benefit of our domain is its compact representation and the ability to leverage the power of state-of-the-art linear programming solvers. It remains for future work to examine the scalability of our domain for large realistic programs and to reduce the number of LP queries.

Acknowledgments

We would like to thank Axel Simon, Ji Wang and the anonymous reviewers for their helpful comments and suggestions.

References

1. APRON numerical abstract domain library. http://apron.cri.ensmp.fr/library/.
2. S. Alexander. *Theory of Linear and Integer Programming*. John Wiley & Sons, June 1998.

3. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006.
4. A. Bemporad, K. Fukuda, and F. D. Torrisi. Convexity recognition of the union of polyhedra. *Computational Geometry*, 18(3):141–154, 2001.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM PLDI'03*, pages 196–207, San Diego, California, USA, June 2003. ACM Press.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM POPL'77*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM POPL'78*, pages 84–96, New York, NY, USA, 1978. ACM.
8. E. Goubault. Static analyses of floating-point operations. In *SAS'01*, pages 234–259. Springer-Verlag, 2001.
9. N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Thèse de 3ème cycle d'informatique, Université scientifique et médicale de Grenoble, Grenoble, France, March 1979.
10. T. Huynh, C. Lassez, and J.-L. Lassez. Practical issues on the projection of polyhedral sets. *Annals of Mathematics and Artificial Intelligence*, 6(4):295–315, 1992.
11. J.-L. Imbert. Fourier's elimination: Which to choose? In *PCPP'93*, pages 117–129, 1993.
12. G. Lalire, M. Argoud, and B. Jeannet. Interproc. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/.
13. H. LeVerge. A note on Chernikova's algorithm. Technical Report 635, IRISA, France, 1992.
14. A. Makhorin. The GNU Linear Programming Kit, 2000. http://www.gnu.org/software/glpk/.
15. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, volume 2986 of *LNCS*, pages 3–17. Springer, Barcelona, Spain 2004.
16. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES'06*, pages 54–63, Ottawa, Ontario, Canada, 2006. ACM Press.
17. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
18. R. Moore. *Interval Analysis*. Prentice-Hall, 1966.
19. A. Neumaier and O. Shcherbina. Safe bounds in linear and mixed-integer linear programming. *Math. Program.*, 99(2):283–296, 2004.
20. D. N. Que. Robust and generic abstract domain for static program analysis: the polyhedral case. Technical report, École des Mines de Paris, July 2006.
21. J. Rohn. Solvability of systems of interval linear equations and inequalities. In *Linear Optimization Problems with Inexact Data*, pages 35–77. Springer, 2006.
22. S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI'05*, volume 3385, pages 21–47.
23. A. Simon and A. King. Exploiting sparsity in polyhedral analysis. In Chris Hankin, editor, *SAS'05*, volume 3672 of *LNCS*, pages 336–351. Springer Verlag, September 2005.
24. A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In *LOPSTR'03*, volume 2664 of *LNCS*, pages 71–89. Springer, 2003.