Three linearization techniques for multivariate polynomials in static analysis using convex polyhedra

Alexandre Maréchal, Michaël Périn

Verimag Research Report n^o TR-2014-7

July 2014

Reports are downloadable at the following address
http://www-verimag.imag.fr

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UJF
Centre Équation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Three linearization techniques for multivariate polynomials in static analysis using convex polyhedra

Alexandre Maréchal, Michaël Périn

July 2014

Abstract

We present three linearization methods to over-approximate non-linear multivariate polynomials with convex polyhedra. The first one is based on the substitution of some variables by intervals. The principle of the second linearization technique is to express polynomials in the Bernstein basis and deduce a polyhedron from the Bernstein coefficients. The last method is based on Handelman's theorem and consists in using products of constraints of a starting polyhedron to over-approximate a polynomial. As a part of the VERASCO project, the goal is to prove such methods with the proof assistant Coq.

Keywords: polyhedra abstract domain, linearization

Reviewers: Bertrand Jeannet, Alexis Fouilhé

How to cite this report:

```
@techreport {TR-2014-7,
  title = {Three linearization techniques for multivariate polynomials in static analysis using
convex polyhedra},
  author = {Alexandre Maréchal, Michaël Périn},
  institution = {{Verimag} Research Report},
  number = {TR-2014-7},
  year = {}
}
```

Contents

1 Toward Certification of a C compiler

1.1 The VERASCO project

The CompCert project (2006-2010) from Inria Paris-Rocquencourt and Rennes consisted in the formal verification of a C compiler. The goal was to avoid miscompilation which is the production of an executable that does not match the source code. The project led to CompCert, the first C compiler certified using the proof assistant Coq. This compiler is proved to produce a correct executable code if the corresponding source code can not lead to a runtime error. Thus, the correctness of the executable code depends on an assumption on the source code. The VERASCO project follows CompCert and gathers Inria Paris-Rocquencourt, Inria Saclay, VERIMAG laboratory, Airbus and the University of Rennes. VERASCO aims at developing a certified static analyzer capable of proving the absence of error in the source code, hence discarding the CompCert assumption. The principle of verification based on static analysis is to compute an over-approximation of all possible reachable states of the program, examining only the source code, then to check that no error state is reachable. As every program, such a verification tool is not protected from a failure and a bug in the analyzer can make it miss errors. That's why the correctness of the analyzer must be certified in Coq too.

1.2 VERIMAG's contributions

1.2.1 Static analysis by abstract interpretation

The VERIMAG laboratory performs research about program verification using static analysis by abstract interpretation [2]. Static analysis differs from test since, first, the source code is not actually executed, and second, abstract interpretation ensures that no reachable state is omitted. It uses abstract domains (such as intervals, polyhedra, etc.) instead of usual variables to approximate the sets of program states. The effect of each instruction is captured by symbolic computations using abstract domains. Static analysis catches the (potentially infinite) set of possible behaviours of the source code at the price of approximating the reachable states. As a result, some states considered during the analysis are not reachable in practice. When precision is needed, programs can be analyzed using the domain of convex polyhedra which is able to reason about linear relations between program variables.

1.2.2 A certified library for convex polyhedra

A convex polyhedron[1] is defined as a conjunction of linear constraints of the form $\sum_{i=1}^{n} \lambda_i x_i \leq c$ where $\forall i \in \{1, ..., n\}, x_i$ is a variable, $\lambda_i \in \mathbb{Q}$ and $c \in \mathbb{Q}$ are constants. For instance, the polyhedron P_1 defined by the system $\{x \geq 1, y \geq -2, x - y \geq 0, x + y \leq 5\}$ defines a geometrical space represented in the plane (Figure 1). A polyhedron which is fully bounded is called a *polytope*.

Polyhedra are used in static analyzers to automatically discover linear relations between variables of the source code. Those relations help deducing necessary properties to prove the correctness of the program. A Coq library was recently created by Fouilhé [5], allowing safe handling of polyhedra operators such as intersection and convex hull, that is the smallest polyhedron containing the union of P_1 and P_2.

Polyhedra suffer from an ontological limitation: they cannot deal with non-linear relations, *i.e.* expressions containing products of variables. Hence, the analyzer cannot exploit the information on a variable z when encountering a non-linear assignment $z := x * y$. Moreover, after such an assignment, all the previously determined linear constraints containing z do not hold anymore. That's why a linearization technique that approximates $x * y$ by a linear expression is necessary to avoid dramatic loss of precision during the analysis.

1.3 Linearization

The goal of linearization is to approximate non-linear relations by linear ones. In this work we do not consider expressions formed of non-algebraic functions like $sin, log, ...$[2] Our linearization methods only

[1] We only deal with convex polyhedra. For readability, we will omit the adjective convex in the following.

[2] We could in principle treat analytic functions by considering their Taylor polynomials.

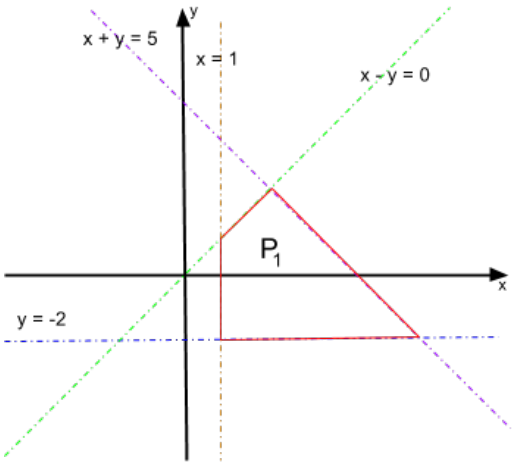


Figure 1 – Graphical representation of $P_1 : \{x \geq 1, y \geq -2, x - y \geq 0, x + y \leq 5\}$

```
1  int x,y,z;
2  if(x >= 1 && y >= -2 && x >= y && x <= 5 - y)
3  {
4          if(x*x + y*y <= 4)
5                  z = y*x;
6          else
7                  z = 0;
8  }
```

Example 2 – C program containing two non-linear expressions $x * x + y * y \leq 4$ *and* $y * x$

address polynomial expressions containing products of variables. Thus, in this paper f represents a polynomial expression on the variables of the program $(x_1, \ldots, x_n)$. The lines 4 and 5 of Example 2 contain such non-linear relations.

Let us explain how the effect of a guard or an assignment on a polyhedron is computed, and how the linearization process is involved. Without loss of generality, we focus in this paper on the treatment of a guard $f \geq 0$ where f denotes a polynomial in the $(x_1, \ldots, x_n)$ variables of the program. For instance the guard $x^2 + y^2 \leq 4$ corresponds to the case $f(x, y) \triangleq 4 - x^2 - y^2$.

The effect of an assignment $x := f$ **on a polyhedron** P corresponds to the intersection of P with a polyhedron G formed of two inequalities $\tilde{x} - f \geq 0 \wedge f - \tilde{x} \geq 0$ (encoding the equality $\tilde{x} = f$). This computation uses a fresh variable $\tilde{x}$ that is renamed in x after elimination of the old value of x by projection. We denote by $P_{/x}$ the polyhedron P where the variable x has been projected. Formally, the effect of $x := f$ on P is the polyhedron $\left((P \sqcap G)_{/x}\right)[\tilde{x}/x]$

The effect of a guard $f \geq 0$ **on a polyhedron** P consists in the intersection of the points of P with the set of points $(x_1, \ldots, x_n)$ satisfying the condition $f(x_1, \ldots, x_n) \geq 0$. This intersection is not necessarily a polyhedron. Let us denote by $\mathscr{P}$ the set of points after the guard, *i.e.* $\mathscr{P} = P \cap \{(x_1, \ldots, x_n) \mid f(x_1, \ldots, x_n) \geq 0\} = \{(x_1, \ldots, x_n) \in P \mid f(x_1, \ldots, x_n) \geq 0\}$.

When the guard is linear, say $x - y \geq 0$, we simply add the constraint $x - y \geq 0$ to P and obtain a polyhedron. With the polyhedron P_1 from Figure 1, we obtain the polyhedron $\mathscr{P} = P_1 \wedge (x - y \geq 0) = \{x \geq 1, y \geq -2, x - y \geq 0, x + y \leq 5,\ x \leq y\}$.

When the guard is not linear, finding its effect on P is done by computing a convex polyhedron P' such

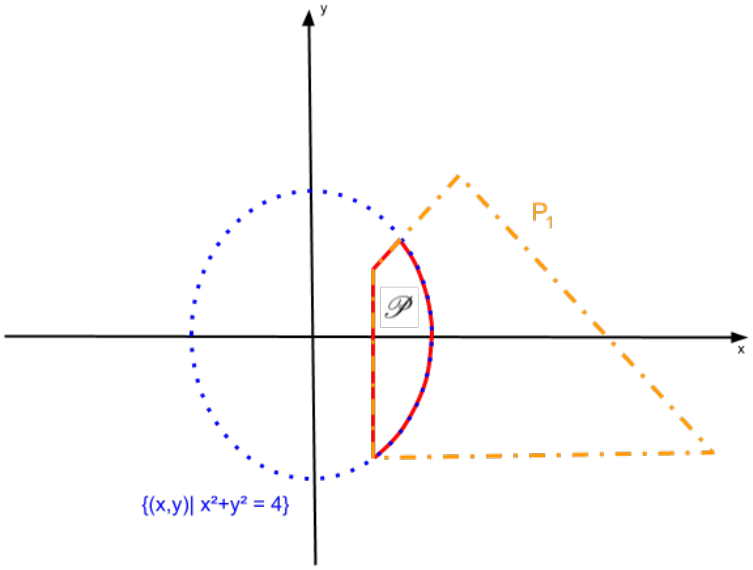


Figure 3 – Graphical representations of $P_1 \triangleq \{x \geq 1, y \geq -2, x - y \geq 0, x + y \leq 5\}$ *in orange and the guard* $\mathscr{G} \triangleq \{(x, y) \mid x^2 + y^2 \leq 4\}$ *in blue. The set* $\mathscr{P} = P_1 \cap \mathscr{G}$ *is represented in red.*

that $\mathscr{P} \subseteq P'$. There is no unique polyhedron P': any choice of P' satisfying $\mathscr{P} \subseteq P'$ would give a correct approximation.

As an example, the effect of the non-linear guard $\mathscr{G} \triangleq x^2 + y^2 \leq 4$ on the polyhedron P_1 of Figure 1 is represented on Figures 3 and 4. The green polyhedron G of Figure 4 is a linear approximation of $\mathscr{G}$ by an octagon. We obtain the red polyhedron P' shown by computing the intersection of polyhedra $P_1 \cap G$.

Shape of real life non-linear expressions We tested the three techniques on statements taken from the benchmarks *debie1*, based on a satellite control software, and *papabench* which is a flight control code. In general, polynomials of such programs contain less than four variables and their power rarely exceed two. Indeed, most non-linear expressions appear in computation of Euclidian distances, that's why we encounter square roots as well. As a consequence, the exponential complexity of some algorithms is manageable.

Overview of the paper Three linearization methods are presented and compared in this paper. The first one uses the principle of variable intervalization introduced by Miné [11], it is discussed in Section 2. This algorithm has been implemented and proven in Coq. The second linearization method consists in representing the polynomial f in the Bernstein basis which allows to deduce a bounding polyhedron from its Bernstein coefficients. This is the subject of Section 3. Section 4 explores a new linearization method based on Handelman's theorem. Given a starting polyhedron P, Handelman's method consists in using products of constraints of P to obtain a linear over-approximation of the polynomial constraint $f \geq 0$. These three methods have been implemented in SAGE (a free open-source mathematics software system similar to maple or mathematica) in order to compare them in terms of precision.

2 Linearization by variable intervalization

In this chapter, we present a linearization method called intervalization [11]. The main idea is to replace some variables by intervals in order to remove products of variables. We shall see how to eliminate these intervals to get into linear expressions over $\mathbb{Z}$. We treat only the case where scalar values are integers and where expressions are composed of additions and multiplications.

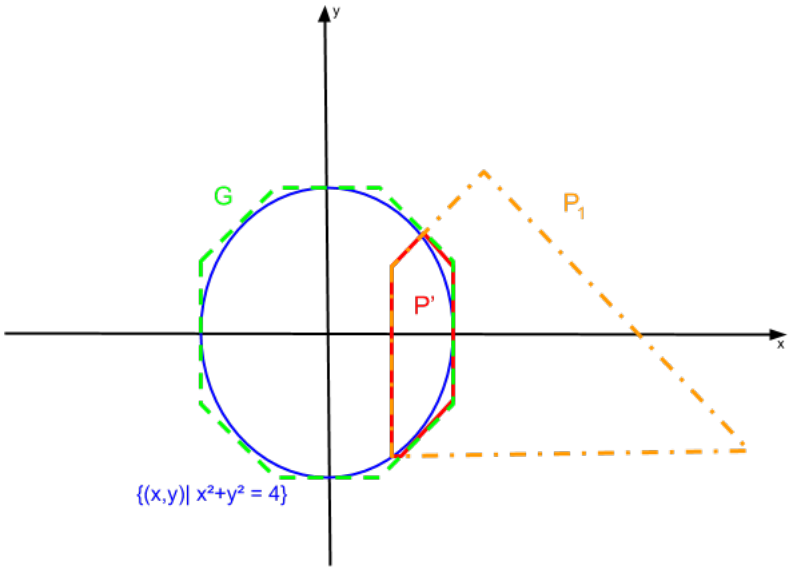


Figure 4 – Graphical representations of $P_1 \triangleq \{x \geq 1, y \geq -2, x - y \geq 0, x + y \leq 5\}$ *in orange and the guard* $\mathscr{G} \triangleq \{(x, y) \mid x^2 + y^2 \leq 4\}$ *in blue. A linear over-approximation* G *of* $\mathscr{G}$ *is drawn in green. The approximation* P' *of* $P \cap \mathscr{G}$ *is represented in red.*

2.1 Principle of the intervalization algorithm

2.1.1 Variable intervalization

Variable bounds can be extracted from a polyhedron, in the sense that each program variable is associated with the interval of all the values it can take at a particular program point[3]. A static analyzer running on the program of Example 1 associates line 2 with the polyhedron $S \triangleq \{x \geq 1, y \geq -2, x - y \geq 0, x + y \leq 5\}$. This polyhedron allows to deduce $x \in [1, 7]$ and $y \in [-2, 3]$.

The linearization method of Miné [11] named intervalization consists in replacing some variables with their corresponding intervals. The idea is to use the known information (the intervals) to over-approximate the behaviours engendered by the non-linear relation. We say that we intervalize x if we choose to replace the variable x with its interval.

Example 1. Consider the following program.

```
1  int x,y,z;
2  if(x >= 1 && y >= -2 && x >= y && x <= 5 - y)
3  {
4          if(x*x + y*y <= 4)
5                  z = y*x;
6          else
7                  z = 0;
8  }
```

If we intervalize $x \in [1, 7]$ at line 4 and 5, and we do the same at line 4 with one occurrence of $y \in [-2, 3]$, we obtain the following program.

[3] Broadly speaking, program points correspond to line numbering. We use this simplification to make easier the reading of examples.

```
1  int x,y,z;
2  if(x <= 1 && y >= -2 && x >= y && x <= 5 - y)
3  {
4          if(x*[1,7] + y*[-2,3] <= 4)
5                  z = y*[1,7];
6          else
7                  z = 0;
8  }
```

This new program is linear: it does not contain any product of variables anymore, they have been replaced with intervals. However, the linearization process is not over because the program contains linear interval expressions[4] whereas coefficients of standard linear expressions must be scalar values.

We then have to eliminate the intervals to obtain standard convex polyhedra. An alternative that avoid interval elimination would be to switch to the abstract domain of interval polyhedra [1]. That domain directly operates on polyhedra of the form $\sum_{i=1}^{n} [a_i, b_i]x_i \leq c$ *i.e.*, with intervals as coefficients. However, this abstract domain uses adapted versions of the polyhedron operators - projection, intersection. Therefore, we would have to certify in Coq these operators to handle interval polyhedra. Since we already have the certified library for standard convex polyhedra of Fouilhé [5], we choose to stay within the abstract domain of convex polyhedra.

2.1.2 Interval elimination and polyhedron approximation

The purpose of interval elimination is to replace an interval polyhedron[5] P_{itv} with a rational polyhedron P such that $\mathscr{S}(P_{itv}) \subseteq \mathscr{S}(P)$, where $\mathscr{S}(P)$ is the set of points $(x_1, \ldots, x_n)$ that satisfy the constraints of P.

Consider Example 1 where we want to eliminate the interval $[1, 7]$. Every number in $[1, 7]$ would lead to a convex polyhedron, but recall that the goal of abstract interpretation is to consider all possible states of the program. That's why the scalar that will replace an interval has to be chosen carefully to perform an over-approximation of the program behaviours.

The elimination technique proposed by Miné [11] consists in replacing every interval by its middle and increasing the constant member c of each linear constraint by a value depending on the intervals size. This technique is efficient but an important imprecision due to the constant can be expected.

2.1.3 The main steps of the algorithm

Given a statement containing a polynomial expression over $\mathbb{Z}$, say $(x + y)(x + y) - 2x * y$, the main steps of our algorithm are :

(1) Expand the polynomial expression into $x*x+y*y+x*y+y*x-2x*y$, which is still a non-linear expression over $\mathbb{Z}$. This step is used to determine which variables appear most frequently.

(2) Intervalize some variables, *e.g.* $x \in [1, 7]$, $y \in [-2, 3]$, to obtain a linear expression $[1, 7] * x + [-2, 3] * y$ with intervals as coefficients.

(3) Perform a case analysis on the sign of the remaining variables in order to replace intervals by their upper or lower bound to get into the linear expressions over $\mathbb{Z}$. Finally the linearization of the assignment $z := (x + y) * (x + y) - 2 * x * y$ produces the non-deterministic program:

$$\text{if } (y \geq 0) \begin{cases} \text{then} & z :\overset{ND}{=} \{z \mid x - 2y \leq z \leq 7x + 3y\} \\ \text{else} & z :\overset{ND}{=} \{z \mid x + 3y \leq z \leq 7x - 2y\} \end{cases}$$

[4] A linear interval expression is a linear expression where coefficients are intervals.

[5] An interval polyhedron is a conjunction of linear intervals constraints.

It uses a non-deterministic assignment that we shall detail further. The output of the linearization algorithm is a program expressed in a guarded command language presented in Section 2.1.3. The choice of variable to intervalize is discussed in Section 2.1.3. The interval elimination is explained in Section 2.1.3 and 2.1.3.

The Guarded Command Language of linearized expression In general, the linearization of an instruction produces a program formed of nested conditionals. The linear programs are expressed in a Guarded Command Langage (GCL) with no loop. GCL reuses expressions and if-then-else of the Cminor intermediate representation of CompCert [7] and introduces guards, denoted if...then, and two non-deterministic statements alt and $:\overset{ND}{=}$.

Non-Deterministic Branching It often happens that the approximation of a guard g and its negation $\neg g$ are not disjoint. The statement if (g) then S_1 else S_2 is therefore analyzed as a non-deterministic alternative:

$$\text{alt} \begin{cases} \text{if } (\textit{approximation}(g)) \text{ then } S_1 \\ \text{if } (\textit{approximation}(\neg g)) \text{ then } S_2 \end{cases}$$

Semantically, the alternative statement alt non-deterministically chooses to execute one branch or the other. Hence, faced with the alt statement, the analyzer considers the two branches as feasible and pursues the analysis on both. It builds a polyhedron that captures the effect of each branch and merges them at the exit of the alt by computing their convex hull.

Non-Deterministic Assignment Non-determinism also arises in the linearization of an assignment. It is captured by the construction $z :\overset{ND}{=} \{z \mid C\}$ that non-deterministically chooses a value satisfying the linear condition C and assigns it to z. More specifically, the linearization will produce assignments of the form $z :\overset{ND}{=} \{\tilde{z} \mid \ell_1 \leq \tilde{z} \leq \ell_2\}$, where ℓ_1 and ℓ_2 are linear expressions. A non-deterministic assignment is a straightforward generalization of the standard assignment $z := \ell$. Instead of intersecting the initial polyhedron P with an equality constraint (see Section 1), we consider the intersection with the constraint C. The standard assignment is thus equivalent to $z :\overset{ND}{=} \{\tilde{z} \mid \ell \leq \tilde{z} \leq \ell\}$. Formally, the effect of $z :\overset{ND}{=} \{\tilde{z} \mid \ell_1 \leq \tilde{z} \leq \ell_2\}$ on P is the polyhedron $\left((P \sqcap \ell_1 \leq \tilde{z} \leq \ell_2)_{/z}\right)[\tilde{z}/z]$ where $\tilde{z}$ is a fresh variable that is renamed into z after the projection of the old value of z.

Choice of the variables to intervalize Before intervalizing, we must determine what variables must be eliminated. This choice influences the accuracy of the result. Several choices are discussed by Miné [11]. In our implementation, we eliminate the variables that appear the most frequently in the expression. This simple heuristic intervalizes as few variables as possible in order to maintain precision.

In our current implementation, it is not possible to intervalize only some occurrences of a variable. When a variable is chosen to be intervalized, all of its occurrences are. This can lead to a loss of precision, especially when the polynomial that needs to be linearized contains powers of variables. Indeed, if a variable x appears in the expression with a power n strictly greater than one, x must be intervalized at least $n-1$ times but we can keep one occurrence of x ; we would end with $x \times [a,b]^{n-1}$. The following example gives a trick to improve precision without modifying the implementation.

Example 2. Consider a statement $z := f(x,y)$ with the polynomial $f \triangleq 4 - x^2 - y^2$. Knowing that $x \in [1,7]$ and $y \in [-2,3]$, a naive intervalization would give $z := 4 - [1,7] * [1,7] - [-2,3] * [-2,3] \triangleq [-54,-1]$, and would loose all dependencies on x and y. To avoid this, one can rename variables before intervalization. With the equalities $x_1 = x$ and $y_1 = y$, the polynomial f is equal to $4 - x \times x_1 - y \times y_1$. Then, the intervalization of x_1 and y_1 is sufficient to get a linear expression over intervals and returns $z := 4 - x \times [1,7] - y \times [-2,3]$.

This process can be generalized to any power of variable: x^n is renamed into $x \times x_1 \times ... \times x_{n-1}$, then $x_1, ..., x_{n-1}$ are intervalized.

Interval elimination on an example We now explain how to eliminate the intervals from linear interval expressions generated by the intervalization of variables. In the following examples, we assume that the sign of the variable y is known: $y \geq 0$. We deal with the general case in Appendix A.

Interval elimination in conditional expressions Let us consider the following program obtained after intervalization of $x \in [1,3]$, and focus on the interval $[1,3]$ from the condition $[1,3]\times y \leq 8$.

Example 3.

```
1  int y,z;
2  if(y*[1,3] <= 8)
3      z = y * [1,3];
4  else
5      z = 0;
```

Suppose that $y \in [3,5]$. Then, the condition $[1,3]\times y \leq 8$ can be true for some values of $[1,3]$ (*e.g.* for 1 or 2) or it can be false for other values (*e.g.* 2 or 3).

A static analyzer must consider all behaviors of the program ; it must then analyze

- the "then" branch in any case where $[1,3]\times y \leq 8$ may be true,
  that is when $\exists i \in [1,3],\ i\times y \leq 8$ (1)
- the "else" branch in any case where $[1,3]\times y > 8$ may be true,
  that is when $\exists i \in [1,3],\ i\times y > 8$ (2)

Recalling that $y \in [3,5]$, if we choose 1 for $i \in [1,3]$ then (1) becomes true. For $i = 3$, then (2) becomes true and if $i = 2$ is chosen, both (1) and (2) can be true depending on the value of y. The if-then-else construction cannot express these non-exclusive branchings, instead we use the non-deterministic construction alt with if-then guards:

$$\text{alt}\begin{cases} \text{if } (\exists i \in [1,3], i\times y \leq 8) \text{ then } z := y\times[1,3] \\ \text{if } (\exists i \in [1,3], i\times y > 8) \text{ then } z := 0 \end{cases}$$

Now, to get an actual program, we have to remove the existential quantifiers in the conditions. This is done following Pugh's techniques for elimination of existential quantifiers [13]. The reduction to quantifier free arithmetic conditions is based on the following equivalences:

$$\exists i \in [a,b], i\times y \leq k \Leftrightarrow \min([a,b]\times y) \leq k \quad (1)$$

$$\exists i \in [a,b], i\times y \geq k \Leftrightarrow \max([a,b]\times y) \geq k \quad (2)$$

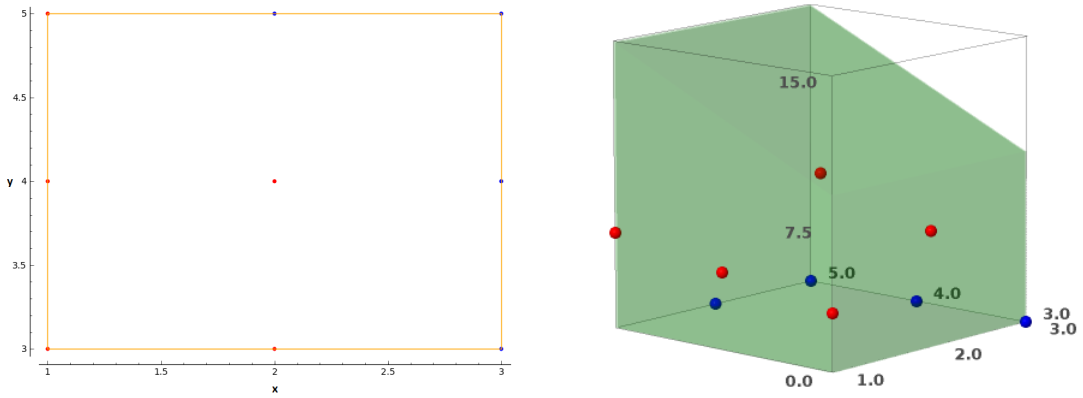
$$\min([a,b]\times y) = \begin{cases} a\times y & \textit{if} \quad y \geq 0 \\ b\times y & \textit{if} \quad y < 0 \end{cases} \quad (3)$$

$$\max([a,b]\times y) = \begin{cases} b\times y & \textit{if} \quad y \geq 0 \\ a\times y & \textit{if} \quad y < 0 \end{cases} \quad (4)$$

Example 3 (following). Back to our example, since we know that $y \geq 0$ we can replace $\exists i \in [1,3],\ i\times y \leq 8$ by $1\times y \leq 8$ and $\exists i \in [1,3],\ i\times y > 8$ by $3\times y > 8$. Finally, the elimination of the interval in the condition $[1,3]\times y \leq 8$ of the program of Example 3 produces the following program fragment:

$$\text{alt}\begin{cases} \text{if } (y \leq 8) & \text{then } z := y\times[1,3] \\ \text{if } (3\times y > 8) & \text{then } z := 0 \end{cases}$$

Interval elimination in assignments Going on with our example, the interval that appears in the assignment $z := [1,3]\times y$ needs to be eliminated. The value of z after the assignment is between y and $3\times y$. This corresponds to the constraint $y \leq z \leq 3\times y$ since $y \geq 0$. The assignment is then replaced by $z :\overset{ND}{=} \{\tilde{z} \mid y \leq \tilde{z} \leq 3\times y\}$.



(a) (b)

Figure 5 – (a): starting polyhedron $P = [1,3] \times [3,5]$. The red points are elements of $\{(x,y) \in \mathbb{Z}^2 \mid x * y \leq 8\}$, blue points are elements of $\{(x,y) \in \mathbb{Z}^2 \mid x * y > 8\}$. (b): representation in green of the effect of the program 2.1.3 on P. The points are the images of the points of (a) by the code of Example 3 .

Finally, the linearization algorithm returns the following non-deterministic linear program that is analyzed with the domain of polyhedra:

$$\text{alt} \begin{cases} \text{if } (y \leq 8) & \text{then } z :\overset{ND}{=} \{\tilde{z} \mid y \leq \tilde{z} \leq 3{\times}y\} \\ \text{if } (3{\times}y > 8) & \text{then } z := 0 \end{cases}$$

A starting polyhedron $P = \{x \geq 1, x \leq 3, y \geq 3, y \leq 5\}$ and the result after the previous program are displayed on Figure 5.

Example 4. The Figure 6 shows the starting polyhedron $\{x \geq 1, y \geq -2, x - y \geq 0, x + y \leq 5\}$ and the polyhedron corresponding to the linearization of Example 1. We can see that the green polyhedron adds a dimension (corresponding to the variable z) to the blue one. The GCL code resulting from this linearization is available in Appendix B, as well as a summary of the main steps of the algorithm.

2.2 Correctness criteria of the linearization process

Recall that, as a part of VERASCO, the aim is to certify this linearization technique in Coq. We start by defining the correctness criteria for assignment and if-then-else. We shall detail the Coq implementation further. Correctness of the linear approximation consists in checking that the linearized program covers all the behaviours of the original statement. The idea is to compare the effects on variable values of a statement to that of its linearization. To this end, we need to define a memory state as the value of the program variables at a given program point.

2.2.1 Valuations of variables

Let us define a memory state as a valuation function $val : variable \mapsto \mathbb{Z}$, that associates a value with each variable. For instance, the valuation $[x \mapsto 1; y \mapsto -2; z \mapsto 0]$ means that x, y and z are respectively 1, -2 and 0. Then, given variables and corresponding intervals, the set of all possible valuations is defined by enumerating possible values of each interval. For instance, the valuations of $[(x,[0,1]);(y,[-1,1])]$ is the set of functions

$$\{[x \mapsto 0; y \mapsto -1], [x \mapsto 0; y \mapsto 0], [x \mapsto 0; y \mapsto 1],$$
$$[x \mapsto 1; y \mapsto -1], [x \mapsto 1; y \mapsto 0], [x \mapsto 1; y \mapsto 1]\}$$

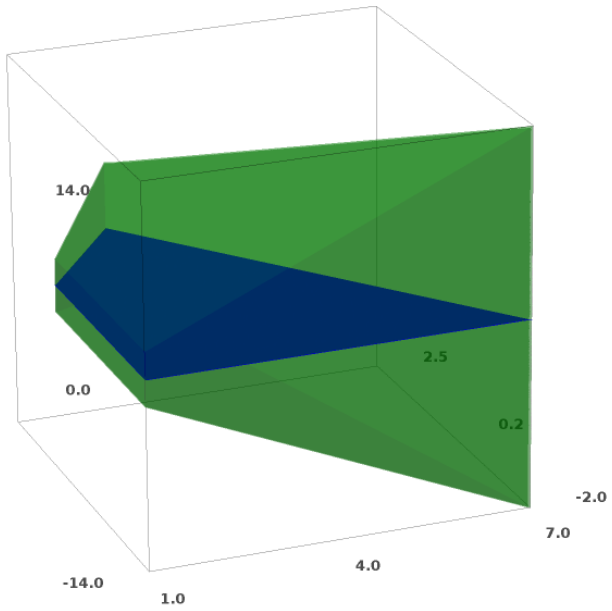


Figure 6 – In blue: the initial polyhedron $\{x \geq 1, y \geq -2, x - y \geq 0, x + y \leq 5\}$. In green: the image of the initial polyhedron by the linearization of Example 1.

During the proof, we shall need to compare the value of variables before and after a statement. In order to compute the value of an expression at a given memory state, we define the following function:

$$\begin{aligned} eval : expression_A \times valuation &\to \mathbb{Z} \\ (f, v) &\mapsto f[x_i \leftarrow v(x_i)] \end{aligned}$$

Given an arithmetical expression f and a valuation v, the function replaces each variable in f with its value in v.

The linearization of a statement returns a linear program prg in a form of a tree of nested conditional statements. We denote by $branch_{t,v}$ the branch of prg corresponding to the valuation v, in the sense that at every node if $(x_i \geq 0)$ of t, we chose the branch corresponding to the sign of $v(x_i)$.

In the case of an assignment $x := f$, $branch_{t,v}$ has the form $x :\overset{ND}{=} \{\tilde{x} \mid \ell_j \leq \tilde{x} \leq \ell'_j\}$ for some $j \in [1, k]$. For the linearization of a test if $(f \leq f')$ then S_1 else S_2, $branch_{t,v}$ has the form

$$\text{alt} \begin{cases} \text{if } (\ell_j \leq \ell'_j) \text{ then } S_1 \\ \text{if } (\ell_j > \ell'_j) \text{ then } S_2 \end{cases}$$

2.2.2 Correctness of assignment linearization

Consider a polyhedron P, an assignment asg of the form $x := f$ where f is a polynomial in the $(x_1, \ldots, x_n)$ variables of the program and $prg = linearization_P(asg)$ the program resulting from the linearization of asg with respect to the polyhedron P. As said previously, prg is a tree where nodes are sign assumptions on the variables $(x_1, \ldots, x_n)$, and leaves have the form $x :\overset{ND}{=} \{\tilde{x} \mid \ell_i \leq \tilde{x} \leq \ell'_i\}$. Given a valuation v, an execution of prg leads into a branch ending by $x :\overset{ND}{=} \{\tilde{x} \mid \ell_j \leq \tilde{x} \leq \ell'_j\}$ for some $j \in [1, k]$. The linear program prg is a correct approximation of $x := f$ if the value of x after this assignment is reached by an execution of prg. Thus, we want to ensure that $eval(f, v) \in [eval(\ell_j, v), eval(\ell'_j, v)]$. We define the correctness criterion for assignments as follows (where v **sat** P means that the valuation v defines a point satisfying the constraints of P):

Definition 1 (Correctness criterion - assignment)**.**

$$\begin{array}{c}\forall \textit{ expression } f,\ \forall \textit{ Polyhedron } P,\ \forall \textit{ valuation } v,\\ \Big(prg \triangleq linearization_P(x := f) \wedge v \textbf{ \textit{sat}} P \wedge branch_{(prg,v)} \triangleq x :\overset{ND}{=} \{\tilde{x} \mid \ell_j \leq \tilde{x} \leq \ell'_j\}\Big)\\ \Rightarrow eval(f,v) \in \big[eval(\ell_j, v), eval(\ell'_j, v)\big]\end{array}$$

2.2.3 Correctness of if-then-else linearization

Consider a polyhedron P, a conditional statement $ineq \triangleq \text{if } (f \leq f') \text{ then } S_1 \text{ else } S_2$ and $prg \triangleq linearization_P(ineq)$. prg is a tree where nodes are sign assumptions on the variables $(x_1, \ldots, x_n)$, and leaves have the form $\ell \leq \ell'$. Given a valuation v, an incomplete execution of prg ends to an alternative of the form

$$\text{alt} \begin{cases} \text{if } (\ell_1 \leq \ell'_1) \text{ then } S_1 \\ \text{if } (\ell_2 > \ell'_2) \text{ then } S_2 \end{cases}$$

When $f \leq f'$, we want S_1 to be executed, thus we want $\ell_1 \leq \ell'_1$ to be true. Conversely, when $f > f'$, we want S_2 to be executed, thus we want $\ell_2 > \ell'_2$ to be true. Thus, the correctness criteria is:

Definition 2 (Correctness criterion - if-then-else)**.**

$$\begin{array}{c}\forall \textit{ expression } f,\ f',\ \forall \textit{ Polyhedron } P,\ \forall \textit{ valuation } v,\\ \left(\begin{array}{ll} & prg \triangleq linearization_P(\textit{if } (f \leq f') \textit{ then } S_1 \textit{ else } S_2)\\ \wedge & v \textbf{ \textit{sat}} P\\ \wedge & branch_{(prg,v)} = \textit{alt} \begin{cases} \textit{if } (\ell_1 \leq \ell'_1) \textit{ then } S_1 \\ \textit{if } (\ell_2 > \ell'_2) \textit{ then } S_2 \end{cases}\end{array}\right)\\ \Rightarrow \begin{cases} (f \leq f' \Rightarrow \ell_1 \leq \ell'_1) \\ \wedge \\ (f > f' \Rightarrow \ell_2 > \ell'_2) \end{cases}\end{array}$$

2.3 Certification in Coq: formalization and correctness proof

Coq is a proof assistant that can be used to develop programs respecting properties or specifications [9]. The Coq specification language is called Gallina and organizes the theories with axioms, lemmas, theorems and definitions in order to express mathematical properties. Proofs are built interactively : the user can apply tactics or lemmas on hypotheses or on the goal. Lots of lemmas are already defined in Coq libraries, helping the manipulation of simple types such as integers or boolean.

2.3.1 Main Coq types

The type $\mathbb{Z} \cup \bot$ of unbounded intervals Recall that during the intervalization process, some variables are replaced by intervals. Besides, a variable can be unbounded on one or both sides and its intervalization would lead to a half-bounded or unbounded interval. In order to manipulate such intervals, let us introduce the type $\mathbb{Z} \cup \bot$, which is the usual coq type $\mathbb{Z}$ of integers which is added a constructor $\bot$. For instance, $x \in [-2, +\infty[$ will be written as $x \in [-2, \bot]$ and means that $x \geq -2$ and x has no upper bound. Remark that we don't need $\bot$ to have an explicit sign because in our algorithm, the context is always sufficient to deduce it. Indeed, recall that intervals represent possible values for variables, that's why we forbid $]\_, -\infty[$ and $]+\infty, \_[$ which have no sense in our use. Thus, when $\bot$ appears in an interval, its sign is determined by the side it belongs to. An ambiguity could appear when intervals are eliminated during expression maxi/minimization. As a matter of fact, if $\bot$ persists after the maximization (resp. minimization) of an expression, it is necessarily positive (resp. negative). Knowing that an interval maximization returns its upper bound, if a negative $\bot$ is the result of an interval maximization, it means that the upper bound of this interval was $-\infty$. This is impossible, for the reason we mentioned earlier. The same reasoning shows that an interval minimization cannot give a positive $\bot$.

$\bot$ should not be thought of as a representation of infinity but as an unbounded value of a variable. That's why we can define $0 \times \bot = 0$. This representation of $\bot$ ensures the associativity of interval multiplication

that is needed in the proof. However, assuming $0 \times \bot = 0$ breaks the distributivity property on $\mathbb{Z} \cup \bot$. Indeed, $(2-2) \times \bot = 0 \times \bot = 0$ and $2 \times \bot - 2 \times \bot = \bot$. This forbids the expansion of expression in $\mathbb{Z} \cup \bot$.

The type `expr` of expressions One way to prove that an element of a type t satisfies a property is to express it in a type that constructively enforces this property, meaning that any elements of type t that can be produced possesses the property by construction. That's why we use several types for expressions, depending on what property we need to ensure. The Coq definition of these types are given in Appendix C.

- Given a piece of code containing a non-linear expression f, we first expand in $\mathbb{Z}$ before intervalization. The reason is that expanding relies on distributivity of multiplication on addition which does not hold in the domain $\mathbb{Z} \cup \bot$ of unbounded intervals. The expansion is performed by a function $expand$: $\texttt{expr}_{\mathbb{Z}} \longrightarrow \texttt{expr\_exp}_{\mathbb{Z}}$, where $\texttt{expr\_exp}_{\mathbb{Z}}$ is a type defining expanded expressions over $\mathbb{Z}$, which constructively only allows expressions of the form $\sum \prod c_i x_i$.
- The second step is to intervalize variables of the expanded expression. This is done by the function $intervalize$: $\texttt{expr\_exp}_{\mathbb{Z}} \longrightarrow \texttt{expr\_lin}_{itv(\mathbb{Z} \cup \bot)}$, where $\texttt{expr\_lin}_{itv(\mathbb{Z} \cup \bot)}$ is the type of linear interval expressions whose constants are intervals over $\mathbb{Z} \cup \bot$.
- The last step is the interval elimination. It returns a GCL program fragment containing expressions in $\texttt{expr}_{\mathbb{Z} \cup \bot}$, which is the type of expressions over $\mathbb{Z} \cup \bot$. As shown previously, interval elimination requires to build a tree where nodes make assumptions about variable sign. This operation is done by the function $build\_tree$: $\texttt{expr\_lin}_{itv(\mathbb{Z} \cup \bot)} \longrightarrow \texttt{stmt}$. At each leaf, the sign of every variable is known and the function $maximinize$: $\texttt{expr\_lin}_{itv(\mathbb{Z} \cup \bot)} \longrightarrow \texttt{expr}_{\mathbb{Z} \cup \bot}$ finishes the linearization, maximizing or minimizing inequalities. We end up with a tree where leaves are linear expressions on $\mathbb{Z} \cup \bot$.

The proof of correctness requires to compare the possible result of the original program according to the semantics of CompCert C to the values returned by the GCL program. To conduct this reasoning, the first step is to provide GCL a semantics.

2.3.2 The semantics of GCL

From now on, we will consider more general programs with the notion of program state, instead of isolated statements. A program state gathers the value of each variable at a given point as well as informations about the next statement. It is represented as $\texttt{stmt} \times \texttt{cont} \times \texttt{valuation}$, and denoted in Coq by `(State stmt cont valuation)`. The `cont` member is used to store the next statement of a state. Our Coq type for statements is

```
Inductive stmt : Type :=
  | Sskip : stmt
  | Sseq : stmt → stmt → stmt
  | Sassign : var → a_expr → stmt
  | SNDassign : a_expr_{Z∪⊥} → var → a_expr_{Z∪⊥} → stmt
  | Sifthen : expr → stmt → stmt
  | Salt : stmt → stmt → stmt
  | Sifthenelse : expr → stmt → stmt → stmt.
```

Knowing that `expr` handles expressions over both $\mathbb{Z}$ and $\mathbb{Z} \cup \bot$, a statement can manipulate at the same time expressions over $\mathbb{Z}$ as well as expressions over $\mathbb{Z} \cup \bot$. This is because we want the input and output language of the linearization to be the same.

To build the proof, we need a link between statements and the memory. This link is done by the semantics, which states how the memory behaves faced to each statement. The semantics of a program is represented as steps from a state to another. In the Appendix D, we define the semantics of our GCL. It is inspired from the Clight semantics of CompCert, which is extended with semantics for `Sifthen`, `Salt` and `SNDassign`. In the following, we give an extract of the semantics which concerns `Salt` and `SNDassign`.

```
Inductive step : state → state → Prop :=
| step_alt1 : ∀ (s1 s2 : stmt) (k : cont) (v : valuation),
    step
      (State (Salt s1 s2) k v)
      (State s1 k v)

| step_alt2 : ∀ (s1 s2 : stmt) (k : cont) (v : valuation),
    step
      (State (Salt s1 s2) k v)
      (State s2 k v)

| step_NDassign : ∀ (x:var) (z z_min z_min:ℤ ∪ ⊥)
    (f_min f_max :a_expr_{ℤ∪⊥} ) (k:cont) (v:valuation),
   z_min = (eval_a_expr_{ℤ∪⊥} f_min v) →
   z_max = (eval_a_expr_{ℤ∪⊥} f_max v) →
   z_min <= z → z <= z_max →
   step
     (State (SNDassign f_min x f_max) k v)
     (State (Sassign x (Aexpr_{ℤ∪⊥} (ZEconst_int z))) k v).
```

The cases `step_alt1` and `step_alt1` show that there exists a step from the program state `(State (Salt s1 s2) k v)` to both the states `(State s1 k v)` and `(State s2 k v)`. This is how we represent non-determinism: both branches can be taken by the program. The case `step_NDassign` states that if $z \in [e_{min}, e_{max}]$, there exists a step from the program state `(State (SNDassign` e_{min} `x` e_{max}`) k v)` to `(State (Sassign x z) k v)`. As a comparison, the semantics of the classical assignment is:

```
| step_assign : ∀ (x:var) (f:a_expr) (k:cont) (v v':valuation),
    v' = (update v x (eval_a_expr f v)) →
    step
      (State (Sassign x f) k v)
      (State Sskip k v')
```

It means that in memory, the value of `x` after the assignment `x := f` is equal to the evaluation of the expression `f` in the valuation `v`, *i.e.* `(eval_a_expr f v)`

This semantics allows only to link two states by a step. During the proof, we will manipulate programs composed of several states and steps. We define the relation $exec$, able to link two states with several steps, as follows:

```
Inductive exec : state → state → Prop :=
  | no_step : ∀ (S : state), exec S S
  | more_step : ∀ (S1 S2 S3 : state), step S1 S2 → exec S2 S3 → exec S1 S3.
```

This definition states that if there exists an execution between two program states `S1` and `S3` (*i.e.* `exec S1 S3`), either `S1` = `S3` or there exists a third state `S2` such that `step S1 S2` and `exec S2 S3`.

2.3.3 Correctness proof of the linearization algorithm

Correctness of the assignment linearization The main theorem ensuring the correctness for assignment is the following one:

```
Theorem lin_assign_wf (x:var) (f:a_expr_ℤ (P:polyhedron)
    (v:valuation) (k:cont) :
   v ∈ P →
   exec
       (State (linearize (Sassign x f) P) k v)
       (State (Sassign x (eval f v)) k v).
```

This theorem ensures that there exists an execution from the linearized GCL program – *i.e.* `(linearize (Sassign x f) P)` – to a program state where x equals to its value in the original program – that is `(eval f v)`. This is the translation of the correctness criterion 1 according to the semantics we have defined.

Correctness of the linearization of if-then-else The property expressed in Definition 2 that we want to ensure about if-then-else is reformulated in two parts here:

```
Lemma lin_ifthenelse_true_wf (s1 s2 : stmt) (op : b_binop)
    (f:b_exprℤ) (P:polyhedron) (v:valuation) (k:cont) :
   v ∈ P →
   bool_of_expr f v true →
   exec
       (State (linearize (Sifthenelse f s1 s2) P) k v)
       (State s1 k v).

Lemma lin_ifthenelse_false_wf (s1 s2 : stmt) (op : b_binop)
    (f:b_exprℤ) (P:polyhedron) (v:valuation) (k:cont) :
   v ∈ P →
   bool_of_expr f v false →
   exec
       (State (linearize (Sifthenelse f s1 s2) P) k v)
       (State s2 k v).
```

The first lemma states that if the condition `f` is true, then there exists an execution of the linearization of `(Sifthenelse f s1 s2)` leading to a state where `s1` is executed. Conversely, the second lemma ensures that if the condition `f` is false, then there exists an execution of the linearization of `(Sifthenelse f s1 s2)` leading to a state where `s2` is executed.

Note that `f` can be an arithmetical expression as well as a boolean expression. As in the C language, an arithmetical expression is false if it is equal to zero, true otherwise.

3 Linearization on polytopes using Bernstein basis

In this section, we introduce a linearization technique based on the approximation of non-linear expressions by multivariate polynomials in the Bernstein basis. Given a polyhedron P and a polynomial f defined on the variables $x_1, ..., x_l$, we want to over-approximate $P \sqcap f \geq 0$. As f is expressed in the canonical basis $\mathfrak{C}$, the method consists in converting f into the Bernstein basis. From the coefficients of f in this basis, we can deduce a polyhedron containing $\{(x_1, ..., x_l) \in P \mid f(x_1, ..., x_l) \geq 0\}$. We begin by giving reminders about the Bernstein basis, and some clue about the conversion from the canonical to the Bernstein basis. Then, we show how to obtain an over-approximating polyhedron from Bernstein coefficients. We will use the polyhedron $P \triangleq \{(x, y) \mid x - 1 \geq 0,\ y + 2 \geq 0,\ x - y \geq 0,\ -x - y + 5 \geq 0\}$ and the polynomial $f(x, y) \triangleq 4 - x^2 - y^2$ as a running example.

3.1 Bernstein representation of polynomials on $[0, 1]^l$

3.1.1 The Bernstein basis

The univariate Bernstein basis We begin by reminding what the univariate Bernstein basis is, following the notations of Farouki's survey [3]. The univariate Bernstein basis $\mathfrak{B}$ of degree n is the set of polynomials b_k^n defined on $x \in [0, 1]$ as

$$b_k^n(x) \triangleq \binom{n}{k}(1 - x)^{n-k}x^k, \quad k = 0, ..., n$$

These polynomials form a basis $\mathfrak{B}$ of the space of polynomials on $[0, 1]$. Thus, any polynomial $p(x)$ with $x \in [0, 1]$ can be written in the Bernstein basis

$$p(x) \triangleq \sum_{k=0}^{n} c_k b_k^n(x), \quad x \in [0, 1],\ c_k \in \mathbb{R}$$

A remarkable property of Bernstein polynomials is that the coefficients c_k allow to deduce control points. As we shall see further, the convex hull of these control points contains the polynomial itself. For

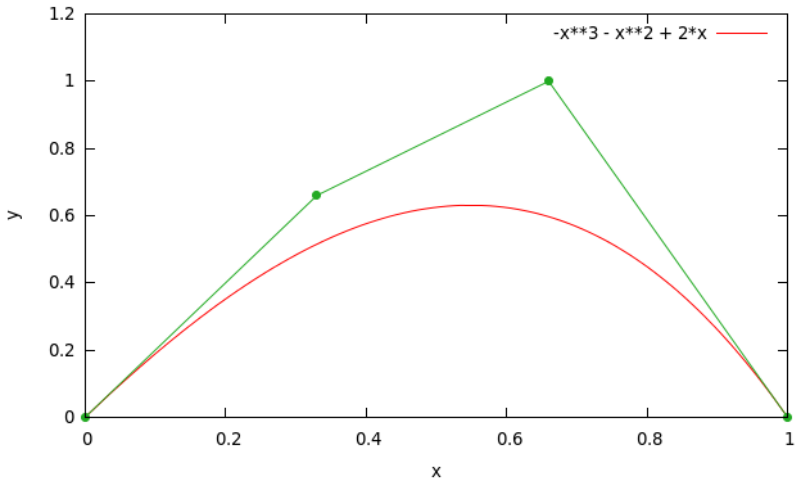


Figure 7 – Representation of $f(x) = -x^3 - x^2 + 2x$ *in red. The green curve is the convex hull of the control points* $(0,0)$, $\left(\frac{1}{3}, \frac{2}{3}\right)$, $\left(\frac{2}{3}, 1\right)$ *and* $(1,0)$.

instance, the Figure 7 shows the polynomial $f(x) = 3 * x^2 - x$ and the convex hull of four control points. The Bernstein polynomial of degree 3 corresponding to f is $0 \times (1-x)^3 x^0 + \frac{1}{3} \times (1-x)^2 x^1 + \frac{2}{3} \times (1-x)^1 x^2 + 1 \times (1-x)^0 x^3$. From this polynomial, we can deduce the four control points $(0,0)$, $\left(\frac{1}{3}, \frac{2}{3}\right)$, $\left(\frac{2}{3}, 1\right)$ and $(1,0)$.

Because we are manipulating multivariate polynomials, we need to define the multivariate Bernstein basis. Let us introduce first some useful notations.

Notations: Multi-indexes Tuples and multi-indexes are typed in boldface. Following the notations from [14], let l be the number of variables, let $\boldsymbol{I} = (i_1, ..., i_l) \in \mathbb{N}^l$ be a multi-index and $\boldsymbol{x}^{\boldsymbol{I}} \triangleq x_1^{i_1} \times ... \times x_l^{i_l}$ be a multi-power. We define a partial order on multi-indexes by $\boldsymbol{I} \leq \boldsymbol{J} \Leftrightarrow \forall k = 1, .., l,\ i_k \leq j_k$. Let $\binom{\boldsymbol{J}}{\boldsymbol{I}} = \binom{j_1}{i_1} ... \binom{j_l}{i_l}$. The degree $\boldsymbol{N}$ of a multi-index $\boldsymbol{I}$ is the index vector $(n_1, ..., n_l)$ such that $\forall k = 1, ..., l,\ i_k \leq n_k$.

The multivariate Bernstein basis The Bernstein basis $\mathfrak{B}$ of degree $\boldsymbol{N} = (n_1, ..., n_l)$ on $\boldsymbol{x} = (x_1, ..., x_l) \in [0,1]^l$ is defined as

$$B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x}) = b_{i_1}^{n_1}(x_1) \times ... \times b_{i_l}^{n_l}(x_l), \quad \boldsymbol{I} \leq \boldsymbol{N}$$

A multivariate polynomial expressed in this basis is written

$$p(\boldsymbol{x}) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} c_{\boldsymbol{I}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x}), \quad \boldsymbol{x} \in [0,1]^l,\ c_{\boldsymbol{I}} \in \mathbb{R}$$

The Bernstein basis $\mathfrak{B}$ respects the following properties:

$$\forall \boldsymbol{I} \leq \boldsymbol{N},\ \forall \boldsymbol{x} \in [0,1]^l,\ B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x}) \in [0,1] \tag{5}$$

$$\forall \boldsymbol{x} \in [0,1]^l,\ \sum_{\boldsymbol{I} \leq \boldsymbol{N}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x}) = 1 \tag{6}$$

The Property 6 is called the partition-of-unity property. These two properties allow us to handle the Bernstein basis elements as coefficients of a convex combination. Indeed, given points $A_{\boldsymbol{0}}, ..., A_{\boldsymbol{N}} \in \mathbb{R}^l$ and $\boldsymbol{x} \in [0,1]^l$, the point $A \triangleq \sum_{\boldsymbol{I} \leq \boldsymbol{N}} A_{\boldsymbol{I}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x})$ is a convex combination of $A_{\boldsymbol{0}}, ..., A_{\boldsymbol{N}}$. It means that A belongs to the convex hull of $A_{\boldsymbol{0}}, ..., A_{\boldsymbol{N}}$.

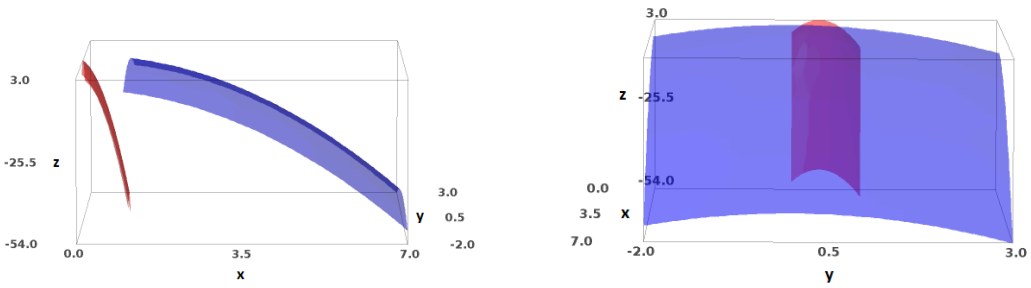


Figure 8 – Scaling of $f(x, y) = -x^2 - y^2 + 4$ from $[1, 7] \times [-2, 3]$ (in blue) to $[0, 1]^2$ (in red).

3.1.2 Change of basis

In this part, we show how to translate a polynomial from $\mathfrak{C}$ to $\mathfrak{B}$ with the conventional method explained by [14] and [12]. The conversion algorithm has been implemented and proven in the Prototype Verification System by [12]. As Bernstein polynomials are defined on $[0, 1]^l$, a polynomial f defined as $f(\boldsymbol{t}) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} d_{\boldsymbol{I}} \boldsymbol{t}^{\boldsymbol{I}}$ on $[a_1, b_1] \times ... \times [a_l, b_l]$ in $\mathfrak{C}$ needs to be scaled and shifted into $[0, 1]^l$. For $k = 1, ..., l$, let $\sigma_k : [0, 1] \to [a_k, b_k]$ be an affine mapping function. We are looking for the coefficients $d'_{\boldsymbol{I}}$ such that $\forall (x_1, ..., x_l) \in [0, 1]^l$, $f'(x_1, ...x_l) \triangleq f(\sigma_1(x_1), ..., \sigma_l(x_l)) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} d'_{\boldsymbol{I}} \boldsymbol{x}^{\boldsymbol{I}}$. These coefficients $d'_{\boldsymbol{I}}$ can be expressed in function of the $d_{\boldsymbol{I}}$ in the following way :

$$d'_{\boldsymbol{I}} = (\boldsymbol{b} - \boldsymbol{a})^{\boldsymbol{I}} \sum_{\boldsymbol{J}=\boldsymbol{I}}^{\boldsymbol{N}} d_{\boldsymbol{J}} \left(\binom{\boldsymbol{J}}{\boldsymbol{I}} \boldsymbol{a}^{\boldsymbol{J}-\boldsymbol{I}} \right), \ \boldsymbol{I} \leq \boldsymbol{N}, \ \boldsymbol{a} = (a_1, ..., a_l), \ \boldsymbol{b} = (b_1, ..., b_l)$$

Example 5. Let us take the example $f = -x^2 - y^2 + 4$ with $x \in [1, 7]$, $y \in [-2, 3]$. The degree of f is the multi-index $N = (2, 2)$, meaning that x and y have maximum degree 2. The coefficients of f are $d_{(0,0)} = 4$, $d_{(2,0)} = d_{(0,2)} = -1$ and $d_{\boldsymbol{I}} = 0$ for all others indexes $\boldsymbol{I}$. The computation of $d'_{(2,0)}$ is done as follows:

$$\begin{aligned} d'_{(2,0)} &= ((7-1)^2) \times ((2+3)^0) \times \sum_{\boldsymbol{J}=(2,0)}^{(2,2)} \left(\binom{\boldsymbol{J}}{(2,0)} \times d_{\boldsymbol{J}} \times \left(1^{j_1-2}\right) \times \left((-2)^{j_2-0}\right) \right) \\ &= 36 \times \binom{(2,0)}{(2,0)} (-1)(1^{2-2}) \times ((-2)^{0-0}) \ \textit{because } d_{(2,1)} = d_{(2,2)} = 0 \\ &= -36 \end{aligned}$$

Finally, the polynomial $f(x, y) = -x^2 - y^2 + 4$ mapped into $[0, 1]^2$ is $f'(x, y) = -36x^2 - 25y^2 - 12x + 20y - 1$, as shown on Figure 8. When (x, y) range over $[0, 1]^2$, the polynomial $f'(x, y)$ covers the image of f on $[1, 7] \times [-2, 3]$.

Now, given a polynomial $\sum_{\boldsymbol{I} \leq \boldsymbol{N}} d'_{\boldsymbol{I}} x^{\boldsymbol{I}}$ in $\mathfrak{C}$ defined on $[0, 1]^l$, the classical method to compute the Bernstein coefficients $c_{\boldsymbol{I}}$ is :

$$c_{\boldsymbol{I}} = \sum_{\boldsymbol{J} \leq \boldsymbol{I}} \frac{\binom{I}{J}}{\binom{N}{J}} d'_{\boldsymbol{J}}$$

As shown in [14], the translation from $\mathfrak{C}$ to $\mathfrak{B}$ using the conventional method has a complexity of $\mathcal{O}(n^{2l})$.

Example 6. We can now compute the Bernstein representation of the scaled polynomial $f' = -36x^2 - 25y^2 - 12x + 20y - 1$:

$$\begin{aligned} f' &= -B^{(2,2)}_{(0,0)} + 9B^{(2,2)}_{(0,1)} - 6B^{(2,2)}_{(0,2)} - 7B^{(2,2)}_{(1,0)} + 3B^{(2,2)}_{(1,1)} \\ &\quad -12B^{(2,2)}_{(1,2)} - 49B^{(2,2)}_{(2,0)} - 39B^{(2,2)}_{(2,1)} - 54B^{(2,2)}_{(2,2)} \end{aligned}$$

3.2 Polyhedron from Bernstein coefficients

Let P be a polyhedron, and assume we want to approximate the effect of the guard $f \geq 0$ on P. As said in Sect. 2.1.1, we can deduce intervals for each variable from P. Let us call $[a_i, b_i]$ the interval associated with the variable $x_i,\ i = 1, ..., l$. Let us call $P_{box} = [a_1, b_1] \times ... \times [a_l, b_l]$ and suppose f is defined on P_{box}. In the following, we denote by $\sigma(\boldsymbol{x})$ the vector $(\sigma_1(x_1), ..., \sigma_l(x_l))$, where σ_k are the affine functions defined in Sect. 3.1.2. The change of basis provided a polynomial f' whose image on $[0,1]^l$ coincides with the image of the original polynomial f on P_{box}. Since f' is defined on $[0,1]^l$, we ended with a Bernstein representation of f' as

$$f'(\boldsymbol{x}) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} c_{\boldsymbol{I}} B^{\boldsymbol{N}}_{\boldsymbol{K}}(\boldsymbol{x}), \quad \boldsymbol{x} \in [0,1]^l,\ c_{\boldsymbol{I}} \in \mathbb{R}$$

In this section, we explain how to build a polyhedron, over-approximating f', from its Bernstein coefficients. We define the set $\mathcal{V}'$, containing control points whose first l dimensions form a l-dimensional mesh:

$$\begin{array}{ll} \mathcal{V}' = & \{v'_{\boldsymbol{I}} \mid \boldsymbol{I} \leq \boldsymbol{N}\} \\ \textit{where} & \boldsymbol{I} = (i_1, \ldots, i_l) \\ & \boldsymbol{N} = (n_1, \ldots, n_l) \\ & v'_{\boldsymbol{I}} = \left(\frac{i_1}{n_1}, ..., \frac{i_l}{n_l}, c_{\boldsymbol{I}}\right) \end{array}$$

Let us define the convex hull $\mathcal{P}'_{\mathcal{V}}$ of the control points $\mathcal{V}'$. Note that the vertices of $\mathcal{P}'_{\mathcal{V}}$ belong to $\mathcal{V}'$, but the points of $\mathcal{V}'$ that are within the interior of $\mathcal{P}'_{\mathcal{V}}$ are not vertices of $\mathcal{P}'_{\mathcal{V}}$. We will now prove that $\mathcal{P}'_{\mathcal{V}}$ is an over-approximating polyhedron of f'. We start by noticing that the Bernstein coefficients of an indeterminate x_k are $\left(\frac{1}{n_k}, \frac{2}{n_k}, \ldots, \frac{n_k}{n_k}\right)$. We prove it through the following lemma:

Lemma 1. *Let $k \in \mathbb{N},\ 1 \leq k \leq l$.*

$$x_k = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} \frac{i_k}{n_k} B^{\boldsymbol{N}}_{\boldsymbol{I}}(\boldsymbol{x}), \quad \boldsymbol{x} \in [0,1]^l$$

Proof. We generalize to multivariate polynomials the proof of [3] for the univariate case. Let us define a truncated multi-index $\boldsymbol{I} \setminus k = (i_1, ..., i_{k-1}, i_{k+1}, ..., i_l)$. Let $\boldsymbol{x} \in [0,1]^l$ and consider the Bernstein polynomial $p(\boldsymbol{x}) \triangleq \sum_{\boldsymbol{I} \leq \boldsymbol{N}} \frac{i_k}{n_k} B^{\boldsymbol{N}}_{\boldsymbol{I}}(\boldsymbol{x})$. We will show that $p(\boldsymbol{x}) = x_k$. Let us focus on the Bernstein monomials issued by the k^{th} component of the index $\boldsymbol{I}$.

$$\begin{array}{rcll} p(\boldsymbol{x}) & = & \displaystyle\sum_{\substack{(\boldsymbol{I}\setminus k) \leq (\boldsymbol{N}\setminus k) \\ i_k = 0, \ldots, n_k}} \frac{i_k}{n_k}\, b^{n_k}_{i_k} \times B^{\boldsymbol{N}\setminus\boldsymbol{k}}_{\boldsymbol{I}\setminus\boldsymbol{k}}(\boldsymbol{x}) & \\ & = & \displaystyle\sum_{\substack{(\boldsymbol{I}\setminus k) \leq (\boldsymbol{N}\setminus k) \\ i_k = 0, \ldots, n_k}} \frac{i_k}{n_k} \binom{n_k}{i_k} x_k^{i_k} (1-x_k)^{n_k - i_k} \times B^{\boldsymbol{N}\setminus\boldsymbol{k}}_{\boldsymbol{I}\setminus\boldsymbol{k}}(\boldsymbol{x}) & \textit{by definition of } b^{n_k}_{i_k} \end{array}$$

As the terms of the sum corresponding to $i_k = 0$ vanish, we can start the summation at $i_k = 1$. We can therefore exploit the property of the binomial coefficients $\frac{i_k}{n_k}\binom{n_k}{i_k} = \binom{n_k - 1}{i_k - 1}$, for $i_k \geq 1$. Thus,

$$p(\boldsymbol{x}) = \sum_{\substack{(\boldsymbol{I}\setminus k) \leq (\boldsymbol{N}\setminus k) \\ i_k = 1, \ldots, n_k}} \binom{n_k - 1}{i_k - 1} x_k^{i_k} (1-x_k)^{n_k - i_k} \times B^{\boldsymbol{N}\setminus\boldsymbol{k}}_{\boldsymbol{I}\setminus\boldsymbol{k}}(\boldsymbol{x})$$

With the change of variable $i'_k = i_k - 1$,

$$\begin{array}{rcll} p(\boldsymbol{x}) & = & \displaystyle\sum_{\substack{(\boldsymbol{I}\setminus k) \leq (\boldsymbol{N}\setminus k) \\ i'_k = 0, \ldots, n_k - 1}} \binom{n_k - 1}{i'_k} x_k^{i'_k + 1} (1-x_k)^{n_k - 1 - i'_k} \times B^{\boldsymbol{N}\setminus\boldsymbol{k}}_{\boldsymbol{I}\setminus\boldsymbol{k}}(\boldsymbol{x}) & \\ & = & x_k \times \displaystyle\sum_{\substack{(\boldsymbol{I}\setminus k) \leq (\boldsymbol{N}\setminus k) \\ i'_k = 0, \ldots, n_k - 1}} \underbrace{\binom{n_k - 1}{i'_k} x_k^{i'_k} (1-x_k)^{n_k - 1 - i'_k}}_{b^{n_k - 1}_{i'_k}} \times B^{\boldsymbol{N}\setminus\boldsymbol{k}}_{\boldsymbol{I}\setminus\boldsymbol{k}}(\boldsymbol{x}) & \textit{we recognize } b^{n_k - 1}_{i'_k} \\ & = & x_k \times \displaystyle\sum_{\substack{(\boldsymbol{I}\setminus k) \leq (\boldsymbol{N}\setminus k) \\ i'_k = 0, \ldots, n_k - 1}} b^{n_k - 1}_{i'_k} \times B^{\boldsymbol{N}\setminus\boldsymbol{k}}_{\boldsymbol{I}\setminus\boldsymbol{k}}(\boldsymbol{x}) & \end{array}$$

Finally, by the partition-of-unity property of the Bernstein basis of degree $\boldsymbol{N'} \triangleq (n_1, ..., n_k - 1, ..., n_l)$, we obtain

$$\begin{aligned} p(\boldsymbol{x}) = x_k \quad \times \quad & \underbrace{\sum_{\boldsymbol{I} \leq \boldsymbol{N'}} B_{\boldsymbol{I}}^{\boldsymbol{N'}}(\boldsymbol{x})}_{1} \\ = x_k \quad \times \quad & \end{aligned}$$

□

Thanks to Lemma 1, we can relate the set $\mathcal{V}'$ of control points to the Bernstein coefficients of f'. For each indeterminate $x_1, \ldots, x_l$, we associate the Bernstein coefficients given in Lemma 1. Similarly, we associate to f' its Bernstein coefficients $c_{\boldsymbol{I}}$ as follows:

$$\forall \boldsymbol{x} \triangleq (x_1, \ldots, x_l) \in [0,1]^l, \quad \begin{pmatrix} x_1 \\ \cdots \\ x_l \\ f'(x) \end{pmatrix} = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} \begin{pmatrix} \frac{i_1}{n_1} \\ \cdots \\ \frac{i_l}{n_l} \\ c_{\boldsymbol{I}} \end{pmatrix} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x}) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} v'_{\boldsymbol{I}} \, B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x}) \tag{7}$$

As a consequence of properties 6 and 5, $\sum_{\boldsymbol{I} \leq \boldsymbol{N}} v'_{\boldsymbol{I}} \, B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x})$ is a convex combination of the points $v'_{\boldsymbol{I}}$. Thus, Equality 7 means that any point of f', say $(x_1, ..., x_l, f'(\boldsymbol{x}))$, can be expressed as a convex combination of the control points $v'_{\boldsymbol{I}}$. Therefore, $(x_1, ..., x_l, f'(\boldsymbol{x}))$ belongs to $\mathcal{P}'_{\mathcal{V}}$, the convex hull of the points $v'_{\boldsymbol{I}}$. It means that $\mathcal{P}'_{\mathcal{V}}$ is an over-approximation of the set $\left\{(x_1, ..., x_l, f'(\boldsymbol{x})) \mid \boldsymbol{x} \in [0,1]^l\right\}$.

We can now make a link between $\mathcal{P}'_{\mathcal{V}}$ and f: let us define $\mathcal{P}_{\mathcal{V}}$ as the convex hull of the elements of

$$\begin{array}{rl} \mathcal{V} = & \{v_{\boldsymbol{I}} \mid \boldsymbol{I} \leq \boldsymbol{N}\} \\ \textit{where} & \boldsymbol{I} = (i_1, \ldots, i_l) \\ & \boldsymbol{N} = (n_1, \ldots, n_l) \\ & v_{\boldsymbol{I}} = \left(\sigma_1\left(\frac{i_1}{n_1}\right), ..., \sigma_l\left(\frac{i_l}{n_l}\right), c_{\boldsymbol{I}}\right) \end{array}$$

The following lemma makes a link between the coefficients of the indeterminates $x_1, \ldots, x_l$ in $\mathfrak{B}$ and their correspondance through σ.

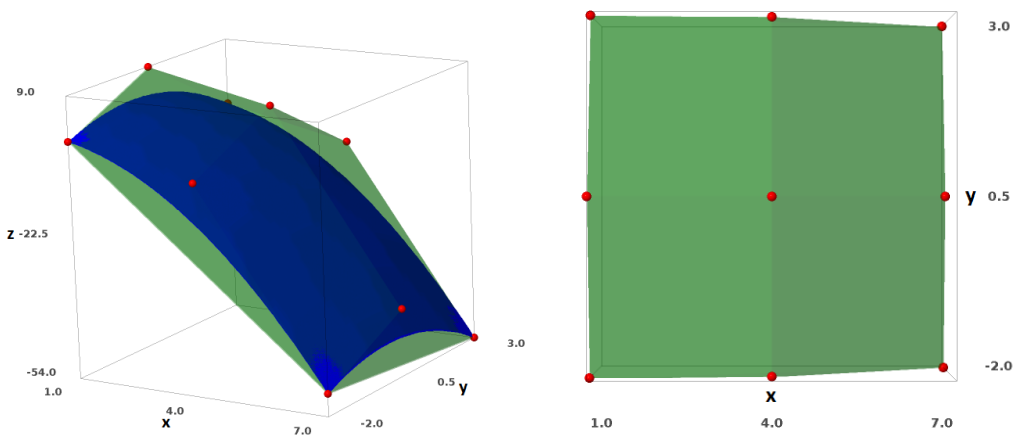
Lemma 2. *Let $k \in \mathbb{N}$, $1 \leq k \leq l$. Let $\sigma_k : t \longmapsto \alpha t + \beta$, $\alpha, \beta \in \mathbb{R}$ be an affine function.*

$$\sigma_k(x_k) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} \sigma_k\left(\frac{i_k}{n_k}\right) B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x}), \quad \boldsymbol{x} \in [0,1]^l$$

Proof. Let $\boldsymbol{x} \in [0,1]^l$ and consider the Bernstein polynomial $p(\boldsymbol{x}) \triangleq \sum_{\boldsymbol{I} \leq \boldsymbol{N}} \sigma_k\left(\frac{i_k}{n_k}\right) B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x})$. We will show that $p(\boldsymbol{x}) = \sigma_k(x_k)$.

$$\begin{aligned} p(\boldsymbol{x}) &= \sum_{\boldsymbol{I} \leq \boldsymbol{N}} \sigma_k\left(\tfrac{i_k}{n_k}\right) B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x}) \\ &= \sum_{\boldsymbol{I} \leq \boldsymbol{N}} \left(\alpha \tfrac{i_k}{n_k} + \beta\right) B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x}) \\ &= \alpha \left(\sum_{\boldsymbol{I} \leq \boldsymbol{N}} \tfrac{i_k}{n_k} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x})\right) + \beta \left(\sum_{\boldsymbol{I} \leq \boldsymbol{N}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x})\right) \\ &= \alpha \left(\sum_{\boldsymbol{I} \leq \boldsymbol{N}} \tfrac{i_k}{n_k} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x})\right) + \beta && \text{by the Property 6} \\ &= \alpha x_k + \beta && \text{by Lemma 1} \\ &= \sigma_k(x_k) \end{aligned}$$

□



(a) (b)

Figure 9 – Representation in different points of view of the polyhedron $\mathcal{P}_\mathcal{V}$ in green. The blue surface corresponds to the polynomial $f = -x^2 - y^2 + 4$. The red points are the points of $\mathcal{V}$.

As previously, Lemma 2 allows us to link the set $\mathcal{V}$ to the Bernstein coefficients of f. Recalling that by definition of σ, $\forall x \triangleq (x_1, ..., x_l) \in [0,1]^l,\ f(\sigma(\boldsymbol{x})) = f'(\boldsymbol{x})$, we obtain that

$$\begin{pmatrix} \sigma_1(x_1) \\ ... \\ \sigma_l(x_l) \\ f(\sigma(\boldsymbol{x})) \end{pmatrix} = \begin{pmatrix} \sigma_1(x_1) \\ ... \\ \sigma_l(x_l) \\ f'(\boldsymbol{x}) \end{pmatrix} = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} \begin{pmatrix} \sigma_1\left(\frac{i_1}{n_1}\right) \\ ... \\ \sigma_l\left(\frac{i_l}{n_l}\right) \\ c_{\boldsymbol{I}} \end{pmatrix} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x}) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} v_{\boldsymbol{I}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x}) \tag{8}$$

Following the same reasoning as above, $\sum_{\boldsymbol{I} \leq \boldsymbol{N}} v_{\boldsymbol{I}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{x})$ is a convex combination of the control points $v_{\boldsymbol{I}}$. Thus, equality 8 implies that any point of f, say $(\sigma_1(x_1), ..., \sigma_l(x_l), f(\sigma(\boldsymbol{x})))$, can be expressed as a convex combination of the $v_{\boldsymbol{I}}$. That's why $(\sigma_1(x_1), ..., \sigma_l(x_l), f(\sigma(\boldsymbol{x})))$ belongs to $\mathcal{P}_\mathcal{V}$, the convex hull of the $v_{\boldsymbol{I}}$. It means that $\mathcal{P}_\mathcal{V}$ is an over-approximation of the set $\{(\sigma_1(x_1), ..., \sigma_l(x_l), f(\sigma(\boldsymbol{x}))) \mid (x_1, ..., x_l) \in [0,1]^l\}$ which is equal to the set $\{(x_1, ..., x_l, f(\boldsymbol{x})) \mid (x_1, ..., x_l) \in P_{box}\}$

Example 7. With our running example, $\mathcal{P}_\mathcal{V}$ is the convex hull of the red points shown on Figure 9(a), where we can see that the surface $\{(x_1, ..., x_l, f(x_1, ..., x_l)) \mid (x_1, ..., x_l) \in P_{box}\}$ is clearly inside $\mathcal{P}_\mathcal{V}$. Moreover, Figure 9(b) shows that the points of $\mathcal{V}$ form a l-dimensional mesh.

Recall that we are looking for a polyhedron over-approximating $P \sqcap f \geq 0$. We have built the polyhedron $\mathcal{P}_\mathcal{V}$ that contains the set $\{(x_1, ..., x_l, f(x_1, ..., x_l)) \mid (x_1, ..., x_l) \in P_{box}\}$, but we are looking for an over-approximation of $\{(x_1, ..., x_l) \in P \mid f(x_1, ..., x_l) \geq 0\}$. In order to approximate $f \geq 0$, consider the polyhedron

$$\mathcal{P}_\mathcal{V}^+ = \mathcal{P}_\mathcal{V} \cap \{(x_1, ..., x_l, x_{l+1}) \in \mathbb{R}^{l+1} \mid x_{l+1} \geq 0\}$$

$\mathcal{P}_\mathcal{V}^+$ is the intersection of $\mathcal{P}_\mathcal{V}$ with the half space where x_{l+1} is positive. It contains the set $\{(x_1, ..., x_l, f(x)) \mid (x_1, ..., x_l) \in P_{box},\ f(x_1, ..., x_l) \geq 0\}$.

We compute $\mathcal{P}_{\mathcal{V}\,/x_{l+1}}^+$, the projection of $\mathcal{P}_\mathcal{V}^+$ on the variable x_{l+1}. $\mathcal{P}_{\mathcal{V}\,/x_{l+1}}^+$ is a polyhedron over-approximating the set $\{(x_1, ..., x_l) \in P_{box} \mid f(x_1, ..., x_l) \geq 0\}$. Knowing that $P \subset P_{box}$, we finally calculate $P' \triangleq \mathcal{P}_{\mathcal{V}\,/x_{l+1}}^+ \sqcap P$ to obtain the approximation of $\{(x_1, ..., x_l) \in P \mid f(x_1, ..., x_l) \geq 0\}$. P' is the effect of the guard $f \geq 0$ on the polyhedron P, as defined in the introduction.

Example 8. The representation of $\mathcal{P}_\mathcal{V}^+$ is given for our example on Figure 10. $\mathcal{P}_{\mathcal{V}\,/x_{l+1}}^+$ and P' are represented on Figure 11.

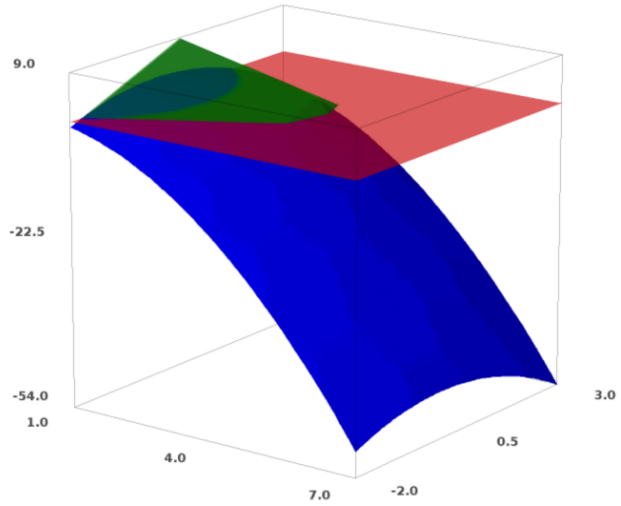
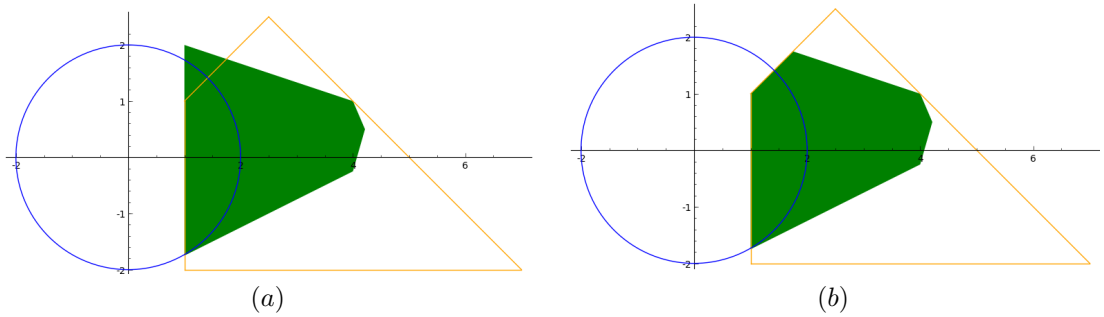


Figure 10 – Representation of the polyhedron $\mathcal{P}_{\mathcal{V}}^{+}$ in green. The blue surface corresponds to the polynomial $f = -x^2 - y^2 + 4$. The red surface is the plane $x_{l+1} = 0$.



(a) (b)

Figure 11 – The starting polyhedron $S = \{(x, y) \mid x - 1 \geq 0,\ y + 2 \geq 0,\ x - y \geq 0,\ -x - y + 5 \geq 0\}$ is represented in orange. The circle $\{x^2 + y^2 \leq 4\}$ is drawn in blue. (a): Representation of $\mathcal{P}_{\mathcal{V}/x_{l+1}}^{+}$ in green. (b): Representation of P' in green. It over-approximates the effect of the guard $x^2 + y^2 \leq 4$ on P.

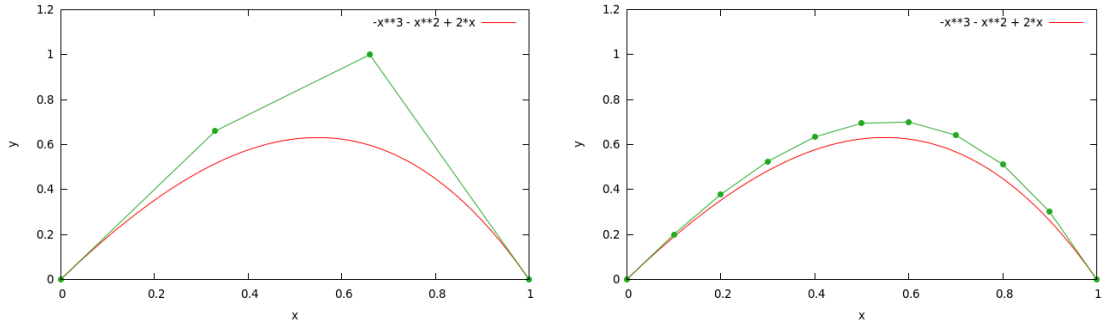


Figure 12 – Representation of $f(x) = -x^3 - x^2 + 2x$ in red. (a): $\mathcal{P}_\mathcal{V}$ deduced from a polynomial of degree 3. (b): $\mathcal{P}_\mathcal{V}$ deduced from a polynomial of degree 10.

3.3 Polyhedron refinement

There exists two methods to improve the approximation precision: degree elevation and interval splitting. The goal of both of them is to to find points that are closer to f.

Degree elevation consists in converting f from $\mathfrak{C}$ to a Bernstein basis of higher degree. Thereby, the set $\mathcal{V}$ contains more points, and as they are regularly spaced, they are closer to f. Thus, their convex hull $\mathcal{P}_\mathcal{V}$ is also closer to f, as shown on Figure 12.

The principle of interval splitting is to split the starting box $[a_1, b_1] \times ... \times [a_l, b_l]$ around a point in some dimensions. This process gives k different boxes , $k = 2^l$ if we split in every dimension. For each of these boxes, we get an expression of f in the basis $\mathfrak{B}$, and we obtain k times more points in $\mathcal{V}$. Note that we don't need to convert f from $\mathfrak{C}$ to $\mathfrak{B}$ for each box, we can deduce the expression of f in $\mathfrak{B}$ on another box directly from its expression in the original one thanks to an algorithm by De Casteljau [12]. The difficulty is to find the good points where to split.

The main drawback of these methods is that although they refine the over-approximating polyhedron $\mathcal{P}_\mathcal{V}$, they increase the number of its faces as well. A polyhedron with many faces leads to more computation.

3.4 Toward certification in Coq

In the following, we give some clues about the certification in Coq of the Bernstein linearization method. Note that we do not need to prove the change of basis in Coq. We can do it externally, and give the result as a certificate to Coq. This certificate is composed of the coefficients $c_\boldsymbol{I}$ of f in $\mathfrak{B}$ and the mapping function σ. Then, some remaining properties have to be verified in Coq:

(1) $\forall \boldsymbol{x} \in [0,1]^l,\ f(\sigma(\boldsymbol{x})) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} c_\boldsymbol{I} B_\boldsymbol{I}^\boldsymbol{N}(\boldsymbol{x})$. This can be done by expanding both sides of the equality, and compare each monomial coefficient.

(2) The properties 5 and 6 of the Bernstein basis.

(3) $\mathcal{P}_\mathcal{V}$ is an over-approximation of $\{(x_1, ..., x_l, f(\boldsymbol{x})) \mid (x_1, ..., x_l) \in P_{box}\}$. This is done by proving in Coq the Lemmas 1 and 2.

Remark that the properties (2) and (3) require to be proven once for all, while (1) needs to be proven for each linearization. Once the set of control points $\mathcal{V}$ is known, we can build a certified polytope P' that approximates the guard using the certified operators provided by the Verasco/Verimag Polyhedra Library developed by Fouilhé [5]. This library deals with polyhedra represented as conjunction of constraints, whereas we have obtained a representation of $\mathcal{V}$ as a set of vertices. The construction of P' can be obtained by adding the control points one after the other to the empty polyhedron, then computing the intersection with the initial polyhedron P. Although this algorithm is correct in principle, its cost is prohibitive: the

union of a polyhedron with a polyhedron reduced to a point is the heaviest operator of the VPL. Some work is still needed to switch to a certified constraint polyhedron at a reasonable cost.

4 Linearization on polytopes using Handelman representation

In this section, we explain how to exploit Handelman's Theorem [6] as a new linearization technique. This theorem gives a characterization of positive polynomials over a compact set. This kind of description is usually called a positivstellensatz. As said in the introduction, we focus on the treatment of a guard $f \geq 0$ where f denotes a polynomial in the $(x_1, \ldots, x_n)$ variables of the program.

Consider a compact polytope $P = \{(x_1, \ldots, x_n) \mid C_1 \geq 0, \ldots, C_p \geq 0\}$ where C_i are linear polynomials over $(x_1, \ldots, x_n)$. Suppose P describes the possible values of $(x_1, \ldots, x_n)$ at a program point before the guard, we seek a polyhedron that approximates $P \wedge f \geq 0$. We will use as a running example

$$P = \{(x, y) \mid x - 1 \geq 0,\ y + 2 \geq 0,\ x - y \geq 0,\ -x - y + 5 \geq 0\} \quad \text{and} \quad f = 4 - x^2 - y^2.$$

The affine approximation problem A naive call $P \sqcap f \geq 0$ to the intersection operator of the polyhedral domain would return P not exploiting the constraint $f \geq 0$ which is not affine. Our approximation problem is to find an affine constraint $\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n$, denoted by $\mathit{aff}(f)$, such that $P \Rightarrow \mathit{aff}(f) > f$ meaning that $\mathit{aff}(f)$ bounds f on the polyhedron P. By transitivity of $\geq$ we will conclude that $P \wedge f \geq 0 \Rightarrow P \wedge \mathit{aff}(f) > 0$. Thus, $P \sqcap \mathit{aff}(f) > 0$ will be a polyhedral approximation of the program state after the polynomial guard $f > 0$.

4.1 Handelman representation of positive polynomials

Notations Following the notations defined in Sect.3.1.1, let $\boldsymbol{I} = (i_1, ..., i_p) \in \mathbb{N}^p$ be a multi-index. Let us define the set of Handelman products

$$\mathscr{H}_P = \left\{C_1^{i_1} \times \cdots \times C_p^{i_p} \mid (i_1, \ldots, i_p) \in \mathbb{N}^p\right\} \textit{ where } P \triangleq C_1 \geq 0 \wedge \ldots \wedge C_p \geq 0$$

This set contains all products of constraints C_i of P. Given a multi-index $\boldsymbol{I} = (i_1, \ldots, i_p)$, $H^{\boldsymbol{I}} \triangleq \prod_{j=1}^{p} C_j^{i_j}$ denotes an element of $\mathscr{H}_P$. Note that the $H^{\boldsymbol{I}}$ are positive polynomials on P as products of positive constraints of P.

Example 9. Considering our running example, $H^{(0,2,0,0)} = (y + 2)^2$, $H^{(1,0,1,0)} = (x - 1)(x - y)$ and $H^{(1,0,0,3)} = (-x - y + 5)^3(x - 1)$ belongs to $\mathscr{H}_P$.

The Handelman representation of a positive polynomial $Q(\boldsymbol{x})$ on P is

$$Q(\boldsymbol{x}) = \sum_{\boldsymbol{I} \in \mathbb{N}^p} \underbrace{\lambda_I}_{\geq 0} \underbrace{H^{\boldsymbol{I}}}_{\geq 0} \textit{ with } \lambda_{\boldsymbol{I}} \in \mathbb{R}^+$$

This representation is used in mathematics as a certificate ensuring that $Q(\boldsymbol{x})$ is positive on P. Obviously if a polynomial can be written in this form, then it is necessarily positive on P. Handelman's theorem [6], that we summarize here, concerns the non trivial opposite implication:

Theorem 1 (Handelman's Theorem)**.** *Let $P = \{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid C_1 \geq 0, \ldots, C_p \geq 0\}$ be a compact polytope where each C_i is a linear polynomial over $\boldsymbol{x} = (x_1, \ldots, x_n)$. Let $Q(\boldsymbol{x})$ be positive polynomial on P. Then there exists $\lambda_{\boldsymbol{I}} \in \mathbb{R}^+$ and $H^{\boldsymbol{I}} \in \mathscr{H}_P$ such that*

$$Q(\boldsymbol{x}) = \sum_{\boldsymbol{I} \in \mathbb{N}^p} \lambda_{\boldsymbol{I}} H^{\boldsymbol{I}}$$

Usually, the Handelman representation of a polynomial $Q(\boldsymbol{x})$ is used to determine a constant lower bound of $Q(\boldsymbol{x})$ on P thanks to Schweighofer's algorithm [8] that focuses on iteratively improving the bound by increasing the degree of the $H^{\boldsymbol{I}}$. In this chapter we present another use of Handelman's theorem: we are not interested in just one (tight) bound but in a polytope wrapping the polynomial $Q(\boldsymbol{x})$.

4.2 Handelman approximation as a Parametric Linear Optimization Problem

We are looking for an affine constraint $\mathit{aff}(f)$, such that $\mathit{aff}(f) \geq f$ on P, which is equivalent to $\mathit{aff}(f) - f \geq 0$ on P. Then, Handelman's theorem applies:

> *The polynomial $\mathit{aff}(f) - f$ which is positive on the polytope P has an Handelman representation as a positive linear combination of products of the constraints of P, i.e.,*
>
> $$\mathit{aff}(f) - f = \sum_{\boldsymbol{I} \in \mathbb{N}^p} \lambda_{\boldsymbol{I}} H^{\boldsymbol{I}} \qquad \text{with } \lambda_{\boldsymbol{I}} \in \mathbb{R}^+,\ H^{\boldsymbol{I}} \in \mathscr{H}_P \tag{9}$$

The relation 9 of Handelman's theorem ensures that there exists some positive combinations of f and some $H^{\boldsymbol{I}} \in \mathscr{H}_P$ that remove the monomials of total degree >1 and lead to affine forms:

$$\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n = \mathit{aff}(f) = 1 \cdot f + \sum_{\boldsymbol{I} \in \mathbb{N}^p} \lambda_{\boldsymbol{I}} H^{\boldsymbol{I}}$$

Note that the polynomials of $\mathscr{H}_P$ are generators of the positive polynomials on P but they do not form a basis. Indeed, it is possible to have one $H \in \mathscr{H}_P$ being a positive linear combination of other elements of $\mathscr{H}_P$.

Example 10. Consider $P = \{(x, y) \mid x \geq 0, y \geq 0, x - y \geq 0, x + y \geq 0\}$.
Then, $H_{(2,0,0,0)} \triangleq x^2$, $H_{(1,1,0,0)} = xy$, $H_{(0,2,0,0)} = y^2$, $H_{(0,0,0,2)} = (x + y)^2$ belongs to Handelman products and they are not independent.
Indeed, $H_{(0,0,0,2)} = x^2 + 2xy + y^2 = H_{(2,0,0,0)} + 2H_{(1,1,0,0)} + H_{(0,2,0,0)}$

As a consequence, a positive polynomial $\mathit{aff}(f) - f$ can have several Handelman representations, even on a given set of Handelman products. Actually, we exploit the non-uniqueness of representation to get a precise approximation of the guard: we look for many affine constraints $\mathit{aff}(f)$ that bound f on P. Their conjunction forms a polyhedron that over-approximates f on P.

We now explain how the determination of all affine constraints bounding f can be expressed as a Parametric Linear Optimization Problem (PLOP).

Notations Given a multi-index $\boldsymbol{I}$ and n variables $x_1, ..., x_n$, let $\boldsymbol{x}^{\boldsymbol{I}}$ be the monomial $x_1^{i_1} \times ... \times x_n^{i_n}$. We define the total degree of the monomial $\boldsymbol{x}^{\boldsymbol{I}}$ as $\deg(\boldsymbol{I}) = \sum_{j=1}^{n} i_j$. For monomial of degree ≤ 1 we simply write $\alpha_i x_i$ instead of $\alpha_{\boldsymbol{I}}\, \boldsymbol{x}^{\boldsymbol{I}}$ when $\boldsymbol{I} = (0, \ldots, 0, 1, 0, \ldots, 0)$ with 1 in its i^{th} coordinate. With this settings, the relation 9 can be rephrased:

> *For some choice of $\lambda_{\boldsymbol{I}}$, the coefficient $\alpha_{\boldsymbol{I}}$ of the monomial $\boldsymbol{x}^{\boldsymbol{I}}$ in the polynomial $f + \sum_{\boldsymbol{I} \in \mathbb{N}^p} \lambda_{\boldsymbol{I}}\, H^{\boldsymbol{I}}$ is null for all multi-index $\boldsymbol{I}$ with $\deg(\boldsymbol{I}) > 1$.*

Now, let $\boldsymbol{d_f} \in \mathbb{N}^n$ be the maximal degree of the monomials of f. We restrict our search to finding a Handelman representation of $\mathit{aff}(f) - f$ on the subset $\{H_1, \ldots, H_q\}$ of all the Handelman products of degree $\leq \boldsymbol{d_f}$, instead of the whole set $\mathscr{H}_P$. If we fail with monomials of degree $\leq d$ we increase d. Handelman's theorem ensures that we will eventually succeed.

Example 11. With $f = 4 - x^2 - y^2$, $\boldsymbol{d_f} = 2$. Therefore, we shall consider the 15 following Handelman products:

$H_1 = H_{(0,0,0,0)} = 1$ $\quad H_2 = H_{(1,0,0,0)} = x - 1$ $\quad H_3 = H_{(0,1,0,0)} = y + 2$
$H_4 = H_{(0,0,1,0)} = x - y$ $\quad H_5 = H_{(0,0,0,1)} = -x - y + 5$ $\quad H_6 = H_{(2,0,0,0)} = (x - 1)^2$
$H_7 = H_{(0,2,0,0)} = (y + 2)^2$ $\quad H_8 = H_{(0,0,2,0)} = (x - y)^2$ $\quad H_9 = H_{(0,0,0,2)} = (-x - y + 5)^2$
$H_{10} = H_{(1,1,0,0)} = (x - 1)(y + 2)$ $\quad H_{11} = H_{(1,0,1,0)} = (x - 1)(x - y)$ $\quad H_{12} = H_{(1,0,0,1)} = (x - 1)(-x - y + 5)$
$H_{13} = H_{(0,1,1,0)} = (y + 2)(x - y)$ $\quad H_{14} = H_{(0,1,0,1)} = (y + 2)(-x - y + 5)$ $\quad H_{15} = H_{(0,0,1,1)} = (x - y)(-x - y + 5)$

With the restriction to $\{H_1, \ldots, H_q\}$, finding the λ_I can be formulated as a PLOP.
The relation 9 on $\{H_1, \ldots, H_q\}$ amounts to find positive $\lambda_1, \ldots, \lambda_q \in \mathbb{R}^+$ such that

$$\begin{array}{ccccc} \mathit{aff}(f) & = & 1 \cdot f + \sum_{i=1}^{q} \lambda_i H_i & = & \underbrace{(\overbrace{\lambda_f}^{=1}, \lambda_1, \ldots, \lambda_q)}_{\boldsymbol{\lambda}} \cdot \underbrace{(f, H_1, \ldots, H_q)^\intercal}_{\mathcal{H}_f \cdot \boldsymbol{m}^\intercal} \\ \| & & \| & & \\ \alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n & & \boldsymbol{\lambda} \cdot \mathcal{H}_f \cdot \boldsymbol{m}^\intercal & & \\ \| & & \| & & \\ \boldsymbol{m} \cdot (\alpha_0, \ldots, \alpha_n, 0, \ldots, 0)^\intercal & = & \boldsymbol{m} \cdot \mathcal{H}_f{}^\intercal \cdot \boldsymbol{\lambda}^\intercal & & \end{array}$$

where:

- $\mathcal{H}_f$ is the matrix of the coefficients of f and the H_i in the canonical basis of monomials of degree $\leq \boldsymbol{d_f}$ denoted by $\boldsymbol{m} = (1, x_1, \ldots, x_n, \boldsymbol{x}^{\boldsymbol{d_1}}, \ldots, \boldsymbol{x}^{\boldsymbol{d_f}})$
- the vector $\boldsymbol{\lambda}$ characterizes the Handelman's positive combination of f and the H_i
- we associated a coefficient $\lambda_f = 1$ to f just to get convenient notations.

Finally, the problem can be rephrased as finding $\boldsymbol{\lambda} \in \{1\} \times (\mathbb{R}^+)^q$ such that

$$\mathcal{H}_f{}^\intercal \cdot \boldsymbol{\lambda}^\intercal = (\alpha_0, \ldots, \alpha_n, 0, \ldots, 0)^\intercal$$

The result of the matrix-vector product $\mathcal{H}_f{}^\intercal \boldsymbol{\lambda}^\intercal$ is a vector $\boldsymbol{\alpha} \triangleq (\alpha_0, \alpha_1, \ldots, \alpha_n, \alpha_{\boldsymbol{d_1}}, \ldots, \alpha_{\boldsymbol{d_f}})$ that represents the constraint $\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n + \sum_{I \leq \boldsymbol{d_f}} \alpha_I \boldsymbol{x}^I$ in the $\boldsymbol{m}$ basis. Since we seek an affine constraints $\mathit{aff}(f)$ we are interested in finding $\boldsymbol{\lambda}$ such that $\mathcal{H}_f{}^\intercal \boldsymbol{\lambda}^\intercal = (\alpha_0, .., \alpha_n, 0, .., 0)^\intercal$. Each such $\boldsymbol{\lambda}$ gives an affine constraint $\mathit{aff}(f)$ that bounds f on P. Therefore, the conjunction of all constraints $\mathit{aff}(f) \geq 0$ form a polyhedron $\mathcal{A}_f$ that approximates the guard $f \geq 0$ on P.

Example 12. Here is the transposed matrix $\mathcal{H}_f{}^\intercal$ of f and our 15 Handelman products with respect to the basis $\boldsymbol{m} = (1, x, y, xy, x^2, y^2)$

	f	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8	H_9	H_{10}	H_{11}	H_{12}	H_{13}	H_{14}	H_{15}
1	*4*	*1*	*-1*	*2*	*0*	*5*	*1*	*4*	*0*	*25*	*-2*	*0*	*-5*	*0*	*10*	*0*
x	*0*	*0*	*1*	*0*	*1*	*-1*	*-2*	*0*	*0*	*-10*	*2*	*-1*	*6*	*2*	*-2*	*5*
y	*0*	*0*	*0*	*1*	*-1*	*-1*	*0*	*4*	*0*	*-10*	*-1*	*1*	*1*	*-2*	*3*	*-5*
xy	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*-2*	*2*	*1*	*-1*	*-1*	*1*	*-1*	*0*
x^2	*-1*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*1*	*1*	*0*	*1*	*-1*	*0*	*0*	*-1*
y^2	*-1*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*0*	*0*	*0*	*-1*	*-1*	*1*

With $\lambda_f = \lambda_6 = \lambda_7 = 1$ and every other $\lambda_i = 0$, we obtain

$$\boldsymbol{\alpha}^\intercal = \mathcal{H}_f{}^\intercal \boldsymbol{\lambda}^\intercal = \begin{pmatrix} 9 \\ -2 \\ 4 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{matrix} 1 \\ x \\ y \\ xy \\ x^2 \\ y^2 \end{matrix}$$

Thus, $\mathit{aff}(f) = -2x+4y+9$ is a constraint that bounds f on P, as shown on the Figure 13(a). Indeed, we can see that the plane $-2x+4y+9$ is above the polynomial f. Figure 13(b) shows the approximation of f that we obtain with the constraint $-2x + 4y + 9 \geq 0$.

In order to obtain a better approximation, we need to find others affine approximations bounding f on P. For instance, with $\lambda_f = \lambda_8 = 1$, $\lambda_{11} = \lambda_5 = 2$, and every other $\lambda_i = 0$, $\boldsymbol{\alpha}^\intercal = (21, -2, -8, 0, 0, 0)$. The Figure 14 shows the approximation of f with the two constraints $-2x + 4y + 9 \geq 0$ and $-2x - 8y + 21 \geq 0$.

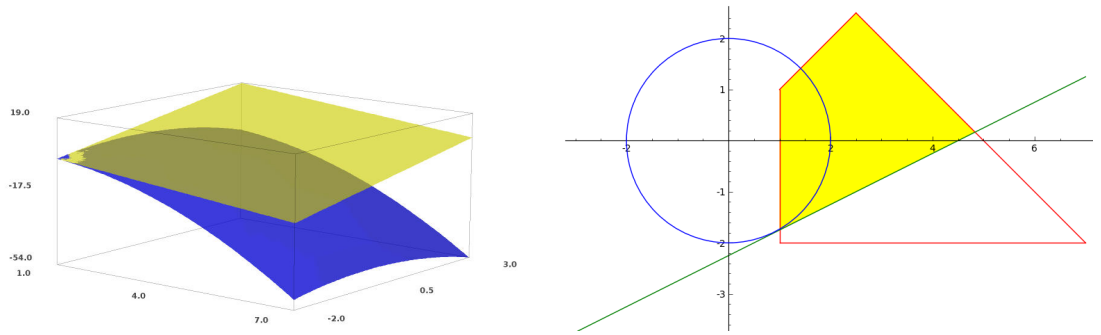


Figure 13 – (a): The surface $z = f(x,y) \triangleq -x^2 - y^2 + 4$ is in blue, the plane $z = aff(f)(x,y) \triangleq -2x + 4y + 9$ in yellow. (b): The representation of the polyhedron $P \triangleq \{(x,y) \mid x - 1 \geq 0,\ y + 2 \geq 0,\ x - y \geq 0,\ -x - y + 5 \geq 0\}$ is in red and the area of the circle in blue represents $\{(x,y) \mid f(x,y) \geq 0\}$. The green line is the constraint $-2x + 4y + 9 = 0$. The yellow area is the polyhedron $P \sqcap \{(x,y) \mid -2x + 4y + 9 \geq 0\}$ that over-approximates the conjunction $P \wedge f \geq 0$.

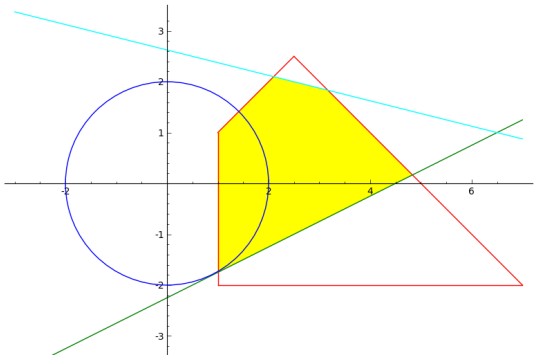


Figure 14 – The material is that of Figure 13 augmented with a cyan line which corresponds to the constraint $-2x - 8y + 21 = 0$. The yellow area is the polyhedron $P \sqcap \{(x,y) \mid -2x + 4y + 9 \geq 0,\ -2x - 8y + 21 \geq 0\}$ that over-approximates the conjunction $P \wedge f \geq 0$.

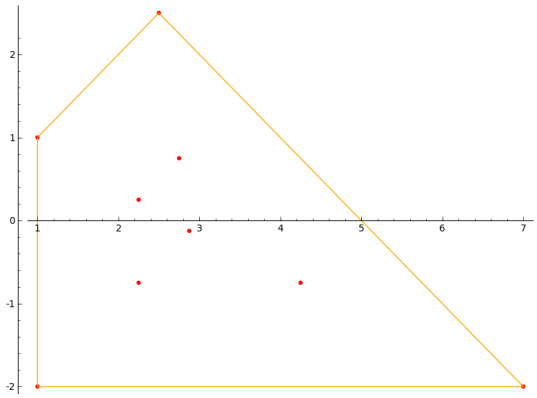


Figure 15 – The 9 points p_k selected to instantiate the parametric simplex. The polyhedron $P = \{(x, y) \mid x - 1 \geq 0,\ y + 2 \geq 0,\ x - y \geq 0,\ -x - y + 5 \geq 0\}$ is in orange.

The Parametric Linear Optimization Problem Finding all and the tightest approximations $\underline{aff}(f)$ that bounds f on P can now be expressed as the following PLOP which can be solved using a Parametric Simplex in the spirit of [4].

Given a set $\{H_1, \ldots, H_q\} \subseteq \mathscr{H}_P$ of Handelman products,

minimize *$\underline{aff}(f)$, that is, $\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n$*
under the constraints

$$\begin{cases} \mathcal{H}_f{}^{\intercal}(\lambda_f, \lambda_1, \ldots, \lambda_q)^{\intercal} = (\alpha_0, \ldots, \alpha_n, 0, \ldots, 0)^{\intercal} \\ \lambda_f = 1 \\ \lambda_i \geq 0,\ i = 1..q \end{cases}$$

where

- *The $\lambda_1, \ldots, \lambda_q$ are the variables of the PLOP.*
- *The $\alpha_0, \ldots, \alpha_n$ are kept for the sake of presentation ; in practise they are substituted by their expression issued from $\mathcal{H}_f{}^{\intercal}\boldsymbol{\lambda}^{\intercal}$.*
- *The $x_1, \ldots, x_n$ are the parameters of the PLOP.*

Each instantiation $(\underline{x}_1, \ldots, \underline{x}_n)$ of the parameters defines a standard Linear Optimization Problem which can be solved by the simplex algorithm, providing the optimum associated to the given parameters. We don't have an implementation of the parametric simplex yet ; we plan to reuse or implement a version of Feautrier's Parametric Simplex, called PIP (for Parametric Integer Programming) [4]. For experimentations in the meantime, we generate several instantiations of the PLOP and execute a standard simplex on each. Each run gives an $\underline{aff}(f)$ constraint and their conjunction forms the final polytope $\mathcal{A}_f$. The points $(\underline{x}_1, \ldots, \underline{x}_n)$ used for instanciation are:

- The vertices v_i of the inital polyhedron P
- The centroid of these vertices (called *isobarycentre* in French)
- Each point p_k satisfying $2\overrightarrow{p_k v_k} + \sum_{j \neq k} \overrightarrow{p_k v_j} = \overrightarrow{0}$. The point p_k is the barycenter of the points $(v_1, ..., v_n)$ where we associate the weight 2 to one vertex and 1 to the others.

Example 13. Back to our example, we show in Figure 15 the points p_k chosen to instantiate the parametric simplex. The result of the linearization is represented on Figure 16. We can see that one constraint is redundant in the sense that it does not constrain the resulting polyhedron.

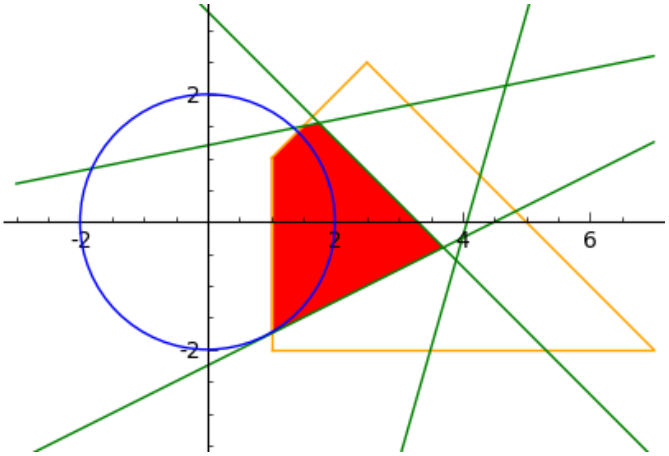


Figure 16 – The polyhedron $P = \{(x, y) \mid x - 1 \geq 0,\ y + 2 \geq 0,\ x - y \geq 0,\ -x - y + 5 \geq 0\}$ is drawn in orange. The area delimited by the blue circle represents the guard $\{(x, y) \mid x^2 + y^2 \leq 4\}$. The red surface is the approximation of $\{(x, y) \mid x^2 + y^2 \leq 4\} \cap P$. The green lines are the results of the parametric simplexes with points of Figure 15.

Applications to other abstract domains The PLOP can be adapted to produce constraints of other abstract domains such as Difference Bound Matrices (DBM) or octagons.

DBM is an abstract domain represented by linear constraints of the form $x_i - x_j \leq c$, with x_i and x_j two distinct variables and c a constant. We can enforce solutions of the PLOP to be DBM constraints by adding constraints that make the general template of the objective $\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n$ be of the form $\alpha_0 + \alpha_i x_i + \alpha_j x_j$, $i \neq j$, $\alpha_i = 1$ and $\alpha_j = -1$. For this purpose, we introduce for each α_k two boolean variables p_k and m_k that model the fact that α_k is positive or negative:

$$p_k = \begin{cases} 1 & \text{if } \alpha_k > 0 \\ 0 & \text{if } \alpha_k \leq 0 \end{cases} \quad m_k = \begin{cases} 1 & \text{if } \alpha_k < 0 \\ 0 & \text{if } \alpha_k \geq 0 \end{cases}$$

We finally end with a Mixed Boolean/Rational PLOP:

minimize $\mathit{aff}(f)$, that is, $\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n$
under the constraints

$$\mathcal{H}_f{}^{\intercal}(\lambda_f, \lambda_1, \ldots, \lambda_q)^{\intercal} = (\alpha_0, \ldots, \alpha_n, 0, \ldots, 0)^{\intercal} \tag{10}$$

$$\lambda_f = 1 \tag{11}$$

$$\alpha_i \leq p_i,\ i = 1, ..., n \tag{12}$$

$$-\alpha_i \leq m_i,\ i = 1, ..., n \tag{13}$$

$$p_i + m_i = 1,\ i = 1, ..., n \tag{14}$$

$$\sum_{i=0}^{n} p_i = 1 \tag{15}$$

$$\sum_{i=0}^{n} m_i = 1 \tag{16}$$

$$-1 \leq \alpha_i \leq 1,\ i = 1, ..., n \tag{17}$$

$$\sum_{i=0}^{n} \alpha_i = \sum_{i=0}^{n} p_i - \sum_{i=0}^{n} m_i \tag{18}$$

$$p_i, m_i \in \{0, 1\},\ \alpha_i \in \mathbb{Q},\ i = 1, ..., n \tag{19}$$

$$\lambda_i \in \mathbb{Q}^+,\ i = 1, ..., q \tag{20}$$

A solution of this PLOP leads to one p_i equal to 1, one m_j ($i \neq j$) equal to 1, and every other p_k and m_k equal to 0, which is encoded by the equations 15 and 16. The equations 12, 13 and 17 ensures that $\alpha_k = 1 \Rightarrow p_k = 1$ and $\alpha_k = -1 \Rightarrow m_k = 1$. Equation 14 ensures that for a given k, p_k and m_k are opposite literals, that is $p_k = \neg m_k$. Note that we do not need α_k to be exactly 1 or -1, actually we need $|\alpha_i| = |\alpha_j|$. Indeed, if $\alpha_i = -\alpha_j$, then $\mathit{aff}(f) = \alpha_0 + \alpha_i x_i + \alpha_j x_j$ is equivalent to $\mathit{aff}(f) = \frac{\alpha_0}{\alpha_i} + x_i - x_j$. In the case of DBM, we can therefore solve the problem with $-1 \leq \alpha_k \leq 1$, $\alpha_k \in \mathbb{Q}$ instead of $\alpha_k \in \{-1, 0, 1\}$. Equation 18 captures the fact that $\alpha_i = -\alpha_j$.

The PLOP can also be adapted to the abstract domain of octagons [10], characterized by linear constraints of the form $\pm x_i \pm x_j \leq c$ with x_i and x_j two distinct variables and c a constant. In this case, we allow the following patterns: $\alpha_i = \alpha_j = 1$ or $\alpha_i = \alpha_j = -1$ or $\alpha_i = 1, \alpha_j = -1$. In order to obtain solutions of this form, one has to replace the equations 15 and 16 by $\sum_{i=0}^{n} p_i + m_i = 2$.

4.3 Toward certification in Coq

In this part, we give intuitions about the certification in Coq of the linearization using Handelman.

We want to prove that $\mathit{aff}(f) > f$, or equivalently that $\mathit{aff}(f) - f > 0$, on the polytope P. Given indexes $(i_{1,1}, ..., i_{1,p}), \ldots, (i_{q,1}, ..., i_{q,p})$ and the coefficients $\lambda_1, .., \lambda_q$, we have to check in Coq that:

(1) $f + \sum_{j=1}^{q} \lambda_j C_1^{i_{j,1}} \ldots C_p^{i_{j,p}}$ is linear. This can be done by expanding the polynomial and looking at the coefficient of monomials of degree > 1.

(2) The product of positive polynomials is positive. This can be proven in Coq once for all. We obtain finally that $f + \sum_{j=1}^{q} \lambda_j C_1^{i_{j,1}} \ldots C_p^{i_{j,p}}$ is a linear over-approximation of f.

The computation of $\mathit{aff}(f)$ does not have to be done in Coq. Thus, $(i_{1,1}, ..., i_{1,p}), \ldots, (i_{q,1}, ..., i_{q,p})$ and $\lambda_1, .., \lambda_q$ are used as a certificate.

5 Comparison of the linearization methods and future work

We have implemented and proved in Coq the linearization algorithm based on intervalization. The Bernstein and Handelman methods have been implemented in SAGE. In order to compare the three methods, we realised a simple analyzer in SAGE. It is able to handle C programs containing guards, assignments and if-then-else but no function call, no loop. Given a starting polyhedron P and a list of statements s, the analyzer computes the effect of s on P with the three techniques. The intervalization algorithm is performed by an Ocaml program obtained by automatic extraction from our Coq development. This Ocaml code is called by the SAGE script. We are then able to measure and compare the volume of the resulting polyhedra using an existing SAGE library.

Comparison We realised experimentations on statements taken from the satellite code. In general, intervalization is the fastest but the less accurate of the three methods. Bernstein's method can be as accurate as needed, but at the price of an high algorithmic cost. Handelman's method is about as accurate as the Bernstein one. Up to now, it is the most expensive method, mainly because it is new, we implemented it in a naive way and no attention has been paid to improve the computations as it has been done, for decades, for Bernstein approximations.

The Bernstein approximation relies on the interval where ranges each variable. Extracting the interval of a variable from a polyhedron requires to solve two linear optimization problems to get the maximum and minimum value of the variable in the polyhedron. This overhead is avoided in Handelman's method which reasons directly on the constraints of the polyhedron. Hence, the Bernstein's method is convenient when the polyhedron is in fact an hypercube – that is the product of the interval of each variable – whereas the Handelman's method is promising at program point associated with a general polyhedron. Specifically, we think the Handelman's method can be more suitable in terms of precision, even in complexity, in case

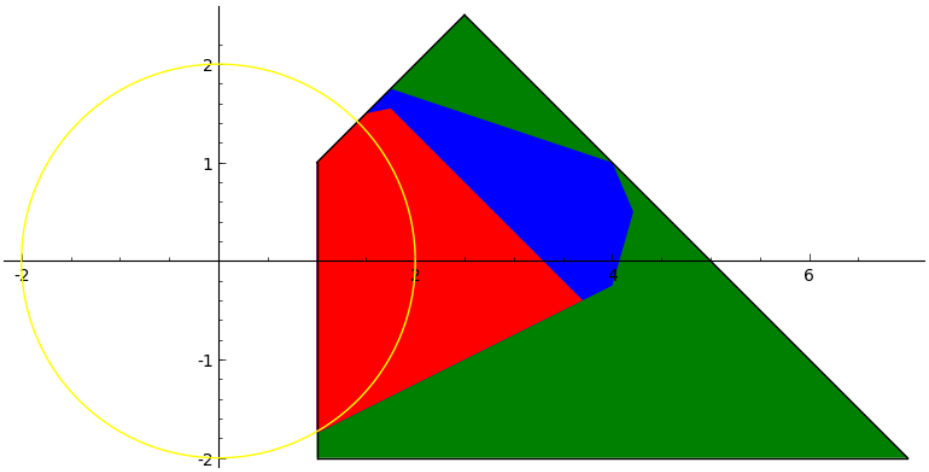


Figure 17 – Representation of the effect of the guard $\{(x,y) \mid x^2+y^2 \leq 4\}$ *(yellow circle) on the polyhedron* $\{(x,y) \mid x-1 \geq 0,\ y+2 \geq 0,\ x-y \geq 0,\ -x-y+5 \geq 0\}$ *(black outline). The green surface is the result of the linearization using intervalization. The blue one is the result of Bernstein's linearization. The red one is the linearization with the Handelman's method.*

of successive linearizations. Indeed, where the Bernstein's method stacks approximation errors at each new linearization, the Handelman one does not degrade. Moreover, in order to certify these methods, the Bernstein one requires to switch from the polyhedron representation using vertices to the polyhedron representation by linear constraints, which is not the case for Handelman.

In practice, the three linearization methods can be combined: analysis is an iterative process that switches to finer methods when the cheapest ones failed to prove correctness of the program. We can imagine starting an analysis with intervals which are cheap and deal with non-linear expressions. Then, switching to the domain of polyhedra if more precision is required. This second phase can reuse the intervals computed by the first one and apply intervalization or Bernstein's linearization without paying the overhead of extracting intervals. This time the analysis associates polyhedra to program points. Then, to gain more precision, a third phase can run the analysis with Handelman's linearization. Other combinations are possible and Handelman can be directly used at any phase since a product of bounded interval is a special case of polytope, called an hypercube.

We show on Figure 17 the results of the three methods to approximate the guard $\{(x,y) \mid x^2+y^2 \leq 4\}$ on $P \triangleq \{(x,y) \mid x-1 \geq 0,\ y+2 \geq 0,\ x-y \geq 0,\ -x-y+5 \geq 0\}$. We can see that intervalization is not precise enough to approximate the guard. Indeed, the resulting polyhedron is the same as the initial one, and the guard does not add any information in the analysis.

We compute Bernstein's method without any interval splitting or degree elevation. Even without any refinement process, Bernstein is more accurate than intervalization but slower.

Handelman's polyhedron is the most precise of the three techniques. However, its execution time is for now significantly slower compared to the other ones. Handelman's polyhedron has been computed by instantiating the parametric simplex with a large number of points - nine in this example. It means that nine simplexes are performed to approximate the guard. Moreover, we chose as subset $\{H_1, \ldots, H_q\}$ the 15 possible products of constraints of P of degree ≤ 2, meaning that we are faced with a Linear Optimization Problem with 15 variables. Industrial linear solvers are able to deal with hundreds of variables, but this is obviously the shortfall of Handelman's linearization.

Future work Along the document we identified several points that still need work prior to the integration of our linearization methods in the VERASCO analyser. We review them quickly and sketch direction of improvement.

Unbounded polyhedron Up to now, we considered only the linearization with Bernstein or Handelman on polytopes. Indeed, the Bernstein basis is defined only on $[0,1]^l$ and Handelman's theorem applies on a compact polyhedron. However, we do not necessarily have full bounds on each variable of the polynomial expression f, therefore we need to be able to manipulate unbounded polyhedra. We already treat this case for intervalization, it was done thanks to the Coq type $\mathbb{Z} \cup \bot$. There exists a method to handle partially unbounded intervals during the transformation of a polynomial into the Bernstein basis [12]. It is based on a bijective transformation of the partially unbounded polynomial to a bounded one which has the same sign. Handelman's theorem addresses only compact polyhedra. Obviously, if a polynomial grows non-linearly in the unbounded direction, it cannot be bounded by any affine function. However, the growth of f with respect to the unbounded variable can be bounded, *e.g.* $f(x) = 1 - x^2$. The adaptation of Handelman's method to this case is an open question.

Certification in Coq Recall that as a part of VERASCO project, the linearization techniques need to be certified. We have done this for the intervalization algorithm, but it still requires to be included in the VERASCO analyzer. The two other linearization methods have not been certified yet. We shall certify them in Coq using certificates, meaning that parts of the computations are done outside of Coq. As mentioned in Sect 3.4, we do not want to compute the change to the Bernstein basis inside Coq. We shall make these computations in an external program producing hints, called certificates, that can be used in Coq to prove the correction of the result. Similarly, we do not plan to implement the Handelman's method in Coq but to use certificates that drive the verification in Coq that $\mathit{aff}(f)$ is an affine constraint and an approximation of f.

Bernstein's Linearization Several improvements can be performed on our Bernstein implementation. First, the change of basis can be done in $\mathcal{O}(n^{l+1})$ - instead of $\mathcal{O}(n^{2l})$ - thanks to a method called the *Matrix method* [14]. Second, as said in Sect 3.3, the refinement method of interval splitting can be done efficiently. The goal is to avoid computing the change of basis for each sub-box. Thereby, we could implement interval splitting with an algorithm by De Casteljau [12] that recursively applies linear interpolations.

Handelman's Linearization The main improvements of Handelman's linearization that we shall work on are:

- **The choice of the subset** $\{H_1, \ldots, H_q\}$. On the one hand, considering a lot of H_i allows lots of Handelman's representations, therefore an improved accuracy. On the other hand, each new H_i adds a variable λ_i in the simplex. In order to minimize the number of H_i to consider, starting with a small subset $\{H_1, \ldots, H_q\}$, we could imagine an incremental approach that adds new H_i when no solution is found. We must pay attention to the algorithm in order to exploit the computations of the previous attempt.

- **A proper implementation of the parametric simplex.** The points that we choose to instantiate our parametric simplex are selected arbitrarily. The difficulty is to instantiate only with points that give the best approximations. Instead, we plan to exploit existing tools for solving Parametric Linear Optimization Problems, such as Paul Feautrier's tool PIP [4]. Instead of instantiating the problem with some particular points, PIP splits the parameter space into polyhedral regions and associates to each region a parametric solution (*i.e.* a linear combination of the parameters). Some work is needed to turn our Handelman problem in the PIP format and to correctly interpret the solution. In particular, it is not obvious that PIP can find the best approximations of f since PIP is designed for lexicographic enumeration, meaning that the definition in PIP of the optimization objective is not that of standard optimization problems.

Experiments in the large Once all the three linearization techniques are integrated in the VERASCO analyzer, series of code benchmarks and testings shall be realised. Then, combination of the three methods shall be adjusted, as well as the heuristics we are using. Indeed, we have to adapt our linearization techniques depending on the type of code we want to analyse. For instance, the satellite code contains lots of

sums of squares, thus we must adjust our heuristics to be more effective on this kind of expressions. Similarly, if an expression appears many times in the program, we should pay a special attention to linearize it precisely.

References

[1] Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. the 16th International Static Analysis Symposium, 5673:309–325, December 2009. 2.1.1

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 238–252. ACM, 1977. 1.2.1

[3] Rida T. Farouki. The bernstein polynomial basis: A centennial retrospective. Computer Aided Geometric Design, 29(6):379–419, 2012. 3.1.1, 3.2

[4] Paul Feautrier. Parametric integer programming. RAIRO Recherche opérationnelle, 22(3):243–268, 1988. 4.2, 5

[5] Alexis Fouilhé, David Monniaux, and Michaël Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In SAS2013, 2013. 1.2.2, 2.1.1, 3.4

[6] David Handelman. Representing polynomials by positive linear functions on compact convex polyhedra. Pac. J. Math, 132(1):35–62, 1988. 4, 4.1

[7] Xavier Leroy. A formally verified compiler back-end. Journal of Automated Reasoning, 43(4):363–446, 2009. 2.1.3

[8] Markus Schweighofer. An algorithmic approach to schmüdgen's positivstellensatz. Elsevier Preprint, June 2001. 4.1

[9] The Coq development team. The Coq proof assistant reference manual. LogiCal Project, 2004. Version 8.0. 2.3

[10] Antoine Miné. The octagon abstract domain. Higher-Order and Symbolic Computation, 19(1):31–100, 2006. 4.2

[11] Antoine Miné. Symbolic methods to enhance the precision of numerical abtract domains. the 7th International Conference on Verification, Model Checking and Abstract Interpretation, 3855:348–363, January 2006. 1.3, 2, 2.1.1, 2.1.2, 2.1.3

[12] César Muñoz and Anthony Narkawicz. Formalization of a representation of Bernstein polynomials and applications to global optimization. Journal of Automated Reasoning, 51(2):151–196, August 2013. 3.1.2, 3.3, 5, 5

[13] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pages 4–13. ACM, 1991. 2.1.3

[14] Shashwati Ray and P. S. V. Nataraj. A matrix method for efficient computation of bernstein coefficients. Reliable Computing, 17(1):40–71, 2012. 3.1.1, 3.1.2, 3.1.2, 5

A Interval elimination: the general case

A.1 Interval elimination in conditions (general case)

We present the interval elimination for a general condition containing an inequality of the form

$$\begin{array}{c}\mathsf{if}\ (\ell_{itv} \leq \ell'_{itv})\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\\ \text{with } \ell_{itv} = \sum [a_i, b_i]{\times}x_i \text{ and } \ell'_{itv} = \sum [a'_i, b'_i]{\times}x'_i\end{array}$$

As explained previously, the linearization of the condition $e_1 \leq e_2$ leads to two conditions which do not exclude each other. Hence, the code provided to the analyzer is non deterministic.

$$\mathsf{alt}\left\{\begin{array}{ll}\mathsf{if} & \left(\begin{array}{l}\exists\lambda_1 \in [a_1, b_1], ..., \lambda_n \in [a_n, b_n],\\ \exists\lambda'_1 \in [a'_1, b'_1], ..., \lambda'_n \in [a'_n, b'_n],\\ \sum \lambda_i x_i \leq \sum \lambda'_i x'_i\end{array}\right) \mathsf{then}\ S_1\\ \mathsf{if} & \left(\begin{array}{l}\exists\lambda_1 \in [a_1, b_1], ..., \lambda_n \in [a_n, b_n],\\ \exists\lambda'_1 \in [a'_1, b'_1], ..., \lambda'_n \in [a'_n, b'_n],\\ \neg\left(\sum \lambda_i x_i \leq \sum \lambda'_i x'_i\right)\end{array}\right) \mathsf{then}\ S_2\end{array}\right.$$

As in the example, the existential quantifiers are eliminated using equivalences 1 and 2. The conditions of the previous code become

- $\min\left(\sum [a_i, b_i]{\times}x_i\right) \leq \max\left(\sum [a'_i, b'_i]{\times}x'_i\right)$ for the first alternative.
- $\max\left(\sum [a_i, b_i]{\times}x_i\right) \leq \min\left(\sum [a'_i, b'_i]{\times}x'_i\right)$ for the second one.

Remark that whatever are the signs of the x_i,

$$\min\left(\sum_i [a_i, b_i]{\times}x_i\right) = \sum_i \min([a_i, b_i]{\times}x_i) \text{ and } \max\left(\sum_i [a_i, b_i]{\times}x_i\right) = \sum_i \max([a_i, b_i{\times}x_i)$$

The conditions are now

- $\sum min\left([a_i, b_i]{\times}x_i\right) \leq \sum max\left([a'_i, b'_i]{\times}x'_i\right)$ for the first alternative.
- $\sum max\left([a_i, b_i]{\times}x_i\right) > \sum min\left([a'_i, b'_i]{\times}x'_i\right)$ for the second one.

Thus, the initial program is translated into the following linear program:

$$\mathsf{alt}\left\{\begin{array}{l}\mathsf{if}\ \left(\sum min\left([a_i, b_i]{\times}x_i\right) \leq \sum max\left([a'_i, b'_i]{\times}x'_i\right)\right)\ \mathsf{then}\ S_1\\ \mathsf{if}\ \left(\sum max\left([a_i, b_i]{\times}x_i\right) > \sum min\left([a'_i, b'_i]{\times}x'_i\right)\right)\ \mathsf{then}\ S_2\end{array}\right.$$

Remark that knowing the sign of x_i, $min\left([a_i, b_i]{\times}x_i\right)$ and $max\left([a_i, b_i]{\times}x_i\right)$ can be simplified in either $a_i * x_i$ or $b_i * x_i$. As a consequence, the previous program would not contain any interval anymore. We shall detail this process in Section A.3.

A.2 Interval elimination in assignments (general case)

Going back to the program of Example 3, the interval that appears in the assignment on line 3 needs to be eliminated. The value of z after the assignment $z := [1, 3]{\times}y$ is between y and $3{\times}y$. This corresponds to the constraint $y \leq z \leq 3{\times}y$ since $y \geq 0$. The assignment on line 3 is then replaced by $z :\overset{ND}{=} \{\tilde{z} \mid y \leq \tilde{z} \leq 3{\times}y\}$.

The interval elimination of an assignment of a linear interval expression to a variable, *e.g.* $x := \sum [a_i, b_i]x_i$ is

$$x :\overset{ND}{=} \left\{\tilde{x} \mid \sum min\left([a_i, b_i]{\times}x_i\right) \leq \tilde{x} \leq \sum max\left([a_i, b_i]{\times}x_i\right)\right\}$$

As explained in the previous section, the final step for interval elimination is to maximize or minimize expressions. This requires to know the sign of some sub-expressions, as we shall see in the next section.

$$x := f \xrightarrow{lin} \textsf{if } (x_1 \geq 0) \begin{cases} \textsf{if } (x_n \geq 0) \begin{cases} \textsf{then} & x \overset{ND}{:=} \{\tilde{x} \mid \ell_1 \leq \tilde{x} \leq \ell'_1\} \\ \textsf{else} & x \overset{ND}{:=} \{\tilde{x} \mid \ell_2 \leq \tilde{x} \leq \ell'_2\} \end{cases} \\ \textsf{then...} \\ \textsf{else...} \\ \textsf{if } (x_n \geq 0) \begin{cases} \textsf{then} & x \overset{ND}{:=} \{\tilde{x} \mid \ell_{k-1} \leq \tilde{x} \leq \ell'_{k-1}\} \\ \textsf{else} & x \overset{ND}{:=} \{\tilde{x} \mid \ell_k \leq \tilde{x} \leq \ell'_k\} \end{cases} \end{cases}$$

Figure 18 – Shape of the result of an assignment linearization

A.3 Expression minimization and maximization

We have seen previously how to handle inequalities. This requires to minimize or maximize expressions, it can be done with two different techniques. Both of them rely on knowledge of sub-expression's sign. The first one consists in expanding the expression, this is the method we have chosen. The particularity of this one is that it requires only the sign of variables. This process is based on the following equalities :

- $\min([a,b]\times x) = \textsf{if } (x \geq 0) \begin{cases} \textsf{then} & \min([a,b]) * x \\ \textsf{else} & \max([a,b]) * x \end{cases}$
- $\max([a,b]\times x) = \textsf{if } (x \geq 0) \begin{cases} \textsf{then} & \max([a,b]) * x \\ \textsf{else} & \min([a,b]) * x \end{cases}$

For instance, assume we want to maximize the expression $f := [-1,2](x + [1,5])$. By expanding f, the code produced is

$$max([-1,2](x+[1,5])) \longrightarrow max([-1,2]x + [-1,2][1,5]) \longrightarrow \textsf{if } (x \geq 0) \begin{cases} \textsf{then} & 2x + 10 \\ \textsf{else} & -x + 10 \end{cases}$$

The second method uses the non-deterministic structure Alt, as defined in Section 2.1.3, to determine the sign of sub-expressions. Applied on f, this technique would give the following code:

$$max([-1,2](x+[1,5])) \longrightarrow \textsf{alt} \begin{cases} \textsf{if } (max(x+[1,5]) \geq 0) \textsf{ then } 2 \times \max(x+[1,5]) \\ \textsf{if } (min(x+[1,5]) < 0) \textsf{ then } -1 \times \min(x+[1,5]) \end{cases}$$

$$\equiv \textsf{alt} \begin{cases} \textsf{if } ((x+5) \geq 0) \textsf{ then } 2 \times (x+5) \\ \textsf{if } ((x+1) < 0) \textsf{ then } -1 \times (x+1) \end{cases}$$

In both cases, the expression maximization or minimization leads to the creation of a tree of Alt or if_then_else whose size depends on the expression size. In particular, the first method allows to bound the size of the tree with the number of non-intervalized variable. This can be interesting in case of a complex expression, or for a conjunction or disjunction. This is detailed in Appendix B. Another reason for our choice is that it makes the proof in Coq easier.

However, this choice implies a loss of precision. Indeed, it allows to eliminate one interval with several values at the same time, which was not possible before expansion. For instance in the expression $[-1,2]x + [-1,2][1,5]$, we can replace the first $[-1,2]$ by -1, and the second one by 2, which gives $-x + 2[1,5]$. This expression can give $-x + 10$ if we replace $[1,5]$ by 5. In the initial expression $[-1,2](x + [1,5])$, we could only replace [-1,2] by one value, and we could not obtain $-x + 10$. By expanding, we have allowed values that were not possible initially.

B Shape of the output of the algorithm

In this section, we summarize the different steps of the algorithm and show the shape of its output. As said previously, the interval elimination gives an expression containing max and min. The minimization

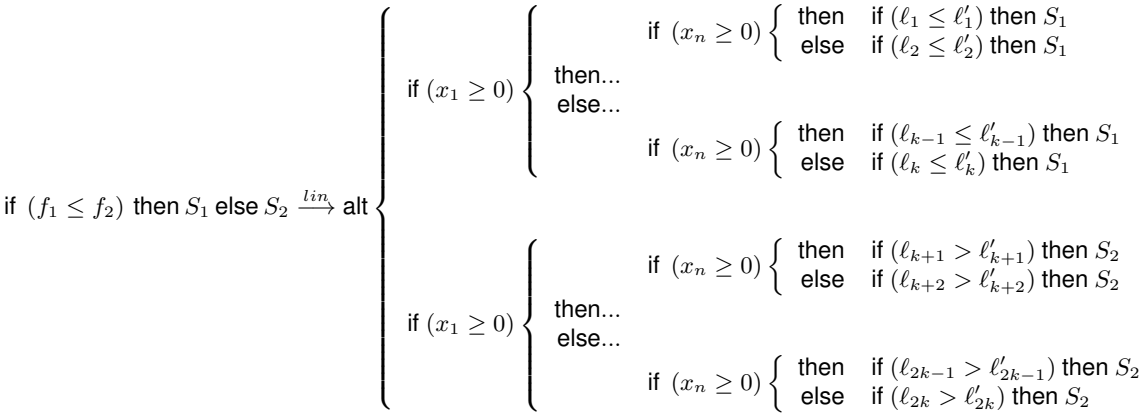


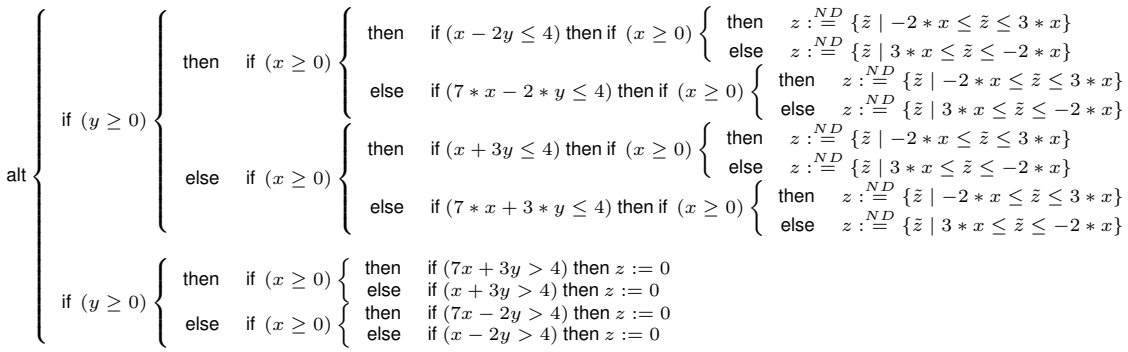
Figure 19 – Shape of the result of the linearization of if $(f1 \leq f2)$ then $S1$ else $S2$

and maximization of expressions require a knowledge about the sign of some variables, as detailed in Appendix A.3. This process leads to a tree where all possible sign of non-intervalized variables are treated. That's why the resulting GCL code is a tree whose nodes are assumptions on the sign of variables.

- Figure 18 shows the result for an assignment linearization. As the interval elimination gives expression of the form $NDassign \sum min\left([a_i, b_i] \times x_i\right) \leq x \leq \sum max\left([a_i, b_i] \times x_i\right)$, the tree leaves have the form $x :\overset{ND}{=} \{\tilde{x} \mid \ell_i \leq \tilde{x} \leq \ell'_i\}$.

- Figure 19 shows the result of the linearization of a if-then-else. The interval elimination for this statement gives an alt structure. That's why the result is a tree starting with an alt. The top half of the tree approximates the part if $(f_1 \leq f_2)$ then S_1 whereas the bottom one approximates if $(f_1 > f_2)$ then S_2.

Example 14. The result of the linearization of the Example 1 is the following GCL code:



Remark that the condition $x^2 + y^2 \leq 4$ and the assignment $z = y * x$ are linearized independently - for the simplicity of the proof. As a consequence, even if we know that $y \geq 0$ in the first branch of the tree, the linearization of $z = y * x$ cannot use directly this assumption. That is the reason why we are doing twice the sign distinction for y. This can be easily improved by a post-processing algorithm which would cut the unfeasible branches. We would obtain the following code:

$$
\text{alt}\left\{\begin{array}{l}
\text{if } (y \geq 0)\left\{\begin{array}{ll}
\text{then} & \text{if } (x \geq 0)\left\{\begin{array}{ll}
\text{then} & \text{if } (x - 2y \leq 4) \text{ then } z :\overset{ND}{=} \{\tilde{z} \mid -2 * x \leq \tilde{z} \leq 3 * x\} \\
\text{else} & \text{if } (7 * x - 2 * y \leq 4) \text{ then } z :\overset{ND}{=} \{\tilde{z} \mid 3 * x \leq \tilde{z} \leq -2 * x\}
\end{array}\right. \\
\text{else} & \text{if } (x \geq 0)\left\{\begin{array}{ll}
\text{then} & \text{if } (x + 3y \leq 4) \text{ then } z :\overset{ND}{=} \{\tilde{z} \mid -2 * x \leq \tilde{z} \leq 3 * x\} \\
\text{else} & \text{if } (7 * x + 3 * y \leq 4) \text{ then } z :\overset{ND}{=} \{\tilde{z} \mid 3 * x \leq \tilde{z} \leq -2 * x\}
\end{array}\right.
\end{array}\right. \\
\text{if } (y \geq 0)\left\{\begin{array}{ll}
\text{then} & \text{if } (x \geq 0)\left\{\begin{array}{ll}
\text{then} & \text{if } (7x + 3y > 4) \text{ then } z := 0 \\
\text{else} & \text{if } (x + 3y > 4) \text{ then } z := 0
\end{array}\right. \\
\text{else} & \text{if } (x \geq 0)\left\{\begin{array}{ll}
\text{then} & \text{if } (7x - 2y > 4) \text{ then } z := 0 \\
\text{else} & \text{if } (x - 2y > 4) \text{ then } z := 0
\end{array}\right.
\end{array}\right.
\end{array}\right.
$$

Conjunctions, disjunctions and equalities Up to now, the only conditional expressions we considered contained inequalities. Let us now treat the case of conjunctions (disjunctions are handled similarly, by inverting $\wedge$ and $\vee$ in the following reasoning). First, given a conditional expression $\text{if } (f_1 \leq f_2 \wedge f_3 \leq f_4) \text{ then } S_1 \text{ else } S_2$, we compute the negation of the condition and get $(f_1 > f_2 \vee f_3 > f_4)$. Second, we apply the same linearization algorithm as detailed above. Thereby, we obtain a tree with the same shape as Figure 19, where leaves have the form $\text{if } (\ell_1 \leq \ell_2 \wedge \ell_3 \leq \ell_4) \text{ then } S_1$ or $\text{if } (\ell_1 > \ell_2 \vee \ell_3 > \ell_4) \text{ then } S_2$. Note that $(f_1 \leq f_2 \wedge f_3 \leq f_4)$ is treated as a single expression, in the sense that a single tree will be generated to treat this condition. It means that if a variable appears at the same time in f_1, f_2, f_3 and f_4, it will be treated only once. This is one reason why we have chosen to eliminate intervals by expanding expressions instead of making assumptions on bigger sub-expressions. Indeed, the second method explained in Appendix A.3 would give a larger tree if f_1, f_2, f_3 and f_4 did not share the same sub-expressions.

The case of equalities is treated simply by replacing $\text{if } (f_1 == f_2) \text{ then } S_1 \text{ else } S_2$ by $\text{if } (f_1 \leq f_2 \wedge f_2 \geq f_1) \text{ then } S_1 \text{ else } S_2$, then apply the linearization algorithm as explained above.

C Coq types

C.1 Type $\texttt{expr}_{\mathbb{Z}}$

```
Inductive var : Type := X : positive → var.

Inductive a_binopℤ : Type :=
| Oadd
| Omul.

Inductive a_exprℤ : Type :=
| Zconst_int : ℤ → a_exprℤ
| Zvar : var → a_exprℤ
| Zbinop : a_binopℤ → a_exprℤ → a_exprℤ → a_exprℤ .

Inductive b_binop : Type :=
| Ole
| Oge
| Olt
| Ogt.

Inductive b_exprℤ : Type :=
| ZBbinop : b_binop → a_exprℤ → a_exprℤ → b_exprℤ
| ZOR : b_exprℤ → b_exprℤ → b_exprℤ
| ZAND : b_exprℤ → b_exprℤ → b_exprℤ.

Inductive exprℤ : Type :=
| Aexprℤ : a_exprℤ → exprℤ
| Bexprℤ : b_exprℤ → exprℤ.
```

C.2 Type $\texttt{expr}_{\mathbb{Z}\cup\bot}$

```
Inductive a_expr_{Z∪⊥} : Type :=
| ZEconst_int : Z ∪ ⊥ → a_expr_{Z∪⊥}
| ZEvar : var → a_expr_{Z∪⊥}
| ZEbinop : a_binop_Z → a_expr_{Z∪⊥} → a_expr_{Z∪⊥} → a_expr_{Z∪⊥}.


Inductive b_expr_{Z∪⊥} : Type :=
| ZEBbinop : b_binop → a_expr_{Z∪⊥} → a_expr_{Z∪⊥} → b_expr_{Z∪⊥}
| ZEOR : b_expr_{Z∪⊥} → b_expr_{Z∪⊥} → b_expr_{Z∪⊥}
| ZEAND : b_expr_{Z∪⊥} → b_expr_{Z∪⊥} → b_expr_{Z∪⊥}.

Inductive expr_{Z∪⊥} : Type :=
| Aexpr_{Z∪⊥} : a_expr_{Z∪⊥} → expr_{Z∪⊥}
| Bexpr_{Z∪⊥} : b_expr_{Z∪⊥} → expr_{Z∪⊥}.
```

C.3 Type $\texttt{expr\_exp}_{\mathbb{Z}}$

```
Inductive monome : Type :=
  | MCste : Z → monome
  | MVar : var → monome
  | MP : monome → monome → monome.

Inductive expr_exp_Z : Type :=
  | Mm : monome → monomes
  | Madd : expr_exp_Z → expr_exp_Z → expr_exp_Z .
```

C.4 Type $\texttt{expr\_lin}_{itv(\mathbb{Z}\cup\bot)}$

```
Record itv : Type := ITV{ min:ZE ; max:ZE}.

Record itv_wf : Type := ITV_WF{ itvl:itv ; wf : min itvl <= max itvl}.

Inductive a_expr_lin_{itv(Z∪⊥)} : Type :=
| Dsum : a_expr_lin_{itv(Z∪⊥)} → a_expr_lin_{itv(Z∪⊥)} → a_expr_lin_{itv(Z∪⊥)}
| Ditv : itv_wf → a_expr_lin_{itv(Z∪⊥)}
| Dvaritv : var → itv_wf → a_expr_lin_{itv(Z∪⊥)}
| DAerror.

Inductive b_expr_lin_{itv(Z∪⊥)} : Type :=
| DOR : b_expr_lin_{itv(Z∪⊥)} → b_expr_lin_{itv(Z∪⊥)} → b_expr_lin_{itv(Z∪⊥)}
| DAND : b_expr_lin_{itv(Z∪⊥)} → b_expr_lin_{itv(Z∪⊥)} → b_expr_lin_{itv(Z∪⊥)}
| DBerror
| DBbinop : b_binop → a_expr_lin_{itv(Z∪⊥)} → a_expr_lin_{itv(Z∪⊥)} → b_expr_lin_{itv(Z∪⊥)}.
```

C.5 Type expr

```
Inductive expr : Type :=
| Expr_Z : expr_Z → expr
| Expr_{Z∪⊥} : expr_{Z∪⊥} → expr.
```

D Semantics of GCL

```
Inductive step : state → state → Prop :=
| step_skip_seq: ∀ (s:stmt) (k:cont) (v:valuation),
    step
      (State Sskip (Kseq s k) v)
```

```
      (State s k v)

| step_assign : ∀ (x:var) (e:a_expr) (k:cont) (v v':valuation),
    v' = (update v x (eval_a_expr e v)) →
    step
      (State (Sassign x e) k v)
      (State Sskip k v')

| step_seq: ∀ (s1:stmt) (s2:stmt) (k:cont) (v:valuation),
    step
      (State (Sseq s1 s2) k v)
      (State s1 (Kseq s2 k) v)

| step_ifthen: ∀ (e:expr) (s:stmt) (k:cont) (v:valuation),
    bool_of_expr e v true →
    step
      (State (Sifthen e s) k v)
      (State s k v)

| step_ifthenelse: ∀ e s1 s2 k v b,
    bool_of_expr e v b →
    step
      (State (Sifthenelse e s1 s2) k v)
      (State (if b then s1 else s2) k v)

| step_alt1 : ∀ (s1 s2 : stmt) (k : cont) (v : valuation),
    step
      (State (Salt s1 s2) k v)
      (State s1 k v)

| step_alt2 : ∀ (s1 s2 : stmt) (k : cont) (v : valuation),
    step
      (State (Salt s1 s2) k v)
      (State s2 k v)

| step_NDassign : ∀ (x:var) (z z_min z_min:ℤ ∪ ⊥)
    (e_min e_max :a_expr_{ℤ∪⊥} ) (k:cont) (v:valuation),
   z_min = (eval_a_expr_{ℤ∪⊥} e_min v) →
   z_max = (eval_a_expr_{ℤ∪⊥} e_max v) →
   z_min <= z → z <= z_max →
   step
      (State (SNDassign e_min x e_max) k v)
      (State (Sassign x (Aexpr_{ℤ∪⊥} (ZEconst_int z))) k v).
```