Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra

Alexis Fouilhé, David Monniaux, Michaël Périn

Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra*

Alexis Fouilhé† David Monniaux‡ Michaël Périn*

April 2, 2013

Abstract

Polyhedra form an established abstract domain for inferring runtime properties of programs using abstract interpretation. Computations on them need to be certified for the whole static analysis results to be trusted. In this work, we look at how far we can get down the road of a posteriori verification to lower the overhead of certification of the abstract domain of polyhedra. We demonstrate methods for making the cost of inclusion certificate generation negligible. From a performance point of view, our single-representation, constraints-based implementation compares with state-of-the-art implementations.

In static analysis by abstract interpretation [6], sets of reachable states, which are in general infinite or at least very large and not amenable to tractable computation, are over-approximated by elements of an *abstract domain* on which the analyzer applies forward (resp. backward) steps corresponding to program operations (assignments, tests...) as well as "joins" corresponding to control points with several incoming (resp. outgoing) edges. When dealing with numerical variables in the analyzed programs, one of the simplest abstract domains consists in keeping one interval per variable, and the forward analysis is known as *interval arithmetic*. Interval arithmetic however does not keep track of relationships between variables. The domain of *convex polyhedra* [7] tracks relationships of the form $\sum_i a_i x_i \bowtie b$ where the a_i and b are integer (or rational) constants, the x_i are rational program variables, and $\bowtie$ is $\leq$, $<$ or $=$.

The implementor of an abstract domain faces two hurdles: the implementation should be reasonably efficient and scalable; it should be reasonably bug-free. As an example, the Parma Polyhedra Library (PPL) [1], version 1.0, which implements several relational numerical abstract domains, comprises

*This work was partially supported by ANR project "VERASCO" (INS 2011).

†Université Joseph Fourier / VERIMAG; VERIMAG is a joint laboratory of Université Joseph Fourier, CNRS and Grenoble-INP.

‡CNRS / VERIMAG

260,000 lines of C++; despite the care put in its development, it is probable that bugs have slipped through. The same applies to the Apron library [11].

Such hurdles are especially severe when the analysis is applied to large-scale critical programs (e.g. in the Astrée system [5], targeting avionics software). For such systems, normal compilers may not be trusted, resulting in expensive post-compilation checking procedures, and prompting the development of the CompCert certified compiler [12]: this compiler is programmed, specified and proved correct in Coq [21]. We wish to extend this approach to obtain a trusted static analyzer; this article focuses on obtaining a trusted library for convex polyhedra, similar in features to the polyhedra libraries at the core of PPL and Apron.

One method for certifying the results of a static analysis is to store the invariants obtained by an untrusted analyzer at (roughly) all program points, then check that they are inductive using a trusted checker: each statement is then a Hoare triple that must be checked. Unfortunately, storing invariants everywhere proved impractical in the Astrée analyzer due to memory consumption; we then opted to recompute them. Our (future) analyzer will thus store invariants only at loop heads, and thus, for control programs consisting of one huge control loop plus small, unrolled, inner loops, will store only a single invariant. It will then enter a checking phase which will recompute, in a trusted fashion, all intermediate invariants. Efficiency is thus important.

The main contribution of our article is an efficient way of implementing a provably correct abstract domain of polyhedra. Efficiency is two-fold:

1. In proof effort: most of the implementation consists in an untrusted oracle providing *certificates* of the correctness of its computations; only a much smaller certificate checker, consisting in simple algorithms (multiplying and adding vectors, replacing a variable by an expression), needs to be proven correct in the proof assistant.

2. In execution time: the expensive parts of the computations (e.g. linear programming) are inside the untrusted oracle and may use efficient programming techniques unavailable in parts that need formal proofs. We do not compute certificates as an afterthought of polyhedral computations: close examination of the algorithms implementing the polyhedral operators revealed that they directly expose the elements needed to build certificates. Simple bookkeeping alleviates the need to rebuild them after the fact. The overhead of making the operators certifying is thus very limited. This contrasts with earlier approaches [4] based on *a posteriori* generation of witnesses, which had to be recomputed from scratch using linear programming.

A second contribution is a complete implementation of the abstract domain of polyhedra in a purely constraints-based representation. Most

libraries used in static analysis, including PPL and Apron, use a double description: a polyhedron is both an intersection of half-spaces (constraints) or the convex hull of vertices, half-lines and lines (generators), with frequent conversions. Unfortunately, the generator representation is exponential in the number of constraints, including for common cases such as hypercubes (e.g. specification of ranges of inputs for the program). We instead chose to represent the polyhedra solely as lists of constraints, with pruning of redundant ones. Our implementation uses sparse matrices of rational numbers and uses efficient techniques for convex hull [20] and emptiness testing by linear programming [9].

We applied our library to examples of polyhedral computations obtained by running the Pagai static analyzer [10] on benchmark programs. Despite a common claim that implementations based on the double representation are more efficient than those based on constraints only, our library reaches performance comparable to the Apron library together with the high-level of trust brought by our Coq certificate checker.

The remainder of this paper is organized as follows. After having stated the conventions we are using (§1), we define correctness criteria for the operators of the abstract domain (§2), which all reduce to inclusion properties for which certificates are presented as *Farkas coefficients* (also known as Lagrange multipliers). Such certificates may also be cheaply generated for the convex hull (§4). Both forward step and convex hull operations reduce internally to a form of projection. Some design choices of our implementation are then described (§5), including how to keep the representation size of the polyhedra reasonable. Last before conclusion (§7), an experimental evaluation and accompanying results are presented (§6).

1 Definitions and Notations

In the remainder of this article, we use the following notations and definitions.

1. C: a linear constraint of the form $\vec{a} \cdot \vec{x} \leq b$ where $\vec{a}$ is a vector of rational constants, b is a rational and $\vec{x} \in \mathbb{Q}^n$ is the vector of the analyzed program variables. Such a linear constraint, or constraint for short, can be viewed as a half-space in an n-dimensional space. We write $\overline{C} \stackrel{\text{def}}{=} \vec{a} \cdot \vec{x} > b$ for the complementary half-space.

2. P: a convex polyhedron, not necessarily closed, represented as a set of constraints. We call "size of the representation of P" the number of constraints that P is made of.

3. satisfaction: saying that point $\vec{y}$ of $\mathbb{Q}^n$ satisfies a constraint $C \stackrel{\text{def}}{=} \vec{a} \cdot \vec{x} \leq b$ means that $\vec{a} \cdot \vec{y} \leq b$. By extension, a point $\vec{y}$ satisfies (or is in) polyhedron P if it satisfies all of its constraints. We write this: Sat P $\vec{y}$.

Given that a constraint C can be regarded as polyhedron with only one constraint, we also write: Sat C $\vec{y}$.

4. Given our focus on the abstract domain of polyhedra we shall adopt the following vocabulary.
   (a) The order relation on polyhedra $\sqsubseteq$ is geometrical inclusion.
   (b) The least upper bound $\sqcup$ is the convex hull.
   (c) The greatest lower bound $\sqcap$ is geometrical intersection.

   We will further distinguish the definition of abstract domain operators from their actual implementation, which can have bugs. The implemented version of the operators will be written with a hat: $\widehat{\sqsubseteq}$, $\widehat{\sqcup}$ and $\widehat{\sqcap}$ implement the ideal operators $\sqsubseteq$, $\sqcup$ and $\sqcap$, respectively.

5. inclusion: a polyhedron P_1 is included in a polyhedron P_2 (noted $P_1 \sqsubseteq P_2$) if and only if
$$\forall \vec{y}, \text{Sat } P_1\ \vec{y} \Rightarrow \text{Sat } P_2\ \vec{y} \tag{1}$$
   Inclusion for constraints $C_1 \stackrel{\text{def}}{=} \vec{a}_1 \cdot \vec{x} \leq b_1$ and $C_2 \stackrel{\text{def}}{=} \vec{a}_2 \cdot \vec{x} \leq b_2$ is a special case which is easy to decide: $C_1 \sqsubseteq C_2$ holds if and only if there exists $k > 0$ such that $k \cdot \vec{a}_1 = \vec{a}_2$ and $k \cdot b_1 \leq b_2$. This latter case is thus proven correct directly inside Coq.

2 Correctness of the Abstract Domain Operators

Let us now see what needs to be proven for the implementation of each operator of an abstract domain so that the correctness of its result can be established.

Inclusion test $P_1 \widehat{\sqsubseteq} P_2 \Rightarrow P_1 \sqsubseteq P_2$

Convex hull $P_1 \sqsubseteq P_1 \widehat{\sqcup} P_2$ and $P_2 \sqsubseteq P_1 \widehat{\sqcup} P_2$

Intersection $\forall \vec{x}, \text{Sat } P_1\ \vec{x} \wedge \text{Sat } P_2\ \vec{x} \Rightarrow \text{Sat } P_1 \widehat{\sqcap} P_2\ \vec{x}$.
For now, we will assume a naive implementation of the intersection: $P_1 \widehat{\sqcap} P_2$ is the union of the constraints of P_1 with these of P_2, which trivially satisfies the desired property (1).

Assignment in a forward analysis, $x := e$ amounts to intersection by the equality constraint $x' = e$ (where x' is a fresh variable), projection of x and renaming of x' to x.[1] When analyzing backward, assignment is just substitution.

[1] Other polyhedra libraries distinguish invertible assignments (e.g. $x := x + 1$, more generally $\vec{x'} = A \cdot \vec{x}$ with A an invertible matrix), which can be handled without projection, from non-invertible ones (e.g. $x := y + z$). Because our library automatically keeps a canonical system of equalities, which it uses if possible when projecting, no explicit detection of invertibility is needed; it is subsumed by the canonicalization.

Projection if P_2 is the returned polyhedron for the projection of P_1 parallel to variables $x_{i_1}, \ldots, x_{i_p}$ we check that $P_1 \sqsubseteq P_2$ and that variables $x_{i_1}, \ldots, x_{i_p}$ do not appear in the constraints defining P_2.

Widening : no correctness check needed. Widening ($\triangledown$) is used to accelerate the convergence of the analysis to a candidate invariant. For partial correctness of the analyzer, no property is formally needed of the widening operator, since iterations stop when the inclusion test reports that an inductive invariant has been obtained. There exist formalizations of the widening operator suitable for proving the total correctness of the analysis (that is, that it eventually converges to an inductive invariant) [15] but we avoided this question by assuming some large upper bound on the number of iterations after which the analyzer terminates with an error message.

Remark that we only prove that the returned polyhedron *contains* the polyhedron that it should ideally be (which is all that is needed for proving that the results of the analysis are sound), not that it *equals* it: for instance, we prove that the polyhedron returned by the convex hull operator includes the convex hull, not that it is the true convex hull. The *precision* of our algorithms (that is, the property that they do not return polyhedra larger than needed) is not proved formally; it is however ensured by usual software engineering methods (informally correct algorithms, comparing the output of our implementation to that of other polyhedra libraries...).

3 A Posteriori Verification of the Inclusion Test

We shall now describe a way to ensure the correctness of the inclusion test. Recall we represent polyhedra as sets of constraints only. Our certificate for proving that a polyhedron P, composed of the constraints $C_1, \ldots, C_n$ satisfies a constraint C relies on the following trivial fact:

Lemma 1 *If a point $\vec{y}$ satisfies a set of constraints $\{C_1, \ldots, C_n\}$, it satisfies any linear positive combination $\sum_{i=1}^{n} \lambda_i C_i$ with $\lambda_i \geq 0$.*

If we can find a constraint C' that is a linear positive combination of C_1, $\ldots, C_n$ and such that $C' \sqsubseteq C$ then it follows that P is included in C. Farkas' lemma states that such linear combinations necessarily exist when inclusion holds, which justifies our approach.

The motivation for *a posteriori* verification of inclusion results stems from this formulation: while finding an appropriate linear combination requires advanced algorithms, a small program checking that a particular set of λ_i's entails $P \sqsubseteq C$ can easily be proven correct in a proof assistant. We call these λ_i's the *certificate* for $P \sqsubseteq C$.

3.1 A Certificate Checker Certified in Coq

Our certificate checker has Coq type:

$$\texttt{inclusion\_checker}\ (P_1\ P_2 : \textbf{Polyhedra})\ (\mathit{cert} : \textbf{Cert}) : \textbf{Exception}\ (P_1 \sqsubseteq P_2)$$

where the type **Polyhedra** is a simple representation of a polyhedron as a list of linear constraints and the type **Cert** is a representation for inclusion certificates. If a proof of $P_1 \sqsubseteq P_2$ can be built from *cert*, then the `inclusion_checker` returns it wrapped in the constructor VALUE. However *cert* might be incorrect due to a bug in $\widehat{\sqsubseteq}$. In this case, the `inclusion_checker` fails to build a proof of $P_1 \sqsubseteq P_2$ and returns ERROR.

When extracting the OCaml program from the Coq development, proof terms are erased and the type of the checker function becomes that which would have been expected from a hand-written OCaml function: [2]

$$\texttt{inclusion\_checker} : \textbf{Polyhedra} \to \textbf{Polyhedra} \to \textbf{Cert} \to \textbf{bool}$$

In reality, our implementation is slightly more complicated because the untrusted part of our library, for efficiency reasons, operates on fast rational and integer arithmetic, while the checker uses standard Coq types that explicitly represent integers as a list of bits (see §5.6).

3.2 A Certificate-Generating Inclusion Test

Let us now go back to the problem of building a proof of $P \sqsubseteq C$ by exhibiting an appropriate linear combination. From [4], this can be rephrased as a pure satisfiability problem in linear programming:

$$\left(\forall y, \neg\mathrm{Sat}\ \left(P \sqcap \overline{C}\right)\ y\right) \Rightarrow P \sqsubseteq C$$

This problem can be solved by the simplex algorithm [8]. For this purpose, the simplex variant proposed by [9], designed for SMT-solvers, is particularly well-suited. This algorithm only implements the first of the two phases of the simplex algorithm: finding a feasible point, that is a point satisfying all the constraints of the problem. If there is no such point, a witness of unsatisfiability is extracted as a set of mutually exclusive bounds on linear terms and suitable Farkas coefficient $\vec{\lambda}$, in the same way that blocking clauses for theory lemmas are obtained for use in SMT-solving modulo linear rational arithmetic. Furthermore, this algorithm is designed for cheap backtracking (addition and removal of constraints), which is paramount in SMT-solving and also very useful in our application (§5.2).

[2] We chose to replace the constructors VALUE and ERROR of the type **Exception** by OCaml booleans instead of letting the extraction define an OCaml type "exception" with two nullary constructors due to proof terms being erased.

Our approach to certificate generation differs from previous suggestions [4] where inclusion is first tested by untrusted means, and, if the answer is positive, a vector of Farkas coefficients is sought as the solution of a dual linear programming problem with optimization, which has a solution, the Farkas coefficients, if and only if the primal problem has no solution. Ours uses a primal formulation without optimization.

3.3 From an Unsatisfiability Witness to an Inclusion Certificate

Inclusion certificates are derived from unsatisfiability witness in a way similar to [18]. To illustrate how they are built as part of the inclusion test, a global idea of the inner workings of the simplex variant from [9] is needed. We insist on the following being a coarse approximation.

We aim at building, given P non-empty and C, an inclusion certificate for $P \sqsubseteq C$, otherwise said $P \wedge \overline{C}$ having no solution. P is composed of n constraints $C_1, \ldots, C_n$ of the form $\sum_{j=1}^{n} a_{ij} \cdot x_j \leq b_i$, where i is the constraint subscript. We refer to $\overline{C} \stackrel{\text{def}}{=} b_0 < \sum_{j=1}^{n} a_{0j} \cdot x_j$ as C_0.

Let us start by describing the organization of data. Each constraint C_i is split into an equation $x'_i = \sum_{j=1}^{n} a_{ij} \cdot x_j$ and a bound $x'_i \leq b_i$ where x'_i is a fresh variable. For the sake of simplicity, in this presentation, a constraint $x_i \leq b_i$ is represented as $x'_i = x_i \wedge x'_i \leq b_i$; the actual implementation avoids introducing such extra variable. Therefore, each x'_i uniquely identifies C_i by construction and the original variables x_i are unbounded. We call *basic* the variables which are defined by an equation (*i.e.* on the left-hand side, with unit coefficient) and *non-basic* the others. Last, the algorithm maintains a candidate feasible point, that is a value for every variable x'_i and x_j, initially set to 0.

From this starting point, the algorithm iterates pivoting steps while ensuring preservation of the invariant: *the candidate feasible point always satisfy the equations and the values of the non-basic variables always satisfy their bounds (‡).* At each iteration and prior to pivoting, a basic variable x'_i is chosen such that its value does not satisfy its bounds. Either there is no such x'_i, and the candidate feasible point is indeed a solution of $P \wedge \overline{C}$, thereby disproving $P \sqsubseteq C$; or there is such a basic variable x'_i. In this case, we shift its value to fit its bounds and we seek a non-basic variable x'_n such that its value can be adjusted to compensate the shift: through a pivoting step, x'_i becomes non-basic, and x'_n becomes basic. If there is no such x'_n (because all the non-basic variables already have reached their bound), the equation which defines x'_i exhibits incompatible bounds of the problem and is of the form $x'_i = \sum_{j \neq i} \lambda_j \cdot x'_j$ (only x'_j's appear in this equation: recall that the x_j's are unbounded). We now show how to transform this unsatisfiability result into an inclusion certificate.

Since we supposed that P is not empty, the unsatisfiability necessarily

involves C_0. Thus, x'_0, which represents C_0, has a non-zero coefficient λ_0 in the equation. Without loss of generality, we suppose that the incompatible bounds involve an upper bound on x'_i and that λ_0 is positive. The above equation can be rewritten so that x'_0 appears on the left-hand side:

$$x'_0 = \sum_{j=1}^{n} \lambda'_j \cdot x'_j$$

where the lower bound $b_0 < x'_0$ and the upper bound $\sum_{j=1}^{n} \lambda'_j \cdot x'_j \leq b'$ are such that $b' \leq b_0$. Recall that the x'_i's were defined as equal to linear terms $l_i \stackrel{\text{def}}{=} \sum_{j=1}^{n} a_{ij} \cdot x_j$ of the constraints C_i. Let us now substitute the x'_i's by their definition, yielding

$$l_0 = \sum_{j=1}^{n} \lambda'_j \cdot l_j$$

Noting that C is $l_0 \leq b_0$ (since $C_0 = \ b_0 < l_0$ is $\overline{C}$), that $\sum_{j=1}^{n} \lambda'_j \cdot l_j \leq b$ and that $b' \leq b_0$, the λ'_j's form an inclusion certificate for $P \sqsubseteq C$.

4 A Posteriori Verification of the Convex Hull

We saw in §2 that the result of the convex hull of two polyhedra P_1 and P_2 must verify inclusion properties with respect to both P_1 and P_2. Computing $P \stackrel{\text{def}}{=} P_1 \widehat{\sqcup} P_2$, then $P_1 \mathrel{\widehat{\sqsubseteq}} P$ and $P_2 \mathrel{\widehat{\sqsubseteq}} P$ and then checking the certificates would produce a certified convex hull result, at the expense of two extra inclusion tests. From a development point of view, this is the lightest approach. However, careful exploitation of the details of $\widehat{\sqcup}$ can save us the extra cost of certificate generation, at the expense of some development effort.

Before delving into the details, let us introduce some more notations for the sake of brevity. In this section, a polyhedron P is regarded as a column vector of the constraints $C_1, \ldots, C_n$ it is composed of. This allows for a matrix notation: $P \stackrel{\text{def}}{=} \left\{ \vec{x} \mid A \cdot \vec{x} \leq \vec{b} \right\}$, where the linear term of C_i is the i^{th} line of A and the constant of C_i is the i^{th} component of $\vec{b}$.

Then, an inclusion certificate, $\lambda_1, \ldots, \lambda_n$, for $P \sqsubseteq C'$ is a line vector $\vec{\Lambda}$, such that $\vec{\Lambda} \cdot P = C$ and $C \sqsubseteq C'$. Now, an inclusion certificate for $P \sqsubseteq P'$ is a set of inclusion certificates $\vec{\Lambda}_1, \ldots, \vec{\Lambda}_n$, one for each constraint C'_i of P'. Such a set can be regarded as a matrix F such that

$$F \stackrel{\text{def}}{=} \begin{pmatrix} \vec{\Lambda}_1 \\ \vdots \\ \vec{\Lambda}_n \end{pmatrix} \text{ and } F \times P \sqsubseteq P'$$

where the i^{th} line of $F \times P$ is a constraint C such that $C \sqsubseteq C'_i$. We call $\vec{\Lambda}$ a *Farkas vector* and F a *Farkas matrix*.

4.1 A Convex Hull Algorithm on Constraints Representation

The convex hull $P_1 \sqcup P_2$ is the smallest polyhedron containing all line segments joining P_1 to P_2. Thus, a point $\vec{x}$ of $P_1 \sqcup P_2$ is the barycenter of a point $\vec{x_1}$ in P_1 and a point $\vec{x_2}$ in P_2. Exploiting this remark, [3] defined $P_1 \sqcup P_2$, with $P_i = \{\vec{x} \mid A_i \cdot \vec{x} \leq \vec{b_i}\}$, as the set of solutions of the constraints $A_1 \cdot \vec{x_1} \leq \vec{b_1} \wedge A_2 \cdot \vec{x_2} \leq \vec{b_2} \wedge \vec{x} = \alpha_1 \cdot \vec{x_1} + \alpha_2 \cdot \vec{x_2} \wedge \alpha_1 + \alpha_2 = 1 \wedge 0 \leq \alpha_1 \wedge 0 \leq \alpha_2$ using $2n + 2$ auxiliary variables $\vec{x_1}, \vec{x_2}, \alpha_1, \alpha_2$ where $n = |\vec{x}|$ is the number of variables of the polyhedron. Still following [3], the variable changes $\vec{x'_1} = \alpha_1 \cdot \vec{x_1}$ and $\vec{x'_2} = \alpha_2 \cdot \vec{x_2}$ remove the non-linearity of the equation $\vec{x} = \alpha_1 \cdot \vec{x_1} + \alpha_2 \cdot \vec{x_2}$.

The resulting polyhedron can regarded as the 3-block system S_{bar} below. The auxiliary variables $\vec{x'_1}, \vec{x'_2}, \alpha_1, \alpha_2$ are then projected out to stick to the tuple $\vec{x}$ of program variables. Therefore, the untrusted convex hull operator $\widehat{\sqcup}$ mainly consists in a sequence of projections: $P_1 \,\widehat{\sqcup}\, P_2 \stackrel{\text{def}}{=} \widehat{proj}\ S_{bar}\ (\vec{x'_1}, \vec{x'_2}, \alpha_1, \alpha_2)$ where

$$S_{bar} = \left(\begin{array}{|c|} \hline A_1 \vec{x'_1} \leq \alpha_1 \vec{b_1} \\ \hline A_2 \vec{x'_2} \leq \alpha_2 \vec{b_2} \\ \hline \begin{array}{c} \vec{x} = \vec{x'_1} + \vec{x'_2} \\ \alpha_1 + \alpha_2 = 1 \\ 0 \leq \alpha_1 \\ 0 \leq \alpha_2 \end{array} \\ \hline \end{array} \right)$$

4.2 Instrumenting the Projection Algorithm

Projecting a variable $\vec{x}_k$ from a polyhedron P represented by constraints can be achieved using Fourier-Motzkin elimination (e.g. [8]). This algorithm partitions the constraints of P into three sets: $E^0_{x_k}$ contains the constraints where the coefficient of $\vec{x}_k$ is nil, $E^+_{x_k}$ contains those having a strictly positive coefficient for $\vec{x}_k$ and $E^-_{x_k}$ contains those which coefficient for $\vec{x}_k$ is strictly negative.

Then, the result P_{proj} of the projection of $\vec{x}_k$ from P is defined as

$$P_{\text{proj}} = proj\ P\ \vec{x}_k \stackrel{\text{def}}{=} E^0_{\vec{x}_k} \cup \left(map\ elim_{\vec{x}_k}\ (E^+_{\vec{x}_k} \times E^-_{\vec{x}_k}) \right)$$

where $E^+_{\vec{x}_k} \times E^-_{\vec{x}_k}$ is the set of all possible pairs of inequalities, one element of each pair belonging to $E^+_{\vec{x}_k}$ and the other belonging to $E^-_{\vec{x}_k}$. The $elim_{\vec{x}_k}$ function builds the linear combination with positive coefficients of the members of a pair such that $\vec{x}_k$ has a zero coefficient in the result.

Illustrating on an example, projecting x from

$$P \stackrel{\text{def}}{=} \{y \leq 1, 2 \cdot x + y \leq 2, -x - y \leq 1\} \text{ gives}$$

$$E_x^0 = \{y \leq 1\} \text{ and } E_x^+ \times E_x^- = \{(2 \cdot x + y \leq 2, -x - y \leq 1)\}$$

From $1 \cdot (2 \cdot x + y \leq 2) + 2 \cdot (-x - y \leq 1) = -y \leq 4$, we get $P_{\text{proj}} = \{y \leq 1, -y \leq 4\}$.

Note that every constraint C of P_{proj} is either a constraint of P, or the result of a linear combination with non-negative coefficients λ_1, λ_2 of two constraints C_1 and C_2 of P, such that $\lambda_1 \cdot C_1 + \lambda_2 \cdot C_2 = C$. It is therefore possible, with some bookkeeping, to build a matrix F such that $F \times P = P_{\text{proj}}$. This extends to the projection of several variables: if *proj* P $\vec{x}_k = P_{\text{proj}} = F \times P$ and *proj* P_{proj} $\vec{x}_l = P'_{\text{proj}} = F' \times P_{\text{proj}}$, then $P'_{\text{proj}} = F'' \times P$ with $F'' = F' \times F$.

Fourier-Motzkin elimination can generate a lot of redundant constraints, which make the representation size of P_{proj} unwieldy. In the worst case, the n constraints split evenly into $E^+_{x_k}$ and $E^-_{x_k}$, and thus, after one elimination, one gets $n^2/4$ constraints; this yields an upper bound of $n^{2^p}/4^p$ where p is the number of elimination steps. Yet, the number of true faces can only grow in single exponential [17, §4.1]; thus most generated constraints are likely to be redundant.

The algorithm inspired from [20], which we use in practice, adds these refinements to Fourier-Motzkin elimination:

1. Using equalities when available to make substitutions. A substitution is no more than a linear combination of two constraints, the coefficients of which can be recorded in F. Note that there is no sign restriction on the coefficient applied to an equality.

2. Discarding trivially redundant constraints. The corresponding line F can be discarded just as well.

3. Discarding constraints proved redundant by linear programming, as in §5.2.

Note that, since discarding a constraint only *adds* points to the polyhedron, there is no need to prove these refinements to be correct or to provide certificates for them. We could thus very easily add new heuristics.

4.3 On-the-Fly Generation of Inclusion Certificates

In order to establish the correctness of static analysis, the convex hull operator should return a superset of the true convex hull; we thus need proofs of $P_1 \sqsubseteq P_1 \mathbin{\widehat{\sqcup}} P_2$ and $P_2 \sqsubseteq P_1 \mathbin{\widehat{\sqcup}} P_2$. The converse inclusion is not needed for correctness, though we expect that it holds; we will not prove it. A certifying operator $\widehat{\sqcup}$ must then produce for each constraint C of $P_1 \mathbin{\widehat{\sqcup}} P_2$ a certificate $\vec{\Lambda}_1$ (resp. $\vec{\Lambda}_2$) proving the inclusion of P_1 (resp. P_2) into the single-constraint polyhedron C. The method we propose for on-the-fly generation of a correctness certificate is based on the following remark.

For each constraint C of $P_1 \,\widehat{\sqcup}\, P_2$, the projection operator $\widehat{proj}$ provides a vector $\vec{\Lambda}$ such that $\vec{\Lambda} \times S_{bar} = C$, where S_{bar} is the system of constraints defined in §4.1. An examination of the certificate reveals that $\vec{\Lambda}$ can be split into three parts $(\vec{\Lambda}_1, \vec{\Lambda}_2, \vec{\Lambda}_3)$ such that $\vec{\Lambda}_1$ refers to the constraints $A_1.\vec{x}'_1 \leq \alpha_1 \vec{b_1}$ derived from P_1 ; $\vec{\Lambda}_2$ refers to the constraints $A_2.\vec{x}'_2 \leq \alpha_2 \vec{b_2}$ derived from P_2 and $\vec{\Lambda}_3$ refers to the barycenter part $\vec{x} = \vec{x}'_1 + \vec{x}'_2 \wedge \alpha_1 + \alpha_2 = 1 \wedge 0 \leq \alpha_1 \wedge 0 \leq \alpha_2$. Let us apply the substitution $\sigma = [\alpha_1/1, \alpha_2/0, \vec{x}'_1/\vec{x}, \vec{x}'_2/\vec{0}]$, that characterizes the points of P_1 as some extreme barycenters, to each terms of the equality $\vec{\Lambda} \times S_{bar} = C$. This only changes S_{bar}: Indeed, $\vec{\Lambda}\sigma = \vec{\Lambda}$ since $\vec{\Lambda}$ is a constant vector and $C\sigma = C$ since none of the substituted variables appears in C (due to projection). We obtain the equality (below) where many constraints of $S_{bar}\sigma$ became trivial.

$$(\vec{\Lambda}_1, \vec{\Lambda}_2, \vec{\Lambda}_3) \times \begin{pmatrix} \boxed{A_1\vec{x} \leq \vec{b_1}} \\ \boxed{0 \leq 0} \\ \boxed{\begin{array}{c} \vec{x} = \vec{x} \\ 1 = 1 \\ 0 \leq 1 \\ 0 \leq 0 \end{array}} \end{pmatrix} = C$$

This equality can be simplified into $\vec{\Lambda}_1 \times (A_1\vec{x} \leq \vec{b}_1) + \lambda(0 \leq 1) = C$ where λ is the third coefficient of $\vec{\Lambda}_3$. This shows that $\vec{\Lambda}_1$ is a certificate[3] for $P_1 \sqsubseteq C$. The same reasoning with $\sigma = [\alpha_1/0, \alpha_2/1, \vec{x}'_1/\vec{0}, \vec{x}'_2/\vec{x}]$ shows that $\vec{\Lambda}_2$ is a certificate for $P_2 \sqsubseteq C$.

5 Notes on the Implementation

The practical efficiency of the abstract domain operators is highly sensitive to implementation details. Let us thus describe our main design choices.

5.1 Extending to Equalities and Strict Inequalities

Everything we discussed so far deals with non-strict inequalities only. The inclusion test algorithm however complements such non-strict inequalities, which yields strict ones. Adaptation could have been restricted to the simplex algorithm on which the inclusion test relies, and such an enhancement is described in [9]. We have however elected to add full support for strict inequalities to our implementation. Once the addition of two constraints has been defined, almost no further change to the algorithms we discussed previously was needed.

Proper support and use of equalities was more involving. As [20] points out, equalities can be used for projecting variables. Such substitutions do not

[3]The shift λ of the bound is lost and will be computed again by our Coq-certified checker.

increase the number of constraints, contrary to Fourier-Motzkin elimination. We ended up splitting the constraint set into a set of equalities, each serving as the definition of a variable, and a set of inequalities in which these variables have been substituted by their definitions. Minimization (see §5.2) was augmented to look for implicit equalities in the set of inequalities. Last, testing inclusion of P in C was split into two phases: substituting in C the variables defined by the equalities of P and then using the simplex-based method described earlier without putting the equalities of P in, which reduces the problem size.

Inclusion certificates were adapted for equalities. If $P \sqsubseteq C$, with $C \stackrel{\text{def}}{=} \vec{a} \cdot \vec{x} = b$, cannot be proven using a linear combination of equalities, it is split as $\{\vec{a} \cdot \vec{x} \leq b, \vec{a} \cdot \vec{x} \geq b\}$ and P is proven to be included in each separately.

5.2 Minimization

The intersection $P_1 \widehat{\cap} P_2$ is a very simple operation. As §2 described, a naive implementation amounts to list concatenation. However, some constraints of P_1 may be redundant with constraints of P_2. Keeping redundant constraints leads to a quick growth of the representation sizes and thus of computation costs. In addition, one condition for the good operations of widening operators on polyhedra is that there should be no implicit equality in the system of inequalities and no redundant constraint [2].

It is therefore necessary to *minimize* the size of the representation of polyhedra, that is, removing all redundant constraints, and to have a system of equality constraints that exactly defines the affine span of the polyhedron. We call P_{min} the result of the minimization on P. The correctness of the result is preserved as long as P_{min} is an over-approximation of P, which means $P \sqsubseteq P_{\text{min}}$.

First, we check whether P has points in it using the simplex algorithm from §3.3. If P is empty, $\bot$ is returned as the minimal representation. The certificate is built from the witness of contradictory bounds returned by the simplex algorithm. It is a linear combination which result is a trivially contradictory constraint involving only constants (e.g. $0 \leq -1$) and which, in other words, has no solution.

The next step is implicit equality detection. It builds on $\vec{a} \cdot \vec{x} \leq b \wedge \vec{a} \cdot \vec{x} \geq b \Rightarrow \vec{a} \cdot \vec{x} = b$. For every $C^{\leq} \stackrel{\text{def}}{=} \vec{a} \cdot \vec{x} \leq b$ of P (by definition $P \sqsubseteq C^{\leq}$), we test whether $P \sqsubseteq C^{\geq} \stackrel{\text{def}}{=} \vec{a} \cdot \vec{x} \geq b$. If the inclusion holds, the certificate of the resulting equality is composed of a linear combination yielding $C^{\geq}$ and a trivial one, $1 \cdot C^{\leq}$, yielding $C^{\leq}$. Once this is done, the representation of P can be split into a system of equalities P_e and a system of inequalities P_i with no implicit equality. P_e is transformed to be in echelon form using Gaussian elimination, which has two benefits. First, redundant equations are detected and removed. Second, each equation can now serve as the definition of one variable. The so-defined variables are then substituted in P_i,

yielding P'_i. Although our implementation tracks evidence of the correctness of this process, it should be noted that the uses of equalities decribed above are standard practice.

At this point, if redundancy remains, it is to be found in P'_i only. It is detected using inclusion tests: for every $C \in P'_i$, if $P'_i \setminus \{C\} \sqsubseteq P'_i$, C is removed. Removing a constraint is, at worst, an over-approximation for which no justification needs to be provided.

All that we describe above involve many runs of the simplex algorithm. The key point which makes this viable in practice is the following: they are all strongly related and many pivoting steps are shared among the different queries. We described (§3.3) the data representation used by the simplex variant we use: it splits each constraint of P in linear term and bound by inserting new variables. These variables can have both an upper and a lower bound. Let us now illustrate the three steps of minimization on constraint $C \stackrel{\text{def}}{=} \vec{a} \cdot \vec{x} \leq b$, split as $x' = \vec{a} \cdot \vec{x}$ and $\vec{x}' \leq b$. The first step, satisfiability, solves this very problem. Then, implicit equalities detection checks whether $x' = \vec{a} \cdot \vec{x}$ and $x' < b$ is unsatisfiable. Last, redundancy elimination operates on $x' = \vec{a} \cdot \vec{x}$ and $x' > b$.

For all these problems, we only changed the bound on x', without ever touching either the constraint $x' = \vec{a} \cdot \vec{x}$ or the other constraints of P. These changes can be done dynamically, while preserving the simplex invariant (‡ of §3.3), by making sure that the affected x' is a basic variable. This remark, once generalized to a whole polyhedron, enables the factorization of the construction of the simplex problem. Actually, it is only done once for each minimization. It is also hoped that the feasible point of one problem is close enough to that of the next problem, so that convergence is quick.

Minimization also plays an important role in the convex hull algorithm. We mentioned (§4.2) that projection increases the representation size of polyhedra and described some simple counter-measures from [20]. When projecting a lot of variables, as is done for computing the convex hull of two polyhedra, each redundant constraint can trigger a lot of extra computation. Applying a complete minimization after the projection of each variable mitigates this. More precisely, only the third of the steps described above is used: projection cannot make a non-empty polyhedron empty and it cannot reduce the dimension of a polyhedron, no implicit equality can be created.

5.3 A More Detailed Intuition on Bookkeeping

We mentioned in §3.2 and §4 that simple bookkeeping makes it possible to build inclusion certificates. We now give a more precise insight on what is involved, on the example of the projection.

The main change is an extension of the notion of constraint, which is now a pair (f, C) of a certificate fragment and a linear constraint as we presented them so far. A certificate fragment f is a list of pairs (n_i, id_i), n_i

being a rational coefficient and id_i a natural number uniquely identifying one constraint of P. The meaning of f is the following

$$\sum_i n_i \cdot C_{id_i} = c, \text{ with } C_{id_i} \in P \text{ and } (n_i, id_i) \in f$$

The $elim_{\vec{x}_k}$ function introduced in §4.2 is extended to take two extended constraints (f_1, C_1) and (f_2, C_2), and return an extended constraint (f, C). Recall that the original $elim_{\vec{x}_k}$ chooses λ_1 and λ_2 such that the coefficient of $\vec{x}_k$ in the resulting C is nil. The extended version returns $(\lambda_1 \cdot f_1 \mathbin{@} \lambda_2 \cdot f_2, \lambda_1 \cdot C_1 + \lambda_2 \cdot C_2)$, where @ is the list concatenation operator and $\lambda_i \cdot f_i$ is a notation for:

$$map\ (fun\ (n, id) \to (\lambda_i \cdot n, id))\ f_i$$

The certificate fragment keeps track of how a constraint was generated from an initial set of constraints. For a single projection $proj\ P\ \vec{x}_k$, the fragments are initialized as $[(1, id_C)]$ for every constraint C before the actual projection starts. For a series of projection as done for the convex hull, the initialization takes place before the first projection.

5.4 Polyhedron Representation Invariants

The data representation our implementation uses for polyhedra satisfies a number of invariants which relate to minimality.

(1) There is no implicit equality among the inequalities.

(2) There is no redundant constraint, equality or inequality.

(3) In a given constraint, factors common to all the coefficients of variables are removed.

(4) Each equality provides a definition for one variable, which is then substituted in the inequalities.

(5) Empty polyhedra are explicitly labeled as such.

(3) helps keeping numbers small, hopefully fitting machine representation, resulting in cheaper arithmetic. (1) implies in particular that if an implicit equality is created when adding a constraint C to a polyhedron P, then C is necessarily involved in that equality. It follows that the search for implicit equalities can be restricted to those involving newly added constraints. Because of (2), the same holds for redundancy elimination: if C is shown to be redundant, P remains unaffected by the intersection. Furthermore, (4) allows for the reduction of the problem dimension when testing for $P_1 \sqsubseteq P_2$. Once the same variables are substituted in P_1 and P_2, only the inequalities need to be inserted in the simplex problem. Last, (3) and (4) give a canonical

form to constraints, which make syntactic criteria for deciding inclusion of constraints more powerful. These criteria, suggested by [20], are used whenever possible in the inclusion test and the projection.

5.5 Data Structures

5.5.1 Radix Trees

Capturing linear relations between program variables with polyhedra generally leads to sparse systems, as noted by [20]. Our implementation uses a tree representation of vectors[4] where the path from the root to a node identifies the variable whose coefficient is stored at that node. This offers a middle ground between dense representation, as used by other widely-used implementation of the abstraction domain of polyhedra, and sparse representation which makes random access costly as sparsity diminishes.

5.5.2 Numbers Representation

Rational vector coefficients can grow so as to overflow native integer representation during an analysis. Working around this shortcoming requires the use of an arithmetic library for arbitrarily large numbers. This has a serious impact on overall performance. Our implementation uses the ZArith[14] Ocaml front-end to GMP[22]. ZArith tries to lower the cost of using GMP by using native integers as long as they don't overflow.

Our experiments show that, in many practical cases, extended precision arithmetic is not used. This echoes similar findings in SMT-solvers such as Z3 or OpenSMT [19]: in most cases, extended precision is not used, thus the great importance of an arithmetic library that operates on machine words as much as possible, without allocating extended precision numbers. In the case of polyhedra, however, the situation occasionally degenerates when the convex hull operator generates large coefficients.

The extracted Ocaml code of `inclusion_checker` does not use this efficient representation. Because of the need for correctness of computations, the checker instead uses the Coq representation of numbers (lists of bits), which is inefficient on numerical computations. Alternatively, assuming trust in ZArith and GMP, it is possible to configure the Coq extractor to base the checker on ZArith.

5.6 A Posteriori Certification vs. Full COQ-Certified Development

Even though our library is planned to be used in a Coq-certified analyzer, we preferred *a posteriori* certification over a fully Coq-certified development.

[4] The idea was borrowed from [4].

Keeping Coq only for the development of checkers of external computations reduces the development cost and reconciles efficiency of the tool and confidence in its implementation through certificates.

First, it reduces the proof effort: verifying that a guess is the solution to a problem involves weaker mathematic arguments than proving correctness and termination of the solver. To illustrate the simplicity of our Coq development, Figure 1 shows some excerpts which are self-explanatory. The last function, `inclusion_checker`, is representative of the difficulty of the proofs. This function is close to its extraction in Ocaml except that it returns either an error or a proof of $P_1 \sqsubseteq P_2$ wrapped in the value constructor (Line 38). In the case where P_1 is an empty polyhedron (established by *eproof*) the proof of inclusion in P_2 is built from that proof of emptiness. The missing proof of Line 38 is done in the interactive prover (Lines 43-45) and automatically placed in the function. It consists in an induction on the list of constraints of P_2 that shows that the empty polyhedron P_1 is included in every constraint of P_2.

Our external library acts as an oracle: it efficiently performs the operation and returns a certificate which serves two purposes: it can be used to check the correctness of the computations but it is also a short cut toward the result. For instance, the convex hull $P_1 \sqcup P_2$ is easy to obtain from the complete inclusion certificates $(F_1, \vec{\lambda_1})$ related to P_1 or $(F_2, \vec{\lambda_2})$ related to P_2. Indeed, $P_1 \sqcup P_2 = F_1 \cdot P_1 + \vec{\lambda_1} = F_2 \cdot P_2 + \vec{\lambda_2}$ (see §4.3). This way, the expensive computations that involve numerous calls to our simplex algorithm are done by our Ocaml implementation using ZArith and the result is reflected in Coq at the cost of just a matrix product using the Coq-certified representation of numbers. If we work in such a manner, we never actually have to transfer polyhedra from the untrusted to the trusted side.

From a general point of view, splitting a tool into an untrusted solver and a correctness checker makes it more amenable to extensions and optimizations. *A posteriori* certification has a cost each time the correctness of a result needs to be proved (only during the last phase of the analysis to ascertain the stability of the inferred properties). However, it allows optimizations whose correctness would be difficult to prove and usage of untrusted components (e.g. GMP).

6 Experimental Results

In order to evaluate the viability of our solution, we compared experimentally our library (referred to as Libpoly) with mature implementations.

In addition to the efficiency of the polyhedra computation, we wished to measure the cost of the inclusion checker. Our approach guarantees that, if our certificate checker terminates successfully on a given verification, the result of the operation which produced the certificate is correct. However,

```
From module LinearCsrt:
Record LinearCstr: Set := mk {coefs: Vec; cmp_op: Cmp; bound: Num}.

Definition Sat (c:LinearCstr) (x:Vec) : Prop :=
  denote  (Vec.eval (coefs c) x) (cmp_op c) (bound c).

From module List:
Inductive Forall (A : Type) (pred : A → Prop) : list A → Prop :=
  | FORALL_NIL: Forall pred nil
  | FORALL_CONS: ∀ (x:A) (l:list A),
                   pred x → Forall pred l → Forall pred (x :: l)

From module Polyhedra:
Definition Polyhedra : Set := list (id * LinearCstr).

Definition Sat (P:Polyhedra) (x:Vec) : Prop :=
  List.Forall (fun c => LinearCstr.Sat (snd c) x) P.

Definition Incl (P:Polyhedra) (C:LinearCstr) : Prop :=
  ∀ x:Vec, Sat P x → LinearCstr.Sat C x.

Definition (infix ⊑) (P1 P2 : Polyhedra) : Prop :=
  ∀ x:Vec, Sat P1 x → Sat P2 x.

Definition CertOneConstraint : Set := list (id * Num)

Inductive Cert : Set :=
 | INCL: list (id * CertOneConstraint) -> Cert
 | EMPTY: CertOneConstraint -> Cert.

Lemma Empty_is_included: ∀ (P:Polyhedra) (C:LinearCstr),
  (Empty P) → (Incl P C).

Definition inclusion_checker (P1 P2:Polyhedra) (cert:Cert) : Exc(P1⊑P2).
refine ( match cert with
         | INCL icert => checkInclusion P1 P2 icert
         | EMPTY ecert => match (checkEmptyness P1 ecert) with
                          | VALUE eproof => VALUE _ ← missing proof
                          | ERROR => ERROR
                          end
         end
). The missing proof is provided by the following proof script:
induction P2 with IH;
 exact (List.FORALL_NIL _ _) ;
 exact (List.FORALL_CONS _ _ c _ (Empty_is_included P1 (snd c) eproof) IH).
Defined.
```

Figure 1: Excerpts of our Coq-certified inclusion checker

this assertion currently only applies to the polyhedra as known to the Coq checker: a translation occurs between the OCaml representation of numbers, ZArith, and their representation in the Coq language as lists of bits. This means that the checker has to compute on this inefficient representation, and thus we wished to ascertain whether the cost was tolerable.[5]

The best approach to evaluating Libpoly would have been to rely on it for building a complete static analyzer. Although this is our long-term goal, a less demanding method was needed for a more immediate evaluation. We chose to compare computation results from Libpoly to those of widely used existing implementations of the abstract domain of polyhedra: the NewPolka library and the PPL. More precisely, we used them through their Apron front end [11].

6.1 The Method

As [16] points out, randomly-generated polyhedra do not give a faithful evaluation: a more realistic approach was needed. Because of the lack of a static analyzer supporting both Apron and Libpoly, we carried out the comparison by logging and then replaying with Libpoly the abstract domain operations done by the existing Pagai analyzer [10] using Apron.

Technically, logging consists in intercepting calls to the Apron shared library (using the wrap functionality of the GNU linker `ld`), analyzing the data structures passed as operands and generating equivalent OCaml code for Libpoly. NewPolka and PPL results are logged too, for comparison purposes. At the end of the analysis, the generated OCaml code forms a complete program which replays all the abstract domain operations executed by the NewPolka library or the PPL on request of the analyzer.

The comparison was done for the following operations: parallel assignment, convex hull, inclusion test and intersection on the analysis of the following programs:

1. `bf`: the Blowfish cryptographic cipher
2. `bz2`: the bzip2 compression algorithm
3. `dbz2`: the bzip2 decompress algorithm
4. `jpg`: an implementation of the jpeg codec

[5] An alternative would be to map, at checker extraction time, Coq numbers to ZArith numbers, at the expense of having both ZArith and GMP in the trusted computing base. One may consider that we already make assumptions about ZArith and GMP: we assume they respect memory safety, and thus will not corrupt the data of the OCaml code extracted from Coq, or at least that, if they corrupt memory, they will cause a crash in the analyzer (probably in the garbage collector) instead of a silent execution with incorrect data. This seems a much less bold assumption than considering that they always compute correctly, including in all corner cases.

5. `re`: the regular expression engine of GNU `awk`

6. `foo`: a hand-crafted program leading to polyhedra with many constraints, large coefficients and few equalities

6.2 Precision and Representation Size Comparison

The result of each operator we evaluated is a well-defined geometrical object. For every logged call, the results from NewPolka, PPL and Libpoly were checked for equality (double inclusion). The certificates generated by Libpoly were then systematically checked. Furthermore, polyhedra have a minimal constraints representation, up to the variable choices in the substitutions of equalities. It was systematically checked whether Libpoly, NewPolka and the PPL computed the same number of equalities and inequalities. In all the cases we tried, the tests of correctness and precision passed. It is to be noted that the PPL does not systematically minimize representations: its results often have redundant constraints.[6]

Besides giving confidence in the results computed by Libpoly, ensuring that our results are identical to those of NewPolka or the PPL lead us to believe that the analyzer behavior would not have been very different, had it used the results from Libpoly. There is no noticeable difference between the analyses carried out using NewPolka and the PPL.

6.3 Timing Measurements

Timing measurements were made difficult because of the importance of the state of polyhedra in the double representation NewPolka and the PPL use. We were concerned that logging and replaying as described above would be unfair towards these libraries, since it would force the systematic recomputation of generator representations that, in a real analyzer, would be kept internally. We thus opted for a different approach.

We measured the timings for NewPolka and the PPL directly inside Pagai by wrapping the function calls between calls to a high precision timer. We made sure that the overhead of the timer system calls was sufficiently small so as to produce meaningful results. For Libpoly, timing measurements were done during the replay and exclude the time needed to parse and rebuild the operand polyhedra.

We present two views of the same timing measurements, carried out on the programs introduced in §6.1. Table 1 gives, for each benchmark program, the total time spent in each operation of the abstract domain. Such a table does not inform us of the typical distribution of problem sizes

[6] This is due to the lazy-by-default implementation of the operators of the PPL. Since support for the eager version of the operators has been deprecated in and is being removed from the PPL (see [23], § A Note on the Implementation of the Operators), we could not configure the library to have the same behavior as NewPolka.

Table 1: Timing comparison between NewPolka (N), PPL (P), Libpoly (L) and Libpoly with certificate checker (C): total time (in milliseconds) spent in each of the operations; trivial problems are excluded.

prog.	assignment			convex hull				inclusion				intersection		
	N	P	L	N	P	L	C	N	P	L	C	N	P	L
`bf`	3.7	11.4	0.5	3.2	1.2	2.7	2.8	0.2	0.4	0.1	0.1	10.7	13.4	1.2
`bz2`	14.6	54.1	2.9	23.5	11.5	66.8	68.7	1.6	2.8	0.7	1.2	52.3	61.1	7.9
`dbz2`	1618	4182	83.8	1393	231.9	532.8	535.3	32.3	35.6	2.1	3.6	1687	1815	28.3
`jpg`	23.7	68.3	3.8	28.2	7.5	24.0	24.9	1.2	1.8	0.5	0.8	39.7	51.0	6.0
`re`	5.7	17.2	0.7	20.2	8.4	17.9	19.2	1.1	1.3	0.5	0.7	37.3	47.2	3.3
`foo`	9.2	14.8	8.5	4.2	0.6	941.8	943.7	0.2	0.2	0.9	0.9	6.7	7.1	5.5

and the relationship between problem size and computation time, thus we compiled Table 2 which shows computation times aggregated according to the "problem size", defined as the sum of the number of constraints of all the operands of a given operation.

For the assignment and the convex hull, all the constraints of the two operands are put together after renaming and many projections follow. The inclusion test $P_1 \mathbin{\widehat{\sqsubseteq}} P_2$, in the worst case, solves as many linear programming problems as there are constraints in P_2, but each is of size the number of constraints of $P_1 + 1$. Last, the intersection operator minimizes the result of the union of the sets of constraints. Note that the sums in Table 1 exclude operations on trivial problems of size zero or one.

The presented results show that Libpoly is efficient on small problems. Yet, the performance gap between Libpoly and the other implementations closes on bigger problems. This is especially true for the convex hull, which is a costly operation in the constraint representation. At least part of the difference in efficiency on small problems can be explained by the generality Apron provides: it provides a unified interface to several abstract domains at the expense of an extra abstraction layer which introduces a significant overhead on small problems.

More generally, the use of ZArith in Libpoly is likely to lower the cost of arithmetic when compared to NewPolka and the PPL, which use GMP directly. The `foo` program illustrates this: the analysis creates constraints with big coefficients, likely to overflow native number representation. However, precise measurement of the effect of using ZArith would be a hard task.

Last, Table 1 seems to show that problems are most often of rather small size, but this may well be due to our limited experimentation means.

In spite of the shortcomings of our evaluation method, these results seem promising for a constraints-only implementation of the abstract domain of polyhedra. Some progress still needs to be made on the convex hull side (see §7). It is also interesting to notice the performance differences between the NewPolka and the PPL, despite their design similarities; we ignore their cause.

Table 2: Timing comparison between NewPolka (N), PPL (P) and Libpoly (L). Computation times (in milliseconds) are aggregated according to operation and problem size. (n) is the total number of problems of the size range in the benckmarks.

problem size		0–1	2–5	6–10	11–15	16–20	21–25	26–30	31+
assignment	N	33.8	601.8	385.4	20.9	78.3	537.4	59.5	13.1
	P	47.5	1176	519.7	87.4	247.6	2111	81.7	77.9
	L	1.1	6.6	14.3	10.7	5.2	39.2	15.2	11.6
	n	*539*	*667*	*381*	*58*	*64*	*480*	*30*	*16*
convex hull	N	687.9	679.7	434.1	119.5	68.8	37.9	6.4	3.5
	P	167.5	141.0	68.4	22.8	16.8	9.2	1.9	0.9
	L	7.0	57.1	133.7	131.2	1050	106.4	50.1	27.8
	n	*3354*	*3373*	*1092*	*354*	*135*	*65*	*14*	*7*
inclusion	N	7.2	9.7	9.7	3.3	5.8	4.0	4.0	0
	P	6.5	12.8	10.6	4.2	7.0	3.9	3.4	0
	L	0.6	1.6	1.3	0.5	1.0	0.3	0.1	0
	n	*1482*	*1881*	*673*	*277*	*111*	*52*	*17*	*4*
intersection	N	1389	1752	52.3	27.4	1.3			
	P	1933	1740	158.6	91.4	4.8			
	L	35.0	30.9	18.4	8.8	0.6			
	n	*11458*	*4094*	*322*	*156*	*6*	*0*	*0*	*0*

6.4 Certificate Checking Overhead

The certificate checking overhead shown in Table 1 includes the translation between OCaml and Coq representations. Inside a certified static analyzer, this overhead could be reduced by only transferring the certificates, as opposed to the full polyhedra, and using them to simulate the polyhedra computations, without bothering to check after every call that the polyhedron inside the OCaml library corresponds to the one inside the certified checker. In addition to translation costs, there is the general inefficiency of computations on Coq integers, which are represented as lists of bits; this is considerably more expensive than using native integers, or even arrays of native integers as GMP would do.

However, it should be noted that the checking of inclusion certificates occurs only during the final step of the certified static analysis which consists in verifying that the inferred invariant candidates are indeed inductive invariants for the program.

Last, the overhead of certificate checking is relatively greater for inclusion than for convex hull. Although the actual checking burden is bigger for the convex hull, due to certificate composition densifying the resulting certificate, the inclusion test algorithm is much cheaper than the convex hull in terms of computations. More precisely, the convex hull algorithm involves inclusion tests as part of representation minimization.

7 Conclusions

The previous sections demonstrated that a realistic implementation of the abstract domain of polyhedra can be certified using a posteriori verification of results. This approach has a key benefit: the time-consuming development inside the Coq proof assistant is reduced to the bare minimum. A tight integration of the certification concern enables on-the-fly certification generation as a by-production of the actual computations, thereby making the associated cost negligible. The same procedures can be used for fixed point iterations (with certificate generation turned off for efficiency) and for fixed point verification (with certificates generated and checked).

The complete implementation which has been developed operates only on a constraints representation of polyhedra; our motivations for this choice were the ease of generation of certificates as well as the absence of combinatorial explosion on common cases such as hypercubes. This is made possible through careful choice of data structures and exploitation of recent algorithmic refinements [20, 9]. Possible future developments include designing efficient techniques for generating Farkas certificates for a library based on the double representation (generators and constraints) and providing heuristics for choosing when to operate over constraints only and when to use the double representation.

Prior to this, however, there remains room for both enhancement and extension of our current implementation. A simple enhancement would be to have both an upper and a lower bound for linear terms, which would further condense the representation of polyhedra. The implicit equality detection algorithm could be made less naive by exploiting the fact that a point in a polyhedron P which has implicit equalities E_i necessarily reaches the bounds of the inequalities involved in the proof of $P \sqsubseteq E_i$.

Finally, our library is planned to be part of a certified static analyzer, such as the one being built in the Verasco project. Beyond a certified implementation of the abstract domain of polyhedra, our library could also serve to verify the numerical invariants discovered by untrusted analysis using a combination of abstract domains (intervals, octagons, ... which are special cases of polyhedra). The discovered invariants could be stored in the form of polyhedra and the verification of their stability could be done with our certified library. Currently, our polyhedron library only deals with linear constraints, but a general-purpose analyzer has to handle nonlinearity. Our library should therefore include linearization techniques [13] at the condition that these be proven correct.

Acknowledgements We would like to thank Bertrand Jeannet for his advice on proper ways to evaluate Libpoly against his NewPolka library.

References

[1] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. "The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems". In: *Science of Computer Programming* 72.1–2 (2008), pp. 3–21. arXiv: cs/0612085.

[2] Roberto Bagnara et al. "Precise Widening Operators for Convex Polyhedra". In: *Science of Computer Programming* 58.1–2 (Oct. 2005), pp. 28–56. DOI: 10.1016/j.scico.2005.02.003.

[3] Florence Benoy, Andy King, and Frédéric Mesnard. "Computing convex hulls with a linear solver". In: *Theory and Practice of Logic Programming* 5.1-2 (2005), pp. 259–271. DOI: 10.1017/S1471068404002261. arXiv: cs/0311002.

[4] Frédéric Besson et al. *Result certification for relational program analysis*. Anglais. Tech. rep. RR-6333. INRIA, 2007. HAL: inria-00166930.

[5] Bruno Blanchet et al. "A Static Analyzer for Large Safety-Critical Software". In: *Programming Language Design and Implementation (PLDI)*. ACM. 2003, pp. 196–207. HAL: hal-00128135.

[6] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Principles of Programming Languages (POPL)*. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[7] Patrick Cousot and Nicolas Halbwachs. "Automatic discovery of linear restraints among variables of a program". In: *Principles of Programming Languages (POPL)*. ACM, 1978, pp. 84–97. DOI: 10.1145/512760.512770.

[8] George Dantzig and Mukund Narain-Dhami Thapa. *Linear Programming*. Springer, 2003. ISBN: 9780387986135.

[9] Bruno Dutertre and Leonardo De Moura. *Integrating simplex with DPLL(T)*. Tech. rep. SRI-CSL-06-01. SRI International, computer science laboratory, 2006.

[10] Julien Henry, David Monniaux, and Matthieu Moy. "PAGAI: a path sensitive static analyser". In: *Tools for Automatic Program Analysis (TAPAS)*. Ed. by Bertrand Jeannet. 2012. HAL: hal-0071843.

[11] Bertrand Jeannet and Antoine Miné. "Apron: A Library of Numerical Abstract Domains for Static Analysis". In: *Computer-aided verification (CAV)*. 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52.

[12] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814. HAL: inria-00415861.

[13] Antoine Miné. "Symbolic methods to enhance the precision of numerical abstract domains". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Vol. 3855. LNCS. Springer, 2006, pp. 348–363. HAL: hal-00136661.

[14] Antoine Miné and Xavier Leroy. *ZArith*. `http://forge.ocamlcore.org/projects/zarith`.

[15] David Monniaux. "A minimalistic look at widening operators". In: *Higher Order and Symbolic Computation* 22.2 (Dec. 2009), pp. 145–154. DOI: 10.1007/s10990-009-9046-8. HAL: hal-00363204.

[16] David Monniaux. "On using floating-point computations to help an exact linear arithmetic decision procedure". In: *Computer-aided verification (CAV)*. Vol. 5643. LNCS. Springer, 2009, pp. 570–583. DOI: 10.1007/978-3-642-02658-4_42. HAL: hal-00354112.

[17] David Monniaux. "Quantifier elimination by lazy model enumeration". In: *Computer-aided verification (CAV)*. Vol. 6174. LNCS. Springer, 2010, pp. 585–599. ISBN: 3642142958. DOI: 10.1007/978-3-642-14295-6_51. HAL: hal-00472831.

[18] George C. Necula and Peter Lee. "Proof Generation in the Touchstone Theorem Prover". In: *Conference on Automated Deduction (CASE)*. Vol. 1831. LNAI. Springer, 2000. DOI: 10.1007/10721959_3.

[19] Diego Caminha Barbosa de Oliveira and David Monniaux. "Experiments on the feasibility of using a floating-point simplex in an SMT solver". In: *Workshop on Practical Aspects of Automated Reasoning (PAAR)*. CEUR Workshop Proceedings, 2012.

[20] Axel Simon and Andy King. "Exploiting Sparsity in Polyhedral Analysis". In: *Static Analysis Symposium (SAS)*. Vol. 3672. LNCS. Springer, 2005, pp. 336–351. DOI: 10.1007/11547662_23.

[21] The Coq Development Team. *The Coq proof assistant reference manual*. 8.4. INRIA. 2012.

[22] *The GNU Multiple Precision Arithmetic Library*. 5.0. Free Software Foundation. 2012. URL: `http://gmplib.org`.

[23] *The Parma Polyhedra Library*. 1.0. Bugseng. 2012. URL: `http://bugseng.com/products/ppl/documentation/user/ppl-user-1.0-html/index.html`.