New Algorithmics for Polyhedral Calculus via Parametric Linear Programming

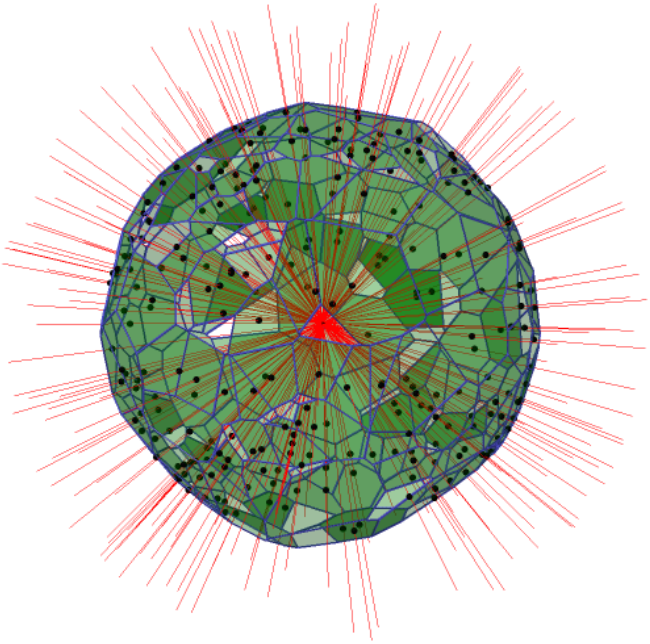
Alexandre Maréchal

Communauté
UNIVERSITÉ Grenoble Alpes

École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

Thèse pour obtenir le grade de
Docteur de l'université de Grenoble

Nouvelle Algorithmique pour le Calcul Polyédral via Programmation Linéaire Paramétrique



Alexandre Maréchal

Réalisée au laboratoire Verimag sous la direction de :

Dr. David Monniaux
CNRS

Dr. Michaël Périn
Université Grenoble Alpes

Soutenue publiquement le 11 Décembre 2017 devant le jury composé de :

Pr. Sylvain Conchon, Université Paris-Sud, Président

Pr. Antoine Miné, Université Pierre et Marie Curie - Paris 6, Rapporteur

Pr. Sriram Sankaranarayanan, University of Colorado Boulder, Rapporteur

Pr. Philippe Clauss, Université de Strasbourg, Examinateur

Dr. Charlotte Truchet, Université de Nantes, Examinatrice

Résumé

Cette thèse présente la nouvelle implémentation de la Verified Polyhedra Library (VPL), une bibliothèque efficace de calcul polyédral. Elle fournit des opérateurs certifiés en Coq, s'appliquant sur des représentations en contraintes. La version précédente souffrait d'inefficacité lors d'opérateurs cruciaux, à savoir l'élimination de variables et l'enveloppe convexe. Dans ce document, je présente des améliorations importantes qui bénéficient à la modularité, la simplicité et au passage à l'échelle de la bibliothèque : le processus de certification est généralisé et simplifié ; les conditions polynomiales sont maintenant traitées ; les calculs qui n'impliquent pas de certification sont effectués en flottants ; de nouveaux algorithmes sont fournis pour la minimisation de représentation et la détection d'égalités implicites.

D'un côté, l'implémentation d'un solveur de problèmes de programmation linéaire paramétrique (PLP) a mené à une meilleure efficacité tant en nombre de contraintes que de générateurs. L'élimination de variables et l'enveloppe convexe sont tous deux encodés en problème PLP. Le PLP est un outil générique possédant de nombreuses applications, et qui permet d'éviter la génération de redondances grâce à l'utilisation d'une contrainte de normalisation. De plus, nous proposons de nouveaux opérateurs pour la gestion des contraintes polynomiales, l'un d'entre eux étant également encodé en tant que problème PLP.

De l'autre, la certification de la bibliothèque a été grandement optimisée et simplifiée. La VPL suit un paradigme de vérification a posteriori, où les calculs non triviaux sont effectués par des oracles externes générant des témoins de correction. Ces témoins sont ensuite validés par un vérifieur écrit en Coq. Grâce à un cadre de certification puissant et innovant, le *polymorphic factory style* (PFS), la plupart des aspects délicats de la génération de témoins sont maintenant évitée. La souplesse du PFS est démontrée par la création d'une tactique en Coq qui découvre les égalités implicites en arithmétique linéaire.

Abstract

This thesis presents the design and implementation of the Verified Polyhedra Library (VPL), a scalable library for polyhedral calculus. It provides Coq-certified polyhedral operators that work on constraints-only representation. The previous version was inefficient on crucial operations, namely variable elimination and convex hull. In this work, I present major improvements that have been made in scalability, modularity and simplicity: The certification process is generalized and simplified; polynomial guards can now be handled; computations that do not involve certification may use floating-point numbers; new algorithms are presented for minimization and detection of implicit equalities.

On the one hand, the implementation of a solver for Parametric Linear Programming (PLP) problems led to improved scalability both in the dimension and in the number of constraints. Variable elimination and convex hull are now encoded as PLP. PLP is a generic tool that has many applications, and that avoids generating redundancies thanks to a normalization constraint. Additionally, we provide new operators for handling multivariate polynomials, one of which also encoded as a PLP problem.

On the other hand, the certification part of the library has been greatly optimized and simplified. The VPL follows a result-verification paradigm, where complex computations are performed by untrusted oracles that generate witnesses of correctness, themselves validated by a certified Coq checker. Thanks to an innovative and powerful certification framework known as Polymorphic Factory Style (PFS), most cumbersome parts of the witness generation are now avoided. The flexibility of PFS is attested by the creation of a Coq tactic for learning equalities in linear arithmetic.

Remerciements

Une fois le diplôme de Master en poche, j'avais le sentiment d'avoir acquis une certaine expérience. Avec le recul, je pense avoir plus appris pendant ces trois années que pendant tout le reste de ma scolarité. La thèse est une expérience unique, pleine de rebondissements et parsemée de surprises, de joies et parfois de peines. Mais le jeu en vaut la chandelle, et je remercie donc tous ceux qui m'ont épaulé tout au long de ce parcours.

Je remercie chaleureusement mon directeur de thèse, David. Véritable mine de connaissance, il a été d'une aide précieuse et ses conseils ont permis de remédier à bien des problèmes. Alors qu'on se débat devant un tableau, agitant frénétiquement un feutre dans l'espoir que recouvrir le blanc de noir nous aidera à expliquer un algorithme qu'on n'est pas sûr d'avoir compris, il apporte calmement la réponse au problème qu'on n'avait pas même identifié.

Je remercie également Michaël, qui a été un encadrant formidable. Toujours souriant, il apporte son soutien indéfectible dans toutes les situations. Adepte de l'art ancestral qu'est la peinture sur manuscrit, il reformule, flèche, barre et encadre, mélangeant sur sa palette toutes les teintes et nuances qu'on peut imaginer, suivant un code couleur compris de lui seul et qui évolue de page en page, comme mû par une volonté propre. La légende rapporte même l'existence d'un tiroir secret, verrouillé et couvert de formes géométriques convexes, dans lequel des grimoires oubliés seraient conservés, attendant d'être consciencieusement annotés par un stylo à la couleur indescriptible.

Je tiens à témoigner ma gratitude envers Sylvain pour m'avoir tenu la tête hors du marécage où Coqide m'avais plongé. Il est le représentant vivant du génie incompris qui, après une longue explication et affublé d'un demi-sourire, cherche d'un regard appuyé la compréhension dans l'oeil de son auditoire. Sa passion de l'élégance et de la précision mathématique se transmet à son contact. Une discussion amène parfois un moment de lucidité exacerbée dans lequel notre conscience semble contempler la raison profonde des choses. L'instant suivant, on n'est plus sûr de rien : ai-je fait un AVC ? À quoi sert la monade déjà ? Car Sylvain seul détient les clefs de ses découvertes.

Je remercie tout aussi chaleureusement Alexis pour sa patience face à mes nombreuses questions, et à mes débuts laborieux avec OCaml. C'est grâce à lui que la VPL existe, et je ne baignerais sûrement pas aujourd'hui dans la grâce du Saint Plaix sans son intervention.

Un grand merci au laboratoire Verimag pour m'avoir accueilli pendant tout ce temps. Merci également à tous les collègues doctorants qui ont vécu cette aventure avec moi ! J'ai une pensée particulière pour Denis avec qui j'ai partagé le café de tous les distributeurs du campus, et qui sourit toujours en voyant la marque de mon laptop. Malgré mes efforts, je ne suis jamais parvenu à lui retirer cette expression mêlant exaspération et bienveillance, similaire à celle d'un parent qui reçoit un collier de nouilles, qu'il arborait en lisant mon code C++. Je tiens aussi à remercier mes co-bureaux Anaïs, Valentin et Hang pour leur patience face aux nombreuses interjections qui émanaient pendant les périodes de programmation, et à mon pianotage incessant alors que j'écoutais de la musique. Merci aussi à mes amis du LJK : J.B. pour ses superbes solos de batteries ("Il manquait une croche là, non ?"), et David pour les parties de jeux de plateau et les discussions stratégiques sur Starcraft II.

Je remercie finalement Jessica, ma douce compagne, qui a rendu toutes ces aventures plus merveilleuses. Qu'il faille débugger un programme au milieu de la nuit ou se frayer péniblement un chemin dans les contrées hostiles de Lordran, le concept d'ennui n'existe pas à ses côtés ! Et quand l'empreinte de la manette est si imprimée dans le mur que le voisin pourrait jouer à notre place, on peut toujours se consoler en louant le soleil.

Contents

Foreword **9**

1 A Certified Domain of Polyhedra **11**
1.1 Checking Programs with Polyhedra 11
1.2 Certifying an Abstract Domain of Polyhedra 15
1.3 Solving Linear Programming Problems 25

I Toward a Scalable Polyhedral Domain **33**

2 Minimization by Raytracing **35**
2.1 Redundancy in Polyhedra 35
2.2 Certifying a Minimization of Polyhedra 38
2.3 An Efficient Minimization Algorithm 40
2.4 Experimental Results 46
2.5 Redundancy in the Double Description Framework 50

3 Minimizing Operators via Parametric Linear Programming **53**
3.1 Projection via Parametric Linear Programming 54
3.2 Polyhedron as Solution of a PLOP 56
3.3 Principle of a PLP solver 59
3.4 The Normalization Constraint 60
3.5 Experiments 64
3.6 Convex Hull via Parametric Linear Programming 67
3.7 Conclusion on PLP-Based Operators 71

4 Linearization **73**
4.1 Polyhedral Approximation of Polynomials 73
4.2 Intervalization 74
4.3 Bernstein's Linearization 80
4.4 Handelman's Linearization 89
4.5 Linearization in the VPL 99

5 Parametric Linear Programming Problem Solving **101**
5.1 Tree-Exploration Parametric Simplex 102
5.2 Algorithm by Generalization 104
5.3 Degeneracy 107

II Certification in Coq **109**

6 Introduction to Coq Certification **111**
6.1 The Struggle of Certification 111
6.2 The Coq Proof Assistant 112
6.3 Three Certification Approaches 115

7 Certification by Polymorphic Factories **119**
7.1 PFS Oracle Explained on Projection 120
7.2 Formalizing the Frontend in Coq 124
7.3 The Flexible Power of PFS Illustrated on Convex Hull 126
7.4 Related Works 130

8 A Coq Tactic for Equality Learning **133**
8.1 Specification of the VPL Tactic 134
8.2 Using the Tactic 135
8.3 The Reduction Algorithm 136
8.4 Conclusion 140

9 Certification of Handelman's Linearization **143**
9.1 Design of the Coq Checker 144
9.2 Computing Handelman Products in Coq 145
9.3 Coq Code of Handelman's Linearization 149

Conclusion **150**
Current Status of the VPL 152
Test Infrastructure 155
Toward a PLP Abstract Domain 156
Future Work 158

A Handelman's Heuristics: LP Problem Definition **159**

B Example of XML Result File **161**

Bibliography **165**

Index **173**

Foreword

This thesis presents the new design of the Verified Polyhedra Library (VPL), which implements an abstract domain of polyhedra. Polyhedra are sets of points delimited by affine constraints. They can be equivalently defined by their generators, *i.e.* their vertices in the bounded case. Written in OCaml, the VPL provides polyhedral operators that are certified in Coq, such as inclusion testing, intersection or convex hull. Polyhedra have many uses in program analysis, coming from their ability to express affine relations between variables. For instance, in compilation, the polytope model allows representing sets of reachable values for indexes in nested loops. The application we will be interested in is static analysis by abstract interpretation, which tries to establish the validity of assertions in programs.

The VPL was originally designed by Fouilhé (2015) during his PhD thesis, under the supervision of Michaël Périn and David Monniaux, at Verimag. It was part of the VERASCO project (Jourdan et al., 2015), which aimed at creating the first static analyzer fully certified in Coq. This project emerged from a need of the Coq-certified C compiler COMPCERT (Leroy, 2009). Roughly, this compiler produces a correct binary code provided that the source code is free of undefined behaviours, which are numerous in the C standard (Yang et al., 2011). The goal of VERASCO was to discard this assumption, by formally ensuring the absence of such behaviours.

The VPL was then created as a relational abstract domain for VERASCO. Thus, the library had to be certified in Coq. Most polyhedra libraries work in double description, using both generators and constraints. This framework has the advantage of taking the best of each side. But switching from one representation to the other is done by Chernikova's algorithm, which suffers from an exponential worst-case complexity in the size of inputs and outputs. Moreover, implementing a certified double description library would imply the proof of this algorithm, which is far from being straightforward. Instead, A. Fouilhé chose to focus on the constraints-only representation. A generators-only library could have been attempted, but it was avoided for two reasons. First, in abstract interpretation, it is common to see hypercubes, for which the generators representation has an exponential size in the number of variables. Second, operators are easy to certify in constraints-only, thanks to Farkas' lemma.

Fouilhé et al. (2013) evaluated the VPL efficiency in terms of execution time compared to other polyhedra libraries: the Parma Polyhedra Library (Bagnara et al., 2008) and the NEWPOLKA library, both available in APRON (Jeannet and Miné, 2009). They concluded that the overall performance of those libraries was similar. The expensive operators of constraints-only representation were found to be variable elimination and convex hull. They both rely on orthogonal projection, that was therefore the bottleneck of VPL, limiting the scalability of the whole library. Indeed, Fourier-Motzkin elimination is the standard algorithm for projection in constraints-only, and it has an exponential complexity in the number of eliminated variables. Thus, improving the projection operator is mandatory for the VPL to achieve scalability.

Contributions

My work takes place in the STATOR project, a 5-year research project started in 2013, that focuses on developing new methods for static analysis of software. My contributions improve several aspects of the VPL:

Scalability. The VPL now contains a solver for Parametric Linear Programming (PLP) problems, which is a powerful and generic tool. In particular, the two expensive operators in constraints-only that are variable elimination and convex hull are now encoded as such. The performance gain comes from a normalization constraint for avoiding the generation of redundancies, which can become numerous during intermediate steps of Fourier-Motzkin elimination.

New Algorithmics. In addition to PLP, the VPL now benefits from two new algorithms. First, a fast minimization algorithm for eliminating redundant constraints from a polyhedron representation. It is based on a raytracing process to avoid as much as possible the solving of costly linear programming (LP) problems. Second, a conflict-based algorithm for extracting implicit equalities from inequalities. It is more efficient in particular when there is no equality to be found: while the standard algorithm solves one LP problem for each inequality of the polyhedron, our solution needs only one in that case.

Handling of Polynomial Guards. The VPL provides two algorithms for over-approximating the effect of polynomial guards on a polyhedron. The first one extends the intervalization process of Miné (2006), that replaces some variables of nonlinear products by intervals. It is implemented directly in Coq, exploiting a proof by refinement. The second one exploits Handelman (1988)'s theorem to find affine constraints dominating the polynomial on the polyhedron. To obtain the tightest constraints in every direction, the problem is once again encoded in PLP.

Floating Point. When certification is not needed, for instance within intermediate computations, we try to avoid the overhead of rational operations by using floating point numbers as much as possible. In particular, we use LP solvers in floating point such as GLPK when possible.

Certification. The certification system has been redesigned in order to be more general and easier to debug. The idea is to exploit type polymorphism to preserve – for free! – some properties from Coq to OCaml by extraction. This led to an innovative and powerful certification framework named Polymorphic Factory Style, in which all VPL operators are now implemented.

Overview of the Thesis

The thesis is split into two parts, one focusing on new algorithmics and the other on Coq development. Before that, Chapter 1 provides the basics about certified polyhedral calculus. It details the different operators required in an abstract domain and defines their soundness criteria, that often boil down to proving polyhedral inclusions. Then, a section is dedicated to Farkas' lemma, a crucial result for the VPL since it provides a straightforward certificate format for proving polyhedral inclusions. The chapter ends with an introduction to linear programming, necessary to compute those certificates.

Part I focuses on improving the library overall performances and scalability. It begins with the minimization algorithm by raytracing in Chapter 2. After detailing the challenges of redundancy removal, the algorithm is expressed in terms of polyhedral cones, so that it can be applied both to constraints and generators. Then, we give the results of experiments performed on random polyhedra, and show how raytracing scales compared to the standard minimization when the dimension and the number of constraints grows.

Chapter 3 presents the encodings of projection and convex hull as PLP problems. We introduce the normalization constraint, and we prove that it avoids the generation of redundancies in results. Again, experiments show that projection by PLP scales much better than the standard Fourier-Motzkin elimination.

Chapter 4 introduces three methods to handle polynomials guards. First, we present our enhanced intervalization, that splits the space depending on the sign of some affine subterms

of the polynomial. This involves an oracle to rewrite the polynomial so that those subterms syntactically show up. After that, we detail a method that relies on the conversion of polynomials into the Bernstein basis. From the coefficients in this basis, one can extract some points whose convex-hull form an over-approximation. It is not implemented in the VPL though, because it works on generators. Finally, the last section deals with Handelman's theorem, and how it can be exploited to determine over-approximating polyhedra. Basically, it extends Bernstein's linearization by considering products of constraints of the input polyhedron, instead of starting from an initial hypercube like Bernstein's bases. To evaluate the method, we implemented it as a decision procedure for the SMT solver cvc4, to solve satisfiability problems of semialgebraic sets. Despite the shallow embedding and the absence of clause learning, the results were pretty convincing compared to other solvers.

Chapter 5 gives some details on the implementation of our PLP solver. It also describes the problem of degeneracy that can occur.

The subject of Part II is the Coq certification of the VPL. It begins with an introduction to the Coq proof assistant. After illustrating the basics of a Coq proof on a simple polyhedral inclusion, Chapter 6 discusses three different approaches of certification, and in particular how to embed untrusted code within a Coq development.

The subject of Chapter 7 is the embedding of untrusted code in Coq proofs without introducing a certificate format, that can be hard to develop and debug. Our solution, the Polymorphic Factory Style (PFS), is illustrated on the certification of the projection operator. Its flexible power is then attested on the more complex case of convex hull.

Chapter 9 deals with the certification of Handelman's linearization.

The last chapter gives some details about the VPL implementation. It ends with a description of an alternative abstract domain of polyhedra based on another representation: the partition in regions resulting from a PLP execution.

Chapter 1

A Certified Domain of Polyhedra

Convex polyhedra are mathematical objects defining a geometrical space. They appear in many fields such as static analysis and compilation, are at the heart of Linear Programming (LP), and are widely used in combinatorial optimization. In this thesis, polyhedra are involved in the formal certification of programs, by static analysis based on abstract interpretation.

This chapter defines the different operators for computing with polyhedra. We will see that the correctness of each operator boils down to proving an inclusion between two polyhedra. This inclusion is decidable thanks to Farkas' Lemma which naturally leads to an encoding of the inclusion as a LP problem. After introducing the theory of linear programming, we will detail on examples the seminal method for solving LP problems: the simplex algorithm.

1.1 Checking Programs with Polyhedra

1.1.1 Static Analysis by Abstract Interpretation

A static analyzer is a software that proves *assertions* on programs. An assertion is a property, sometimes user-defined, that must hold for every executions of the program. Focusing on numeric variables, properties of interest could be the absence of arithmetic overflow or array index out of bounds. To establish the validity of an assertion, the analyzer must find an inductive invariant that entails it. An *invariant* is a predicate that is true at each execution of the program. In a loop, we say that an invariant is *inductive* when it is true at the beginning and at the end of the loop body.

Analyzers based on abstract interpretation consider invariants within a particular class of properties called an abstract domain (Cousot and Cousot, 1977), for which operations meet ($\sqcap$) and join ($\sqcup$) are computable, and $\sqsubset$ is decidable. Roughly, each program point is associated to an *abstract value*, which represents the set of values that program variables can take at this point. For example, the abstract domain of intervals attaches to each variable its range. Initially, each variable at each program point has a range defined by its type, *e.g.* $[0, 1]$ for Booleans, $\mathbb{Z}$ for integers or $\mathbb{Q}$ for floats. Then, the analysis refines these ranges until a *fixpoint* (*i.e.* an inductive invariant) is reached. An interval analysis is cheap and particularly useful to detect arithmetic overflows. However, it is not powerful enough to validate an assertion that requires handling relations between variables.

For instance, consider the C program of Program 1.1. Using the precondition, an interval-based analysis could determine that at the end of the loop x and y both belong to $[100, 110]$, but it is not able to prove the assertion of line 11 because it requires keeping the information that $x = y$ at each loop step. The abstract domain of convex polyhedra has the ability to handle any kind of affine relation between variables. We will return to this example after introducing the domain of polyhedra.

```
1 //precondition : 0 ≤ x ≤ 10
2 void f(x){
3   int i;
4   int y = x;
5
6   for (i = 0 ; i < 100 ; i++){
7     x++;
8     y++;
9   }
10
11  assert (x == y);
12 }
```

Program 1.1 – C loop example

1.1.2 Convex Polyhedra

A set X is said *convex* if it fulfills the following property

$$\forall x, y \in X,\ \forall \alpha \in [0,1],\ \alpha x + (1-\alpha)y \in X$$

Intuitively it means that, for any two elements x, y of X, any point of the segment $[x, y]$ also belongs to X. A convex polyhedron[1] is a convex set of points defined by flat facets that meet on edges and vertices. In two dimensions, it is a *polygon*. A polyhedron can be bounded, in which case it is called a *polytope*, or unbounded as in the figure of Example 1.2.

We restrict our program analysis to integer variables only. Floating-point variables are more tricky to handle, since they induce rounding errors that must be over-approximated in order to stay sound. Thus, we will mainly be interested in the integer points of a polyhedron $\mathcal{P}$, *i.e.* $\mathbb{Z}^n \cap [\![\mathcal{P}]\!]$. But, computing with such sets is not easy, and sticking to standard convex polyhedra is usually sufficient to prove the required assertions. We will therefore consider convex polyhedra with rational coefficients. More formally, a convex polyhedron can be defined equivalently in two ways: as constraints or as generators.

1.1.2.1 Constraint Representation

A *convex polyhedron* $\mathcal{P}$ represents a set of points that satisfy a conjunction of *constraints*:

$$[\![\mathcal{P}]\!] = \left\{ \boldsymbol{x} \mid \bigwedge_{j=1}^{p} \sum_{i=1}^{n} a_{i,j} x_i \bowtie_j b_j \right\} \tag{1.1}$$

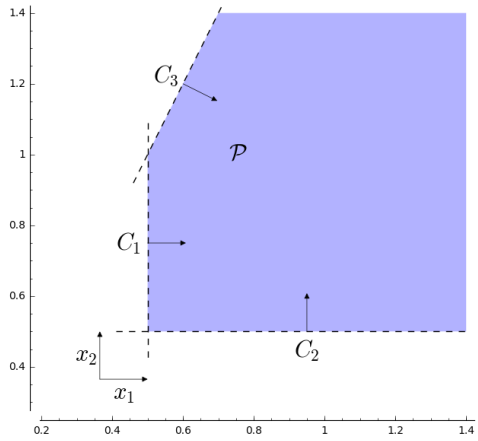
where $\boldsymbol{x} = (x_1, \ldots, x_n)$ is a vector of rational variables, $\bowtie_j \in \{\leq, <, =\}$, and $a_{i,j},\ b_j \in \mathbb{Q}$ (see Example 1.2). The notation $[\![\mathcal{P}]\!]$ describes the set of points represented by $\mathcal{P}$, while $\mathcal{P}$ refers to the set of constraints of the polyhedron. Thus, we say that a point $\boldsymbol{x} \in \mathbb{Q}^n$ lies within $[\![\mathcal{P}]\!]$, and that a constraint $C : \boldsymbol{ax} \bowtie b$ belongs to $\mathcal{P}$. We qualify a constraint as *affine* when the right-hand side b_j is not zero. If $b_j = 0$, we simply call it *linear*.

To simplify case studies in algorithms, we often convert constraints written with $\{\leq, <, =\}$ into equivalent ones using only $\{\geq, >, =\}$. Similarly, the equality symbol could be avoided as any equation $\sum_{i=1}^{n} a_i x_i = b$ can be rewritten into the conjunction $(\sum_{i=1}^{n} a_i x_i \geq b) \wedge (\sum_{i=1}^{n} a_i x_i \leq b)$.

We refer to the number of constraints of $\mathcal{P}$ as $|\mathcal{P}|$, or sometimes simply p. The polyhedron that is unconstrained, meaning that it contains all points of $\mathbb{Q}^n$, is noted $\top$. In other words, $[\![\top]\!] = [\![\{\}]\!] = \mathbb{Q}^n$.

1. In all the thesis, we deal only with convex polyhedra. For readability, we often omit the adjective *convex*.

Example 1.2



The unbounded polyhedron $\mathcal{P}$ defined by the three affine constraints $C_1 : 2x_1 \geq 1$, $C_2 : 2x_2 \geq 1$, $C_3 : 2x_1 - x_2 \geq 0$ is shown on the figure above. A constraint is represented as a dotted line, with a vector showing its direction. For instance, constraint C_1 is the half-space on the right of the vertical half-plane $x_1 = \frac{1}{2}$. A polyhedron is the intersection of half-spaces.

Matrix Notations. Convex polyhedra can be conveniently denoted using matrix notations. In all the thesis, vectors are written in boldface lowercase, *e.g.* $\boldsymbol{x} = (x_1, \ldots, x_n)$, and matrices in boldface uppercase, to be distinguished from scalar values. For instance, a vector of 0 is written $\mathbf{0}$. The definition (1.1) of a polyhedron can be rewritten as $\boldsymbol{Ax} \geq \boldsymbol{b}$, where $\boldsymbol{A}$ is the matrix of $a_{i,j}$'s, and $\boldsymbol{b}$ is the vector of b_j's. This notation works similarly with $\leq$ or $=$. However, a system $\boldsymbol{Ax} \geq \boldsymbol{b}$ cannot represent strict constraints. When a polyhedron contains both large and strict constraints, we therefore represent it with two systems $\boldsymbol{A_1 x} \geq \boldsymbol{b_1}$ and $\boldsymbol{A_2 x} > \boldsymbol{b_2}$.
We will denote by $\mathbb{Q}_{p\times n}$ the set of matrices with p rows and n columns with coefficients in $\mathbb{Q}$. Here, $\boldsymbol{A} \in \mathbb{Q}_{p\times n}$ and $\boldsymbol{b}, \boldsymbol{x} \in \mathbb{Q}_{n\times 1}$. The j^{th} row of the system $\boldsymbol{Ax} \geq \boldsymbol{b}$ corresponds to the j^{th} constraint $\sum_{i=1}^{n} a_{i,j} x_i \geq b_j$.
We will sometimes use the augmented matrix $[\boldsymbol{A}|-\boldsymbol{b}] \in \mathbb{Q}_{p\times n+1}$ associated to a system $\boldsymbol{Ax} \geq \boldsymbol{b}$, that is the matrix $\boldsymbol{A}$ concatenated with the column vector $-\boldsymbol{b}$. Indeed, a system $\boldsymbol{Ax} \geq \boldsymbol{b}$ is equivalent to $\boldsymbol{Ax} - \boldsymbol{b} \geq \mathbf{0}$, *i.e.*

$$[\boldsymbol{A}|-\boldsymbol{b}] \begin{pmatrix} \boldsymbol{x} \\ 1 \end{pmatrix} \geq \mathbf{0}$$

Dimension of a Polyhedron. Given a polyhedron $\mathcal{P} : \boldsymbol{Ax} \geq \boldsymbol{b}$ made only of inequalities, we say that $\boldsymbol{a}_i\boldsymbol{x} = b_i$ is an *implicit* (or *implied*) equality if $\mathcal{P} \Rightarrow (\boldsymbol{a}_i\boldsymbol{x} = b_i)$, meaning that all points of $[\![\mathcal{P}]\!]$ satisfy the equality. For instance, $x_1 = 1$ is an implicit equality that can be deduced from $x_1 \geq 1 \wedge x_1 \leq 1$. Let $\mathcal{P} = \{\boldsymbol{x} \in \mathbb{Q}^n \mid \boldsymbol{A}_1\boldsymbol{x} \geq \boldsymbol{b}_1,\ \boldsymbol{A}_2\boldsymbol{x} > \boldsymbol{b}_2\}$ and let $\overline{\boldsymbol{A_1}}\boldsymbol{x} = \overline{\boldsymbol{b_1}}$ be the system of implicit equations in $\boldsymbol{A}_1\boldsymbol{x} \geq \boldsymbol{b}_1$. Then the dimension of $\mathcal{P}$ is $dim\,(\mathcal{P}) = n - \rho$, where ρ is the rank of matrix $\overline{\boldsymbol{A_1}}$ Cook et al. (1998, Property 6.15 p.212).
From a more topological point of view, we can think of $dim\,(\mathcal{P})$ as the maximal dimension of an open ball that can be fully contained in $\mathcal{P}$. A polyhedron $\mathcal{P}$ defined on $\mathbb{Q}^n$ and of dimension n is said to be of *nonempty interior*, or of *full dimension*. The interior of a polyhedron $\mathcal{P}$ is written $\mathring{\mathcal{P}}$, and a point within $[\![\mathring{\mathcal{P}}]\!]$ is noted $\mathring{\boldsymbol{x}}$.
Given a polyhedron $\mathcal{P} : \boldsymbol{Ax} \geq \boldsymbol{b}$ of dimension n, the set $\boldsymbol{a}_i\boldsymbol{x} = b_i$ associated to a constraint $\boldsymbol{a}_i\boldsymbol{x} \geq b_i \in \mathcal{P}$ is a *facet* of $\mathcal{P}$ if $\boldsymbol{a}_i\boldsymbol{x} = b_i$ is an hyperplane of dimension $n-1$. In particular, the hyperplane of any irredundant constraint of $\mathcal{P}$ that does not induce an implicit equality is a facet of $\mathcal{P}$. In the following, we may abusively designate by facet both a constraint and its bounding hyperplane. The context should not leave any doubt on the nature of the so called object.
Most polyhedra library – including VPL– handle equalities and inequalities separately. When building a polyhedron, the first step often consists in extracting explicit and implicit equalities from the input set of constraints. Note that the VPL uses a particularly efficient algorithm

for extracting implicit equalities, presented in Chapter 8. Operators on polyhedra usually have a special treatment for equalities, much simpler as they rely on linear algebra instead of linear programming. Then, algorithms can be applied more easily by assuming that polyhedra have nonempty interior. Therefore, we will generally present algorithms on full-dimension polyhedra.

1.1.2.2 Generator Representation

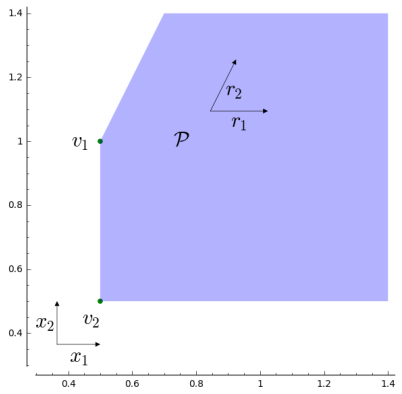
A convex polyhedron $\mathcal{P}$ can also be defined as the convex combination of *generators*: *vertices* $\boldsymbol{v_i}$ and *rays* $\boldsymbol{r}_i$. A ray represent a direction in which the polyhedron is unbounded.

$$[\![\mathcal{P}]\!] = \left\{ \boldsymbol{x} \mid \boldsymbol{x} = \sum_{i=1}^{|\boldsymbol{v}|} \beta_i \boldsymbol{v_i} + \sum_{i=1}^{|\boldsymbol{r}|} \lambda_i \boldsymbol{r}_i,\ \ \beta_i, \lambda_i \geq 0,\ \ \sum_{i=1}^{|\boldsymbol{v}|} \beta_i = 1 \right\}$$

By convexity, any convex combination of the vertices – *i.e.* $\sum_i \beta_i \boldsymbol{v_i}$ for some β_i's ≥ 0 such that $\sum_i \beta_i = 1$ – belongs to $[\![\mathcal{P}]\!]$. $\mathcal{P}$ is also stable by translating any point of $[\![\mathcal{P}]\!]$ along rays.

The generator representation usually contains a third structure called *line*, which reduces the representation size: one line stands for two opposite rays.

Example 1.3



The polyhedron $\mathcal{P}$ of Example 1.2 can be defined by the two vertices $\boldsymbol{v_1} : \left(\frac{1}{2}, 1\right)$, $\boldsymbol{v_2} : \left(\frac{1}{2}, \frac{1}{2}\right)$ and the two rays $\boldsymbol{r}_1 : (1, 0)$, $\boldsymbol{r}_2 : (1, 1)$.

1.1.2.3 Double Description

Most polyhedra libraries — such as the Parma Polyhedra Library (Bagnara et al., 2008), the NEWPOLKA library included in APRON (Jeannet and Miné, 2009), Polylib (Loechner and Clauss, 2010), and CDD (Fukuda, 2016) — use both representations of polyhedra, which is known as the double description framework (Motzkin et al., 1953; Fukuda and Prodon, 1996). The motivation behind that is simple: some operators have a better complexity in one representation. Complexity will be discussed at the end of the next section, after the introduction of polyhedral operators. For now, simply notice that having the double description of a polyhedron makes computations cheaper, because one can choose the best representation for each operator.

Chernikova's Algorithm. The bottleneck of double description is the conversion from one representation to the other, which is done by Chernikova (1968)'s algorithm. It is often assimilated to the Fourier-Motzkin elimination (that we will introduce later in §1.2.2.1), because they solve closely linked problems (Fukuda and Prodon, 1996). Let us give the idea of a conversion from constraints to generators. Starting from a set of generators representing $\top$ (*i.e.* an origin vertex and one line per variable), Chernikova's algorithm iteratively adds

constraints one by one. When adding a constraint, some generators are created while some others become redundant. To limit the combinatorial explosion, the fast removal of those redundancies is crucial. Several criteria have been proposed for that purpose. For instance, Le Verge (1992) gives an upper bound on the number of constraints saturated by a ray beyond which it is redundant. Still, Chernikova's algorithm has an exponential complexity in the size of the inputs and outputs.

Unlike other libraries, the VPL has the particularity to use the constraints-only representation of polyhedra. This choice was made because of certification: there is no known method to check in polynomial time that a system of inequalities is equivalent to a system of generators (Fukuda, 2004). Therefore, certifying a polyhedra library in double description would require to develop and prove Chernikova's algorithm directly in Coq. Moreover, to ensure the soundness of a set of constraints, one can check constraints independently one by one. In particular, the analysis stays sound if some constraints are missing. On the contrary, the generator system must be certified globally.

1.1.2.4 Operations on Constraints

Let us introduce some useful notations for affine constraints. Recall that vectors are written in boldface lowercase, to be distinguished from scalar values.

An affine constraint is of the form $\boldsymbol{a}\boldsymbol{x} \bowtie b$, where $\boldsymbol{a}, \boldsymbol{x} \in \mathbb{Q}^n$, $b \in \mathbb{Q}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$. The inner product of vectors $\boldsymbol{a}$ and $\boldsymbol{x}$, *i.e.* $\sum_{i=1}^{n} a_i x_i$, is denoted by $\langle \boldsymbol{a}, \boldsymbol{x} \rangle$ or simply $\boldsymbol{a}\boldsymbol{x}$. We often refer to a constraint with an affine function $C(\boldsymbol{x}) \bowtie 0$, where $C(\boldsymbol{x}) = \boldsymbol{a}\boldsymbol{x} - b$. Sometimes, we simply write $C \bowtie 0$, or even C when the context leaves no doubt about the sign of the constraint.

Consider two constraints $C_1 : \boldsymbol{a}_1\boldsymbol{x} \bowtie_1 b_1$ and $C_2 : \boldsymbol{a}_2\boldsymbol{x} \bowtie_2 b_2$ with $\bowtie_1, \bowtie_2 \in \{\leq, <, =\}$. We define the following operations on constraints:

- **Addition**: $C_1 + C_2 \stackrel{\text{def}}{=} \boldsymbol{a}_1\boldsymbol{x} + \boldsymbol{a}_2\boldsymbol{x} \bowtie b_1 + b_2$ with $\bowtie = max(\bowtie_1, \bowtie_2)$ for the total increasing order induced by the sequence $\{=, \leq, <\}$.
- **Product by a scalar**: $\alpha \times C_1 \stackrel{\text{def}}{=} \alpha \cdot \boldsymbol{a}_1\boldsymbol{x} \bowtie_1 \alpha \cdot b_1$ and is defined only if $\alpha > 0$ or if $\bowtie_1$ is symbol $=$.

These operations work similarly with $\bowtie_1, \bowtie_2 \in \{\geq, >, =\}$. Keep in mind that, to avoid confusion, we will allow the addition of two constraints only if they share the same comparison set (either $\{\leq, <, =\}$ or $\{\geq, >, =\}$).

1.2 Certifying an Abstract Domain of Polyhedra

Following abstract interpretation theory, an abstract domain is a lattice, and therefore must provide several operators such as inclusion testing, join or meet. This section introduces the specification of the required polyhedral operators, and some additional ones that are useful in practice, such as linearization. It also sketches the idea of their computation. We will give criteria for their correctness, and explain how to verify their results. Actually, VPL operators are not directly implemented and proved in Coq: they are certified *a posteriori* (Fouilhé et al., 2013). Each operator has an *oracle* written in OCaml, that performs most complex computations and returns a *certificate* in addition to its results. This certificate is then used by a Coq checker to rebuild the certified result in Coq datastructures. The Coq part of the VPL is detailed in Part II.

1.2.1 Polyhedral Operators

Abstract interpretation is a theory of *sound approximation* of the semantics of programs. In this context, soundness means that each abstract value – for us, a polyhedron – must over-approximate its associated concrete value, which is the set of actual values that can be taken by program variables at a program point. Intuitively, the result of an analysis is sound if it does not miss any behaviour: all reachable states of the program are covered. To ensure

that abstract values are always over-approximating their concrete value, each operator must maintain this property.

We can distinguish two kinds of operators: the purely geometrical ones (*e.g.* join, meet, inclusion tests, projection) and those linked to program statements like guards and assignments. The correctness of geometrical operators involves an inclusion, to check that their actual result over-approximates the expected one. Other operators, like assignment, require the checker to verify that their result satisfy their semantics. Still, we can always express such operators in terms of geometrical ones. For instance, an assignment can be encoded as a guard followed by a projection and a renaming, as detailed below.

Soundness does not require to prove that operators are precise, even if we could do it for most of them. In the rest of the thesis, we will not distinguish between the terms soundness and correctness, as they are similar to us.

Operators will be expressed using constraints-only representation. All along the section, we consider three polyhedra

$$\mathcal{P} : \bigwedge_{i=1}^{p} C_i(\boldsymbol{x}) \geq 0,\ \mathcal{P}' : \bigwedge_{i=1}^{p'} C'_i(\boldsymbol{x}) \geq 0 \text{ and } \mathcal{P}'' : \bigwedge_{i=1}^{p''} C''_i(\boldsymbol{x}) \geq 0$$

Inclusion Test. *The inclusion of polyhedra*, denoted by $\mathcal{P} \sqsubseteq \mathcal{P}'$ is the inclusion of their set of points, *i.e.* $[\![\mathcal{P}]\!] \subseteq [\![\mathcal{P}']\!]$. Being able to generate certificates for inclusion is crucial: the correctness of most operators boils down to proving polyhedral inclusions. The proof of an inclusion can be split into p' independent sub-proofs $\mathcal{P} \sqsubseteq C'_i$, one for each constraint C'_i of $\mathcal{P}'$. Each one-constraint inclusion can then be tested thanks to Farkas' lemma 1.5, given later. Basically, $\mathcal{P} \sqsubseteq C'_i$ if and only if C'_i can be expressed as a nonnegative combination of constraints of $\mathcal{P}$, *i.e.*

$$\exists (\lambda_k)_{k\in\{0,\dots,p\}} \in \mathbb{Q}^+,\ \forall \boldsymbol{x} \in \mathbb{Q}^n,\ \lambda_0 + \sum_{k=1}^{p} \lambda_k C_k(\boldsymbol{x}) = C'_i(\boldsymbol{x})$$

As the whole domain correctness is based on it, this lemma is dedicated the whole section §1.2.2.

Emptiness Test. *A polyhedron is empty* if its constraints are unsatisfiable. The empty polyhedron is noted $\mathcal{P}_\emptyset$ or $\bot$, and is represented by a single contradictory (constant) constraint, such as $-1 \geq 0$. Testing the emptiness of a polyhedron $\mathcal{P}$ is the same as proving $\mathcal{P} \sqsubseteq \mathcal{P}_\emptyset$. Theorem 1.6, stated later, is a variant of Farkas' lemma for deciding this particular inclusion. The idea is similar: $\mathcal{P} \sqsubseteq \mathcal{P}_\emptyset$ if $-1 \geq 0$ can be expressed as a nonnegative combination of constraints of $\mathcal{P}$, *i.e.*

$$\exists (\lambda_k)_{k\in\{0,\dots,p\}} \in \mathbb{Q}^+,\ \forall \boldsymbol{x} \in \mathbb{Q}^n,\ \lambda_0 + \sum_{k=1}^{p} \lambda_k C_k(\boldsymbol{x}) = -1$$

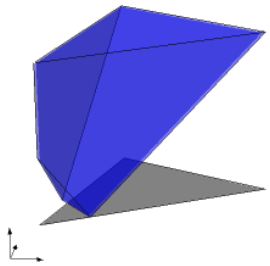
Variable Elimination. Given $n > k$, *the projection onto* $\mathbb{Q}^k$ *of a polyhedron* $\mathcal{P}$ that constrains points of $\mathbb{Q}^n$ is the polyhedron $\mathcal{P}'$ such that

$$[\![\mathcal{P}']\!] \stackrel{\text{def}}{=} \{(x_1,\dots,x_k) \in \mathbb{Q}^n \mid \exists x_{k+1},\dots,x_n \in \mathbb{Q},\ (x_1,\dots,x_k,x_{k+1},\dots,x_n) \in [\![\mathcal{P}]\!]\}$$

Without loss of generality, we can reorder the variables so that the eliminated ones are the last $n-k$. In static analysis, projection is used to eliminate variables from a polyhedron, for instance at the end of a statement block in a program analysis and also in the assignment operator, defined below. Given a subset of indices $I \subsetneq \{1,\dots,n\}$, we denote by $\mathcal{P}_{\setminus\{(x_i)_{i\in I}\}}$ the polyhedron $\mathcal{P}$ where the variables $(x_i)_{i\in I}$ have been eliminated by projection. A resulting polyhedron $\mathcal{H}$ is a correct over-approximation of $\mathcal{P}_{\setminus\{(x_i)_{i\in I}\}}$ if $\mathcal{P} \sqsubseteq \mathcal{H}$ and if x_i is unbounded in $\mathcal{H}$, for all i in I.

In constraints-only, convex hull and assignment are implemented via projection, which makes it a central part of our polyhedral domain. The classical algorithm for projecting variables is Fourier-Motzkin elimination, the principle of which is sketched later in §1.2.2.1. Fourier-Motzkin suffers from an exponential complexity in the number of projected variables. A major contribution of the VPL is to perform variable projection by Parametric Linear Programming (PLP) (Maréchal et al., 2017), which is the subject of Chapter 3.

Example 1.4



Elimination of x_3 from $\mathcal{P} : \{-x_1 - 2x_2 + 2x_3 \geq -7,\ -x_1 + 2x_2 \geq 1, 3x_1 - x_2 \geq 0, x_3 \leq 10, x_1 + x_2 + x_3 \geq 5\}$ by projection. The projected polyhedron is the grey shadow.

Join. *The convex hull of two polyhedra,* denoted by $\mathcal{P}' \sqcup \mathcal{P}''$, is the smallest convex polyhedron that includes $\mathcal{P}'$ and $\mathcal{P}''$. Due to convexity $[\![\mathcal{P}' \sqcup \mathcal{P}'']\!]$ can be defined as the set of all convex combinations between a point of $[\![\mathcal{P}']\!]$ and a point of $[\![\mathcal{P}'']\!]$:

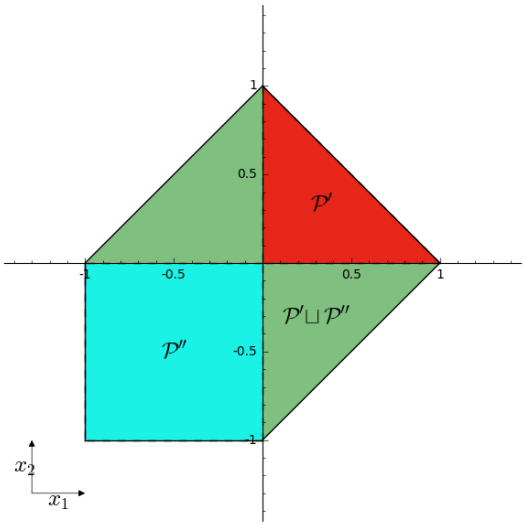
$$[\![\mathcal{P}' \sqcup \mathcal{P}'']\!] \stackrel{\text{def}}{=} \{\boldsymbol{x} = \alpha \cdot \boldsymbol{x}' + (1 - \alpha) \cdot \boldsymbol{x}'' \mid \boldsymbol{x}' \in [\![\mathcal{P}']\!],\ \boldsymbol{x}'' \in [\![\mathcal{P}'']\!],\ 0 \leq \alpha \leq 1\} \tag{1.2}$$

Elaborating on this remark, Benoy et al. (2005) expressed the convex hull as a projection problem: $\mathcal{P}' \sqcup \mathcal{P}''$ can be obtained by eliminating variables $\boldsymbol{x}'$, $\boldsymbol{x}''$ and α from 1.2, to get constraints mentioning only $\boldsymbol{x}$. This process and its certification are detailed in §7.3. A result $\mathcal{H}$ is a correct over-approximation of $[\![\mathcal{P}' \sqcup \mathcal{P}'']\!]$ if it satisfies:

$$\mathcal{P}' \sqsubseteq \mathcal{H} \wedge \mathcal{P}'' \sqsubseteq \mathcal{H} \tag{1.3}$$

Additionally, the VPL provides another implementation of convex hull, directly encoded as a PLP problem exploiting (1.3), hence not involving projection. This is discussed in §3.6.

Example 1.5



The convex hull of $\mathcal{P}' : \{x_1 \geq 0,\ x_2 \geq 0,\ x_1 + x_2 \leq 1\}$ and $\mathcal{P}'' : \{-1 \leq x_1 \leq 0,\ -1 \leq x_2 \leq 0\}$ is $\{\ x_1 \geq -1,\ x_2 \geq -1,\ x_1 + x_2 \leq 1,\ x_2 - x_1 \leq 1,\ x_1 - x_2 \leq 1\}$.

Minimization. A constraint $C_k \geq 0$ is *redundant* w.r.t. $\mathcal{P}$ if it is a logical consequence of the other constraints of $\mathcal{P}$, *i.e.* $\bigwedge_{i \neq k} C_i(\boldsymbol{x}) \geq 0 \Rightarrow C_k(\boldsymbol{x}) \geq 0$. Informally, it implies that C_k is useless in the sense that adding it does not change the geometrical space delimited by $\mathcal{P}$. Redundancies must be eliminated in order to maintain concise representations and avoid useless computations in subsequent operations. The previous implication can be reformulated as $\boldsymbol{x} \in [\![\mathcal{P} \setminus \{C_k\}]\!] \Rightarrow \boldsymbol{x} \in [\![C_k \geq 0]\!]$ which reveals that deciding the redundancy of C_k w.r.t. $\mathcal{P}$ amounts to decide the inclusion $\mathcal{P} \sqsubseteq \{C_k \geq 0\}$. The minimization is sound as long as it only removes constraints, since this process can only build over-approximations of the input polyhedron. We also want the minimization to be precise, meaning that it does not remove any irredundant constraint, but we do not have to prove it. The VPL benefits from an efficient algorithm for minimizing (Maréchal and Périn, 2017), which is the subject of Chapter 2.

Meet. The intersection of two polyhedra, denoted by $\mathcal{P}' \sqcap \mathcal{P}''$, is the conjunction of their constraints followed by a minimization to remove redundant ones. In other words, given $\mathcal{P}' = \boldsymbol{A}'\boldsymbol{x} \leq \boldsymbol{b}' = \{ C'_1, \ldots, C'_{p'} \}$ and $\mathcal{P}'' = \boldsymbol{A}''\boldsymbol{x} \leq \boldsymbol{b}'' = \{ C''_1, \ldots, C''_{p''} \}$, then

$$\mathcal{P}' \sqcap \mathcal{P}'' = \begin{pmatrix} \boldsymbol{A}' \\ \boldsymbol{A}'' \end{pmatrix} \boldsymbol{x} \leq \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{b}'' \end{pmatrix} = \{ C'_1, \ldots, C'_{p'} \} \cup \{ C''_1, \ldots, C''_{p''} \}$$

The correctness criterion is the same as for minimization: as long as the result constraints are a subset of $\{ C'_1, \ldots, C'_{p'} \} \cup \{ C''_1, \ldots, C''_{p''} \}$, it is a correct over-approximation of $\mathcal{P}' \sqcap \mathcal{P}''$.

Guard. The effect of a guard g on a polyhedron $\mathcal{P}$ is the set of points of $[\![\mathcal{P}]\!]$ that satisfy the predicate g, *i.e.* $\{\boldsymbol{x} \in [\![\mathcal{P}]\!] \mid g(\boldsymbol{x})\}$. When the guard g is a conjunction of affine constraints, *i.e.* a polyhedron $\mathcal{G}$, the result is simply the intersection $\mathcal{P} \sqcap \mathcal{G}$. If the guard can be transformed into an equivalent disjunction of polyhedra, $\mathcal{G}_1 \vee \ldots \vee \mathcal{G}_k$, the effect of the guard is approximated by the polyhedron $(\mathcal{P} \sqcap \mathcal{G}_1) \sqcup \ldots \sqcup (\mathcal{P} \sqcap \mathcal{G}_k)$. The correctness of this approximation is guaranteed by the correctness of $\sqcap$ and $\sqcup$.
When a guard cannot be expressed as an abstract value of the domain, it is always possible to return $\mathcal{P}$. In the case of polynomial expressions, specific linearization algorithms can exploit the guard.

Linearization. When a guard is a polynomial constraint $Q(\boldsymbol{x}) \geq 0$, the abstract domain must still provide an output. The result must be a polyhedral over-approximation of the constraint, that is a polyhedron $\mathcal{P}'$ such that $\{\boldsymbol{x} \in [\![\mathcal{P}]\!] \mid Q(\boldsymbol{x}) \geq 0\} \subseteq [\![\mathcal{P}']\!]$. This operation, called *linearization*, has two variants in the VPL that are addressed in Chapter 4. The first one extends the intervalization process of Miné (2006), where some variables of products are replaced by intervals (Boulmé and Maréchal, 2015). The second one is a new algorithm involving Handelman's theorem for representing positive polynomials over a polyhedron (Maréchal et al., 2016).

Assignment. When the right-hand side of an assignment is an affine expression f, the effect of $x := f$ on a polyhedron $\mathcal{P}$ corresponds to the intersection of $\mathcal{P}$ with the polyhedron encoding the equality $\{\tilde{x} = f\}$, where $\tilde{x}$ is a fresh variable that denotes the new value of x. The fresh variable $\tilde{x}$ will be renamed into x after elimination of the old value of x by projection. Formally, the effect of $x := f$ on $\mathcal{P}$ is the polyhedron

$$\Big((\mathcal{P} \sqcap \{\tilde{x} = f\})_{\setminus \{x\}} \Big) [\tilde{x}/x].$$

Assignment is thus treated as a combination of a guard, a projection and a renaming.

Widening. The principle of abstract interpretation is to automatically find inductive loop invariants. The standard fixpoint computation based on Kleene iteration can fail if the abstract domain does not satisfy the ascending chain condition, or it can simply be too long to converge toward a fixpoint. The usual approach is to use a *widening* operator, which consists in guessing an invariant by analyzing the first iterations of a loop. The VPL uses the widening operator

Table 1.6 – Complexity of polyhedral operators where p is the number of constraints, n the dimension, and g the number of generators. $LP(p,n)$ is the time required to solve a LP problem with p constraints and n variables.

Operator	Constraints	Generators	Both
inclusion	$\mathcal{O}(p\ LP(p,n))$	$\mathcal{O}(g\ LP(g,n))$	$\mathcal{O}(pmg)$
join	$\mathcal{O}(np^{2n+1})$	$\mathcal{O}(ng)$	$\mathcal{O}(ng)$
meet	$\mathcal{O}(np)$	$\mathcal{O}(ng^{2n+1})$	$\mathcal{O}(np)$
widening	$\mathcal{O}(p\ LP(p,n))$	$\mathcal{O}(g\ LP(g,n))$	$\mathcal{O}(pmg)$
guard	$\mathcal{O}(n)$	$\mathcal{O}(ng^{2n+1})$	$\mathcal{O}(n)$
assignment	$\mathcal{O}(np^2)$	$\mathcal{O}(ng)$	$\mathcal{O}(ng)$

of Halbwachs (1979) that finds an invariant by discarding constraints whose constant term changes from one iteration to the other. The abstract domain does not need to certify the candidate invariants provided by widening: the analyzer checks their inductiveness by itself.

Example of Polyhedral Analysis. Going back to the C code of Program 1.1 (p. 12), let us summarize the behaviour of a polyhedral abstract domain, focusing on variables x and y. Before entering the loop, at line 5, the abstract value is $\mathcal{P}_0 \stackrel{\text{def}}{=} \{y = x,\ 0 \leq x \leq 10\}$. Let us unroll the first iteration of the loop: After statement `x++`, the polyhedron becomes $\{y = x - 1,\ 1 \leq x \leq 11\}$. Then, the effect of `y++` gives $\mathcal{P}_1 \stackrel{\text{def}}{=} \{y = x,\ 1 \leq x \leq 11\}$. Trying to infer an inductive invariant, we compute the convex hull of $\mathcal{P}_0$ and $\mathcal{P}_1$, which is $\mathcal{P}'_0 \stackrel{\text{def}}{=} \{y = x,\ 0 \leq x \leq 11\}$. We can then unroll a second time the loop, starting from $\mathcal{P}'_0$. After the i^{th} unrolling of the loop body, our abstract value will be $\{y = x,\ 0 \leq x \leq 10 + i\}$. To avoid unrolling the loop a hundred times, the widening operator removes the constraints that change from one iteration to the other, *i.e.* $x \leq 10 + i$. Finally, we obtain the inductive invariant $\{y = x,\ 0 \leq x\}$, which is able to prove the assertion of line 11.

1.2.1.1 Operators Complexity

Table 1.6, taken from (Singh et al., 2017), gives the complexity for each operator in constraints-only, generators-only, and double description frameworks. It shows that operator join is the weakness of constraints-only representation, whereas the meet operator is the most expensive one for generators. Obviously, knowing both representations allows using the cheapest representation for each operator.

As mentioned before, the bottleneck of double description is Chernikova's conversion of representation. Some double description libraries, such as PPL, can apply lazy techniques: by delaying computations until a result is really needed, they can chain operations in one representation without triggering Chernikova's algorithm if it is not worth it.

1.2.2 Farkas' Lemma

Farkas' lemma allows deciding a polyhedral inclusion. This is a seminal result that we shall use all along the thesis, hence before giving the variant that suits our needs in the VPL, let us see a brief overview of the other ones. Originally established in (Farkas, 1902), it was stated in several variants through the years. Some of them are given in the book "Combinatorial Optimization" by Cook et al. (1998, Corollary A.2 p. 326), from which most following proofs are taken. Note that all results given in that section are valid for both $\mathbb{R}$ and $\mathbb{Q}$, but we will give them in $\mathbb{Q}$ since we use them in that way. One of the most famous variant is analogous to the following well-known result in Gaussian elimination.

Theorem 1.1 (Gauss) *Given $n, m \in \mathbb{N}^*$, $\boldsymbol{A} \in \mathbb{Q}_{m\times n}$ and $\boldsymbol{b} \in \mathbb{Q}^m$, exactly one of the following assertions is true:*

— $\exists \boldsymbol{x} \in \mathbb{Q}^n,\ \boldsymbol{Ax} = \boldsymbol{b}$

— $\exists \boldsymbol{y} \in \mathbb{Q}^m,\ \boldsymbol{y}^\intercal \boldsymbol{A} = \boldsymbol{0}$ *and* $\boldsymbol{y}^\intercal \boldsymbol{b} \neq 0$

This result is known as a *theorem of alternatives*, because it states that exactly one of two systems of equations has a solution. Farkas' lemma extends this theorem to the existence of a solution in a system $\boldsymbol{Ax} \leq \boldsymbol{b}$. Instead of Gaussian elimination, it is based on the Fourier-Motzkin elimination procedure that eliminates a variable from a system of inequalities.

1.2.2.1 Fourier-Motzkin Elimination

Suppose we want to eliminate variable x_l from polyhedron $\boldsymbol{Ax} \leq \boldsymbol{b}$, which corresponds to the projection operator defined above. Fourier-Motzkin elimination is a classical algorithm to perform this projection. The idea is to combine every possible pair of rows where x_l has opposite signs, in order to make it vanish. We split rows of $\boldsymbol{Ax} \leq \boldsymbol{b}$ into three groups I^+, I^- and I^0 according to their coefficient for x_l: $\boldsymbol{a}_i\boldsymbol{x} \leq b_i$ is in I^+ if $a_{il} > 0$, in I^- if $a_{il} < 0$ and in I^0 otherwise. As we may multiply any row by a positive scalar without changing the set of solutions, we can isolate x_l by rewriting the system into

$$\begin{array}{rcll} x_l + \frac{1}{a_{il}}\boldsymbol{a}'_i\boldsymbol{x'} & \leq & \frac{b_i}{a_{il}}, & \text{if } (\boldsymbol{a}_i\boldsymbol{x} \leq b_i) \in I^+ \\ -x_l + \frac{1}{-a_{il}}\boldsymbol{a}'_i\boldsymbol{x'} & \leq & \frac{b_i}{-a_{il}}, & \text{if } (\boldsymbol{a}_i\boldsymbol{x} \leq b_i) \in I^- \\ \boldsymbol{a}'_i\boldsymbol{x'} & \leq & b_i, & \text{if } (\boldsymbol{a}_i\boldsymbol{x} \leq b_i) \in I^0 \end{array} \quad \text{where } \boldsymbol{x'} = \begin{pmatrix} x_1 \\ \cdots \\ x_{l-1} \\ x_{l+1} \\ \cdots \\ x_n \end{pmatrix}$$

and $\boldsymbol{a}'_i$ is the i^{th} row of $\boldsymbol{A}$ with the l^{th} coefficient deleted. This system has a solution if and only if the following one has a solution.

$$\begin{array}{c} \frac{1}{-a_{jl}}(\boldsymbol{a}'_j\boldsymbol{x'} - b_j) \leq x_l \leq \frac{1}{a_{il}}(b_i - \boldsymbol{a}'_i\boldsymbol{x'}), \quad \forall(\boldsymbol{a}_i\boldsymbol{x} \leq b_i) \in I^+,\ \forall(\boldsymbol{a}_j\boldsymbol{x} \leq b_j) \in I^- \\ \boldsymbol{a}'_k\boldsymbol{x'} \leq b_k, \quad \forall(\boldsymbol{a}_k\boldsymbol{x} \leq b_k) \in I^0 \end{array} \tag{1.4}$$

The value for x_l can be chosen if and only if the gap between its lower and upper bounds is nonnegative. By rewriting inequalities in the usual form, $\boldsymbol{Ax} \leq \boldsymbol{b}$ is thus equivalent to

$$\begin{array}{rcll} (\frac{1}{a_{il}}\boldsymbol{a}'_i + \frac{1}{-a_{jl}}\boldsymbol{a}'_j)\boldsymbol{x'} & \leq & \frac{b_i}{a_{il}} + \frac{b_j}{a_{jl}}, & \forall(\boldsymbol{a}_i\boldsymbol{x} \leq b_i) \in I^+,\ \forall(\boldsymbol{a}_j\boldsymbol{x} \leq b_j) \in I^- \\ \boldsymbol{a}'_k\boldsymbol{x'} & \leq & b_k, & \forall(\boldsymbol{a}_k\boldsymbol{x} \leq b_k) \in I^0 \end{array}$$

Variable x_l no longer appears in this system. Note that it may contain much more constraints than the initial one: the Fourier-Motzkin elimination could produce $\mathcal{O}\big((\frac{|\mathcal{P}|}{2})^{2^k}\big)$ constraints when eliminating k variables of a polyhedron with $|\mathcal{P}|$ constraints (Simon and King, 2005). Indeed, eliminating variable x_l from $\mathcal{P}$ generates a polyhedron with $\big|I^0\big| + |I^+| * |I^-|$ constraints. The worst case is obtained when $I^0 = \{\}$ and $|I^+| = |I^-| = \frac{|\mathcal{P}|}{2}$, meaning that x_l appears in all constraints, half of the time with a negative coefficient. Then, $\big|\mathcal{P}_{\setminus\{x_l\}}\big| = \big(\frac{|\mathcal{P}|}{2}\big)^2$. Eliminating k variables one after the other leads to the complexity claimed above.

Proposition 1.2 *Fourier-Motzkin elimination preserves unsatisfiability.*

Proof. The proposition is a direct consequence of the following result: Let $\boldsymbol{A'x'} \leq \boldsymbol{b'}$ be the result of the Fourier-Motzkin elimination of x_l from $\boldsymbol{Ax} \leq \boldsymbol{b}$. Then for any point $\boldsymbol{x'} = (x_1, \ldots, x_{l-1}, x_{l+1}, \ldots, x_n)$ satisfying $\boldsymbol{A'x'} \leq \boldsymbol{b'}$, there exists x_l such that $\boldsymbol{x} = (x_1, \ldots, x_n)$ satisfies $\boldsymbol{Ax} \leq \boldsymbol{b}$. This holds because, as pointed out above in (1.4), the constraints of $\boldsymbol{A'x'} \leq \boldsymbol{b'}$ are built exactly in such a way.

In other words, if $\boldsymbol{A'x'} \leq \boldsymbol{b'}$ is non empty, then so is $\boldsymbol{Ax} \leq \boldsymbol{b}$. The contraposition of this result leads to the proposition. □

1.2.2.2 Proofs of Farkas' Lemma

Now, let us prove our first variant of Farkas' lemma. Note that, given a vector $\boldsymbol{y} \in \mathbb{Q}^m$, notation $\boldsymbol{y} \geq \boldsymbol{0}$ stands for element-wise comparison, *i.e.* $\forall i \in \{1, \ldots, m\},\ y_i \geq 0$.

Lemma 1.3 (Farkas (for inequalities)) *Given $n, m \in \mathbb{N}^*$, $\boldsymbol{A} \in \mathbb{Q}_{m\times n}$ and $\boldsymbol{b} \in \mathbb{Q}^m$, exactly one of the following assertions is true:*

— $\exists \boldsymbol{x} \in \mathbb{Q}^n,\ \boldsymbol{Ax} \leq \boldsymbol{b}$

— $\exists \boldsymbol{y} \in \mathbb{Q}^m,\ \boldsymbol{y} \geq \boldsymbol{0}$ *and* $\boldsymbol{y}^\intercal \boldsymbol{A} = \boldsymbol{0},\ \boldsymbol{y}^\intercal \boldsymbol{b} < 0$

Proof. Inspired from Cook et al. (1998, Theorem A.1 p. 326).

Suppose $\boldsymbol{Ax} \leq \boldsymbol{b}$ has a solution $\tilde{\boldsymbol{x}}$ and suppose that there exists a vector $\tilde{\boldsymbol{y}} \geq \boldsymbol{0}$ such that $\tilde{\boldsymbol{y}}^\intercal \boldsymbol{A} = \boldsymbol{0}$ and $\tilde{\boldsymbol{y}}^\intercal \boldsymbol{b} < 0$. Then, we obtain the contradiction

$$0 > \tilde{\boldsymbol{y}}^\intercal \boldsymbol{b} \geq \tilde{\boldsymbol{y}}^\intercal (\boldsymbol{A}\tilde{\boldsymbol{x}}) = (\tilde{\boldsymbol{y}}^\intercal \boldsymbol{A})\tilde{\boldsymbol{x}} = 0$$

Therefore, both assertions cannot be simultaneously true.

Now, suppose that $\boldsymbol{Ax} \leq \boldsymbol{b}$ has no solution. Let us prove by induction on the dimension of $\boldsymbol{x}$ that $\exists \boldsymbol{y} \in \mathbb{Q}^m,\ \boldsymbol{y} \geq \boldsymbol{0}$ and $\boldsymbol{y}^\intercal \boldsymbol{A} = \boldsymbol{0},\ \boldsymbol{y}^\intercal \boldsymbol{b} < 0$. When $\boldsymbol{x}$ is a single variable $x \in \mathbb{Q}$, then $\boldsymbol{A}$ is a single column $\boldsymbol{a}$. In that case, the unsatisfiability of $\boldsymbol{A}x \leq \boldsymbol{b}$ can be reduced to two contradictory constraints, say C_i and C_j, that yield an empty interval for x. In other words, there are two rows of $\boldsymbol{a}x \leq \boldsymbol{b}$ satisfying $a_i x \leq b_i$ and $a_j x \leq b_j$ where $a_i > 0$, $a_j < 0$, and $\frac{b_i}{a_i} < \frac{b_j}{a_j}$. These two constraints define the empty interval $\frac{b_j}{a_j} \leq x \leq \frac{b_i}{a_i}$. Now, by defining the vector $\boldsymbol{y}$ as 1 in the i^{th} coordinate, $-\frac{a_i}{a_j}$ for the j^{th} and 0 otherwise, we have $\boldsymbol{y}^\intercal \boldsymbol{a} = a_i - \frac{a_i}{a_j} a_j = 0$ and $\frac{b_i}{a_i} < \frac{b_j}{a_j} \Leftrightarrow b_i < \frac{a_i}{a_j} b_j \Leftrightarrow b_i - \frac{a_i}{a_j} b_j < 0 \Leftrightarrow \boldsymbol{y}^\intercal \boldsymbol{b} < 0$. Thus, the property holds in dimension one.

When the system $\boldsymbol{Ax} \leq \boldsymbol{b}$ has a greater dimension, say $\boldsymbol{x} = (x_1, \ldots, x_n)$, apply Fourier-Motzkin elimination to obtain a system $\boldsymbol{A'x'} \leq \boldsymbol{b'}$ with one less variable, *i.e.* $\boldsymbol{x'} = (x_1, \ldots, x_{n-1})$. Since Fourier-Motzkin preserves unsatisfiability (Proposition 1.2, page 20), $\boldsymbol{A'x'} \leq \boldsymbol{b'}$ has no solution either. By induction hypothesis, $\exists \boldsymbol{y'} \geq \boldsymbol{0}$ and $\boldsymbol{y'}^\intercal \boldsymbol{A'} = \boldsymbol{0},\ \boldsymbol{y'}^\intercal \boldsymbol{b'} < 0$. The last step is to reconstruct $\boldsymbol{y}$ from $\boldsymbol{y'}$ such that $\boldsymbol{y} \geq 0$, $\boldsymbol{y}^\intercal \boldsymbol{A} = \boldsymbol{0}$ and $\boldsymbol{y}^\intercal \boldsymbol{b} < 0$.

Let p (resp. p') be the number of constraints of $\boldsymbol{Ax} \leq \boldsymbol{b}$ (resp. $\boldsymbol{A'x'} \leq \boldsymbol{b'}$). As we saw in the Fourier-Motzkin elimination, each constraint of $\boldsymbol{A'x'} \leq \boldsymbol{b'}$ is a positive combination of constraints of $\boldsymbol{Ax} \leq \boldsymbol{b}$. Thus, $\forall i \in \{1, \ldots, p'\},\ \exists (\lambda_{ik})_{k\in\{1,\ldots,p\}} \geq 0$ such that

$$\left.\begin{array}{r} \boldsymbol{a'_i} = \sum_{k=1}^{p} \lambda_{ik} \boldsymbol{a_k} \\ \text{or equivalently} \\ \forall j \in \{1, \ldots, n\},\ a'_{ij} = \sum_{k=1}^{p} \lambda_{ik} a_{kj} \end{array}\right\} \tag{1.5}$$

$$b'_i = \sum_{k=1}^{p} \lambda_{ik} b_k \tag{1.6}$$

Let us build a vector $\boldsymbol{y}$ from $\boldsymbol{y'}$ such that $\boldsymbol{y}^\intercal \boldsymbol{A} = \boldsymbol{0}$:

$$\begin{array}{lll} & \boldsymbol{y'}^\intercal \boldsymbol{A'} = \boldsymbol{0} & \text{by induction hypothesis} \\ \Leftrightarrow & \forall j \in \{1, \ldots, n\},\ \sum_{i=1}^{p'} y'_i a'_{ij} = 0 & \\ \Leftrightarrow & \forall j \in \{1, \ldots, n\},\ \sum_{i=1}^{p'} y'_i \left(\sum_{k=1}^{p} \lambda_{ik} a_{kj} \right) = 0 & \text{by (1.5)} \\ \Leftrightarrow & \forall j \in \{1, \ldots, n\},\ \sum_{k=1}^{p} \left(\sum_{i=1}^{p'} y'_i \lambda_{ik} \right) a_{kj} = 0 & \end{array}$$

Thus, by defining the vector $\boldsymbol{y}$ as $\forall k \in \{1, \ldots, p\},\ y_k \stackrel{\text{def}}{=} \sum_{i=1}^{p'} y'_i \lambda_{ik}$, we have

$$\forall j \in \{1, \ldots, n\},\ \sum_{k=1}^{p} y_k a_{kj} = \sum_{k=1}^{p} \left(\sum_{i=1}^{p'} y'_i \lambda_{ik} \right) a_{kj} = 0$$

which is equivalent to $\boldsymbol{y}^\intercal \boldsymbol{A} = \boldsymbol{0}$. Moreover, $\boldsymbol{y} \geq 0$ since λ_{ik}'s are nonnegative and $\boldsymbol{y'} \geq 0$ by induction hypothesis. Finally, let us verify that $\boldsymbol{y}^\intercal \boldsymbol{b} < 0$:

$$\begin{aligned}
\boldsymbol{y}^\intercal \boldsymbol{b} &= \sum_{k=1}^{p} y_k b_k \\
&= \sum_{k=1}^{p} \left(\sum_{i=1}^{p'} y'_i \lambda_{ik} \right) b_k && \text{by definition of } y_k \\
&= \sum_{i=1}^{p'} y'_i \left(\sum_{k=1}^{p} \lambda_{ik} b_k \right) \\
&= \sum_{i=1}^{p'} y'_i b'_i && \text{by (1.6)} \\
&= \boldsymbol{y'}^\intercal \boldsymbol{b'} \\
&< 0 && \text{by induction hypothesis}
\end{aligned}$$

□

From this lemma, we can easily prove the variant that was actually stated by Farkas.

Corollary 1.4 (Farkas) *Given $n, m \in \mathbb{N}^*$, $\boldsymbol{A} \in \mathbb{Q}_{m \times n}$ and $\boldsymbol{b} \in \mathbb{Q}^m$, exactly one of the following assertions is true:*

— $\exists \boldsymbol{x} \in \mathbb{Q}^n,\ \boldsymbol{x} \geq \boldsymbol{0}$ *and* $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$

— $\exists \boldsymbol{y} \in \mathbb{Q}^m,\ \boldsymbol{y}^\intercal \boldsymbol{A} \geq \boldsymbol{0},\ \boldsymbol{y}^\intercal \boldsymbol{b} < 0$

Proof. From Cook et al. (1998, Corollary A.2 p. 327).

Let us encode the equality $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ as two inequalities $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$ and $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$. The system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ and the constraint $\boldsymbol{x} \geq 0$ corresponds to

$$\underbrace{\begin{pmatrix} \boldsymbol{A} \\ -\boldsymbol{A} \\ -\boldsymbol{I} \end{pmatrix}}_{\boldsymbol{A'}} \boldsymbol{x} \leq \underbrace{\begin{pmatrix} \boldsymbol{b} \\ -\boldsymbol{b} \\ 0 \end{pmatrix}}_{\boldsymbol{b'}}$$

Then, $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ has a nonnegative solution if and only if $\boldsymbol{A'}\boldsymbol{x} \leq \boldsymbol{b'}$ has a solution. Applying the previous Lemma 1.3 to $\boldsymbol{A'}\boldsymbol{x} \leq \boldsymbol{b'}$ ends the proof. □

The variant that we will use to determine polyhedral inclusions is the following one. It builds upon Corollary 1.4 to show that the non-empty polyhedron $\{C_1 \geq 0, \ldots, C_p \geq 0\}$ is included into the single-constraint polyhedron $\{C' \geq 0\}$ if and only if C' can be expressed as a nonnegative affine combinations of $C_1, \ldots, C_p$.

Theorem 1.5 (Farkas generalized) *Let $C_1, \ldots, C_p$ and C' be $p+1$ affine forms from $\mathbb{Q}^n$ to $\mathbb{Q}$. Assume*

$$\left\{ \boldsymbol{x} \in \mathbb{Q}^n \mid \bigwedge_{i=1}^{p} C_i(\boldsymbol{x}) \geq 0 \right\} \neq \emptyset \tag{†}$$

Then,

$$\left\{ \boldsymbol{x} \in \mathbb{Q}^n \mid \bigwedge_{i=1}^{p} C_i(\boldsymbol{x}) \geq 0 \right\} \subseteq \{ \boldsymbol{x} \in \mathbb{Q}^n \mid C'(\boldsymbol{x}) \geq 0 \} \tag{‡}$$

if and only if

$$\exists \lambda_0, \ldots, \lambda_p \in \mathbb{Q}+,\ \forall \boldsymbol{x} \in \mathbb{Q}^n,\ \lambda_0 + \sum_{i=1}^{p} \lambda_i C_i(\boldsymbol{x}) = C'(\boldsymbol{x})$$

Proof.
($\Leftarrow$) : Assume $\exists \lambda_0, \ldots, \lambda_p \in \mathbb{Q}+,\ \forall \boldsymbol{x} \in \mathbb{Q}^n,\ \lambda_0 + \sum_{i=1}^{p} \lambda_i C_i(\boldsymbol{x}) = C'(\boldsymbol{x})$. Let $\tilde{\boldsymbol{x}} \in \mathbb{Q}^n$ such that $\forall i \in \{1, \ldots, p\}$, $C_i(\tilde{\boldsymbol{x}}) \geq 0$. Then, $C'(\tilde{\boldsymbol{x}}) = \lambda_0 + \sum_{i=1}^{p} \lambda_i C_i(\tilde{\boldsymbol{x}}) \geq 0$ since the λ_i's are nonnegative. Thus, the required inclusion holds.

($\Rightarrow$) : Assume (‡), and let us prove that C' is a nonnegative affine combination of $C_1, \ldots, C_p$. Let us define matrices

$$\boldsymbol{A}^\intercal = \begin{pmatrix} 0 & \ldots & 0 & 1 \\ a_{11} & \ldots & a_{1n} & b_1 \\ \ldots & & & \\ a_{p1} & \ldots & a_{pn} & b_p \end{pmatrix} \text{ and } \boldsymbol{b} = \begin{pmatrix} a'_1 \\ \ldots \\ a'_n \\ b' \end{pmatrix}$$

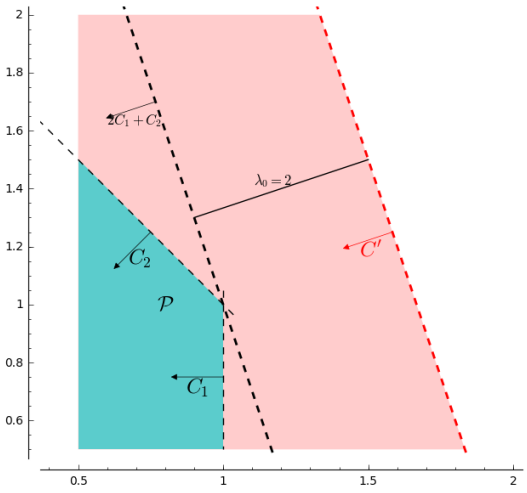
such that for $i \in \{1, \ldots, p\}$, $C_i(\boldsymbol{x}) = \sum_{j=1}^{n} a_{ij}x_j + b_i$ and $C'(\boldsymbol{x}) = \sum_{j=1}^{n} a'_j x_j + b' = \boldsymbol{b}^\intercal \boldsymbol{x}$. By Corollary 1.4, there are two cases:

Either $\exists \boldsymbol{\lambda} \in \mathbb{Q}^{p+1}$, $\boldsymbol{\lambda} \geq \boldsymbol{0}$ and $\boldsymbol{A\lambda} = \boldsymbol{b}$. Then, $\boldsymbol{\lambda}^\intercal \boldsymbol{A}^\intercal = \boldsymbol{b}^\intercal$, and we are done.

Or $\exists \boldsymbol{y} \in \mathbb{Q}^{n+1}$, $\boldsymbol{y}^\intercal \boldsymbol{A} \geq \boldsymbol{0}$, $\boldsymbol{y}^\intercal \boldsymbol{b} < 0$, which is equivalent to $\boldsymbol{A}^\intercal \boldsymbol{y} \geq \boldsymbol{0}$ and $\boldsymbol{b}^\intercal \boldsymbol{y} < 0$. A point $\boldsymbol{y}$ that satisfies $\boldsymbol{y}^\intercal \boldsymbol{A} \geq \boldsymbol{0}$ may exist because we assumed in (†) that the polyhedron $\{C_1 \geq 0, \ldots, C_p \geq 0\}$ is not empty; it means that $\forall i \in \{1, \ldots, p\}$, $C_i(\boldsymbol{y}) \geq 0$. However, $\boldsymbol{b}^\intercal \boldsymbol{y} < 0$, *i.e.* $C'(\boldsymbol{y}) < 0$. Thus, $\boldsymbol{y}$ is a point of $\{C_1 \geq 0, \ldots, C_p \geq 0\}$ that does not belong to $\{C' \geq 0\}$, which contradicts (‡). Hence, this alternative of Corollary 1.4 cannot happen.

□

Example 1.7



The Farkas combination of the form $\lambda_0 + \lambda_1 C_1 + \lambda_2 C_2$, with $\lambda_0 = 2$, $\lambda_1 = 2$ and $\lambda_2 = 1$, proves that $\mathcal{P} : \{C_1 : -x_1 \geq -1,\ C_2 : -x_1 - x_2 \geq -2\}$ implies $C' : -3x_1 - x_2 \geq -6$. Indeed, $C' = 2 + 2C_1 + C_2$.

In other words, the non-empty polyhedron defined by constraints $C_1(\boldsymbol{x}) \geq 0, \ldots, C_p(\boldsymbol{x}) \geq 0$ implies $C'(\boldsymbol{x}) \geq 0$ if and only if C' is a nonnegative affine combination of $C_1, \ldots, C_p$. This combination is called the *Farkas combination* of C' in terms of $\mathcal{P}$. It exists if and only if the system $\boldsymbol{A\lambda} = \boldsymbol{b}$ defined in the proof has a nonnegative solution $\boldsymbol{\lambda}$. This system being itself a polyhedron, we have therefore expressed the polyhedral inclusion as an emptiness test. This problem is addressed in the following lemma.

Theorem 1.6 (Unsatisfiability criterion) *Let $C_1, \ldots, C_p$ be p affine forms from $\mathbb{Q}^n$ to $\mathbb{Q}$. The polyhedron defined by*

$$\mathcal{P} = \left\{ \boldsymbol{x} \in \mathbb{Q}^n \mid \bigwedge_{i=1}^{p} C_i(\boldsymbol{x}) \geq 0 \right\}$$

is empty if and only if

$$\exists \lambda_0, \ldots, \lambda_p \in \mathbb{Q}+,\ \forall \boldsymbol{x} \in \mathbb{Q}^n,\ \lambda_0 + \sum_{i=1}^{p} \lambda_i C_i(\boldsymbol{x}) = -1$$

Proof.
($\Leftarrow$) : Suppose $\exists \lambda_0, \ldots, \lambda_p \in \mathbb{Q}+,\ \forall \boldsymbol{x} \in \mathbb{Q}^n,\ \lambda_0 + \sum_{i=1}^{p} \lambda_i C_i(\boldsymbol{x}) = -1$. Assume there exists a point $\tilde{\boldsymbol{x}} \in \mathcal{P}$. By definition of $\mathcal{P}$, $\lambda_0 + \sum_{i=1}^{p} \lambda_i C_i(\tilde{\boldsymbol{x}}) \geq 0$, which contradicts $\lambda_0 + \sum_{i=1}^{p} \lambda_i C_i(\tilde{\boldsymbol{x}}) = -1$. Thus, there is no such $\tilde{x}$ and $\mathcal{P}$ is empty.
($\Rightarrow$) : Suppose $\mathcal{P} = \emptyset$, meaning that the constraints of $\mathcal{P}$ are unsatisfiable. Then, by Proposition 1.2, the projection of $\mathcal{P}$ on any subspace is also empty. The proof is similar to that of Lemma 1.3. We eliminate each variable of $\mathcal{P}$ by Fourier-Motzkin elimination, until there is only one left, say x_n. Since $\mathcal{P} = \emptyset$, we end up with a contradictory constraint $a \geq x_n \geq b$ with $b > a$. Then, $a \geq b \equiv a - b \geq 0 \equiv \frac{a-b}{|a-b|} \geq 0 \equiv -1 \geq 0$ as $b > a$. Recall that Fourier-Motzkin eliminates one variable by making nonnegative combinations of rows of the input polyhedron. Thus, by multiplying the coefficients obtained by the successive variable eliminations, we can express constraint $-1 \geq 0$ as a nonnegative combination of constraints of $\mathcal{P}$. □

Certifying a Polyhedral Inclusion. Farkas' lemma 1.5 gives a criterion to test the inclusion of a polyhedron in a single constraint. Generalizing this result to the inclusion $\mathcal{P} \sqsubseteq \mathcal{P}'$ of two polyhedra $\mathcal{P} : \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$ and $\mathcal{P}' : \boldsymbol{A'}\boldsymbol{x} \geq \boldsymbol{b'}$ is simple. It suffices to concatenate into a matrix $\boldsymbol{\Lambda}$ each Farkas combination $\boldsymbol{\lambda_i}$ that proves the one-constraint inclusion $\mathcal{P} \sqsubseteq [\![\boldsymbol{A'_i}\boldsymbol{x} \geq b'_i]\!]$. Thus, $\mathcal{P} \sqsubseteq \mathcal{P}'$ if and only if there exists a matrix $\boldsymbol{\Lambda} \in \mathbb{Q}+_{|\mathcal{P}'| \times |\mathcal{P}|}$ of Farkas combinations such that

$$\underbrace{\begin{pmatrix} \boldsymbol{\lambda_1^\intercal} \\ \cdots \\ \boldsymbol{\lambda_{|\mathcal{P}'|}^\intercal} \end{pmatrix}}_{\boldsymbol{\Lambda}} \underbrace{\begin{pmatrix} \boldsymbol{0} & 1 \\ \boldsymbol{A} & \boldsymbol{-b} \end{pmatrix}}_{\boldsymbol{F}} = [\boldsymbol{A'} | \boldsymbol{-b'}] \tag{1.7}$$

If $\boldsymbol{\Lambda}$ exists, its i^{th} row is a vector $\boldsymbol{\lambda_i^\intercal}$ representing the Farkas combination of the i^{th} constraint of $\mathcal{P}'$ in terms of $\mathcal{P}$. The first row of matrix $\boldsymbol{\Lambda}$ is similar to the one that appears within the proof of Theorem 1.5, to handle the λ_0's.

Example 1.8

Let us show that $\mathcal{P} : \{C_1 : x_1 + x_2 \geq 1,\ C_2 : x_1 - x_2 \geq 2,\ C_3 : x_1 \geq 0\}$ is included into $\mathcal{P}' : \{C'_1 : 3x_1 + x_2 \geq 4,\ C'_2 : 2x_1 - x_2 \geq 2\}$. $\mathcal{P}$ is included in C'_1 because $C'_1 = 1 + 2C_1 + C_2$. Thus, its associated Farkas combination is $\boldsymbol{\lambda_1} = (1, 2, 1, 0)^\intercal$. Similarly, the Farkas combination of C'_2 in terms of $\mathcal{P}$ is $\boldsymbol{\lambda_2} = (0, 0, 1, 1)$ since $C'_2 = C_2 + C_3$.

Thus, the matrix $\boldsymbol{\Lambda} \stackrel{\text{def}}{=} \begin{pmatrix} \boldsymbol{\lambda_1^\intercal} \\ \boldsymbol{\lambda_2^\intercal} \end{pmatrix}$ shows that $\mathcal{P} \sqsubseteq \mathcal{P}'$, because

$$\underbrace{\begin{pmatrix} 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}}_{\boldsymbol{\Lambda}} \underbrace{\begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & -1 \\ 1 & -1 & -2 \\ 1 & 0 & 0 \end{pmatrix}}_{\boldsymbol{F}} = \underbrace{\begin{pmatrix} 3 & 1 & -4 \\ 2 & -1 & -2 \end{pmatrix}}_{[\boldsymbol{A'}|\boldsymbol{-b'}]}$$

Matrix $\boldsymbol{\Lambda}$ is a *certificate* that proves $\mathcal{P} \sqsubseteq \mathcal{P}'$. Computing $\boldsymbol{\Lambda}$ requires solving LP problems, but checking that the inclusion holds is easy once $\boldsymbol{\Lambda}$ is known. It suffices to compute the matrix product $\boldsymbol{\Lambda F}$ of (1.7), and check that it is equal to the right hand side (the operator result). This is how the correctness of the VPL is ensured: an untrusted oracle produces a matrix of certificates $\boldsymbol{\Lambda}$, verified by a checker developed and certified in Coq. Actually, the VPL provides an optimization of this verification process: instead of checking that equation (1.7) holds and returning the oracle result $[\boldsymbol{A'}|\boldsymbol{-b'}]$, the certified operator can directly output the product $\boldsymbol{\Lambda F}$. Indeed, this result is sound by construction, and it saves a matrix equality check in Coq. See Part II for more details about the VPL certification.

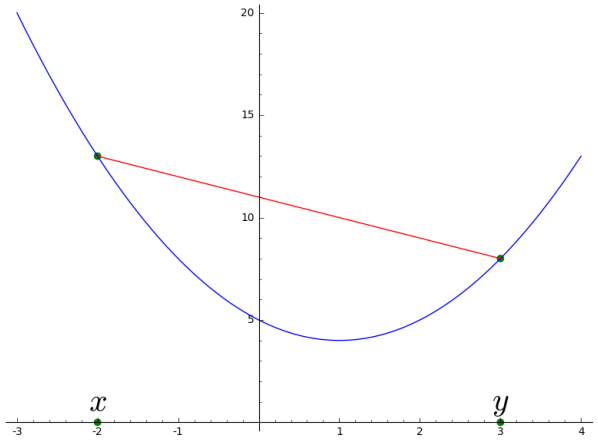


Figure 1.8 – Example of convex function: $f(x) = x^2 - 2x + 5$.

1.3 Solving Linear Programming Problems

Convex Optimization. *Convex optimization* is a field of mathematical optimization problems, that focuses on problems of the form

$$\begin{aligned} \textbf{minimize } \mathbf{Z} &\stackrel{\text{def}}{=} f_0(\boldsymbol{x}) \\ \textbf{subject to } & f_i(\boldsymbol{x}) \geq 0, \ i = 1, \ldots, m \end{aligned}$$

where functions $f_0, f_1, \ldots, f_m : \mathbb{R}^n \to \mathbb{R}$ are convex, *i.e.* satisfy

$$f_i(\alpha \boldsymbol{x} + (1-\alpha)\boldsymbol{y}) \leq \alpha f_i(\boldsymbol{x}) + (1-\alpha) f_i(\boldsymbol{y}), \ \forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n, \ \forall \alpha \in [0,1]$$

This property means that, for every two points $\boldsymbol{x} \stackrel{\text{def}}{=} (x_1, \ldots, x_n)$ and $\boldsymbol{y} \stackrel{\text{def}}{=} (y_1, \ldots, y_n)$, the segment $[(x_1, \ldots, x_n, f_i(\boldsymbol{x})), (y_1, \ldots, y_n, f_i(\boldsymbol{y}))]$ dominates the curve of f_i on $[\boldsymbol{x}, \boldsymbol{y}]$, as illustrated on Fig. 1.8. Note that we denote by $[\boldsymbol{x}, \boldsymbol{y}]$ the segment joining $\boldsymbol{x}$ and $\boldsymbol{y}$, *i.e.* the set of points $\{\alpha \boldsymbol{x} + (1-\alpha)\boldsymbol{y} \mid 0 \leq \alpha \leq 1\}$. Some interesting properties of convex functions make them particularly well-suited for optimization. For instance, a local minimum is also a global one and if the objective is strictly convex – *i.e.* satisfies the previous property with a strict comparison sign, and $\alpha \in]0,1[$ – then the minimum is unique. Linear programming problems, introduced in the next section, are particular instances of convex optimization problems, focusing on affine functions. For more information on convex optimization, I refer the interested reader to the book of Boyd and Vandenberghe (2004).

1.3.1 Linear Programming

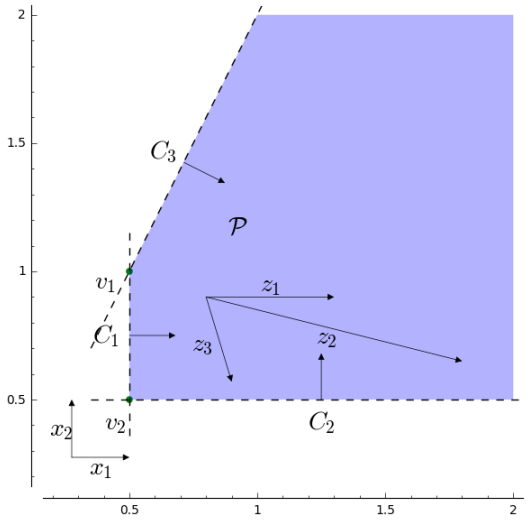
As §1.2.2 pointed out, deciding a polyhedral inclusion requires solving a system of affine inequalities, *i.e.* finding a point within a polyhedron. A *Linear Programming* (LP) problem is a convex optimization problem of the form

$$\begin{aligned} \textbf{minimize } \mathbf{Z} &\stackrel{\text{def}}{=} \boldsymbol{c}^\intercal \boldsymbol{x} \\ \textbf{subject to } & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \end{aligned}$$

Solving this problem consists in minimizing the linear *objective function* $\boldsymbol{c}^\intercal \boldsymbol{x} = \sum_{i=1}^{n} c_i x_i$ over polyhedron $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$. Variables $\boldsymbol{x} = (x_1, \ldots, x_n)$ are called *decision variables*. An *optimal solution* is a point $\boldsymbol{x}^\star$ – often noted with a superscript star – of the polyhedron $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$ such that for all $\boldsymbol{x}$ satisfying $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$, $\boldsymbol{c}^\intercal \boldsymbol{x}^\star \leq \boldsymbol{c}^\intercal \boldsymbol{x}$. The value $\boldsymbol{c}^\intercal \boldsymbol{x}^\star$ is called the *optimal value*.
A LP problem is said *satisfiable* if the polyhedron $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$, called the *feasible space*, is nonempty. If so, a point $\boldsymbol{x}$ of the polyhedron is called a *feasible solution*. Otherwise, the

problem is *unsatisfiable*. A problem is *unbounded* if it is feasible and if there is no optimal solution, meaning that the objective function can decrease infinitely on the polyhedron. If the problem is bounded, having a polyhedral feasible space ensures that at least one of its vertices is an optimal solution. An optimal solution is not necessarily unique, for several points can lead to the optimal value.

Example 1.10



Consider the polyhedron $\mathcal{P} : \{C_1 : 2x_1 \geq 1,\ C_2 : 2x_2 \geq 1,\ C_3 : 2x_1 - x_2 \geq 0\}$ of Example 1.2 and the directions of optimization $\boldsymbol{Z_1} = (\frac{1}{2}, 0)$, $\boldsymbol{Z_2} = (1, -\frac{1}{4})$ and $\boldsymbol{Z_3} = (\frac{1}{10}, -\frac{1}{3})$. The following LP problem has an optimal value of $\frac{1}{4}$, which is reached in all points of segment $[\boldsymbol{v_1}, \boldsymbol{v_2}]$. In particular, both vertices $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ are optimal solutions.

$$\begin{aligned} \textbf{minimize } \mathbf{Z} \stackrel{\text{def}}{=} \boldsymbol{Z_1^\intercal x} \textit{ i.e. } & \frac{1}{2}x_1 \\ \textbf{subject to } & 2x_1 \geq 1 \\ & 2x_2 \geq 1 \\ & 2x_1 - x_2 \geq 0 \end{aligned}$$

The next LP problem has also an optimal value of $\frac{1}{4}$, which is only reached in $\boldsymbol{v_1}$.

$$\begin{aligned} \textbf{minimize } \mathbf{Z} \stackrel{\text{def}}{=} \boldsymbol{Z_2^\intercal x} \textit{ i.e. } & x_1 - \frac{1}{4}x_2 \\ \textbf{subject to } & 2x_1 \geq 1 \\ & 2x_2 \geq 1 \\ & 2x_1 - x_2 \geq 0 \end{aligned} \qquad \text{(LP 1.8)}$$

The last problem is unbounded.

$$\begin{aligned} \textbf{minimize } \mathbf{Z} \stackrel{\text{def}}{=} \boldsymbol{Z_3^\intercal x} \textit{ i.e. } & \frac{1}{10}x_1 - \frac{1}{3}x_2 \\ \textbf{subject to } & 2x_1 \geq 1 \\ & 2x_2 \geq 1 \\ & 2x_1 - x_2 \geq 0 \end{aligned}$$

Solving LP Problems. There exists two main categories of algorithms for solving LP problems:

- *Interior point methods* start from a point in the interior of the feasible space. These methods are iterative algorithms that compute a sequence of interior points, as the name suggests, each time closer to the optimum. One of the most famous is the barrier method (Boyd and Vandenberghe, 2004), in which the sequence of interior points follows a *central path*. This path is determined by a combination of the objective function, that guides the path towards the optimum, and a *barrier function* that keeps points inside the feasible space. A barrier function is designed to have a reasonable behaviour inside the feasible space, and to diverge close to frontiers. The barrier method has a guaranteed precision, meaning that given an error ϵ, there exists a (computable) bound on the number of steps needed to reach an ϵ-close of the optimum.
- The *simplex algorithm* starts from a vertex of the feasible space, and moves from one vertex to another until an optimum is reached. The "moving" operation (actually called the *pivot*) is designed so that each iteration increases the current objective value.

Solving LP problems is a mean to check polyhedral inclusions. Proving $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \sqsubseteq \boldsymbol{A'}\boldsymbol{x'} \leq \boldsymbol{b'}$ boils down to determining a Farkas combination as given in Equation (1.7) (p.24), that we recall here:

$$\underbrace{\begin{pmatrix} \boldsymbol{\lambda}_1^\intercal \\ \cdots \\ \boldsymbol{\lambda}_{|\mathcal{P}'|}^\intercal \end{pmatrix}}_{\boldsymbol{\Lambda}} \begin{pmatrix} \boldsymbol{0} & 1 \\ \boldsymbol{A} & -\boldsymbol{b} \end{pmatrix} = [\boldsymbol{A'}| - \boldsymbol{b'}]$$

To prove that this equation holds, we need the exact coefficients of the Farkas combination (the matrix $\boldsymbol{\Lambda}$). The simplex algorithm is more suitable for certification, since the interior point method gives an approximated solution.

The next section introduces the simplex algorithm. The reader may notice that we introduced linear programming to find solutions of a system of affine inequalities, which is a satisfiability problem. But what we defined so far were optimization problems. Actually, finding a feasible solution is the *initialization* step of the simplex algorithm. We will see that initialization and optimization are very similar steps, for they are based on the same operation: the pivot.

1.3.2 The Simplex Algorithm

Understanding the simplex algorithm is important to follow the rest of the thesis, especially when we will address Parametric Linear Programming. This section summarizes the basics and introduces the vocabulary that will be needed later. We restrict our presentation to our use case where decision variables are nonnegative rationals. Note that nonnegativity constraints are not explicitly written in the system of constraints: they are assumed by the restricted algorithm that we will see below. In the general algorithm, variables are instead attached to an interval of values – not necessarily bounded. Here, we thus consider that each variable belongs to $[0, +\infty[$. For more details about LP and the general simplex algorithm, refer to the seminal book of Chvátal (1983).

The overall principle of the simplex algorithm is to start from a vertex of polyhedron $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$ and travel from one vertex to another, until the optimum is reached. If the optimum exists, it is always reached on a vertex. The first phase – finding a starting vertex – is called *initialization*, and the second one is the *optimization* phase.

Echelon Form. In practice, the algorithm puts the system of constraints and the objective function in *echelon form* [2], as in Gaussian elimination. We will see that it syntactically exposes the currently visited vertex and that it indicates how to find a neighbor vertex that improves the objective value. An *echelon form* matrix of n rows and m columns is the concatenation of the identity matrix I_n and a $n \times (m - n)$ matrix.

2. What we call *echelon form* is usually known as *reduced row echelon form*.

Example 1.11

Here is an example of echelon form matrix.

$$\begin{array}{ccccc} 1 & 0 & 0 & -2 & 1 \\ 0 & 1 & 0 & 3 & 0 \\ 0 & 0 & 1 & 0 & 2 \end{array}$$

We abusively say that

$$\begin{array}{ccccc} 0 & 1 & 0 & -2 & 1 \\ 1 & 0 & 0 & 3 & 0 \\ 0 & 0 & 1 & 0 & 2 \end{array}$$

is also in echelon form.

We abusively say that a system is in echelon form if we can find a reordering of the columns that syntactically gives an echelon form matrix as defined above.

Before solving the problem, the simplex algorithm transforms each inequality of the problem into an equation, by adding *slack variables*. An inequality $\boldsymbol{ax} \geq b$ is changed into $\boldsymbol{ax} - s = b$, introducing an additional constraint $s \geq 0$. These two constraints are equivalent provided that the added slack variable s, which represents the gap between $\boldsymbol{ax}$ and b, is nonnegative. Recall that, in our restricted version of the simplex algorithm, nonnegativity constraints are not explicitly written in the system of constraints. Thus, s is implicitly considered nonnegative.[3]

Dictionaries. To put a system of affine equations in echelon form, we partition the set of variables – both decision and slack variables – into a set $\mathcal{B}$ of *basic* variables and a set $\mathcal{N}$ of *nonbasic* ones. The set of basic variables forms the *basis*. Once the partition is chosen, the system of equations and the objective function are then rewritten so that basic variables become expressed in terms of nonbasic ones, leading to an echelon form. When rewritten in this way, the system is called a *dictionary*: each basic variable is associated with an equation that gives its expression in terms of nonbasic variables.

Example 1.12 (follows 1.10)

Adding slack variables to the problem (LP 1.8) of Example 1.10 gives

$$\begin{array}{rcl} 2x_1 - s_1 & = & 1 \\ 2x_2 - s_2 & = & 1 \\ 2x_1 - x_2 - s_3 & = & 0 \\ \boldsymbol{Z} & = & x_1 - \frac{1}{4}x_2 \end{array}$$

where $\boldsymbol{Z}$ denotes the objective function. We can rewrite this system of equations into an echelon form by selecting a set of basic variables. With $\mathcal{B} = \{x_1, x_2, s_3\}$ and $\mathcal{N} = \{s_1, s_2\}$, we obtain the following dictionary

3. Note that for a strict inequality $\boldsymbol{ax} > b$, the additional variable s is required to be strictly positive. To represent such variable in the general simplex algorithm, we would attach an open interval $]0, +\infty[$ to s.

$$\begin{aligned} x_1 &= \frac{1}{2}s_1 + \frac{1}{2} \\ x_2 &= \frac{1}{2}s_2 + \frac{1}{2} \\ s_3 &= s_1 - \frac{1}{2}s_2 + \frac{1}{2} \\ \boldsymbol{Z} &= \frac{1}{2}s_1 - \frac{1}{8}s_2 + \frac{3}{8} \end{aligned} \qquad \text{(Dict. 1.9)}$$

To clearly see that this system is in echelon form, let us write its augmented matrix. For a better understanding, each row of the matrix is preceded by the basic variable that is defined by the row, and the system of equations is written in the same form on the right hand side. The last row represents the objective function.

$$S \stackrel{\text{def}}{=} \begin{array}{c|cccccc|c} & \boldsymbol{Z} & x_1 & x_2 & s_1 & s_2 & s_3 & \\ x_1 & 0 & 1 & 0 & -\frac{1}{2} & 0 & 0 & -\frac{1}{2} \\ x_2 & 0 & 0 & 1 & 0 & -\frac{1}{2} & 0 & -\frac{1}{2} \\ s_3 & 0 & 0 & 0 & -1 & \frac{1}{2} & 1 & -\frac{1}{2} \\ \hline \boldsymbol{Z} & 1 & 0 & 0 & -\frac{1}{2} & \frac{1}{8} & 0 & -\frac{3}{8} \end{array} \qquad \begin{array}{rcl} x_1 - \frac{1}{2}s_1 - \frac{1}{2} & = & 0 \\ x_2 - \frac{1}{2}s_2 - \frac{1}{2} & = & 0 \\ s_3 - s_1 + \frac{1}{2}s_2 - \frac{1}{2} & = & 0 \\ \hline \boldsymbol{Z} - \frac{1}{2}s_1 + \frac{1}{8}s_2 - \frac{3}{8} & = & 0 \end{array}$$

By reordering the columns to take into account the partition into basic and nonbasic variables, the identity matrix appears on the left hand side.

$$\begin{array}{c|cccccc|c} & x_1 & x_2 & s_3 & \boldsymbol{Z} & s_1 & s_2 & \\ x_1 & 1 & 0 & 0 & 0 & -\frac{1}{2} & 0 & -\frac{1}{2} \\ x_2 & 0 & 1 & 0 & 0 & 0 & -\frac{1}{2} & -\frac{1}{2} \\ s_3 & 0 & 0 & 1 & 0 & -1 & \frac{1}{2} & -\frac{1}{2} \\ \hline \boldsymbol{Z} & 0 & 0 & 0 & 1 & -\frac{1}{2} & \frac{1}{8} & -\frac{3}{8} \end{array}$$

A dictionary is a way of syntactically representing a point and its associated solution. A dictionary (or equivalently its basis) is said *feasible* if, by giving the value 0 to each nonbasic variable and by evaluating basic ones accordingly, the obtained point belongs to the feasible space. Note that, when giving the value 0 to each nonbasic variable, the value of a basic variable is reduced to the constant term of its associated equation and the value of the objective function $\boldsymbol{Z}$ is the constant term of its associated equation. For instance in Example 1.12, the dictionary $\mathcal{B} = \{x_1, x_2, s_3\}$ and $\mathcal{N} = \{s_1, s_2\}$ gives the feasible point $(s_1 = 0,\ s_2 = 0,\ s_3 = \frac{1}{2},\ x_1 = \frac{1}{2},\ x_2 = \frac{1}{2})$ associated to the value $\boldsymbol{Z} = \frac{3}{8}$. Looking at the feasible space in Example 1.10, we see that the point $\left(x_1 = \frac{1}{2}, x_2 = \frac{1}{2}\right)$ corresponds to vertex $\boldsymbol{v_2}$.

Any partition into basic and nonbasic does not necessarily give an initial dictionary that is feasible. Indeed, each equation of the dictionary must define exactly one basic variable. For instance, in Example 1.12, the basis $\mathcal{B} = \{x_1, s_1, s_3\}$ is not allowed because the second equation $x_2 = \frac{1}{2}s_2 + \frac{1}{2}$ would be left with no basic variables. We say that such a basis is not *well-defined*: it cannot express the system of equations. Note that a dictionary can be well-defined but infeasible. For instance, the basis $\mathcal{B} = \{x_1, s_2, s_3\}$ gives the following dictionary, which is well-defined but infeasible because s_2 has the negative value -1.

$$\begin{aligned} x_1 &= \frac{1}{2}s_1 + \frac{1}{2} \\ s_2 &= 2x_2 - 1 \\ s_3 &= s_1 - x_2 + 1 \\ \boldsymbol{Z} &= \frac{1}{2}s_1 - \frac{1}{4}x_2 + \frac{4}{8} \end{aligned}$$

The simplex algorithm can start with an infeasible but well-defined dictionary that will be repaired during the initialization phase.

Pivoting. The core operation of the simplex algorithm is the *pivot*. It consists in swapping a basic variable $x_b \in \mathcal{B}$ with a nonbasic one $x_n \in \mathcal{N}$: x_b leaves the basis while x_n enters it. To do so, x_n is expressed in terms of x_b and other nonbasic variables. This expression is then propagated in the dictionary to eliminate x_n from other equations. Obviously, this is possible only if x_n appears in the equation associated to x_b. Otherwise, there is no way to express x_n in terms of x_b. If well chosen, a pivot changes a feasible dictionary (and its associated feasible point) into another feasible one: this is how the simplex algorithms travels from one vertex to another. In practice, a pivot is applied on dictionaries like a pivot in the Gaussian elimination, but the choice of the pivoting variables (x_n, x_b) must obey some rules that will be detailed.

Example 1.13 (follows 1.12)

Let us perform a pivot on dictionary (Dict. 1.9) (and it associated matrix S) of Example 1.12 between variables s_3 (the leaving variable) and s_2 (the entering variable). The equation defining s_3 in terms of nonbasic variables in (Dict. 1.9) is $s_3 = s_1 - \frac{1}{2}s_2 + \frac{1}{2}$, which corresponds to the third row of S. As we want the coefficient of s_2 to become 1, let us multiply this equation by 2 and isolate s_2: $s_2 = 2s_1 - 2s_3 + 1$. Then, we use this definition of s_2 to eliminate it in other rows. This operation is equivalent to the substitution of s_2 by $2s_1 - 2s_3 + 1$. The resulting dictionary (and its associated matrix) are:

$$\begin{aligned} x_1 &= \frac{1}{2}s_1 + \frac{1}{2} \\ x_2 &= s_1 - s_3 + 1 \\ s_2 &= 2s_1 - 2s_3 + 1 \\ \boldsymbol{Z} &= \frac{1}{4}s_1 + \frac{1}{4}s_2 + \frac{1}{4} \end{aligned} \qquad \text{(Dict. 1.10)}$$

$$\begin{array}{c} \\ x_1 \\ x_2 \\ s_2 \\ \hline \boldsymbol{Z} \end{array} \begin{array}{c} \begin{array}{ccccccc} \boldsymbol{Z} & x_1 & x_2 & s_1 & s_2 & s_3 & \end{array} \\ \left(\begin{array}{cccccc|c} 0 & 1 & 0 & -\frac{1}{2} & 0 & 0 & -\frac{1}{2} \\ 0 & 0 & 1 & -1 & 0 & 1 & -1 \\ 0 & 0 & 0 & -2 & 1 & 2 & -1 \\ \hline 1 & 0 & 0 & -\frac{1}{4} & 0 & -\frac{1}{4} & -\frac{1}{4} \end{array}\right) \end{array}$$

The new basis is $\{x_1, x_2, s_2\}$. The feasible point corresponding to this dictionary is $(x_1 = \frac{1}{2}, x_2 = 1)$, which is vertex $\boldsymbol{v_1}$ from Example 1.10.

1.3.2.1 Optimization Phase

Let us see how to reach the optimum value from a feasible system of constraints. Since we wish to minimize the objective value, the idea is to look for a nonbasic variable x_n with a negative coefficient in the objective function. Since variables are assumed nonnegative, increasing the value of x_n will decrease the objective value. As nonbasic variables have value 0 in dictionaries, increasing the value of x_n can only be done if it enters the basis. Then, we need a basic variable to leave the basis to let some room for x_n.

For instance, in dictionary (Dict. 1.9) from Example 1.12, the objective is expressed as $\boldsymbol{Z} = \frac{1}{2}s_1 - \frac{1}{8}s_2 + \frac{3}{8}$. As $s_2 \geq 0$, by increasing s_2, the objective value will decrease by the quantity $-\frac{1}{8}s_2$. Optimality is eventually reached because the simplex algorithm always increases the entering variable (s_2 here) as much as possible, being careful not to violate any constraint.

Let us study equations of the dictionary to find an equation that could limit the increase of s_2.

- $x_1 = \frac{1}{2}s_1 + \frac{1}{2}$: s_2 does not appear in this equation.
- $x_2 = \frac{1}{2}s_2 + \frac{1}{2}$: By increasing s_2, we must ensure that x_2 stays nonnegative. Here, s_2 has a positive coefficient in the definition of x_2. Thus, increasing s_2 will increase as well the value of x_2: this equation does not limit the increase of s_2.
- $s_3 = s_1 - \frac{1}{2}s_2 + \frac{1}{2}$: By changing s_2, we must ensure that s_3 stays nonnegative. This time, s_2 has a negative coefficient in the definition of s_3. Thus, increasing s_2 will decrease the value of s_3. The maximal suitable value for s_2 is reached when $-\frac{1}{2}s_2 + \frac{1}{2} = 0$, which is $s_2 = 1$.

s_2 increase is limited by the equation of s_3, thus this equation is chosen as definition of s_2: s_2 enters the basis and s_3 leaves it. The pivot operation given in Example 1.13 performs the swapping and leads to the new objective function of (Dict. 1.10) which is $Z = \frac{1}{4}s_1 + \frac{1}{4}s_2 + \frac{1}{4}$. It no longer contains any negative coefficient, meaning that there is no more way to minimize the objective: an optimum has been reached!

To summarize, the optimization phase consists in iterating the three following steps:

1. Find a variable $x_n \in \mathcal{N}$ with a negative coefficient in the objective function. If there is no such x_n, an optimum has been reached.
2. Find the variable $x_b \in \mathcal{B}$ that limits the most the increase of x_n. If there is no such x_b, it means that there is no limit to the increase of x_n, and therefore the problem is unbounded.
3. Perform the pivot $x_b \leftrightarrow x_n$. Go back to step (1).

1.3.2.2 Initialization Phase

Before the optimization phase, we must find a feasible dictionary (*i.e.* associated to a feasible point). Suppose the initial LP problem was defined as follows.

$$\begin{aligned} \textbf{minimize } \mathbf{Z} \stackrel{\text{def}}{=} & \ \boldsymbol{c}^\intercal \boldsymbol{x} \\ \textbf{subject to} & \ \boldsymbol{a_i x} \geq b_i, \ \forall i \in \{1, \dots, p\} \\ & \ x_i \geq 0, \ \forall i \in \{1, \dots, n\} \end{aligned} \tag{LP 1.11}$$

Chvátal (1983) introduces an auxiliary variable $y \geq 0$ and defines an *auxiliary problem* that finds a feasible basis, if such exists:

$$\begin{aligned} \textbf{minimize } \mathbf{Z} \stackrel{\text{def}}{=} & \ y \\ \textbf{subject to} & \ \boldsymbol{a_i x} + y \geq b_i, \ \forall i \in \{1, \dots, p\} \\ & \ x_i \geq 0, \ \forall i \in \{1, \dots, n\} \\ & \ y \geq 0 \end{aligned} \tag{LP 1.12}$$

Problem (LP 1.11) has a solution if and only if the optimal value for y in (LP 1.12) is 0. Let us try to solve (LP 1.12). First, we add the slack variables $(s_i)_{i \in \{1,\dots,p\}}$:

$$\begin{aligned} \textbf{minimize } \mathbf{Z} \stackrel{\text{def}}{=} & \ y \\ \textbf{subject to} & \ \boldsymbol{a_i x} + y - s_i = b_i, \ \forall i \in \{1, \dots, p\} \\ & \ x_i \geq 0, \ \forall i \in \{1, \dots, n\} \\ & \ s_i \geq 0, \ \forall i \in \{1, \dots, p\} \\ & \ y \geq 0 \end{aligned}$$

Let us write the dictionary where all slack variables are basic:

$$\begin{aligned} s_i &= \boldsymbol{a_i x} + y - b_i, \ \forall i \in \{1, \dots, p\} \\ z &= y \end{aligned}$$

This LP problem has a feasible point: choosing $x_i = 0$ and $y = max\ \{0\} \cup \{b_i \mid i = 1, \ldots, p\}$ produces nonnegative values for all slack variables $s_i = y - b_i$. Then, if $y = 0$, the basis $\mathcal{B} = \{s_i \mid i \in \{1, \ldots, p\}\}$ leads to a feasible dictionary for the initial problem (LP 1.11). Otherwise, it means that this dictionary gives a negative value for at least one s_i. The dictionary can be fixed by performing one single pivot, with y entering the basis so that it can take a value greater than 0. Actually, y should have value $max\ \{0\} \cup \{b_i \mid i = 1, \ldots, p\}$ to ensure a nonnegative value for all $s_i = y - b_i$ when $x_i = 0$. In fact, the standard pivoting rule that we described before will adjust the value y.

The leaving variable, say s_ℓ, is the slack variable with the minimal value in the dictionary. After this pivot $y \leftrightarrow s_\ell$, s_ℓ will have value 0, and every other s_i will be nonnegative, leading to a feasible dictionary for (LP 1.12). Then,

- Either the minimal value for y is 0, in which case the final dictionary of the auxiliary problem can easily be converted into a feasible dictionary for the initial problem, by simply replacing y by 0.
- Or the minimal value for y is strictly positive, in which case the initial problem (LP 1.11) is infeasible.

Complexity of the Simplex Algorithm. Although the ellipsoid algorithm shows that rational LP problem resolution belongs to P, the simplex algorithm has an exponential worst case complexity in the number of vertices of the feasible space (Chvátal, 1983). Still, it behaves much better in practice.
The efficiency of the algorithm is sensitive to the choice of the entering variable in the first step of the optimization phase. The simplest choice is the use of a lexicographic order on variables: when two or more variables are eligible for entering the basis at the same time, pick the minimal one for the lexicographic order. Bland (1977) showed that this heuristic ensures the termination of the simplex algorithm.

Implementation in the VPL. A. Fouilhé implemented the general simplex algorithm in the VPL, meaning that decision variables are not assumed nonnegative. It can handle strict inequalities by manipulating a symbolic error ϵ: it gives symbolic values of the form $(q, d{\cdot}\epsilon)$ with $q, d \in \mathbb{Q}$ to coordinates of vertices. (q, d) is a solution to $x > 0$ if $q > 0$ or $(q = 0 \wedge d > 0)$.

Part I

Toward a Scalable Polyhedral Domain

Chapter 2

Minimization by Raytracing

We presented in §1.2.1 the minimization operator in constraints-only, for removing redundant constraints from a polyhedron representation. In general, deciding redundancies is expensive as it requires solving one LP problem for each constraint. The goal of this chapter is to introduce a new and efficient minimization algorithm based on raytracing, implemented in the VPL. It consists in launching rays starting from a point within the polyhedron and orthogonal to its bounding hyperplanes. A constraint first encountered by one of these rays is irredundant. Since this procedure is incomplete (for a finite number of launched rays), LP problem resolutions are still required for the remaining undetermined constraints. This work was published and presented at the 18^{th} International Conference on Verification, Model Checking, and Abstract Interpretation (Maréchal and Périn, 2017), in Paris. As we will see, this algorithm is expressed in terms of polyhedral cones and can therefore be applied either on constraints or generators.

2.1 Redundancy in Polyhedra

The addition of new constraints or generators introduces redundancies which must be removed to reduce memory consumption and avoid useless computations in subsequent operations. The emergence of redundancies is illustrated by the figure of Example 2.1: when constraint C' is added into $\mathcal{P}_a$ to form $\mathcal{P}_b$, constraints C_3 and C_4 become redundant. Conversely, the addition of points $\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}$ into $\mathcal{P}_b$ generates $\mathcal{P}_a$ and makes $\boldsymbol{v'_1}$ and $\boldsymbol{v'_2}$ redundant.

As explained in §1.2.2.1 (p. 20), in constraints-only representations, redundant constraints tend to grow exponentially during the computation of a projection by Fourier-Motzkin elimination (Simon and King, 2005). For a description by generators, the same pitfall occurs in Chernikova's conversion when a polyhedron is sliced with a constraint (Fukuda and Prodon, 1996). This motivates the search for efficient ways of detecting redundancies.

Characterization of Redundancy. A ray $\boldsymbol{r}_k$ is redundant if it is a nonnegative combination of some other rays. A point $\boldsymbol{v_k}$ is redundant if it is a convex combination of the other generators, *i.e.* $\boldsymbol{v_k} = \sum_{i \neq k} \beta_i \boldsymbol{v_i} + \sum_i \lambda_i \boldsymbol{r}_i$ for some $\beta_i, \lambda_i \geq 0$ with $\sum \beta_i = 1$. Back to the figure of Example 2.1, equations $\boldsymbol{v'_1} = 1 \times \boldsymbol{v_1} + 2 \times \boldsymbol{r}_1$ and $\boldsymbol{v'_2} = 1 \times \boldsymbol{v_3} + 1 \times \boldsymbol{r}_1$ prove the redundancy of $\boldsymbol{v'_1}$ and $\boldsymbol{v'_2}$ in $\mathcal{P}_a$. Therefore, these equations account for *certificates of redundancy*.
Intuitively, a constraint is redundant if it is useless, in the sense that adding it does not change the geometrical space delimited by the polyhedron. Formally, as explained by Farkas' lemma 1.5 in §1.2.2, a constraint C_k is redundant if it is a nonnegative combination of other constraints.

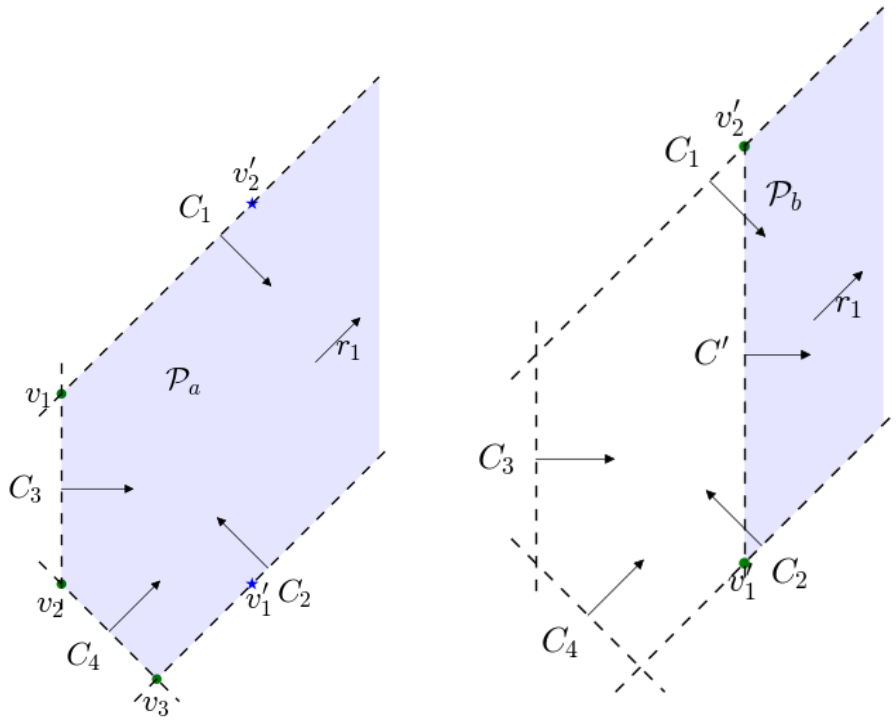
If only one representation is available – as generators or as constraints – discovering redundancy requires solving LP problems of the form "does there exist nonnegative scalars satisfying some linear equations?":

$$\exists \lambda_0, \ldots, \lambda_{|p|} \geq 0,\ C_k = \sum_{i=1, i\neq k}^{|p|} \lambda_i C_i + \lambda_0 \qquad \textit{(1) for constraints}$$

$$\exists \lambda_1, \ldots, \lambda_{|r|} \geq 0,\ \boldsymbol{r}_k = \sum_{i=1, i\neq k}^{|r|} \lambda_i \boldsymbol{r}_i \qquad \textit{(2) for rays}$$

$$\exists \beta_1, \ldots, \beta_{|v|}, \lambda_1, \ldots, \lambda_{|r|} \geq 0,\ \boldsymbol{v_k} = \sum_{i=1, i\neq k}^{|v|} \beta_i \boldsymbol{v_i} + \sum_{i=1}^{|r|} \lambda_i \boldsymbol{r}_i \quad \textit{(3) for vertices}$$
$$\wedge \sum_{i=1, i\neq k}^{|v|} \beta_i = 1$$

Example 2.1



Let $\mathcal{P}_a$ be defined equivalently by constraints $\{C_1 : x_2 - x_1 <= 1,\ C_2 : x_2 - x_1 \geq 5,\ C_3 : x_1 \geq 1,\ C_4 : x_1 + x_2 \geq 0\}$ or by generators $\{\boldsymbol{v_1} : (1,2),\ \boldsymbol{v_2} : (1,-1),\ \boldsymbol{v_3} : (\frac{5}{2}, -\frac{5}{2}),\ \boldsymbol{r}_1 : (1,1)\}$. Let $\mathcal{P}_b$ be defined equivalently by constraints $\{C_1 : x_2 - x_1 <= 1,\ C_2 : x_2 - x_1 \geq 5,\ C' : x_1 \geq 4\}$ or by generators $\{\boldsymbol{v_1'} : (4,-1),\ \boldsymbol{v_2'} : (4,5),\ \boldsymbol{r}_1 : (1,1)\}$. The equations $C_3 = 3 + C'$ and $C_4 = 13 + (2 \times C') + (1 \times C_2)$ are the Farkas decompositions of C_3 and C_4. They act as certificates of redundancy. Indeed,

$$C_3 : x_1 - 1 \geq 0 \equiv 3 + (x_1 - 4 \geq 0)$$

and

$$C_4 : x_1 + x_2 \geq 0 \equiv 13 + 2 \times (x_1 - 4 \geq 0) + (x_2 - x_1 - 5 \geq 0)$$

Polyhedral Cones. The way to reconcile those three definitions of redundancy is to switch to *polyhedral cones*. A polyhedral cone is defined by

$$\left\{ \boldsymbol{x} \in \mathbb{Q}^n \mid \boldsymbol{x} = \sum_{i=1}^{|r|} \lambda_i \boldsymbol{r}_i,\ \lambda_i \geq 0 \right\}$$

where $\boldsymbol{r}_i$'s are vectors of $\mathbb{Q}^n$. These vectors act as rays in the generator representation of a polyhedron. Therefore, a polyhedral cone is a special case of polyhedron, defined only by a finite set of rays and a single vertex, that is the origin $\mathbf{0} \stackrel{\text{def}}{=} (0, \ldots, 0)$. This vertex can be kept implicit since $\mathbf{0}$ is necessarily part of any such cone (obtained with $\lambda_i = 0, \forall i = 1..|r|$). As a consequence, a constraint of a polyhedral cone has no constant term, which leads to a homogeneous system of constraints. From now on, since we only consider cones that are polyhedra, we will omit the adjective "polyhedral" and simply talk about "cones".

The trick to change a polyhedron $\mathcal{P}$ – represented as constraints – into a cone is to associate an extra variable η to the constant term $\boldsymbol{b}$ as follows (Wilde, 1993) (see Example 2.2 below): for any $\eta > 0$,

$$\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \equiv \eta(\boldsymbol{A}\boldsymbol{x}) \leq \eta\boldsymbol{b} \equiv \boldsymbol{A}(\eta\boldsymbol{x}) - \eta\boldsymbol{b} \leq 0 \equiv [\boldsymbol{A}| - \boldsymbol{b}] \begin{pmatrix} \eta\boldsymbol{x} \\ \eta \end{pmatrix} \leq 0$$

Goldman and Tucker (1956) proved that $\boldsymbol{x} \in \mathbb{Q}^n$ belongs to $\mathcal{P}$ if and only if $(\boldsymbol{x} \quad 1) \in \mathbb{Q}^{n+1}$ belongs to the cone $\{\boldsymbol{x'} \in \mathbb{Q}^{n+1} \mid [\boldsymbol{A}| - \boldsymbol{b}]\, \boldsymbol{x'} \leq 0\}$. Using this transformation, operators on polyhedra can be implemented as computations on their associated cones producing a cone that, once intersected with the hyperplane $\eta = 1$, is the expected polyhedron. We switch back to general polyhedra in illustrations as they are easier to draw (they have one dimension less than their cone).

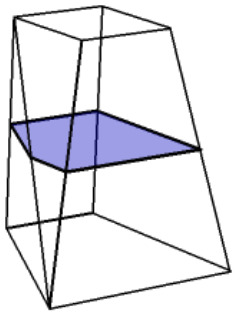
Note that there is a technicality with this transformation into cone: when testing the emptiness of a polyhedron $\mathcal{P}$ represented as a cone, one must have in mind that a cone is never empty, it necessarily contains $\mathbf{0}$. The emptiness test then becomes $\mathcal{P} \neq \emptyset$ iff $\exists (x_1, \ldots, x_n, \eta) \in Cone(\mathcal{P})$ with $\eta > 0$.

Considering cones simplifies the presentation: the constant term of constraints vanishes and the vertices disappear from definitions. This reconciles constraints-only and generators representations, in the sense that the minimization algorithm works on both. We end up with the same definition of redundancy for constraints (1') and for generators (2'): a vector is redundant if it is a nonnegative combination of the other vectors. In particular, the constant term λ_0 that appears in the Farkas decomposition of a redundant constraint is no longer necessary because of the absence of constant terms in constraints. Thus, a constraint C_k (resp. a ray $\boldsymbol{r}_k$) is redundant if

$$\exists \lambda_1, \ldots, \lambda_{|p|} \geq 0, \quad C_k = \sum_{i=1, i\neq k}^{|p|} \lambda_i C_i \qquad \textit{(1') for constraints}$$

$$\exists \lambda_1, \ldots, \lambda_{|r|} \geq 0, \quad \boldsymbol{r}_k = \sum_{i=1, i\neq k}^{|r|} \lambda_i \boldsymbol{r}_i \qquad \textit{(2') for rays}$$

Example 2.2



Consider the unbounded polyhedron $\mathcal{P} = \{2x_1 \geq 1,\ x_2 \geq \frac{1}{2},\ 2x_1 - x_2 \geq 0\}$, that was the running example of Chapter 1. Following the transformation given above, its associated cone is $Cone(\mathcal{P}) = \{2x_1\eta \geq \eta,\ x_2\eta \geq \frac{1}{2}\eta,\ 2x_1\eta - x_2\eta \geq 0,\ \eta \geq 0\} \equiv \{2y_1 - y_3 \geq 0,\ y_2 - \frac{1}{2}y_3 \geq 0,\ 2y_1 - y_2 \geq 0,\ y_3 \geq 0\}$ with the change of variable $y_1 = x_1\eta$, $y_2 = x_2\eta$ and $y_3 = \eta$. This cone is displayed (truncated) on the figure above. It starts from the origin $(0, 0, 0)$ and extends to infinity in the direction of four rays (one for each constraint of $Cone(\mathcal{P})$. When $\eta = 1$, we retrieve $\mathcal{P}$, which is the blue area.

Deciding Redundancy. The redundant/irredundant status of a constraint or a ray depends on the satisfiability of an existential problem *(1',2')* involving linear equations but also inequalities ($\bigwedge_i \lambda_i \geq 0$). As we saw in §1.2.2, such a problem does not fall within the realm of linear algebra but in that of linear programming for which the simplex algorithm is a standard solver (see §1.3 for a introduction). In practice, the simplex performs much better than its theoretical exponential complexity – but still remains a costly algorithm. So, much research has been devoted to identifying many cases where the simplex can be avoided. Wilde (1993) and Lassez et al. (1993) suggest several *fast redundancy-detection criteria* before switching to the general LP problem:

- **The quasi-syntactic redundancy test** considers pairs of constraints and looks for single constraint redundancies of the form $C' = \lambda C$ with $\lambda > 0$, *e.g.* $C' : 4x_1 - 6x_2 \geq 2$ is redundant w.r.t. $C : x_1 - 3x_2 \geq 1$ since $C' = 2 \times C$.
- **The bound shifting test** exploits the implication $\boldsymbol{ax} \leq b \Longrightarrow \boldsymbol{ax} \leq b'$ if $b \leq b'$. Hence, when the coefficients of two constraints C and C' only differ on b and b' with $b \leq b'$ then C' is redundant and the certificate is $C' = C + (b' - b)$.
- **The combination of single variable inequalities** such as $x_1 \leq b_1$ and $x_2 \leq b_2$ entails for instance the redundancy of $C : 2x_1 + 3x_2 \leq b$ with $2b_1 + 3b_2 \leq b$. The corresponding certificate is $C = 2 \times (x_1 \leq b_1) + 3 \times (x_2 \leq b_2) + (2b_1 + 3b_2 - b)$.

While these criteria can detect certain redundancies at a low cost, the raytracing algorithm that we shall present exploits the other side of redundancy and provides a *fast criterion to detect irredundant constraints*. The combination of the two approaches limits the usage of the simplex to constraints that are neither decided by our criteria nor by those of Wilde (1993) and Lassez et al. (1993).

2.2 Certifying a Minimization of Polyhedra

In this section we recall the standard algorithm for minimizing a *polyhedral cone represented as a set of constraints*. It has been extended in the VPL 0.1 to produce on-the-fly *certificates of correctness, precision and minimality*.

Minimizing a polyhedral cone $\mathcal{P}$ consists in removing all redundant constraints such that the result, $\mathcal{P}_M$, represents the same geometrical space, *i.e.* $[\![\mathcal{P}]\!] = [\![\mathcal{P}_M]\!]$. Two certificates are needed to prove that equality:

(1) one for the inclusion $[\![\mathcal{P}]\!] \subseteq [\![\mathcal{P}_M]\!]$ which guarantees *the correctness of the minimization* and

(2) another one for $[\![\mathcal{P}_M]\!] \subseteq [\![\mathcal{P}]\!]$ which justifies its *precision*.

A third certificate (3) ensures *the minimality of the result* showing that all constraints of $\mathcal{P}_M$ are irredundant.

Certificate (1) must prove that each point of $[\![\mathcal{P}]\!]$ belongs to $[\![\mathcal{P}_M]\!]$, which means that each constraint of $\mathcal{P}_\mathcal{M}$ must be a logical consequence of the constraints of $\mathcal{P}$. In the particular case of minimization, inclusion (1) is trivial because $\mathcal{P}_M$ is obtained by only removing constraints from $\mathcal{P}$, which necessarily leads to a larger set of points. A syntactic test is sufficient to retrieve the constraints of $\mathcal{P}_M$ in $\mathcal{P}$ in order to prove inclusion (1). By contrast, the existence of certificates for (2) and (3) is not straightforward but the consequence of the following corollary, that is a rephrasing of Farkas' lemma for polyhedral cones.

Corollary 2.1 (Farkas' lemma for cones) *Let $C_1, \ldots, C_p$ and C' be vectors in a n-dimensional space. Then,*

(I) either C' **is redundant** *and there exists a* Farkas decomposition *of C' that is a nonnegative linear combination of linearly independent vectors from $C_1, \ldots, C_p$, i.e. $C' = \lambda_1 C_1 + \ldots + \lambda_p C_p$ for some scalars $\lambda_1, \ldots, \lambda_p \geq 0$.*

(II) or C' **is irredundant** *and there exists a n-dimensional vector $\boldsymbol{w}$ such that $C'(\boldsymbol{w}) > 0$ and $C_1(\boldsymbol{w}), \ldots, C_p(\boldsymbol{w}) \leq 0$.*

Proof. It is a straightforward consequence of Theorem 1.5 (p.22).
(I): It is exactly implication ($\Rightarrow$) of Theorem 1.5. As we said earlier, because we work on cones, the constant term λ_0 of the Farkas decomposition $C' = \lambda_0 + \lambda_1 C_1 + \ldots + \lambda_p C_p$ vanishes.
(II): It is proven by the contraposition of implication ($\Leftarrow$) of Theorem 1.5. □

Algorithm 2.3: The standard minimization algorithm

Input : A set of constraints $\{C_1, \ldots, C_p\}$.
Output: $\mathcal{P}_M$ = the irredundant constraints of $\{C_1, \ldots, C_p\}$
(R, I) = the redundancy and irredundancy certificates

$\mathcal{P}_M \leftarrow \{C_1, \ldots, C_p\}$
for C' *in* $\{C_1, \ldots, C_p\}$ **do**
  switch simplex $\left(\exists \lambda_i \geq 0,\ C' = \sum_{C_i \in \mathcal{P}_M \setminus C'} \lambda_i C_i\right)$ **do**
    case SUCCESS ($\boldsymbol{\lambda}$): **do** $R \leftarrow R \cup \{(C', \boldsymbol{\lambda})\}$; $\mathcal{P}_M \leftarrow \mathcal{P}_M \setminus C'$
    case FAILURE ($\boldsymbol{w}$): **do** $I \leftarrow I \cup \{(C', \boldsymbol{w})\}$
return $(\mathcal{P}_M, R, I)$

The standard minimization algorithm (Algorithm 2.3) exploits the redundancy criterion (I) of the corollary which was already illustrated in Example 2.1. The existence of a Farkas decomposition of C' is decided by solving a LP problem. The feasibility phase of the simplex algorithm is designed to answer such questions. The second phase of the simplex, called optimization phase, is not triggered in this case. If the simplex algorithm returns a solution $\boldsymbol{\lambda}$ then the pair $(C', \boldsymbol{\lambda})$ is recorded as a *certificate of precision* (2) which proves that the removed constraint was indeed redundant. To get rid of all the redundancies, Algorithm 2.3 executes the simplex algorithm for each constraint.

Given an existential LP problem, the simplex can return either a solution or an explanation of the lack of solution. The proof of Farkas' lemma and the simplex algorithm have strong connections which result in an interesting feature of the VPL simplex: calling simplex($\exists \lambda_i \geq 0,\ C' = \sum_i \lambda_i C_i$) returns either SUCCESS($\boldsymbol{\lambda}$) or FAILURE($\boldsymbol{w}$) such that $C'(\boldsymbol{w}) > 0 \ \bigwedge_i C_i(\boldsymbol{w}) \leq 0$. Conversely, simplex($\exists \boldsymbol{w}, C'(\boldsymbol{w}) > 0 \ \bigwedge_i C_i(\boldsymbol{w}) \leq 0$) returns either SUCCESS($\boldsymbol{w}$) or FAILURE($\boldsymbol{\lambda}$) such that $C' = \sum_i \lambda_i C_i$. This feature is a consequence of Corollary 2.1 and requires no additional computation. For more details about the computation of $\boldsymbol{\lambda}$ and $\boldsymbol{w}$, the interested reader can refer to (Fouilhé, 2015).

When the simplex returns FAILURE($\boldsymbol{w}$), the irredundancy criterion (II) of the corollary tells that C' is irredundant and must be kept in the set of constraints. Algorithm 2.3 builds the *certificate of minimality* (3) by associating a *witness point* $\boldsymbol{w}$ to each constraint of the minimized polyhedron $\mathcal{P}_M$.

While the standard algorithm focuses on criterion (I), we revisit the corollary paying attention to the geometrical interpretation of criterion (II): when a constraint C' is irredundant, its associated bounding hyperplane is *a facet*[1] of the polyhedron separating the inside from the outside. Part (II) of the corollary ensures that we can exhibit a witness point $\boldsymbol{w}$, outside of $[\![\mathcal{P}]\!]$, satisfying all constraints of $\mathcal{P}$ except C'. The raytracing algorithm that we present in the next section efficiently discovers such witness points.

2.3 An Efficient Minimization Algorithm

Building up on the geometric interpretation of Corollary 2.1, we present here a new minimization algorithm for polyhedral cones that brings two major improvements: it reduces the number of calls to the simplex algorithm and limits the number of constraints they involve. The key idea of the algorithm is to trace rays starting from a point in the interior of the cone. The first hyperplane encountered by a ray is a facet of the polyhedron, *i.e.* an irredundant constraint. Unfortunately, with a limited number of rays, some facets can be missed depending on the cone and the position of the interior point. This *raytracing procedure* is thus incomplete and LP problem resolutions are still required for the remaining undetermined constraints.

While the simplex algorithm is used in the standard minimization to discover Farkas decompositions, we rather use it to get closer to a witness point, and only when all previous rays failed to prove the irredundancy of a constraint. Of course, if the constraint is redundant, the simplex algorithm returns no witness point $\boldsymbol{w}$ at all but an explanation of its failure which is nothing else than a Farkas decomposition proving the redundancy.

2.3.1 The Facet Detection Criterion

We now detail the process of finding witness points by raytracing (Algorithm 2.6). We consider a cone $\mathcal{P}$ with a nonempty interior.[2] Then, there exists a point $\mathring{\boldsymbol{x}}$ in $[\![\mathring{\mathcal{P}}]\!]$. The basic operation of our algorithm consists in sorting the constraints of $\mathcal{P}$ with respect to the order in which they are hit by a *ray*, *i.e.* a half-line starting at the interior point $\mathring{\boldsymbol{x}}$ and extending along a given direction $\boldsymbol{d}$.

Consider the constraint $C(\boldsymbol{x}) \leq 0$. The hyperplane of the constraint is $\{\boldsymbol{x} \mid C(\boldsymbol{x}) = 0\}$, *i.e.* the set of points orthogonal to vector C, since $C(\boldsymbol{x})$ is $\langle C, \boldsymbol{x}\rangle$ for cones. The ray starting at $\mathring{\boldsymbol{x}}$ and extending in direction $\boldsymbol{d}$ is the set of points $\{\boldsymbol{x}(t) \mid \boldsymbol{x}(t) = \mathring{\boldsymbol{x}} + t \times \boldsymbol{d},\ t \geq 0\}$. Let us assume that the ray hits the C-hyperplane at point $\boldsymbol{x_c}$. Then, there exists $t_c \geq 0$ such that $\boldsymbol{x_c} = \mathring{\boldsymbol{x}} + t_c \times \boldsymbol{d}$ and so, $\boldsymbol{x_c} - \mathring{\boldsymbol{x}} = t_c \times \boldsymbol{d}$. Therefore, the distance $\|\mathring{\boldsymbol{x}} - \boldsymbol{x_c}\|$ is just a scaling by $|t_c|$ of the norm $\|\boldsymbol{d}\|$. Hence, *by computing $|t_c|$ for each constraint we will be able to know in which order the constraints are hit by the ray*. Prior to computing t_c we check if the ray can hit the constraint, meaning that C and $\boldsymbol{d}$ are not orthogonal, *i.e.* $C(\boldsymbol{d}) \neq 0$. Then, we use the fact that $\boldsymbol{x_c} \in \{\boldsymbol{x} \mid C(\boldsymbol{x}) = 0\}$ to get $t_c = -\frac{C(\mathring{\boldsymbol{x}})}{C(\boldsymbol{d})}$. Indeed,

$$0 = C(\boldsymbol{x_c}) = C(\mathring{\boldsymbol{x}} + t_c \times \boldsymbol{d}) = C(\mathring{\boldsymbol{x}}) + t_c \times C(\boldsymbol{d}).$$

Hence, the basic operation of our raytracing algorithm consists in two evaluations of each constraint C of $\mathcal{P}$ at $\mathring{\boldsymbol{x}}$ and $\boldsymbol{d}$ in order to compute the scalar t_c. Let us explain how we exploit this information to discover actual facets of $\mathcal{P}$.

Note that any direction could be used to sort the constraints w.r.t. the order of intersection by a ray. We choose successively as direction $\boldsymbol{d}$ of the ray the opposite direction of the normal vector of each bounding hyperplane of $\mathcal{P}$. This heuristic (from lines 7 to 9 in Algorithm 2.6) ensures that each hyperplane will be hit by at least one ray. As illustrated by Fig. 2.4, a direction $\boldsymbol{d} \stackrel{\text{def}}{=} -C$ necessarily intersects the C-hyperplane and may potentially cross many other

1. Actually, a facet here abusively designates both a constraint and its bounding hyperplane. What is important here is that a facet is *irredundant*.

2. Equalities are extracted beforehand. Finding redundancies in a set of equalities is the same as finding dependent vectors in a family, which is standard in linear algebra.

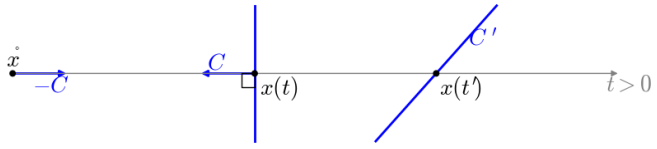
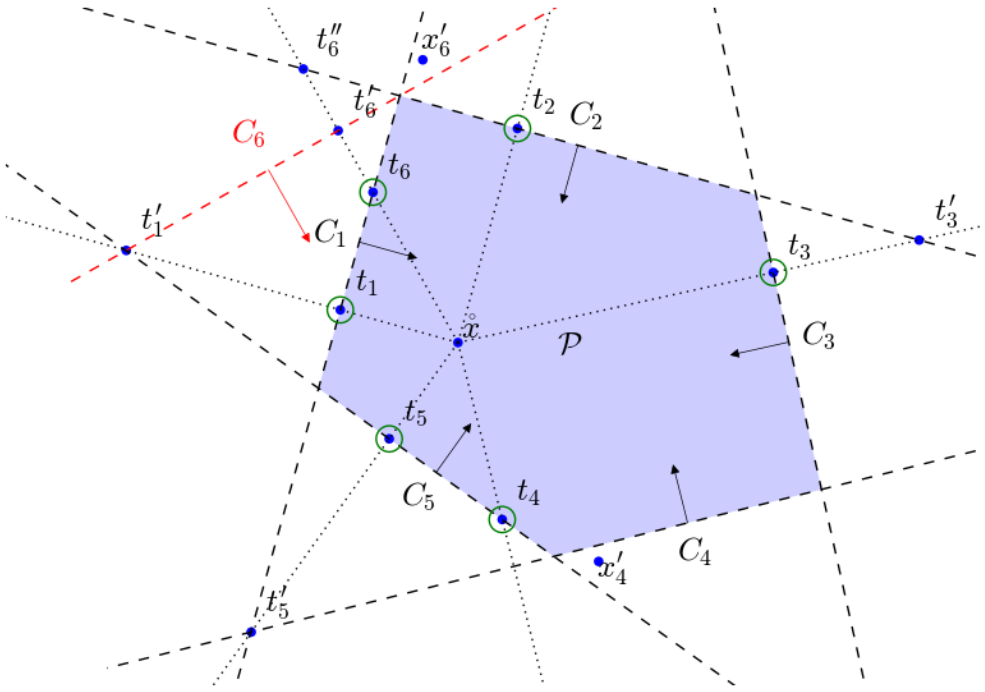


Figure 2.4 – The ray starting at the interior point $\mathring{\boldsymbol{x}}$ and orthogonal to a constraint C meets C and possibly others constraints.

constraints for some values of t. Considering a direction $\boldsymbol{d_i} = -C_i$, we sort the intersected hyperplanes w.r.t. the increasing order of the scalar t, which is proportional to the distance between the interior point $\mathring{\boldsymbol{x}}$ and the intersection point $\boldsymbol{x}(t)$ of an hyperplane and the ray $\textsc{ray}(\mathring{\boldsymbol{x}}, \boldsymbol{d}_i)$. We obtain a sorted *intersection list* of pairs (t, S_t) where S_t is the set of the (possibly many) constraints vanishing at $\boldsymbol{x}(t)$. If a constraint C is not hit by the ray (because $C(\boldsymbol{d}_i) \leq 0$), then C is not added to the intersection list. The head pair provides the constraints which are encountered first by the ray. At the heart of our algorithm is the following proposition: "If the head of an intersection list is a pair $(t, \{C\})$ with *a single constraint*, then C is a facet of $\mathcal{P}$; otherwise we cannot conclude from this list." This will be proved in §2.3.2 (Proposition 2.3) when we will come to the generation of witness points.

Example 2.5



This figure shows the detection of some facets of a polyhedron by looking at their intersections with rays starting from an interior point $\mathring{\boldsymbol{x}}$ and orthogonal to a constraint. The doubly-circled intersection points show the first constraint hit by a ray.

Here are the sorted intersection lists obtained for this polyhedron. The list I_i records the constraints met along $\textsc{ray}(\mathring{\boldsymbol{x}}, -C_i)$ from $\mathring{\boldsymbol{x}}$ orthogonally to the hyperplane of C_i. It satisfies $t_i < t'_i < t''_i < t'''_i$.

$I_1 = [\ (t_1, \{C_1\});\ (t'_1, \{C_5, C_6\});\ (t''_1, \{C_2\})\ ]$
$I_2 = [\ (t_2, \{C_2\});\ (t'_2, \{C_6\});\ (t''_2, \{C_3\});\ (t'''_2, \{C_1\})\ ]$
$I_3 = [\ (t_3, \{C_3\});\ (t'_3, \{C_2\});\ (t''_3, \{C_4\})\ ]$
$I_4 = [\ (t_4, \{C_5\});\ (t'_4, \{C_4\});\ (t''_4, \{C_3\})\ ]$
$I_5 = [\ (t_5, \{C_5\});\ (t'_5, \{C_1, C_4\})\ ]$
$I_6 = [\ (t_6, \{C_1\});\ (t'_6, \{C_6\});\ (t''_6, \{C_2\})\ ]$

These lists reveal that $C_1,\ C_2,\ C_3$ and C_5 are facets of $\mathcal{P}$; C_1 and C_5 are even confirmed twice. Our criterion fails to decide the status of C_4 and C_6 because, in any of the considered directions, they are never encountered first. This situation is legitimate for the redundant constraint C_6 but also happens for C_4 even though it is a facet of $\mathcal{P}$.

At this point (line 10 of Algorithm 2.6), we run the simplex algorithm to determine the irredundancy of the remaining constraints. In order to keep LP problems as small as possible, we build them incrementally as follows. Consider an undetermined constraint C_i and let I_i be the intersection list resulting from the direction $\boldsymbol{d}_i = -C_i$. We illustrate the algorithm on the case of a single head constraint as it is the most frequent one. We pose a LP problem to find a point $\boldsymbol{x}'_i$ satisfying $C_i(\boldsymbol{x}'_i) > 0 \wedge C'(\boldsymbol{x}'_i) \leq 0$, where C' is the single constraint that appears at the head of I_i. As said earlier, C' is a facet because it is the first hyperplane encountered by the ray. If the head set contains several constraints we cannot know which one is a facet, thus we add all of them in the LP problem (lines 13-14 of Algorithm 2.6). We distinguish two cases depending on the satisfiability of the existential LP problem: If the problem of line 15 is unsatisfiable, the simplex returns FAILURE(λ), C_i is redundant w.r.t. C' and the Farkas decomposition of C_i is $\lambda \times C'$. Otherwise, the simplex exhibits a point $\boldsymbol{x}'_i$ which satisfies $C_i(\boldsymbol{x}'_i) > 0 \wedge C'(\boldsymbol{x}'_i) \leq 0$. Here, we cannot conclude on C_i's redundancy since $\boldsymbol{x}'_i$ is a witness showing that C_i is irredundant w.r.t. C' alone, but C_i could still be redundant w.r.t. the other constraints.

Algorithm 2.6: Raytracing algorithm

Input : A set of constraints $\mathcal{P} = \{C_1, \ldots, C_p\}$; a point $\mathring{\boldsymbol{x}} \in \mathring{\mathcal{P}}$
Output : $\mathcal{P}_M$: minimized version of $\mathcal{P}$
Data : $LP[i]$: Linear Programming problem associated to C_i ; $I[i]$: intersection list of C_i
Function: `intersectionList`($\boldsymbol{d}$, $\{C_1, \ldots, C_q\}$) returns the intersection list obtained by intersecting $\{C_1, \ldots, C_q\}$ with ray $\boldsymbol{d}$

```
1  Function updateFacets (I[i], P_M, P)
2      if head (I[i]) = (t_F, {F}) then
3          P_M ← P_M ∪ {F}
4          P ← P \ F
5      return (P_M, P)
6  P_M ← ∅ ; LP ← arrayOfSize(p) ; I ← arrayOfSize(p)
7  for C_i in P do                 /* First step of raytracing with orthogonal rays */
8      I[i] ← intersectionList (RAY(x̊, −C_i), P)
9      (P_M, P) ← updateFacets (I[i], P_M, P)
10 while P ≠ ∅ do
11     for C_i in P do
12         (t, S) ← head(I[i])
13         for C in S do
14             LP[i] ← LP[i] ∧ C(x'_i) ≤ 0
15         switch simplex (∃x'_i, C_i(x'_i) > 0 ∧ LP[i]) do
16             case SUCCESS (x'_i): do
17                 I[i] ← intersectionList (RAY(x̊, x'_i − x̊), P ∪ P_M)
18                 (P_M, P) ← updateFacets (I[i], P_M, P)
19             case FAILURE (λ): do  P ← P \ C_i        /* C_i is redundant */
20
21 return P_M
```

To check the irredundancy of C_i, we launch a new ray RAY($\mathring{\boldsymbol{x}}, \boldsymbol{x}'_i - \mathring{\boldsymbol{x}}$) from $\mathring{\boldsymbol{x}}$ to $\boldsymbol{x}'_i$. As

before, we compute the intersection list of this ray with all the constraints but this time we know for sure that C_i will precede C' in the list. This property is a pure technicality given in Proposition 2.2 below. Then, we analyze the head of the list: if C_i is the *single* first element, then it is a facet. Otherwise the first element, say C'', is added to the LP problem, which is now asked for a point $\boldsymbol{x_i''}$ such that $C_i(\boldsymbol{x_i''}) > 0 \wedge C'(\boldsymbol{x_i''}) \leq 0 \wedge C''(\boldsymbol{x_i''}) \leq 0$ resulting in a new $\textsc{ray}(\mathring{\boldsymbol{x}}, \boldsymbol{x_i''} - \mathring{\boldsymbol{x}})$. The way we choose rays guarantees that the previous constraints $C', C'', \ldots$ will always be hit after C_i by the next ray. Therefore, ultimately the constraint C_i will be hit first by a ray, or it will be proved redundant. Termination is guaranteed because the first constraint struck by the new ray is either C_i and we are done, or a not already considered constraint and there is a finite number of constraints in $\mathcal{P}$. Observe that this algorithm builds incremental LP problems which contain only facets that were between $\mathring{\boldsymbol{x}}$ and the hyperplane of C_i at some step.

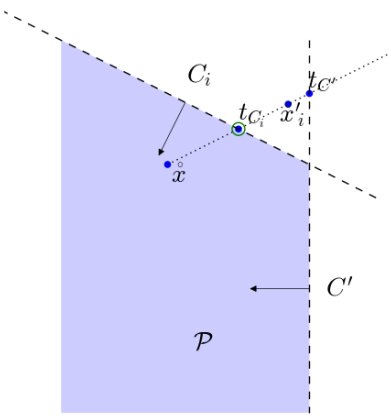
Example 2.7 (follows 2.5)

In the previous example, we found out that C_1, C_2, C_3 and C_5 were facets. To determine the status of C_4, we solve the LP problem $\exists \boldsymbol{x'_4}$, $C_4(\boldsymbol{x'_4}) > 0 \wedge C_5(\boldsymbol{x'_4}) \leq 0$ because C_5 is the head of I_4. The simplex finds such a point $\boldsymbol{x'_4}$ and the next step is to compute the intersection list corresponding to $\textsc{ray}(\mathring{\boldsymbol{x}}, \boldsymbol{x'_4} - \mathring{\boldsymbol{x}})$. This list will reveal C_4 as an actual facet.

Similarly, the intersection list I_6 of the example suggests to solve the LP problem $\exists \boldsymbol{x'_6}$, $C_6(\boldsymbol{x'_6}) > 0 \wedge C_1(\boldsymbol{x'_6}) \leq 0$ to launch a new ray toward C_6. This problem is satisfiable and the simplex returns $\textsc{success}(\boldsymbol{x'_6})$. Then, we compute the intersection list corresponding to $\textsc{ray}(\mathring{\boldsymbol{x}}, \boldsymbol{x'_6} - \mathring{\boldsymbol{x}})$ and this time the head of the list is C_2. We thus add C_2 to the previous LP problem and call the simplex on $\exists \boldsymbol{x''_6}$, $C_6(\boldsymbol{x''_6}) > 0 \wedge C_1(\boldsymbol{x''_6}) \leq 0 \wedge C_2(\boldsymbol{x''_6}) \leq 0$. This problem has no solution: the simplex returns $\textsc{failure}(\boldsymbol{\lambda} = (1, 1))$ showing that C_6 is redundant and its Farkas decomposition is $C_6 = 1 \times C_1 + 1 \times C_2$.

Proposition 2.2 *Let C_i, C' be two constraints of a polyhedron $\mathcal{P}$, and let $\mathring{\boldsymbol{x}} \in \mathring{\mathcal{P}}$. Let $\boldsymbol{x'_i}$ be a point such that $C_i(\boldsymbol{x'_i}) > 0$ and $C'(\boldsymbol{x'_i}) \leq 0$. Then $\textsc{ray}(\mathring{\boldsymbol{x}}, \boldsymbol{x'_i} - \mathring{\boldsymbol{x}})$ intersects C_i at some point $\boldsymbol{x}(t_{C_i})$. Moreover, assume it crosses C' at $\boldsymbol{x}(t_{C'})$, then $t_{C_i} < t_{C'}$.*

Proof. Because $C_i(\mathring{\boldsymbol{x}}) < 0$ and $C_i(\boldsymbol{x'_i}) > 0$ then $\textsc{ray}(\mathring{\boldsymbol{x}}, \boldsymbol{x'_i} - \mathring{\boldsymbol{x}})$ necessarily crosses C_i, say at $\boldsymbol{x}(t_{C_i})$ with $0 < t_{C_i} < 1$, since $\boldsymbol{x}(t = 0)$ is $\mathring{\boldsymbol{x}}$ and $\boldsymbol{x}(t = 1)$ is $\boldsymbol{x'_i}$. As both ends of the line segment $[\mathring{\boldsymbol{x}}, \boldsymbol{x'_i}]$ satisfy C', then $1 < t_{C'}$. Thus, $t_{C_i} < 1 < t_{C'}$. □



Strict Inequalities. So far, we presented the raytracing algorithm with an input cone made of nonstrict inequalities only. But, there is no obstacle to the use of strict inequalities. Indeed, they already appear during the call to the simplex (line 15 of Algorithm 2.6), when we specify to violate one constraint. Therefore, the LP-solver that we use must be able to handle strict inequalities. As we saw at the end of §1.3, the rational LP solver implemented in the VPL has this ability: it gives symbolic values of the form $(q, d \cdot \epsilon)$ with $q, d \in \mathbb{Q}$ to coordinates of vertices. (q, d) is a solution to $x > 0$ if $q > 0$ or $(q = 0 \wedge d > 0)$. For floating-point solving, we can rewrite strict inequalities of the form $C(\boldsymbol{x}) > 0$ into $C(\boldsymbol{x}) \geq \epsilon$, where the value for ϵ must be provided, close to 0, but not too much to avoid rounding errors.

2.3.2 Irredundancy Certificates

Let us explain how we compute witness points from the intersection lists defined in the previous section. In the following, to ease understanding, we will denote a constraint by $\boldsymbol{F}$ if we know that it is a facet, and by C when we do not know if it is redundant. Let us come back to the list of the intersections of constraints of $\mathcal{P}$ with a ray $\{\boldsymbol{x}(t) \mid \boldsymbol{x}(t) = \mathring{\boldsymbol{x}} + t \times \boldsymbol{d},\ t \geq 0\}$ for a direction $\boldsymbol{d}$.

Proposition 2.3 *If the head of an intersection list contains a single constraint $\boldsymbol{F}$, then we can build a* witness point *satisfying the irredundancy criterion of Corollary 2.1 which proves that $\boldsymbol{F}$ is a facet:*

(a) For a list $[(t_{\boldsymbol{F}}, \{\boldsymbol{F}\})]$, we take the witness $\boldsymbol{w}_a = \mathring{\boldsymbol{x}} + (t_{\boldsymbol{F}} + 1) \times \boldsymbol{d}$

(b) For a list $[(t_{\boldsymbol{F}}, \{\boldsymbol{F}\})\ ;\ (t', S')\ ;\ \ldots]$ with at least two pairs, we define the witness $\boldsymbol{w}_b = \mathring{\boldsymbol{x}} + \frac{t_{\boldsymbol{F}}+t'}{2} \times \boldsymbol{d}$.

Proof. Let us prove that these witness points attest that $\boldsymbol{F}$ is an irredundant constraint. According to Corollary 2.1, it amounts to proving that, for $\boldsymbol{w}_a$ (resp. $\boldsymbol{w}_b$),

$$\bigwedge_{C \in \mathcal{P} \setminus \boldsymbol{F}} C(\boldsymbol{w}) \leq 0 \wedge \boldsymbol{F}(\boldsymbol{w}) > 0$$

Let us first study the sign of $\boldsymbol{F}(\boldsymbol{x}(t))$ at point $\boldsymbol{x}(t) = \mathring{\boldsymbol{x}} + t \times \boldsymbol{d}$. Note that

$$\boldsymbol{F}(\boldsymbol{x}(t)) = \boldsymbol{F}(\mathring{\boldsymbol{x}} + t \times \boldsymbol{d}) = \boldsymbol{F}(\mathring{\boldsymbol{x}}) + t \times \boldsymbol{F}(\boldsymbol{d}). \qquad (\text{Ŧ})$$

By construction, $\boldsymbol{F}(\boldsymbol{x}(t_{\boldsymbol{F}})) = 0$ then, by equation (Ŧ), $-\boldsymbol{F}(\mathring{\boldsymbol{x}}) = t_{\boldsymbol{F}} \times \boldsymbol{F}(\boldsymbol{d})$. Recall that $t_{\boldsymbol{F}} \geq 0$, $\boldsymbol{F}(\boldsymbol{d}) \neq 0$ since the ray hits $\boldsymbol{F}$ and $\boldsymbol{F}(\mathring{\boldsymbol{x}}) < 0$ because $\mathring{\boldsymbol{x}} \in \mathring{\mathcal{P}}$. Thus, $\boldsymbol{F}(\boldsymbol{d})$ and $t_{\boldsymbol{F}}$ are necessarily positive. Consequently, $\boldsymbol{F}(\boldsymbol{x}(t)) = \boldsymbol{F}(\mathring{\boldsymbol{x}}) + t \times \boldsymbol{F}(\boldsymbol{d})$ is positive for any $t > t_{\boldsymbol{F}}$. Hence, in case (a) $\boldsymbol{F}(\boldsymbol{w}_a) \stackrel{\text{def}}{=} \boldsymbol{F}(\boldsymbol{x}(t_{\boldsymbol{F}} + 1)) > 0$ and in case (b) $\boldsymbol{F}(\boldsymbol{w}_b) \stackrel{\text{def}}{=} \boldsymbol{F}(\boldsymbol{x}(\frac{t_{\boldsymbol{F}}+t'}{2})) > 0$ since $t_{\boldsymbol{F}} < \frac{t_{\boldsymbol{F}}+t'}{2} < t'$.

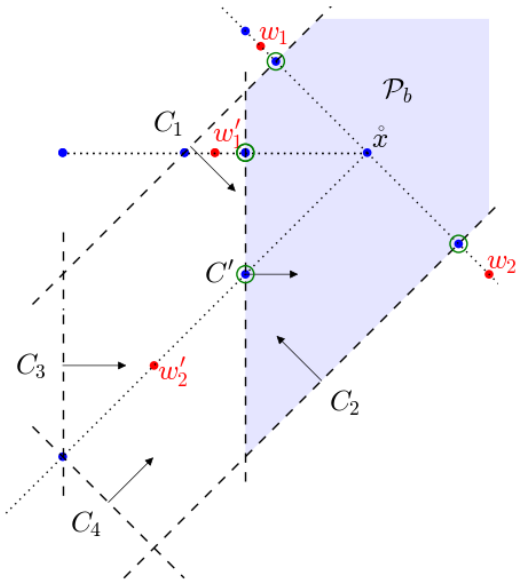
Let us now study the sign of $C(\boldsymbol{x}(t))$ for constraints other than $\boldsymbol{F}$:

(a) Consider the list $[(t_{\boldsymbol{F}}, \{\boldsymbol{F}\})]$. By construction, it means that no other constraint C of $\mathcal{P}$ is struck by the $\textsc{ray}(\mathring{\boldsymbol{x}}, \boldsymbol{d})$, *i.e.* whatever the value $t \geq 0$, the sign of $C(\boldsymbol{x}(t)) = C(\mathring{\boldsymbol{x}}) + t \times C(\boldsymbol{d})$ does not change. As $C(\boldsymbol{x}(t=0)) = C(\mathring{\boldsymbol{x}}) < 0$ because $\mathring{\boldsymbol{x}} \in \mathring{\mathcal{P}}$, we can conclude that $\forall t \geq 0,\ C(\boldsymbol{x}(t)) < 0$. Thus, in particular, $C(\boldsymbol{w}_a) \stackrel{\text{def}}{=} C(\boldsymbol{x}(t_{\boldsymbol{F}} + 1)) < 0$ for any $C \in \mathcal{P} \setminus \boldsymbol{F}$.

(b) Consider now the list $[(t_{\boldsymbol{F}}, \{\boldsymbol{F}\});\ (t', S'); \ldots]$. A constraint C that appears in the set S' vanishes at point $\boldsymbol{x}(t')$ with $t' > t_{\boldsymbol{F}} \geq 0$. The previous reasoning (Ŧ) (on $\boldsymbol{F}$) based on equation $C(\boldsymbol{x}(t)) = C(\mathring{\boldsymbol{x}}) + t \times C(\boldsymbol{d})$ is valid for C, hence proving $C(\boldsymbol{d}) > 0$. Thus, $C(\boldsymbol{x}(t))$ is negative for $t < t'$ (zero for $t = t'$ and positive for $t' < t$). Finally, $C(\boldsymbol{w}_b) \stackrel{\text{def}}{=} C(\boldsymbol{x}(\frac{t_{\boldsymbol{F}}+t'}{2})) < 0$ since $\frac{t_{\boldsymbol{F}}+t'}{2} < t'$. The same reasoning applies to any other pair (t, S_t) in the tail of the list.

□

Example 2.8



Consider the polyhedron $\mathcal{P}_b$ from Example 2.1, made of constraints C_1, C_2, C_3, C_4 and C', where C_3 and C_4 are redundant. The figure shows the witness points w_1, w_2, w'_1, w'_2 of constraints C_1, C_2 and C'. The irredundancy of C' is confirmed three times by different rays respectively orthogonal to C', C_3 and C_4, leading to witnesses w'_1 (twice) and w'_2. Our algorithm could take advantage of this situation: if we pick up the ray orthogonal to C_3 (or C_4) before that of C', constraint C' will be accidentally confirmed and thus the ray orthogonal to C' does not need to be considered.

2.3.3 Using Floating Point in Raytracing

It is possible to make raytracing even more efficient by using floating point numbers instead of rationals, both in LP problem resolutions and distance computations. The rational coefficients of constraints are translated into floating point numbers. It introduces a loss in precision which does not jeopardize the result because the certificate checking controls the minimization process. Therefore, we must generate *exact* (*i.e.* rational) certificates from floating point computations. The solution we propose differs depending on the kind of certificate.

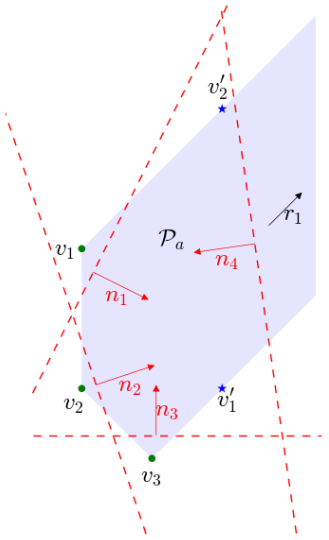
Witness Points. Checking a certificate of irredundancy consists in evaluating the sign of $C_i(\boldsymbol{w})$ for all constraints C_i of $\mathcal{P}$ with the provided witness point $\boldsymbol{w}$. A witness point $\boldsymbol{w}$ must then be given with rational coefficients to avoid sign errors if $C_i(\boldsymbol{w})$ is too close to 0. Thus, the witness point $\boldsymbol{w}^{\mathbb{F}}$ obtained with floating point computations is translated into a rational one $\boldsymbol{w}^{\mathbb{Q}}$, without loss of precision (each floating point $0.d_1 \ldots d_m 10^e$ is changed into a rational $\frac{d_1 \ldots d_m}{10^m} 10^e$). Then we check the irredundancy certificate with $\boldsymbol{w}^{\mathbb{Q}}$ and the rational version of the constraints. If the verification passes, then $\boldsymbol{w}^{\mathbb{Q}}$ is indeed a witness point. In the rare case of failure, *i.e.* when a constraint is wrongly labeled irredundant, using the exact simplex of the VPL on the LP problem will fix the approximation error by directly providing a rational witness point or a Farkas witness of redundancy.

Farkas Decompositions. To prove a redundancy we need to exhibit the Farkas decomposition of the redundant constraint. To obtain an exact decomposition from the floating LP solution, we record which constraint is actually part of the decomposition. What is needed from the floating point solution is the set of basic variables and an ordering of the nonnull λ_i coefficients to speed up the search in exact simplex. Then, we run the exact simplex on a LP problem involving only those constraints to retrieve the exact Farkas decomposition.

2.3.4 Minimizing Generators

So far, to ease the understanding, we presented the raytracing for the constraints-only representation of polyhedra, but it works as well for generators. Indeed, we manipulated constraints as vectors and all our explanations and proofs are based on inner product. Moreover, Corollary 2.1 is not limited to constraints, it holds for any vector space and can be rephrased for generators. This time the irredundancy certificate for a generator $\boldsymbol{g}'$ is a vector $\boldsymbol{n}$ such that $\langle \boldsymbol{g}_1, \boldsymbol{n} \rangle, \ldots, \langle \boldsymbol{g}_p, \boldsymbol{n} \rangle \leq 0$ and $\langle \boldsymbol{g}', \boldsymbol{n} \rangle > 0$. Such a vector defines a hyperplane orthogonal to $\boldsymbol{n}$, *i.e.* $\{\boldsymbol{x} \mid \langle \boldsymbol{n}, \boldsymbol{x} \rangle = 0\}$. It is called a *separating hyperplane* because it isolates generator $\boldsymbol{g}'$ from the other ones.

Example 2.9



The figure shows the separating hyperplanes defined by $\boldsymbol{n_1}, \boldsymbol{n_2}, \boldsymbol{n_3}$ and $\boldsymbol{n_4}$. They respectively justify the irredundancy of $\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}$ and $\boldsymbol{r}_1$ in $\mathcal{P}_a$. The hyperplane defined by $\boldsymbol{n_4}$ vouches for the irredundancy of $\boldsymbol{r}_1$ because it satisfies $\langle \boldsymbol{n_4}, \boldsymbol{g} \rangle \geq 0$ for all $\boldsymbol{g} \in \{\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{v_1'}, \boldsymbol{v_2'}\}$ and $\langle \boldsymbol{n_4}, \boldsymbol{r}_1 \rangle < 0$. No hyperplane $\boldsymbol{n}$ can isolate $\boldsymbol{v_1'}$ or $\boldsymbol{v_2'}$ from other vertices while ensuring $\langle \boldsymbol{n}, \boldsymbol{r}_1 \rangle \geq 0$ because vectors $\boldsymbol{v_2'} - \boldsymbol{v_1}$ and $\boldsymbol{v_1'} - \boldsymbol{v_3}$ are colinear with $\boldsymbol{r}_1$.

2.4 Experimental Results

This section is devoted to the comparison of three minimization algorithms:

— **The Standard Minimization Algorithm (sma).** The standard Algorithm 2.3 of §2.2 is available in the VPL since version 0.1. It works on rationals and can generate certificates of precision, minimality and correctness. The VPL implementation carries an optimization: the LP problem of Algorithm 2.3 is built only once before testing the redundancy of each constraint. Initially it is composed of all the constraints $\{C_1, \ldots, C_n\}$. Then, for each $C_i \in \{C_1, \ldots, C_n\}$, it checks C_i's redundancy w.r.t. other constraints present in the LP problem. If C_i is shown redundant, it is removed from the LP problem and the algorithm continues with C_{i+1}. It means that the redundancy of the following constraints $C_{i+1}, \ldots, C_n$ will be tested taking into account only irredundant constraints of $\{C_1, \ldots, C_i\}$.

— **The Rational Raytracing Algorithm (rra).** RRA and SMA use the same LP solver, thus comparing their running time is relevant to estimate the efficiency of raytracing w.r.t. the standard algorithm.

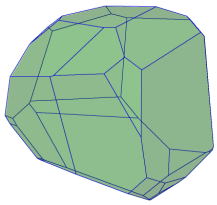
— **The Floating point Raytracing Algorithm (fra).** FRA implements raytracing with floating point as explained in §2.3.3. LP problems are solved by the GNU Linear Programming Kit (Makhorin, 2000–2017) which provides an efficient simplex algorithm in floating point.

These three algorithms are all implemented in the current version (0.2) of the VPL. For computing the exact Farkas decomposition that proves a constraint's irredundancy, the three

algorithms ultimately rely on the VPL simplex algorithm in rational. They use the same datastructures (*e.g.* for constraints), allowing reliable timing comparisons between them. Moreover, they share the same pre-processing step of finding a point within the polyhedron interior. This point is obtained after extraction of implicit equalities from the set of constraints (see Chapter 8). The time measurements given below include this step but not the reconstruction of exact certificates from floating-point witnesses.

The representativeness of the polyhedra encountered in practice is a recurrent issue in experiments, since the generated polyhedra depend on the verification tool and the program under analysis. Their dimension can range from four variables, *e.g.* in a tuned analysis of a C program, to a thousand in analyzes of Lustre programs. Moreover, verification tools pay attention to limit their use of polyhedra to a small number of variables and constraints, as polyhedra are known to be costly. They often switch to interval domains if these limits are exceeded, hence polyhedra encountered in actual analyzes are often small. This is an obstacle for evaluating our minimization as it is designed to be more efficient with higher numbers of variables and constraints. Therefore, we created our own benchmarks made of polyhedra randomly generated from different characteristics that are detailed in the next paragraph.

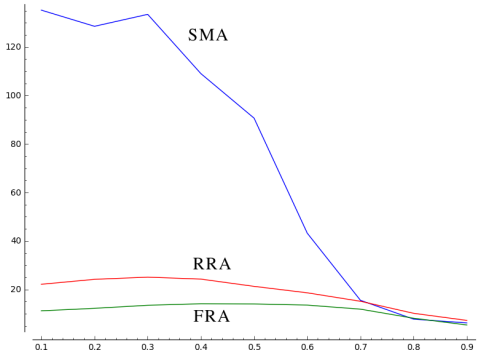
Benchmarks. Throughout the paper, we focused on cones to simplify both notations and explanations. However, our algorithm works for general convex polyhedra and we build our experiments as follows. To compare the three algorithms, we asked them to minimize polyhedra that were generated randomly from four parameters that will be detailed further: the number of variables ($V \in [2, 50]$), the number of constraints ($C \in [2, 50]$), the redundancy rate ($R \in [0\%, 90\%]$) and the density rate ($D \in [10\%, 80\%]$). Each constraint is created by giving a random integer between -100 and 100 to the coefficient of each variable, within the density rate. All constraints are attached the same constant bound ≤ 20. Such polyhedra have a convex potatoid shape, shown on the right hand side. We do not directly control the number of generators but we count them using the APRON interface [3] to polyhedral libraries in double description. Among all our measurements, the number of generators ranged from 10 to 6400 and this number grows polynomially in the number of constraints. These benchmarks cover a wide variety of polyhedra and our experiments show that raytracing is always more efficient.



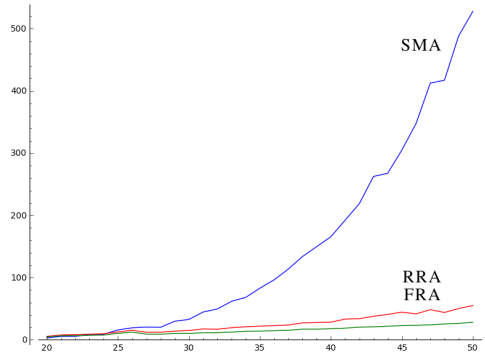
Redundancy Rate. The effect of redundancy on execution time is displayed on Fig. 2.10(a). These measures come from the minimization of polyhedra with 10 variables and 35 constraints, and a redundancy rate ranging from 0% to 90% of the number of constraints. To generate a redundant constraint, we randomly pick two constraints and produce a nonnegative combination of them. We took care to avoid redundancies that can be discarded by the fast detection criteria of §2.1. The graph clearly shows that raytracing has a big advantage on polyhedra with few redundancies. This phenomenon was expected: raytracing is good at detecting irredundancy at a low cost. sma becomes similar to raytracing when the redundancy rate is high. This is explained by the implementation details given in previous paragraphs: when a redundant constraint is found, it is removed from the LP problem. Thus, if the redundancy rate reaches a very high level, the LP problem becomes smaller and smaller at each iteration, lowering the impact of using floating point. Moreover, the heuristic used by our algorithm never hits if almost all constraints are redundant, which makes the raytracing computations useless. To be fair between raytracing and the standard algorithm, we set the redundancy rate at 50% in other experiments.

Number of Constraints. Fig. 2.10(b) measures the minimization time depending on the number of constraints for polyhedra with 10 variables. fra and rra scale better w.r.t. the number of constraints than sma: experiments show that when C ranges from 20 to 50 constraints, sma has a quadratic evolution compared to raytracing algorithms.

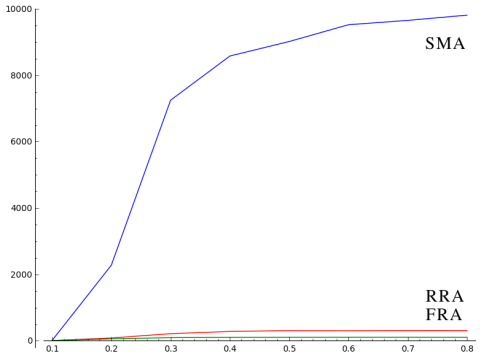
3. http://apron.cri.ensmp.fr/library/0.9.10/mlapronidl/index.html



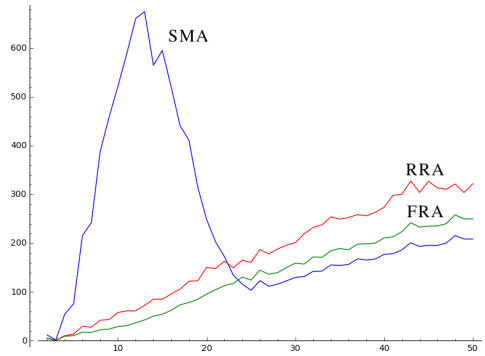
$(a) : C = 35, V = 10, D = 50\%, R = [0\%, 90\%]$



$(b) : C = [20, 50], V = 10, D = 50\%, R = 50\%$

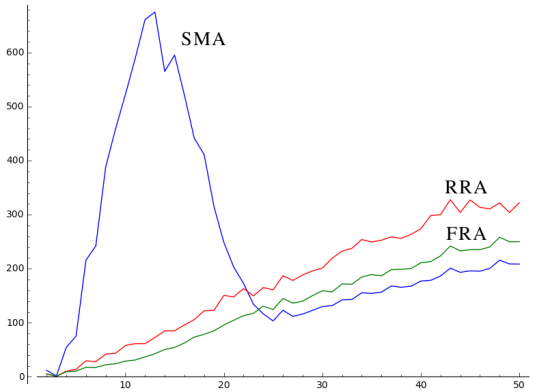


$(c) : C = 100, V = 10, D = [10\%, 80\%], R = 50\%$

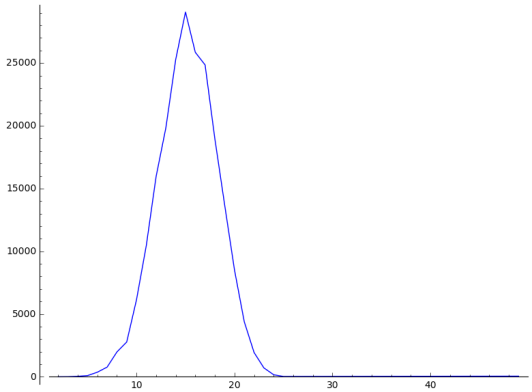


$(d) : C = 50, V = [2, 50], D = 50\%, R = 50\%$

Figure 2.10 – Execution time in milliseconds of SMA (blue), RRA (red) and FRA (green) depending on respectively (a) redundancy, (b) number of constraints, (c) density and (d) number of variables. On each curve, a point is the average time for minimizing 50 polyhedra that share the same characteristics.

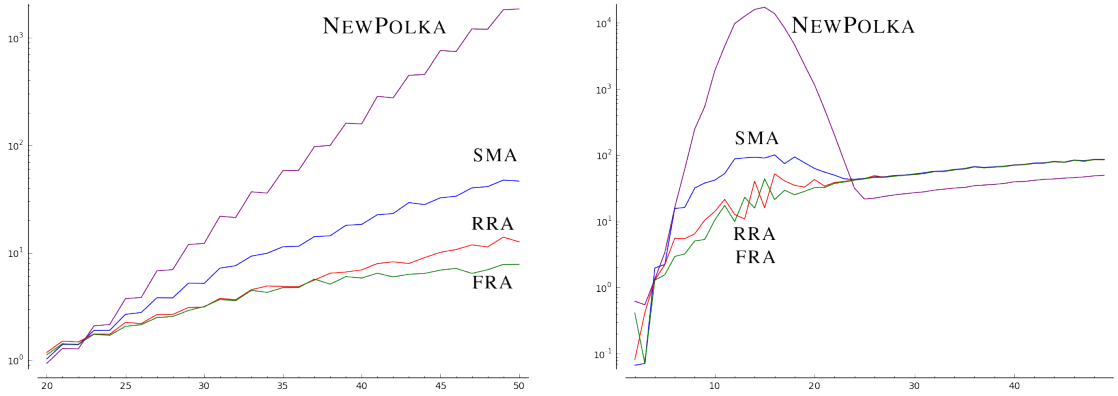


(a)



(b)

Figure 2.11 – $(a) : C = 50, V = [2, 50], D = 50\%, R = 50\%$; (b): number of generators associated to polyhedra tested in (a).



$(a) : C = [20, 50], V = 10, D = 50\%, R = 50\%$ $(b) : C = 50, V = [2, 50], D = 50\%, R = 50\%$

Figure 2.12 – Execution time in milliseconds (log scale) of SMA (blue), RRA (red), FRA (green) and NEWPOLKA (purple) depending on respectively (a) the number of constraints and (b) the number of variables.

Density Rate. The density of a polyhedron is the (average) rate of nonnull coefficients within a constraint. For instance, a density of 60% with 10 variables means that on average, constraints have 6 nonnull coefficients. Fig. 2.10(c) shows the execution time for 10-dimensional polyhedra with 100 constraints, where the density rate D goes from 10% to 80%. The raytracing algorithms are almost insensitive to density, whereas the execution time of the standard algorithm blows up with density. Actually, having a lot of nonnull coefficients in constraints tends to create huge numerators and denominators because a pivot in the simplex performs many combinations of constraints. The coefficient explosion does not happen in RRA because LP problems are much smaller in the raytracing algorithms.

Number of Variables. The effect of the dimension on execution time is shown on Fig. 2.10(d). Whereas raytracing seems linearly impacted by the dimension, SMA has a behaviour that may look a bit strange. After a dramatic increase of execution time, the curve falls down when the dimension reaches the number of irredundant constraints, which is half the given number of constraints as the redundancy rate equals 50%. SMA finally joins and sticks to FRA curve. This phenomenon may be explained if we take a look at the number of generators. Indeed, when $V \geq C$, the number of generators becomes close to the number of constraints, as shown on Fig. 2.11. Recall that the simplex algorithm travels from one vertex to another until the optimal value is reached. If the number of generators is low, few pivots are then needed to solve the LP problem. This makes SMA competitive even with more LP problems to solve.

Table 2.13 (p. 51) shows results for several values of dimension and number of constraints. Again, each cell of this table gives the average values resulting from the minimization of 50 convex potatoids, with a density and a redundancy both fixed at 50%. For each pair (number of variables $\times$ number of constraints), Table 2.13 gives the number of LP problems that were solved and their size (*i.e.* the number of constraints they involve) on average. It contains also the computation time of the minimization in milliseconds and the speed up of raytracing compared to SMA. Results of Table 2.13 show that for small polyhedra, either in dimension or in number of constraints, raytracing does not help. Indeed, for such small LP problems, the overhead of our algorithm is unnecessary and leads to time losses. Raytracing becomes interesting for larger polyhedra, where the speed improvement is significant. For instance, FRA is 80 times faster with 10 variables and 100 constraints than SMA. The gain can be explained by the number of LP problems solved and their average size, noticeably smaller in raytracing than in SMA. As expected, raytracing is faster with floating point.

We also compare our algorithms with the NEWPOLKA, a double description library available in APRON.[4] As an illustration, Fig. 2.12 adds the minimization time computed with

4. http://apron.cri.ensmp.fr/library/

APRON for the same tests as Fig. 2.10(b) and (d), with a log scale. Not surprisingly the minimization time of APRON is significantly larger than that of SMA, FRA and RRA, as it must first compute the representation as generators using Chernikova's algorithm. The linear behaviour of APRON in log scale (Fig. 2.12(a)) shows the exponential complexity of the conversion from constraints to generators. The behaviour of APRON in Fig. 2.12(b) is the same as SMA and is directly correlated to the number of generators, as mentioned above.

2.5 Redundancy in the Double Description Framework

The raytracing algorithm has been designed to minimize polyhedra in single representation, but its principle can be reused in the double description framework, where it could *quickly detect irredundant constraints*. Redundancy is easier to detect when the two representations of a polyhedron are available. Let the pair $(\mathscr{C},\mathscr{G})$ denote the set of constraints and the set of generators of a polyhedron in $\mathbb{Q}^n$ and $(\mathscr{C}_M,\mathscr{G}_M)$ be its minimal version. A constraint $C \in \mathscr{C}$ is irredundant if it is saturated by at least n irredundant generators, *i.e.* $\exists \boldsymbol{g}_1,\ldots,\boldsymbol{g}_n \in \mathscr{G}_M, \langle C,\boldsymbol{g}_i\rangle = 0$. Similarly, a generator $\boldsymbol{g} \in \mathscr{G}$ is irredundant if it is the intersection of at least n irredundant constraints *i.e.* $\exists C_1,\ldots,C_n \in \mathscr{C}_M, \langle C_i,\boldsymbol{g}\rangle = 0$. Think for instance of a line in 2D being defined by two points and a point being the intersection of at least two lines.

The principle of the minimization algorithm is the following (Halbwachs, 1979): build the *Boolean saturation matrix* $\boldsymbol{S}$ of size $|\mathscr{C}|\times|\mathscr{G}|$ defined by $\boldsymbol{S}[C][\boldsymbol{g}] := (\langle C,\boldsymbol{g}\rangle = 0)$, then iteratively remove constraints (and the corresponding rows of $\boldsymbol{S}$) which are insufficiently saturated and do the same for generators (and columns of $\boldsymbol{S}$) until reaching a stable matrix. The remaining constraints and generators form the minimal version $(\mathscr{C}_M,\mathscr{G}_M)$ which mutually justify the irredundancy of each other. This algorithm is appealing compared to its counterpart in single representation but the number of evaluation of $\langle C,\boldsymbol{g}\rangle$ is huge when each variable x_i ranges in an interval $[l_i,u_i]$. Such a product of intervals can be represented by $2n$ constraints (two inequalities $l_i \le x_i \wedge x_i \le u_i$ per variable) which corresponds to 2^n vertices. The opposite phenomenon ($2n$ vertices corresponding to 2^n constraints) also exists but hardly ever occurs in practice (Benoy et al., 2005). Therefore, the size of $\boldsymbol{S}$ is $n2^{n+1}$.

To limit the computations, the saturation matrix is not fully constructed. Let us summarize the improved algorithm (Wilde, 1993):

(1) Some constraints are removed by the *fast redundancy detection* mentioned in §2.1.

(2) The irredundant generators of $\mathscr{G}_M$ are constructed from the remaining constraints using Chernikova's algorithm with some optimized adjacency criteria (Le Verge, 1992; Fukuda and Prodon, 1996; Zolotykh, 2012). The adjacency criterion ensures that the construction cannot produce redundant generators (Motzkin et al., 1953).

(3) Finally, the saturation matrix is built to remove the constraint redundancies but a row is only completed if the constraint never finds enough saturating generators, otherwise the computation of the row is interrupted.

We believe that our orthogonal raytracing phase can be used at step (3) to *quickly discover irredundant constraints*, which therefore do not have to be confirmed by the saturation matrix. The cost of this initial raytracing is reasonable: $\mathscr{C}$ rays and $2\times|\mathscr{C}|$ evaluations per ray resulting in $2\times|\mathscr{C}|^2$ computations of inner products. It could therefore benefit to minimization in the double description framework especially when $|\mathscr{C}| \ll |\mathscr{G}|$ as in hypercubes.

Table 2.13 – Time measures (in ms) of the three minimization algorithms SMA, RRA and FRA for different values of variables and constraints. $D = 50\%, R = 50\%$.

#vars		5 constraints			10 constraints			25 constraints			50 constraints			100 constraints		
		SMA	RRA	FRA	SMA	RRA	FRA	SMA	RRA	FRA	SMA	RRA	FRA	SMA	RRA	FRA
2	# lp	5	2	2	10	5	5	25	18	18	50	42	42	100	90	90
	lp size	4	3	3	7	3	3	16	4	3	29	4	3	53	5	4
	time	0.12	0.19	0.46	0.54	0.51	0.75	3.1	1.7	2.1	12.4	5.4	6.1	65.4	23.3	25.0
	speed up	-	0.65	0.26	-	1.1	0.73	-	**1.9**	**1.5**	-	**2.3**	**2.0**	-	**2.8**	**2.6**
5	# lp	5	2	2	10	6	6	25	15	15	50	35	35	100	78	78
	lp size	4	3	3	8	3	3	20	5	4	37	6	5	75	7	6
	time	0.13	0.31	0.36	0.60	0.83	0.85	8.5	3.4	3.0	74.3	12.7	10.3	571	57.6	41.6
	speed up	-	0.42	0.36	-	0.73	0.71	-	**2.5**	**2.8**	-	**5.8**	**7.2**	-	**9.9**	**13.7**
10	# lp	5	2	2	10	5	5	25	12	12	50	27	27	100	58	58
	lp size	4	4	3	8	4	4	19	6	6	38	9	7	74	13	10
	time	0.24	0.30	0.31	0.64	1.1	1.0	11.2	8.7	6.9	639	59.7	31.4	9s	331	117
	speed up	-	0.82	0.77	-	0.59	0.64	-	**1.3**	**1.6**	-	**10.7**	**20.4**	-	**28.3**	**80.1**

Chapter 3

Minimizing Operators via Parametric Linear Programming

In Chapter 1, we presented the projection operator as the bottleneck of constraints-only representation. It is used in the computation of assignments, Minkowski sums and convex hulls. In §1.2, we introduced Fourier-Motzkin elimination which is the standard algorithm for projection, but suffers from an exponential complexity in the number of eliminated variables, due to the generation of many redundant constraints during intermediate projection steps. Fourier-Motzkin elimination is not an issue for assignment, which only requires the elimination of a single variable. But, its complexity becomes a problem for convex hull computation: Benoy et al. (2005)'s encoding of the convex hull $\mathcal{P}' \sqcup \mathcal{P}'' \subseteq \mathbb{Q}^n$ as a projection yields the elimination of $n+1$ variables in a polyhedron of dimension $2n+1$. The same pitfall occurs for Minkowski sum, as it is computed with a similar encoding.

The high cost of general convex polyhedra was long deplored. It motivated studying restricted classes of polyhedra, with simpler and faster algorithms, such as *octagons* (Miné, 2006); and even these were found to be too slow, motivating recent algorithmic improvements (Singh et al., 2015). Instead, this thesis revisits the domain of polyhedra with different algorithms.

Our work on projection was inspired by Howe and King (2012)'s attempt to avoid generating redundant constraints by replacing Fourier-Motzkin elimination with a formulation in Parametric Linear Programming (PLP), which they solved by an ad hoc algorithm. Their PLP encoding was formulated as the enumeration of all vertices in a sliced polyhedral cone of Farkas combinations. Unfortunately, their implementation is not available. We took the more direct approach and expressed the PLP encoding of projection directly in terms of constraints: the objective function becomes a Farkas combination to minimize. Fouilhé (2015) showed that a result free of redundancies requires adding a normalization constraint in the encoding, which is somehow equivalent to slicing the cone of solutions in Howe and King (2012). Here, I take a step further and developed a generic PLP-solver exploiting insights by Jones et al. (2007, 2008). The solver is implemented in OCaml, works over rationals and generates Coq-certificates of correctness of its computations.

This work was published and presented during the 24^{th} Static Analysis Symposium (Maréchal et al., 2017), in New York.

Parametric Linear Programming. Parametric Linear Programming is an extension of Linear Programming, that was presented in §1.3, where the constants in the constraints or the coefficients in the objective function may be replaced by affine combinations of parameters (Gal and Nedoma, 1972).

Parameters may appear in the right-hand side of constraints or in the objective function to optimize, but not in both. In the primal version, the unknowns $\boldsymbol{v}$ and $\boldsymbol{p}$ of a standard linear optimization problem $[\boldsymbol{A'}|\boldsymbol{A''}](\boldsymbol{v}|\boldsymbol{p})^\intercal \leq \boldsymbol{b}$ can be split into decision variables, the $\boldsymbol{v}$, which will be set to an optimal value whereas the parameters $\boldsymbol{p}$ will remain free. The system is

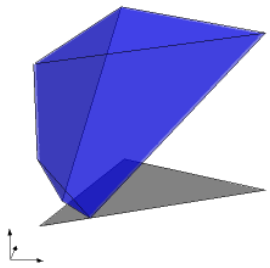
equivalent to $\boldsymbol{A'v} \leq \boldsymbol{b} - \boldsymbol{A''p}$ where the right hand side now depends on parameters. Note that instantiating $\boldsymbol{p}$ brings us back to a standard LP problem. The goal of this parametric problem is to compute a generic solution parameterized by $\boldsymbol{p}$. Parameters naturally arise when the objective function to optimize is a bilinear form on products $\lambda_i \times x_j$. Such nonlinear problems fall into the field of PLP if one unknown is a nonnegative decision variable and the other is considered as a parameter. This problem is the dual version of the previous one. In the thesis, we will be interested only in this dual form.

In the following, we start by introducing PLP on projection. Then, we will see how to encode convex hull directly as a Parametric Linear Optimization Problem (PLOP). The PLP-solver that we developed in the VPL is detailed in Chapter 5.

3.1 Projection via Parametric Linear Programming

Naive Fourier-Motzkin elimination produces $\mathcal{O}((\frac{|\mathcal{P}|}{2})^{2^k})$ constraints when eliminating k variables of a polyhedron with $|\mathcal{P}|$ constraints (Simon and King, 2005). Most of them are redundant: indeed, the number of faces of the projected polyhedron is $\mathcal{O}(|\mathcal{P}|^k)$ (Monniaux, 2010, §4.1).This follows from McMullen's bound on the number of $n-k-1$-faces of the polyhedron (McMullen, 1970; McMullen and Shepard, 1971). Therefore, removing the redundant constraints is costly, whatever the efficiency of the minimization algorithm.

Example 3.1



This figure shows the geometrical space defined by the polyhedron $\mathcal{P} = \{C_1 : -x_1 - 2x_2 + 2x_3 \geq -7, C_2 : -x_1 + 2x_2 \geq 1, C_3 : 3x_1 - x_2 \geq 0, C_4 : -x_3 \geq -10, C_5 : x_1 + x_2 + x_3 \geq 5\}$ and its projection on dimensions (x_1, x_2) resulting from the elimination of variable x_3. Eliminating variable x_3 from $\mathcal{P}$ – noted $\mathcal{P}_{\setminus\{x_3\}}$ – by Fourier-Motzkin elimination consists in combining constraints with opposite signs for x_3. Constraints that do not involve x_3 remain unchanged. This process retains constraints C_2, C_3 and produces two new constraints: $C_1 + 2 \times C_4 : -x_1 - 2x_2 \geq -27$ and $C_4 + C_5 : x_1 + x_2 \geq -5$. By Farkas' Lemma, the latter is redundant w.r.t. C_2 and C_3 as it can be expressed as a nonnegative combination of C_2 and C_3.

Jones et al. (2008) then Howe and King (2012) noticed that the projection of a polyhedron can be expressed as a PLOP. In fact, PLP naturally arises when trying to generalize Fourier-Motzkin method to eliminate several variables simultaneously. In this chapter, we achieve the work initiated by Howe and King (2012) and followed by Fouilhé (2015), whose goal was to compute the projected polyhedron without generating redundant constraints. Let us first explain their approach.

Example 3.2 (follows 3.1)

As a consequence of Farkas' lemma, any constraint implied by $\{C_1, \ldots, C_5\}$ is a nonnegative combination of them, written $\lambda_0 + \sum_{i=1}^{5} \lambda_i C_i$ with $\lambda_i \geq 0$, *i.e.*

$$\begin{aligned} &\lambda_0 + \lambda_1(-x_1 - 2x_2 + 2x_3) + \lambda_2(-x_1 + 2x_2) + \lambda_3(3x_1 - x_2) \\ &+ \lambda_4(-x_3) + \lambda_5(x_1 + x_2 + x_3) \quad \geq \quad -7\lambda_1 + \lambda_2 - 10\lambda_4 + 5\lambda_5 \end{aligned}$$

The left-hand side of the inequality can be rearranged to reveal the coefficient of each

variable x_i and we can bring the right-hand side term to the left.

$$\begin{aligned}\lambda_0 + (-\lambda_1 - \lambda_2 + 3\lambda_3 + \lambda_5)\, x_1 + (-2\lambda_1 + 2\lambda_2 - \lambda_3 + \lambda_5)\, x_2 \\ + (2\lambda_1 - \lambda_4 + \lambda_5)\, x_3 - (-7\lambda_1 + \lambda_2 - 10\lambda_4 + 5\lambda_5) \geq 0\end{aligned} \tag{3.1}$$

Then, any instantiation of that inequality with λ_i canceling the coefficient of x_3, *i.e.* that satisfies (α) $2\lambda_1 - \lambda_4 + \lambda_5 = 0$, is an over-approximation of $\mathcal{P}_{\setminus\{x_3\}}$. Indeed, it does not involve x_3 and as a Farkas combination, it is by construction a logical consequence of $\mathcal{P}$. Constraints found by the Fourier-Motzkin elimination of x_3 correspond to the solutions $(\lambda_0, \ldots, \lambda_5) \in \{(0,0,1,0,0,0), (0,0,0,1,0,0), (0,1,0,0,2,0), (0,0,0,0,1,1)\}$ of Equation (α). Note that it is possible to eliminate several variables simultaneously by setting an elimination equation for each variable that must be discarded.

We give here a first formulation of a projection as a PLOP. We will refine it later, as it is not sufficient to avoid redundancies in the projected polyhedron. Given a polyhedron $\mathcal{P} = \{C_1 : \boldsymbol{a_1}(\boldsymbol{x}) \geq b_1, \ldots, C_p : \boldsymbol{a_p}(\boldsymbol{x}) \geq b_p\}$ on variables $x_1, \ldots, x_n$, the projection of $\mathcal{P}$ by elimination of k variables $x_{e_1}, \ldots, x_{e_k}$ can be obtained as the solution of the optimization problem:

$$\begin{aligned}&\textbf{minimize } \mathbf{Z}(\boldsymbol{x}) \stackrel{\text{def}}{=} \lambda_0 + \sum_{i=1}^{p} \lambda_i \times (\boldsymbol{a_i}(\boldsymbol{x}) - b_i) \\ &\quad \textbf{subject to} \\ &\qquad (F)\ \lambda_0 \geq 0, \ldots, \lambda_p \geq 0 \\ &\qquad (\dagger)\ \sum_{i=0}^{p} \lambda_i = 1 \\ &\qquad (\alpha)\ \boldsymbol{\alpha}_{e_1}(\boldsymbol{\lambda}) = 0, \ldots, \boldsymbol{\alpha}_{e_k}(\boldsymbol{\lambda}) = 0\end{aligned} \tag{PLOP 3.2}$$

where $\boldsymbol{x} \stackrel{\text{def}}{=} (x_1, \ldots, x_n)$, $\boldsymbol{\lambda} \stackrel{\text{def}}{=} (\lambda_0, \ldots, \lambda_p)$ and $\boldsymbol{\alpha}_i(\boldsymbol{\lambda})$ denotes the coefficient of x_i in the reformulation of the objective as

$$\boldsymbol{\alpha}_0(\boldsymbol{\lambda}) + \boldsymbol{\alpha}_1(\boldsymbol{\lambda}) \times x_1 + \ldots + \boldsymbol{\alpha}_n(\boldsymbol{\lambda}) \times x_n$$

like in Equation (3.1) of Example 3.2. The unknowns λ_i's are the *decision variables* of the PLOP: the solver must find a solution for them. Note the inequalities (F) from Farkas' Lemma in addition to the (α) equations defining a projection. This problem has a *parametric objective*: the objective function $\boldsymbol{Z}(\boldsymbol{x})$ depends on parameters $x_1, \ldots, x_n$ due to the terms $\boldsymbol{a}_i(\boldsymbol{x})$ in the coefficients of the decision variables. This problem belongs to *parametric linear programming* because once $x_1, \ldots, x_n$ are fixed, it boils down to linear programming: both the objective function and the constraints become affine functions of the decision variables.

An additional constraint $(\dagger)$, here $\sum_i \lambda_i = 1$, is needed to prevent the solver from obtaining the optimal solution $\boldsymbol{\lambda} = \mathbf{0}$ which is always valid in a projection problem, whatever the parameter values. The $(\dagger)$ condition only excludes this useless null solution because any other solution can be scaled so that $\sum_i \lambda_i = 1$. The presence of λ_0 in the objective can seem useless and strange to readers who are familiar with linear programming: the solution $\lambda_0 = 1$ and $\lambda_1 = \ldots = \lambda_p = 0$ becomes feasible and generates a trivially redundant constraint $C_{triv} : 1 \geq 0$. The role of λ_0 will become clear when we will introduce the normalization constraint in §3.4.

Example 3.3 (follows 3.2)

The elimination of x_3 via PLP is defined by two matrices: $\boldsymbol{O}$ is built from $[-\boldsymbol{b}|\boldsymbol{A}]^\intercal$ and encodes the objective. The other one captures the requirement (α) and $(\dagger)$. As usual in solvers, Farkas constraints (F) are left implicit.

$$\textbf{minimize } \mathbf{Z}(\boldsymbol{x}) \stackrel{\text{def}}{=} (1, x_1, x_2, x_3)^\intercal \overbrace{\left(\begin{array}{ccccccc} 0 & 1 & 7 & -1 & 0 & 10 & -5 \\ 0 & 0 & -1 & -1 & 3 & 0 & 1 \\ 0 & 0 & -2 & 2 & -1 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & -1 & 1 \end{array}\right)}^{\boldsymbol{O}} \begin{pmatrix} 1 \\ \lambda_0 \\ \vdots \\ \lambda_5 \end{pmatrix}$$

$$\textbf{subject to } \overbrace{\underbrace{\left(\begin{array}{ccccccc} -1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 2 & 0 & 0 & -1 & 1 \end{array}\right)}_{\boldsymbol{\alpha}}}^{(\dagger)} \begin{pmatrix} 1 \\ \lambda_0 \\ \vdots \\ \lambda_5 \end{pmatrix} = \mathbf{0} \qquad \text{(PLOP 3.3)}$$

This formulation of the projection is sound. Unfortunately, it may still generate redundant constraints: the solutions $(\lambda_0, \ldots, \lambda_5) \in \{$(1,0,0,0,0,0), (0,0,1,0,0,0), (0,0,0,1,0,0), (0,$\frac{1}{3}$,0,0,$\frac{2}{3}$,0), (0,0,0,0,$\frac{1}{2}$,$\frac{1}{2}$)$\}$ include the trivial constraint $1 \geq 0$ and $\frac{1}{2} \times C_4 + \frac{1}{2} \times C_5$ which is equivalent to the redundant constraint $C_4 + C_5$ found by Fourier-Motzkin elimination. The normalization constraint of §3.4 will solve this point.

3.2 Polyhedron as Solution of a PLOP

In the previous section we encoded the projection of a polyhedron as a PLOP. For interpreting the result of a PLP-solver as a polyhedron we need to go one step further into the field of PLP and look at the solutions of a PLOP.

The general form of a PLOP that stems from projection is

$$\begin{aligned} \textbf{minimize } \mathbf{Z}(\boldsymbol{x}) \stackrel{\text{def}}{=} & \ \lambda_0 + \sum_{i=1}^{p} \lambda_i \times (\boldsymbol{a}_i(\boldsymbol{x}) - b_i) \\ \textbf{subject to} & \ \lambda_0, \ldots, \lambda_p \geq 0 \\ (\dagger) & \ \sum_{i=0}^{p} \lambda_i = 1, \\ & \ \boldsymbol{\alpha}\boldsymbol{\lambda} = \mathbf{0} \end{aligned} \qquad \text{(PLOP 3.4)}$$

where $\boldsymbol{x}$ is the vector of parameters $(x_1, \ldots, x_n)$; $(\boldsymbol{a}_i(\boldsymbol{x}) - b_i)$ are *affine forms on the parameters*; and $\boldsymbol{\alpha}$ is a matrix. In a projection problem the system of equations $\boldsymbol{\alpha}\boldsymbol{\lambda} = \mathbf{0}$ constrains the *decision variables* $\lambda_1, \ldots, \lambda_p$ but not λ_0.

The solution $\boldsymbol{Z}^\star$ is a concave, piecewise affine function, mapping the parameters to the optimal solution:

$$\boldsymbol{Z}^\star \stackrel{\text{def}}{=} \boldsymbol{x} \mapsto \begin{cases} \boldsymbol{Z}_1^\star(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in \mathcal{R}_1 \\ \vdots & \\ \boldsymbol{Z}_r^\star(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in \mathcal{R}_r \end{cases} \qquad (3.5)$$

Each piece $\boldsymbol{Z}_i^\star$ is an affine form over $\boldsymbol{x}$, obtained by instantiating the objective function $\boldsymbol{Z}$ with a solution $\boldsymbol{\lambda}$; hence a piece can also be denoted by $\boldsymbol{Z}_{\boldsymbol{\lambda}}^\star$. Each $\boldsymbol{Z}_i^\star$ is associated to a *region of optimality* $\mathcal{R}_i$ that designates the set of $\boldsymbol{x}$ for which the minimum of $\boldsymbol{Z}^\star(\boldsymbol{x})$ is $\boldsymbol{Z}_i^\star(\boldsymbol{x})$. Regions of optimality are polyhedra; that will be clear in §3.3 when we will explain how they are computed by our solver. They form a *quasi-partition* of the space of parameters: their union covers $\mathbb{Q}^n$ and the intersection of *the interior* of two distinct regions is empty, *i.e.* $\forall i \neq j$,

$[\![\mathring{\mathcal{R}}_i]\!] \cap [\![\mathring{\mathcal{R}}_j]\!] = \emptyset$. They however do not form a partition because two regions $\mathcal{R}_i, \mathcal{R}_j$ may overlap on their frontiers; then, their solutions $\boldsymbol{Z}^\star_i, \boldsymbol{Z}^\star_j$ coincide on the intersection.

From optimal function to polyhedron. A PLOP can be thought of as a *declarative description of the projection operator*. The solution $\boldsymbol{Z}^\star$ can be interpreted as a polyhedron $\mathcal{P}^\star$ that is the projection of an input polyhedron $\mathcal{P}$. This requires some explanations:

- Due to the Farkas conditions $\lambda_0, \ldots, \lambda_p \geq 0$ which preserve the direction of inequalities, the objective function of (PLOP 3.4), *i.e.* $\lambda_0 + \sum_{i=1}^{p} \lambda_i \times (\boldsymbol{a}_i(\boldsymbol{x}) - b_i)$ can be interpreted as a constraint implied by the input polyhedron $\mathcal{P} = \{C_1 : \boldsymbol{a}_i(\boldsymbol{x}) \geq b_1, \ldots, C_p : \boldsymbol{a}_i(\boldsymbol{x}) \geq b_p\}$. Actually, for a given $\boldsymbol{\lambda}$, the statement $\boldsymbol{Z}^\star_{\boldsymbol{\lambda}}(\boldsymbol{x}) \geq 0$ is equivalent to the constraint
$$\lambda_0 + \sum_{i=1}^{p} \lambda_i \times \boldsymbol{a}_i(\boldsymbol{x}) \geq \sum_{i=1}^{p} \lambda_i \times b_i \tag{3.6}$$
- Minimizing the objective ensures that the λ_0-shift of the constraint will be minimal, meaning that the constraint $\boldsymbol{Z}^\star_{\boldsymbol{\lambda}}(\boldsymbol{x}) \geq 0$ will be tightly adjusted.
- The requirement $\boldsymbol{\alpha\lambda} = \boldsymbol{0}$ captures the expected effect of the projection. Thus, any solution $\boldsymbol{\lambda}$ defines a constraint $\boldsymbol{Z}_{\boldsymbol{\lambda}}(\boldsymbol{x}) \geq 0$ of the polyhedron $\mathcal{P}^\star$.

Now recall that a polyhedron is a set of points that satisfy affine inequalities. Therefore, it is natural to define $[\![\mathcal{P}^\star]\!]$ as $\{\boldsymbol{x} \mid \boldsymbol{Z}^\star(\boldsymbol{x}) \geq 0\}$. The following lemma proves that this set of points is a polyhedron.

Lemma 3.1
$$\{\boldsymbol{x} \mid \boldsymbol{Z}^\star(\boldsymbol{x}) \geq 0\} = \bigcap_{k=1}^{r} \{\boldsymbol{x} \mid \boldsymbol{Z}^\star_k(\boldsymbol{x}) \geq 0\}$$

Proof. Let us prove the mutual inclusion.

- ($\subseteq$) Pick up a point $\boldsymbol{x'} \in \{\boldsymbol{x} \mid \boldsymbol{Z}^\star(\boldsymbol{x}) \geq 0\}$. By definition of $\boldsymbol{Z}^\star$ as a piecewise function defined on the whole space of parameters, there exists i such that $\boldsymbol{x'} \in \mathcal{R}_i$ and $\boldsymbol{Z}^\star(\boldsymbol{x'}) = \boldsymbol{Z}^\star_i(\boldsymbol{x'})$. It follows that $\boldsymbol{Z}^\star_i(\boldsymbol{x'}) \geq 0$ since $\boldsymbol{x'}$ belongs to the set of points where $\boldsymbol{Z}^\star$ is nonnegative. Moreover, the fact that $\boldsymbol{x'}$ belongs to $\mathcal{R}_i$ – the region of optimality of $\boldsymbol{Z}^\star_i$ in a minimization problem – ensures that $\boldsymbol{Z}^\star_k(\boldsymbol{x'}) \geq \boldsymbol{Z}^\star_i(\boldsymbol{x'})$ for all k and therefore, $\boldsymbol{Z}^\star_k(\boldsymbol{x'}) \geq 0$ for all k. Thus, $\boldsymbol{x'} \in \{\boldsymbol{x} \mid \boldsymbol{Z}^\star_k(\boldsymbol{x}) \geq 0\}$ for all $k = 1..r$. Finally, $\boldsymbol{x'} \in \bigcap_{k=1}^{r} \{\boldsymbol{x} \mid \boldsymbol{Z}^\star_k(\boldsymbol{x}) \geq 0\}$.
- ($\supseteq$) Pick up a point $\boldsymbol{x'} \in \bigcap_{k=1}^{r} \{\boldsymbol{x} \mid \boldsymbol{Z}^\star_k(\boldsymbol{x}) \geq 0\}$. Then, $\boldsymbol{x'}$ belongs to at least one region $\mathcal{R}_i$ because the regions form a (pseudo) partition of the whole space of parameters $\mathbb{Q}^n$, *i.e.* $\bigcup_{k=1}^{r} \mathcal{R}_k = \mathbb{Q}^n$. Yet, the affine piece that defines $\boldsymbol{Z}^\star$ on $\boldsymbol{x'}$ is $\boldsymbol{Z}^\star_i$ and $\boldsymbol{Z}^\star(\boldsymbol{x'}) = \boldsymbol{Z}^\star_i(\boldsymbol{x'})$. Moreover, all the affine pieces of $\boldsymbol{Z}^\star$ are nonnegative on $\boldsymbol{x'}$ since $\boldsymbol{x'} \in \bigcap_{k=1}^{r} \{\boldsymbol{x} \mid \boldsymbol{Z}^\star_k(\boldsymbol{x}) \geq 0\}$. Then, in particular $\boldsymbol{Z}^\star_i(\boldsymbol{x'}) \geq 0$ and the same goes for $\boldsymbol{Z}^\star(\boldsymbol{x'})$. Finally, $\boldsymbol{x'} \in \{\boldsymbol{x} \mid \boldsymbol{Z}^\star(\boldsymbol{x}) \geq 0\}$.

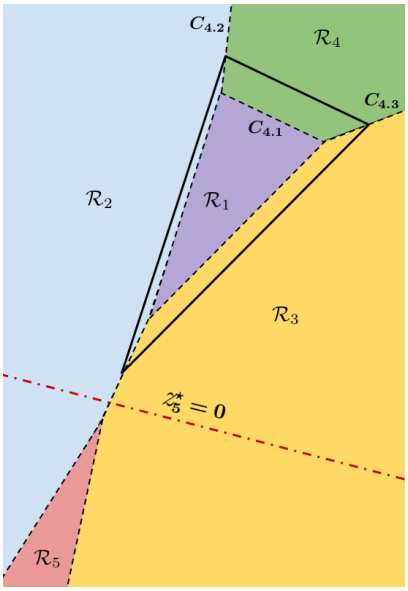
□

Constructing the vector inequality $\boldsymbol{Z^\star x} \geq \boldsymbol{b^\star}$ that defines the polyhedron $\mathcal{P}^\star$ is straightforward from the solution $\boldsymbol{Z}^\star$. It suffices to get rid of the regions of optimality and to interpret each affine piece of $\boldsymbol{Z}^\star$ as an inequality:

$$\begin{array}{ll} \{\boldsymbol{x} \mid \boldsymbol{Z}^\star(\boldsymbol{x}) \geq 0\} & = \bigcap_{k=1}^{r} \{\boldsymbol{x} \mid \boldsymbol{Z}^\star_k(\boldsymbol{x}) \geq 0\} \qquad \text{by Lemma 3.1} \\ & = \{\boldsymbol{x} \mid \bigwedge_{k=1}^{r} \boldsymbol{Z}^\star_k(\boldsymbol{x}) \geq 0\} \\ & = \{\boldsymbol{x} \mid \bigwedge_{k=1}^{r} \langle z^\star_k, x\rangle - b^\star_k \geq 0\} \\ & = \{\boldsymbol{x} \mid \boldsymbol{Z^\star x} \geq \boldsymbol{b^\star}\}. \end{array}$$

Let us detail this construction. Each piece $\boldsymbol{Z}^\star_k$ of the solution is a affine form over $\boldsymbol{x}$ and $\boldsymbol{Z}^\star_k(\boldsymbol{x}) \geq 0$ defines a constraint in the form (3.6) which can be written $\sum_{i=1}^{n} z^\star_{ki}\ x_i \geq b^\star_k$ *i.e.* $\langle z^\star_k, x\rangle \geq b^\star_k$ for some vector $\boldsymbol{z^\star_k} = (z^\star_{k1}, \ldots, z^\star_{kn})$ and some constant $b^\star_k$. It follows from Lemma 3.1 that the set of points $\boldsymbol{x}$ where $\boldsymbol{Z}^\star(\boldsymbol{x})$ is nonnegative is a polyhedron defined by the vector inequality $\boldsymbol{Z^\star x} \geq \boldsymbol{b^\star}$ where the rows of $\boldsymbol{Z}^\star$ are the vectors $\boldsymbol{z^\star_1}, \ldots, \boldsymbol{z^\star_r}$ and $\boldsymbol{b^\star}$ is the column vector $(b^\star_1, \ldots, b^\star_r)^\intercal$.

Example 3.4



On our running projection problem (PLOP 3.3), the PLP-solver returns the following optimal function, and the instantiation of the decision variables λ_i that defines each affine piece:

$$\boldsymbol{Z}^\star \stackrel{\text{def}}{=} (x_1,x_2) \mapsto \begin{cases} \boldsymbol{Z}_2^\star : & -x_1 & +2x_2 & -1 & \text{on} & \mathcal{R}_2 & (\text{for } \lambda_2 = 1) \\ \boldsymbol{Z}_3^\star : & 3x_1 & -x_2 & & \text{on} & \mathcal{R}_3 & (\text{for } \lambda_3 = 1) \\ \boldsymbol{Z}_4^\star : & -\frac{1}{3}x_1 & -\frac{2}{3}x_2 & +9 & \text{on} & \mathcal{R}_4 & (\text{for } \lambda_1 = \frac{1}{3},\ \lambda_4 = \frac{2}{3}) \\ \boldsymbol{Z}_5^\star : & \frac{1}{2}x_1 & +\frac{1}{2}x_2 & +\frac{5}{2} & \text{on} & \mathcal{R}_5 & (\text{for } \lambda_4 = \frac{1}{2},\ \lambda_5 = \frac{1}{2}) \\ \boldsymbol{Z}_1^\star : & & & 1 & \text{on} & \mathcal{R}_1 & (\text{for } \lambda_0 = 1) \end{cases}$$

from which we construct the polyhedron

$$\mathcal{P}^\star = \overbrace{\begin{pmatrix} -1 & 2 & 0 \\ 3 & -1 & 0 \\ -\frac{1}{3} & -\frac{2}{3} & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 \end{pmatrix}}^{\boldsymbol{Z}^\star} \overbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}^{\boldsymbol{x}} \geq \overbrace{\begin{pmatrix} 1 \\ 0 \\ -9 \\ -\frac{5}{2} \\ -1 \end{pmatrix}}^{\boldsymbol{b}^\star} = \left\{ \begin{array}{rrr} C: & -x_1 + 2x_2 \geq & 1 \\ C: & 3x_1 - x_2 \geq & 0 \\ \frac{1}{3}C + \frac{2}{3}C: & -\frac{1}{3}x_1 - \frac{2}{3}x_2 \geq & -9 \\ \frac{1}{2}C + \frac{1}{2}C: & \frac{1}{2}x_1 + \frac{1}{2}x_2 \geq & -\frac{5}{2} \\ C_{triv}: & 0 \geq & -1 \end{array} \right\}$$

Variable x_3 does not appear anymore in the constraints of $\mathcal{P}^\star$ because its column in $\boldsymbol{Z}^\star$ is made of 0. The regions of optimality, shown on the figure above, form a pseudo-partition of the whole space of parameters (x_1,x_2): regions $\mathcal{R}_2,\ldots,\mathcal{R}_5$ are unbounded; the central triangle is the region $\mathcal{R}_1$ associated to the constant affine form $\boldsymbol{Z}_1^\star = 1$ which produces the trivial constraint $C_{triv} : 0 \geq -1$. Each facet of $\mathcal{P}^\star$ (shown as bold lines in the figure) is the intersection of a region of optimality $\mathcal{R}_i$ with the space where the associated affine form $\boldsymbol{Z}_i^\star$ evaluates to zero. We retrieve constraints equivalent to those of Example 3.1, including the redundant constraint $\frac{1}{2} \times C_4 + \frac{1}{5} \times C_5$, generated by $\boldsymbol{Z}_5^\star$. Examining the drawing of the regions reveals that $\boldsymbol{Z}_5^\star$ does not vanish on its region of optimality, *i.e.* $[\![\boldsymbol{Z}_5^\star = 0]\!] \cap [\![\mathring{\mathcal{R}}_5]\!] = \emptyset$. Actually, this phenomenon characterizes redundant constraints. We will prove in Lemma 3.6 (§3.4) that $[\![\boldsymbol{Z}_i^\star = 0]\!] \cap [\![\mathring{\mathcal{R}}_i]\!] \neq \emptyset$ ensures the irredundancy of the constraint $\boldsymbol{Z}_i^\star \geq 0$ in $\mathcal{P}^\star$.

3.3 Principle of a PLP solver

In the following, we only sketch how our PLP-solver works. The details of implementation will be discussed later, in Chapter 5. Our algorithm is based on recent work by Jones et al. (2007) with some improvements: our implementation reuses the fast simplification of regions from Chapter 2 and performs exact computations in rationals so as to avoid rounding errors.

The algorithm for solving a PLOP is a generalization of the simplex algorithm which we presented in §1.3. First, each inequality $C_\ell : \sum_{i=1}^{n} a_{\ell i}\,\lambda_i \leq b_\ell$ is changed into an equality $\sum_{i=1}^{n} a_{\ell i}\,\lambda_i + \lambda_{n+\ell} = b_\ell$ by introducing a slack variable variable $\lambda_{n+\ell} \geq 0$. Second, the objective function is added to the system as an extra equation defining the variable $\boldsymbol{Z}$ as a linear form $\boldsymbol{Z} = \sum_{i=1}^{n} o_i \lambda_i$. Then, as we saw for the simplex algorithm, it pivots as in Gaussian elimination until its reaches an equivalent system of equations where the optimality of $\boldsymbol{Z}$ becomes syntactically obvious. Let us take an example.

Example 3.5 (follows 3.4)

To illustrate the behavior of a LP-solver, such as the simplex, let us instantiate the objective of (PLOP 3.3), *e.g.* with $x_1 = 5, x_2 = 11, x_3 = 1$, to obtain a non-parametric version: $\boldsymbol{Z} \stackrel{\text{def}}{=} \lambda_0 - 18\,\lambda_1 + 16\,\lambda_2 + 4\,\lambda_3 + 9\,\lambda_4 + 12\,\lambda_5$. The simplex chooses the basis $\mathcal{B} = \{\lambda_1, \lambda_4\}$, meaning that λ_1 and λ_4 are defined in terms of the other decision variables $\mathcal{N} = \{\lambda_0, \lambda_2, \lambda_3, \lambda_5\}$. It exploits equations $(\dagger)$ and (α) of (PLOP 3.3) and gets

$$\lambda_1 = -\frac{1}{3}\lambda_0 - \frac{1}{3}\lambda_2 - \frac{1}{3}\lambda_3 - \frac{2}{3}\lambda_5 + \frac{1}{3} \qquad \text{(i)}$$

$$\lambda_4 = -\frac{2}{3}\lambda_0 - \frac{2}{3}\lambda_2 - \frac{2}{3}\lambda_3 - \frac{1}{3}\lambda_5 + \frac{2}{3} \qquad \text{(ii)}$$

$$\boldsymbol{Z} = \lambda_0 + 16\lambda_2 + 4\lambda_3 + 21\lambda_5 \qquad \text{(iii)}$$

Now, it is clear that choosing $\lambda_0, \lambda_2, \lambda_3, \lambda_5$ greater than 0 would increase the value of $\boldsymbol{Z}$ because their coefficient is positive in (iii). Thus, since decision variables are nonnegative, the minimum value of $\boldsymbol{Z}$ is reached for $\lambda_0 = \lambda_2 = \lambda_3 = \lambda_5 = 0$ which entails $\lambda_1 = \frac{1}{3}$ and $\lambda_4 = \frac{2}{3}$ using equations (i) and (ii). This example summarizes the principle of the standard simplex algorithm.

Now consider our projection problem (PLOP 3.3) with its *parametric objective*

$$\begin{aligned}\boldsymbol{Z}(x_1, x_2, x_3) \stackrel{\text{def}}{=}\ & \lambda_1(-x_1 - 2\,x_2 + 2\,x_3 + 7) + \lambda_2(-x_1 + 2\,x_2 - 1) + \lambda_3(3\,x_1 - x_2) \\ & + \lambda_4(-x_3 + 10) + \lambda_5(x_1 + x_2 + x_3 - 5) + \lambda_0\end{aligned}$$

Our PLP-solver uses the previous instantiated problem to discover the useful pivots (i) and (ii). Then, it replays the same rewritings on the parametric objective: λ_1 and λ_4 are replaced with their expression in (i) and (ii). Those substitutions are efficiently implemented using the matrix representation of (PLOP 3.3): they boil down to the addition of combinations of rows of $(\dagger)$ and $\boldsymbol{\alpha}$ to those of $\boldsymbol{O}$. We end up with the following parametric objective:

$$\begin{aligned}\underbrace{-\frac{1}{3}\,x_1 - \frac{2}{3}\,x_2 + 9}_{\boldsymbol{Z}^\star_4} \quad &+ \quad \lambda_0\ \underbrace{\frac{1}{3}\,(x_1 + 2\,x_2 - 24)}_{\geq 0 : C_{4.1}} \quad + \quad \lambda_2\ \underbrace{\frac{2}{3}\,(-x_1 + 4\,x_2 - 15)}_{\geq 0 : C_{4.2}} \\ &+ \quad \lambda_3\ \underbrace{\frac{1}{3}\,(10\,x_1 - x_2 - 27)}_{\geq 0 : C_{4.3}} \quad + \quad \lambda_5\ \underbrace{\frac{1}{3}\,(5\,x_1 + 7\,x_2 - 39)}_{\geq 0 : C_{4.4}}\end{aligned}$$

We recognize $\boldsymbol{Z}^\star_4$ which is the 4^{th} piece of $\boldsymbol{Z}^\star$ and other terms involving the remaining λ-variables; their coefficients are labelled $C_{4.1}, \ldots, C_{4.4}$. The argument for optimality

used in the non-parametric version can be generalized: the minimality of $\boldsymbol{Z}^\star_4$ holds if *the parametric coefficients of the remaining variables are nonnegative*. Indeed, increasing the values of $\lambda_0, \lambda_2, \lambda_3, \lambda_5$ (which must be nonnegative) would make the objective value grow. The nonnegativity of $C_{4.1}, \ldots, C_{4.4}$ defines the region of optimality $\mathcal{R}_4$ of $\boldsymbol{Z}^\star_4$ as the polyhedron $\{C_{4.1}, C_{4.2}, C_{4.3}\}$ constraining the parameters (x_1, x_2), as shown on the figure of Example 3.4. Note that $C_{4.4}$ is actually redundant with respect to $C_{4.1}$, $C_{4.2}$ and $C_{4.3}$. It is thus eliminated from the representation of $\mathcal{R}_4$ using our efficient minimization algorithm, and therefore does not appear on the figure.

3.4 The Normalization Constraint

The previous sections showed how to compute the optimal solution of a PLOP and how to interpret the solution $\boldsymbol{Z}^\star$ as a polyhedron $\mathcal{P}^\star = \bigwedge_{k=1}^{r} (\boldsymbol{Z}^\star_k(\boldsymbol{x}) \geq 0)$. Still, the representation of $\mathcal{P}^\star$ may not be minimal: some constraints $\boldsymbol{Z}^\star_k(\boldsymbol{x}) \geq 0$ may be redundant in $\mathcal{P}^\star$, *e.g.* $\boldsymbol{Z}^\star_5(\boldsymbol{x}) \geq 0$ is Example 3.4. We could remove those redundancies afterwards but, as noticed by Howe and King (2012), it is highly preferable to prevent their generation by adding a *normalization constraint* to the PLOP. We adapt their intuition to our formulation of the problem and we bring the proof that it indeed avoids redundancies. This requires making a detour via normalized solutions to explain the expected effect of a normalization constraint.

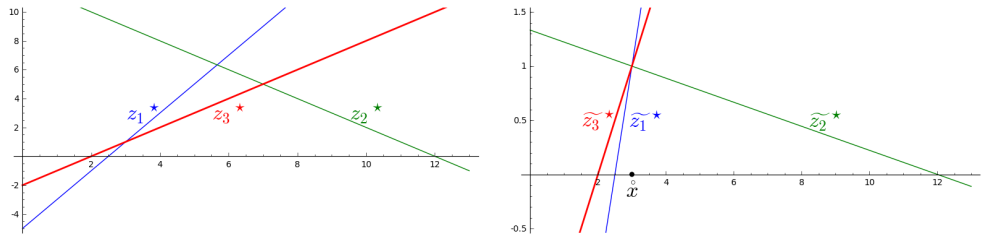
3.4.1 Normalizing the Solution of the Projection Problem

Let us normalize the function $\boldsymbol{Z}^\star$ so that it evaluates to 1 on a given point $\mathring{\boldsymbol{x}}$ chosen within the interior of $\mathcal{P}^\star$. Formally, we consider a solution

$$\widetilde{\boldsymbol{Z}}^\star(\boldsymbol{x}) \stackrel{\text{def}}{=} \frac{\boldsymbol{Z}^\star(\boldsymbol{x})}{\boldsymbol{Z}^\star(\mathring{\boldsymbol{x}})} \text{ or equivalently } \forall k,\ \widetilde{\boldsymbol{Z}}^\star_k(\boldsymbol{x}) \stackrel{\text{def}}{=} \frac{\boldsymbol{Z}^\star_k(\boldsymbol{x})}{\boldsymbol{Z}^\star_k(\mathring{\boldsymbol{x}})}$$

The key point of this transformation is that the space $[\![\boldsymbol{Z}^\star \geq 0]\!]$, which is the polyhedron $\mathcal{P}^\star$ of interest, is unchanged. The normalized solution $\widetilde{\boldsymbol{Z}}^\star$ will differ from the original one but must fulfills $[\![\widetilde{\boldsymbol{Z}}^\star \geq 0]\!] = [\![\boldsymbol{Z}^\star \geq 0]\!]$ which is true on the main functions if it holds on each of their pieces, *i.e.* $\forall k, [\![\widetilde{\boldsymbol{Z}}^\star_k \geq 0]\!] = [\![\boldsymbol{Z}^\star_k \geq 0]\!]$. The normalization preserves the nonnegativity space of each $\boldsymbol{Z}^\star_k$ because $\frac{1}{\boldsymbol{Z}^\star_k(\mathring{\boldsymbol{x}})}$ is a positive scalar: indeed, $\mathring{\boldsymbol{x}}$ belongs to the interior of $\mathcal{P}^\star$, *i.e.* $[\![\bigwedge_k \boldsymbol{Z}^\star_k > 0]\!]$ by Lemma 3.1.

Example 3.6



Normalizing the solution only changes the inclination of the $\boldsymbol{Z}^\star_k$'s, not the space where they cross 0. This can easily be illustrated on one-variable constraints. Consider three constraints $C_1 : 2x \geq 5$, $C_2 : x \leq 12$ and a redundant one $C_3 : x \geq 2$, corresponding to three affine forms $\boldsymbol{Z}^\star_1(x) = 2x - 5$, $\boldsymbol{Z}^\star_2(x) = 12 - x$ and $\boldsymbol{Z}^\star_3(x) = x - 2$. On the left-hand side we plotted the functions $z = \boldsymbol{Z}^\star_i(x)$ for $i \in \{1, 2, 3\}$ and, on the right-hand side, their normalizations w.r.t. the point $\mathring{x} = 3$.

The most interesting consequence of the normalization is that *a constraint is redundant iff its normalized affine form is nowhere minimal*. This property does not hold on the non-normalized

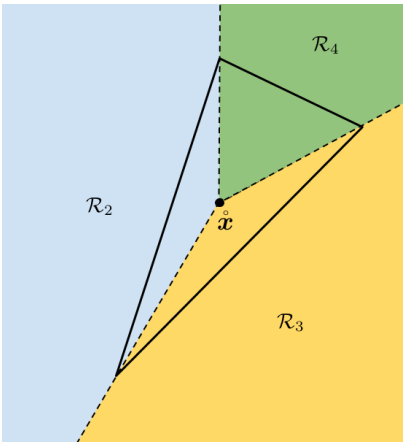
forms. For instance, in Example 3.6, although C_3 is redundant w.r.t. C_1 and C_2, $\boldsymbol{Z}_3^\star$ is minimal w.r.t. $\boldsymbol{Z}_1^\star$ and $\boldsymbol{Z}_2^\star$ on $x \in [3,7]$. On the contrary, considering the normalized forms, $\widetilde{\boldsymbol{Z}}_3^\star$ is no longer minimal, thus *it will be absent from the piecewise solution of a minimization problem*. The proof of this result (stated later as Theorem 3.2), which validates Howe and King (2012)'s intuition, is one contribution of this thesis.

Last, but not least, the normalized pieces are not computed *a posteriori* from the original solutions: our goal is to prevent their generation. Instead, irredundant pieces are obtained directly by enforcing the normalization of the objective through an additional constraint $\boldsymbol{Z}(\mathring{\boldsymbol{x}}) = 1$. Recall from (PLOP 3.4) that the objective of the PLOP is $\boldsymbol{Z}(\boldsymbol{x}) \stackrel{\text{def}}{=} \lambda_0 + \sum_{i=1}^{p} \lambda_i \times (\boldsymbol{a}_i(\boldsymbol{x}) - b_i)$. Then, the *normalization constraint* becomes

$$\lambda_0 + \sum_{i=1}^{p} \lambda_i \times (\boldsymbol{a}_i(\mathring{\boldsymbol{x}}) - b_i) = 1 \qquad (\maltese)$$

where the $\boldsymbol{a}_i(\mathring{\boldsymbol{x}})$'s are coefficients in $\mathbb{Q}$, obtained by evaluating the constraints of the input polyhedron at $\mathring{\boldsymbol{x}}$. The normalization constraint replaces the previous requirement (†) $\sum_i \lambda_i = 1$ in the PLOP: like (†) it excludes the solution $\lambda_0 = \ldots = \lambda_p = 0$.

Example 3.7 (follows 3.4)



$$\mathcal{R}_2 \stackrel{\text{def}}{=} -23x_1 + 51x_2 < 58 \ \wedge \ 9x_1 - 13x_2 > -6$$

$$\mathcal{R}_3 \stackrel{\text{def}}{=} 9x_1 - 13x_2 < -6 \ \wedge 321x_1 - 37x_2 < 810$$

$$\mathcal{R}_4 \stackrel{\text{def}}{=} 321x_1 - 37x_2 > 810 \ \wedge \ -23x_1 + 51x_2 > 58$$

Let us go back to our running projection example of Example 3.4. On the figure, bold lines represent the space $[\![\widetilde{\boldsymbol{Z}}^\star(\boldsymbol{x}) = 0]\!]$. Our PLP-solver applied on the normalized PLOP only builds the irredundant constraints $\widetilde{\boldsymbol{Z}}_2^\star \geq 0$, $\widetilde{\boldsymbol{Z}}_3^\star \geq 0$ and $\widetilde{\boldsymbol{Z}}_4^\star \geq 0$ associated to the regions displayed on the figure above.

Note that we must be able to provide a point $\mathring{\boldsymbol{x}}$ in the interior of $\mathcal{P}^\star$ while $\mathcal{P}^\star$ is not already known. Finding such a point is obvious for projection, convex-hull and Minkowski sum. It is feasible because the operators based on PLP are applied on polyhedra with non-empty interior; the treatment of polyhedra with equalities is explained below. For projection, $\mathring{\boldsymbol{x}}$ is obtained from a point $\boldsymbol{x}$ in the interior of the input polyhedron $\mathcal{P}$. Removing the coordinates of variables marked for elimination provides a point $\mathring{\boldsymbol{x}}$ that will be in the interior of the projected polyhedron $\mathcal{P}^\star$.

Handling of Equalities. In the VPL, polyhedra are stored as a set of inequalities I and a set of equalities E such that I contains no equality, neither implicit nor explicit. The process of extracting implicit equalities from inequalities is detailed in Chapter 8. When eliminating a variable x_i that belongs to an equality e from E, say $e : \sum_j a_j x_j = b$, we rewrite e so that x_i is expressed in terms of the other variables: $x_i = b - \sum_{j \neq i} a_j x_j$. Then, we replace each occurency of x_i in I by $b - \sum_{j \neq i} a_j x_j$. Now, I does not talk anymore about x_i, and we simply remove equation e from E.

3.4.2 Projection via Normalized PLP is Free of Redundancy

The advantage of PLP over Fourier-Motzkin comes from the following theorem:

Theorem 3.2 *Let $\widetilde{Z}^\star \stackrel{\text{def}}{=} \min\{\widetilde{Z}_1^\star, \ldots, \widetilde{Z}_r^\star\}$ be the optimal solution of a normalized parametric minimization problem. Then each solution $\widetilde{Z}_k^\star$ that is not the constant function $\boldsymbol{x} \mapsto 1$ is irredundant w.r.t. polyhedron $[\![\widetilde{Z}^\star \geq 0]\!]$.*

Proof. Theorem 3.2 is a direct consequence of three intermediate results:

(1) Each region of optimality in a normalized PLOP is a cone pointed in $\mathring{\boldsymbol{x}}$ (Lemma 3.4);

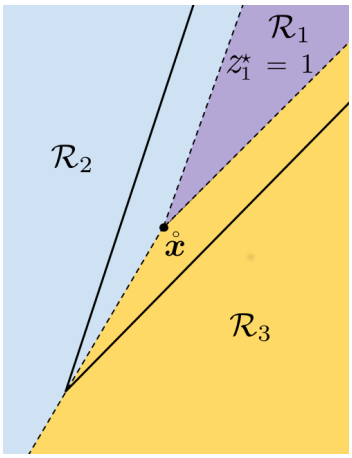
(2) Each piece $\widetilde{Z}_k^\star$ which is not constant, is decreasing on its region of optimality along half-lines starting at $\mathring{\boldsymbol{x}}$ (Lemma 3.5);

(3) Each piece $\widetilde{Z}_k^\star$ that crosses 0 on its region produces an irredundant constraint (Lemma 3.6).

□

Let us summarize the key facts that are needed for exposing the proof of the lemmata: projection via PLP leads to a parametric linear minimization problem whose solution is a function $\widetilde{Z}^\star$ defined by pieces $\{\widetilde{Z}_1^\star$ on $\mathcal{R}_1, \ldots, \widetilde{Z}_r^\star$ on $\mathcal{R}_r\}$; each $\mathcal{R}_k$ is the *region of optimality* of $\widetilde{Z}_k^\star$, meaning that among all the pieces $\widetilde{Z}_k^\star$ is the minimal one on $\mathcal{R}_k$, *i.e.* $\mathcal{R}_k = \{\boldsymbol{x} \mid \widetilde{Z}^\star(\boldsymbol{x}) = \widetilde{Z}_k^\star(\boldsymbol{x})\}$. By construction, $\widetilde{Z}^\star(\boldsymbol{x})$ is the minimum of $\{\widetilde{Z}_1^\star(\boldsymbol{x}), \ldots, \widetilde{Z}_r^\star(\boldsymbol{x})\}$ and $\widetilde{Z}^\star(\mathring{\boldsymbol{x}}) = \widetilde{Z}_1^\star(\mathring{\boldsymbol{x}}) = \ldots = \widetilde{Z}_r^\star(\mathring{\boldsymbol{x}}) = 1$ is enforced by the $(\ddagger)$-normalization constraint.

This is where λ_0 comes into play: the fact that $\boldsymbol{\lambda} = (1, 0, \ldots, 0)$ fulfills $(\ddagger)$ and (α), hence leading to the constant function $Z_{\boldsymbol{\lambda}}^\star = 1$, sets an upper-bound on $Z^\star$. The constant piece $Z_{\boldsymbol{\lambda}}^\star = 1$ arises among the solutions of a normalized PLOP when the resulting polyhedron $\mathcal{P}^\star$ is unbounded as illustrated alongside. Therefore, any minimal piece $\widetilde{Z}_k^\star$, which evaluates to 1 on $\mathring{\boldsymbol{x}}$, can not grow on its region of optimality otherwise it would not be minimal compared to $Z_{\boldsymbol{\lambda}}^\star = 1$. Thus, $\widetilde{Z}_k^\star$ is either constant and equal to 1 or it satisfies



$$\forall \boldsymbol{x} \in \mathring{\mathcal{R}}_k,\ \widetilde{Z}_k^\star(\boldsymbol{x}) < 1 \tag{3.7}$$

which entails its decline on the *infinite cone* $\mathring{\mathcal{R}}_k$ as meant by the forthcoming Lemma 3.5, causing its nullification in $\mathring{\mathcal{R}}_k$, hence its irredundancy (Lemma 3.6).

The proofs make an intensive usage of the following lemma.

Lemma 3.3 *For any affine form "af", any points $\mathring{\boldsymbol{x}}$ and $\boldsymbol{x}$ and any scalar μ,*

$$af(\mathring{\boldsymbol{x}} + \mu \times (\boldsymbol{x} - \mathring{\boldsymbol{x}})) = af(\mathring{\boldsymbol{x}}) + \mu \times af(\boldsymbol{x}) - \mu \times af(\mathring{\boldsymbol{x}})$$

Proof. An affine form af is a linear form f plus a constant a, that is $af(\boldsymbol{x}) = a + f(\boldsymbol{x})$. Then,

$$\begin{aligned}
af(\mathring{\boldsymbol{x}} + \mu \times (\boldsymbol{x} - \mathring{\boldsymbol{x}})) &= a + f(\mathring{\boldsymbol{x}} + \mu \times (\boldsymbol{x} - \mathring{\boldsymbol{x}})) \\
&= a + f(\mathring{\boldsymbol{x}}) + \mu \times f(\boldsymbol{x}) - \mu \times f(\mathring{\boldsymbol{x}}) \quad \text{because } f \text{ is linear} \\
&= a + f(\mathring{\boldsymbol{x}}) + \mu \times f(\boldsymbol{x}) + (\mu \times a - \mu \times a) - \mu \times f(\mathring{\boldsymbol{x}}) \\
&= (a + f(\mathring{\boldsymbol{x}})) + \mu \times (a + f(\boldsymbol{x})) - \mu \times (a + f(\mathring{\boldsymbol{x}})) \\
&= af(\mathring{\boldsymbol{x}}) + \mu \times af(\boldsymbol{x}) - \mu \times af(\mathring{\boldsymbol{x}})
\end{aligned}$$

□

Lemma 3.4 (Normalized Regions are cones pointed in $\mathring{\boldsymbol{x}}$)

$$\forall \boldsymbol{x} \in \mathbb{Q}^n,\ \boldsymbol{x} \in \mathring{\mathcal{R}}_i \Rightarrow \mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}}) \in \mathring{\mathcal{R}}_i,\ \forall \mu > 0.$$

Proof. Consider $\boldsymbol{x} \in \mathring{\mathcal{R}}_i$, $\mu > 0$ and let $\widetilde{\boldsymbol{Z}}_j^\star$ be the piece that is minimal at $\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})$, *i.e.*

$$\widetilde{\boldsymbol{Z}}_j^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})) = \widetilde{\boldsymbol{Z}}^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})) \stackrel{\text{def}}{=} \min_k \left\{ \widetilde{\boldsymbol{Z}}_k^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})) \right\} \tag{3.8}$$

Let us prove that $j \neq i$ leads to a contradiction. Since $\widetilde{\boldsymbol{Z}}_j^\star$ is affine, $\widetilde{\boldsymbol{Z}}_j^\star(\mathring{\boldsymbol{x}}+\mu(\boldsymbol{x}-\mathring{\boldsymbol{x}})) = \widetilde{\boldsymbol{Z}}_j^\star(\mathring{\boldsymbol{x}})+\mu\times \widetilde{\boldsymbol{Z}}_j^\star(\boldsymbol{x})-\mu\times\widetilde{\boldsymbol{Z}}_j^\star(\mathring{\boldsymbol{x}})$. And $\widetilde{\boldsymbol{Z}}_j^\star(\mathring{\boldsymbol{x}}) = 1$ by normalization. Thus, $\widetilde{\boldsymbol{Z}}_j^\star(\mathring{\boldsymbol{x}}+\mu(\boldsymbol{x}-\mathring{\boldsymbol{x}})) = 1-\mu+\mu\times\widetilde{\boldsymbol{Z}}_j^\star(\boldsymbol{x})$. The same reasoning leads to $\widetilde{\boldsymbol{Z}}_i^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})) = 1 - \mu + \mu \times \widetilde{\boldsymbol{Z}}_i^\star(\boldsymbol{x})$. Moreover, $\widetilde{\boldsymbol{Z}}_j^\star(\boldsymbol{x}) > \widetilde{\boldsymbol{Z}}_i^\star(\boldsymbol{x})$ as $\boldsymbol{x} \in \mathring{\mathcal{R}}_i$, $\widetilde{\boldsymbol{Z}}_i^\star$'s region of optimality. Then, $1 - \mu + \mu \times \widetilde{\boldsymbol{Z}}_j^\star(\boldsymbol{x}) > 1 - \mu + \mu \times \widetilde{\boldsymbol{Z}}_i^\star(\boldsymbol{x})$ as $\mu > 0$ and finally, $\widetilde{\boldsymbol{Z}}_j^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})) > \widetilde{\boldsymbol{Z}}_i^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}}))$ which contradicts (3.8). Thus, $j = i$ and $\widetilde{\boldsymbol{Z}}^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})) = \widetilde{\boldsymbol{Z}}_i^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}}))$ meaning that $\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}}) \in \mathcal{R}_i$.

It remains to be proven that $\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})$ cannot lie in a boundary of $\mathcal{R}_i$ and thus belongs to $\mathring{\mathcal{R}}_i$. Recall that, by construction, a boundary is the intersection of two adjacent regions, say $\mathcal{R}_i$ and $\mathcal{R}_j$ (with $i \neq j$), and their affine forms are equal on the boundary. This would mean $\boldsymbol{Z}_j^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})) = \widetilde{\boldsymbol{Z}}^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})) = \widetilde{\boldsymbol{Z}}_i^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}}))$. We already proved that it is unsatisfiable. □

Lemma 3.5 (Normalized solutions decrease along their region of optimality) *Either $\widetilde{\boldsymbol{Z}}_k^\star$ is the constant function $\boldsymbol{x} \mapsto 1$, or it decreases on lines of $\mathcal{R}_k$ starting at $\mathring{\boldsymbol{x}}$, i.e.*

$$\forall \boldsymbol{x} \in \mathring{\mathcal{R}}_k,\ \forall \mu > 1,\ \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x}) > \widetilde{\boldsymbol{Z}}_k^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})).$$

Proof. Assume $\boldsymbol{x} \in \mathring{\mathcal{R}}_k$. Let $\mu > 1$, then

$$\begin{array}{lrll}
& \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x}) &> \widetilde{\boldsymbol{Z}}_k^\star(\mathring{\boldsymbol{x}} + \mu(\boldsymbol{x} - \mathring{\boldsymbol{x}})) & \\
\Leftrightarrow & \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x}) &> \widetilde{\boldsymbol{Z}}_k^\star(\mathring{\boldsymbol{x}}) + \mu \times \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x}) - \mu \times \widetilde{\boldsymbol{Z}}_k^\star(\mathring{\boldsymbol{x}}) & \text{because } \widetilde{\boldsymbol{Z}}_k^\star \text{ is affine} \\
\Leftrightarrow & \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x}) &> 1 - \mu + \mu \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x}) & \text{because } \widetilde{\boldsymbol{Z}}_k^\star(\mathring{\boldsymbol{x}}) = 1 \\
\Leftrightarrow & 0 &> (1 - \mu) - \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x})(1 - \mu) & \\
\Leftrightarrow & 0 &> (1 - \mu)(1 - \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x})) & \\
\Leftrightarrow & 0 &< 1 - \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x}) & \text{because } (1 - \mu) < 0 \\
\Leftrightarrow & \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x}) &< 1 &
\end{array}$$

This last inequality holds as we already noticed in Equation (3.7) that if $\widetilde{\boldsymbol{Z}}_k^\star$ is not the constant function $\boldsymbol{x} \mapsto 1$, then $\widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x}) < 1,\ \forall \boldsymbol{x} \in \mathring{\mathcal{R}}_k$. □

Lemma 3.6 (Irredundancy of solutions that meet 0 on their region of optimality)

$$\left([\![\widetilde{\boldsymbol{Z}}_k^\star = 0]\!] \cap [\![\mathring{\mathcal{R}}_k]\!] \right) \neq \emptyset \Rightarrow \widetilde{\boldsymbol{Z}}_k^\star \geq 0 \text{ is irredundant w.r.t. } \widetilde{\boldsymbol{Z}}^\star \geq 0.$$

Proof by contradiction. Consider $\widetilde{\boldsymbol{Z}}_k^\star$, a piece of $\widetilde{\boldsymbol{Z}}^\star$ such that $[\![\widetilde{\boldsymbol{Z}}_k^\star = 0]\!] \cap [\![\mathring{\mathcal{R}}_k]\!] \neq \emptyset$. Let us assume that $\widetilde{\boldsymbol{Z}}_k^\star$ is redundant. Then, by Farkas' lemma, $\exists\, (\lambda_j)_{j \neq k} \geq 0,\ \forall \boldsymbol{x} \in \mathbb{Q}^n,\ \sum_{j \neq k} \lambda_j \widetilde{\boldsymbol{Z}}_j^\star(\boldsymbol{x}) \leq \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x})$. Let $\boldsymbol{x}$ be a point of the nonempty set $[\![\widetilde{\boldsymbol{Z}}_k^\star = 0]\!] \cap [\![\mathring{\mathcal{R}}_k]\!]$. Then $\widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x}) = 0$, as $\boldsymbol{x} \in [\![\widetilde{\boldsymbol{Z}}_k^\star = 0]\!]$, and the previous Farkas inequality becomes

$$\sum_{j \neq k} \lambda_j \widetilde{\boldsymbol{Z}}_j^\star(\boldsymbol{x}) \leq 0 \tag{3.9}$$

Since $\boldsymbol{x} \in \mathring{\mathcal{R}}_k$, then, $\widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x}) < \widetilde{\boldsymbol{Z}}_j^\star(\boldsymbol{x})$ for $j \neq k$ by definition of $\mathcal{R}_k$ as the region of optimality of $\widetilde{\boldsymbol{Z}}_k^\star$. More precisely, $0 < \widetilde{\boldsymbol{Z}}_j^\star(\boldsymbol{x})$ since $\boldsymbol{x} \in [\![\widetilde{\boldsymbol{Z}}_k^\star = 0]\!]$. Therefore, $0 < \lambda_j \widetilde{\boldsymbol{Z}}_j^\star(\boldsymbol{x})$ for $j \neq k$ as $\lambda_j \geq 0$. Then, summing up this inequality for all $j \neq k$, we obtain

$$0 < \sum_{j \neq k} \lambda_j \widetilde{\boldsymbol{Z}}_j^\star(\boldsymbol{x}) \tag{3.10}$$

Inequalities (3.9) and (3.10) are contradictory, proving thereby that $\widetilde{\boldsymbol{Z}}_k^\star$ is irredundant. □

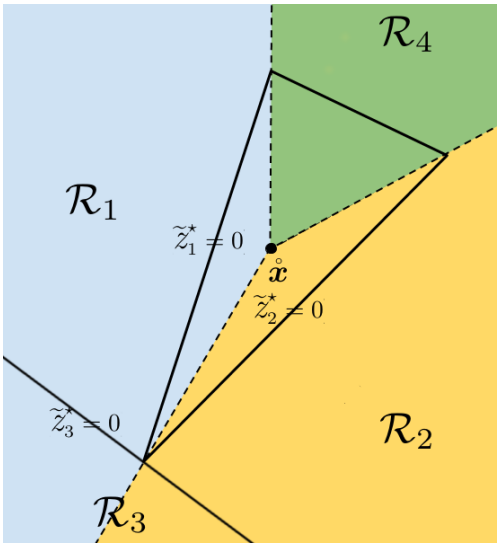


Figure 3.8 – $\mathcal{R}_3 \stackrel{\text{def}}{=} \mathcal{R}_1 \sqcap \mathcal{R}_2$ has an empty interior.

Regions with Empty Interior. They can appear as solutions of a PLOP. Yet, Lemmas 3.4, 3.5 and 3.6 speak only about regions' interior. This is because the reasoning we developed to prove that normalization ensures irredundancy is not valid on regions with empty interior (*i.e.* regions lying within the frontier of another region). For instance, consider two adjacent regions $\mathcal{R}_1$ and $\mathcal{R}_2$ with nonempty interiors. By continuity of the PLP solution, $\widetilde{\boldsymbol{Z}}_1^\star$ and $\widetilde{\boldsymbol{Z}}_2^\star$ coincide on $\mathcal{R}_1 \sqcap \mathcal{R}_2$, *i.e.* $\widetilde{\boldsymbol{Z}}_1^\star(\boldsymbol{x}) = \widetilde{\boldsymbol{Z}}_2^\star(\boldsymbol{x}),\ \forall \boldsymbol{x} \in \mathcal{R}_1 \sqcap \mathcal{R}_2$. Let us call $\mathcal{R}_3$ the region reduced to the frontier between $\mathcal{R}_1$ and $\mathcal{R}_2$, *i.e.* $\mathcal{R}_3 = \mathcal{R}_1 \sqcap \mathcal{R}_2$. We can prove that $\mathcal{R}_3$ has an empty interior (this comes from the regions forming a quasi-partition of the space). Moreover, $\widetilde{\boldsymbol{Z}}_3^\star(\boldsymbol{x}) = \widetilde{\boldsymbol{Z}}_1^\star(\boldsymbol{x}) = \widetilde{\boldsymbol{Z}}_2^\star(\boldsymbol{x}),\ \forall \boldsymbol{x} \in \mathcal{R}_3$. In particular, this equality holds where $\widetilde{\boldsymbol{Z}}_3^\star$ vanishes, hence

$$\begin{cases} \Big(\llbracket \widetilde{\boldsymbol{Z}}_3^\star = 0 \rrbracket \sqcap \llbracket \mathcal{R}_3 \rrbracket\Big) \sqsubset \Big(\llbracket \widetilde{\boldsymbol{Z}}_1^\star = 0 \rrbracket \sqcap \llbracket \mathcal{R}_1 \rrbracket\Big) \\ \wedge \\ \Big(\llbracket \widetilde{\boldsymbol{Z}}_3^\star = 0 \rrbracket \sqcap \llbracket \mathcal{R}_3 \rrbracket\Big) \sqsubset \Big(\llbracket \widetilde{\boldsymbol{Z}}_2^\star = 0 \rrbracket \sqcap \llbracket \mathcal{R}_2 \rrbracket\Big) \end{cases}$$

Thus, $\widetilde{\boldsymbol{Z}}_3^\star$ is redundant w.r.t. $\widetilde{\boldsymbol{Z}}_1^\star$ and $\widetilde{\boldsymbol{Z}}_2^\star$: $\widetilde{\boldsymbol{Z}}_3^\star$ could be either equal to $\widetilde{\boldsymbol{Z}}_1^\star$, $\widetilde{\boldsymbol{Z}}_2^\star$, or even a combination of them, as illustrated on Fig. 3.8.
Without further precautions, the PLP solver could unfortunately discover such undesirable regions. A simple trick avoids generating regions with empty interiors: the solver exclusively focuses on open regions, *i.e.* regions that are defined only by strict inequalities. The missing value $\widetilde{\boldsymbol{Z}}^\star(\boldsymbol{x})$ for a point $\boldsymbol{x}$ lying in a frontier $\mathcal{F}$ can then be retrieved easily by continuity: $\widetilde{\boldsymbol{Z}}^\star(\boldsymbol{x}) = \widetilde{\boldsymbol{Z}}_k^\star(\boldsymbol{x})$ where $\widetilde{\boldsymbol{Z}}_k^\star$ is the solution associated to a region adjacent to $\mathcal{F}$.

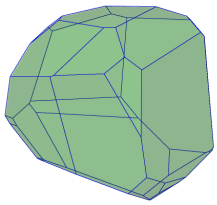
3.5 Experiments

Benchmarks. We reused the benchmark suite that we designed for minimization (§2.4). It contains polyhedra generated randomly from several characteristics: number of constraints, number of variables and density (ratio of the number of zero coefficients by the number of variables). Constraints are created by picking up a random integer between -100 and 100 as coefficient of each variable. All constraints are attached the same constant bound ≤ 20. These polyhedra have a potatoid shape, as shown on the right-hand side figure.

We compare three libraries on projection/minimization problems: NEWPOLKA (Jeannet and Miné, 2009) as representative of the double description framework, VPL 0.1 based on

Fourier-Motzkin elimination (Fouilhé, 2015), and our implementation based on PLP. As we produce polyhedra in minimized form, we asked NEWPOLKA and VPL to perform a minimization afterwards.

We did not consider the Parma Polyhedra Library (PPL) in our experiments because, as far as we know, PPL and NEWPOLKA rely on the same underlying algorithms, and both are based on the double description framework. Our objective is not to be faster than these libraries for the projection operator; it is rather to improve the scalability of the constraints-only representation. Thus, the most relevant comparison lies between the PLP-based projection and Fourier-Motzkin elimination. The comparison with NEWPOLKA is done to give a general picture.



In addition to the number of constraints C, the density D and the number of variables V, we consider the effect of the projection ratio P (number of projected variable over dimension, which is the total number of variables). Fig. 3.9 shows the effect of these characteristics on execution time (in seconds). The vertical axis is always displayed in log scale, for readability. Each point is the average execution time for the projection and minimization of 100 polyhedra sharing the same characteristics. On each problem we measure the execution time, with a timeout fixed at 300 seconds. We tuned the timeout value to let NEWPOLKA and PLP provide a result, excluding only Fourier-Motzkin on too large computations.

Fourier-Motzkin Elimination in the VPL. Projection in VPL 0.1 is done by Fourier-Motzkin elimination that can only remove variables one by one. As mentioned earlier, this algorithm generates many redundant constraints and the challenge of a good implementation is their fast removal. Fouilhé's implementation in the VPL 0.1 uses well-known tricks for removing constraints that can be shown redundant by syntactic arguments (see §2.1). When syntactic criteria fail to decide the redundancy of a constraint, the VPL calls a LP solver. However, as Fouilhé (2015, 3.2.3, p. 76) shown, removing redundancies between the successive elimination of two variables forbids the use of Kohler's criterion, which says that when eliminating k variables, a constraint resulting from the combination of $k+2$ constraints is redundant.

Projection Ratio. Fig. 3.9(a) gives the time measurements when projecting polyhedra of 15 constraints, 10 variables and a density of 50%, with a projection ratio varying from 10 to 90%. Fourier-Motzkin is very efficient when projecting a small number of variables. Its exponential behavior mainly occurs for high projection ratio, as it eliminates variables one after the other and the number of faces tends to grow at each projection. PLP is not suitable when there are only few variables to project, *e.g.* in the case of a single assignment. On the contrary, it becomes interesting compared to Fourier-Motzkin elimination when the projection ratio exceeds 50%, *i.e.* when projecting more than half of the variables. This ratio is always reached when computing Minkowski sums or convex hulls by projection (§3.6). It can also be the case on exits of program blocks where a whole set of local variables must be forgotten. As PLP usefulness grows with a high projection ratio we will focus on the case $P = 75\%$, studying the effect of other characteristics.

Number of Constraints. Fig. 3.9(b) shows the time measurements when projecting polyhedra with 8 variables, a density of 50% and a projection ratio of 75% (*i.e.* elimination of 6 variables). The number of constraints varies in $[8, 60]$. While Fourier-Motzkin blows up when reaching 15 constraints, PLP and NEWPOLKA scale better and the curves shows that PLP wins when the number of constraints exceeds 35.

Dimension. The evolution of execution time in terms of dimension is given in Fig. 3.9(c). With 20 constraints, the exponential behavior of Fourier-Motzkin elimination emerges. PLP and NEWPOLKA show similar curves with an overhead for PLP on a log scale, *i.e.* a proportionality factor on execution time. It would be interesting to see the effect of dimension

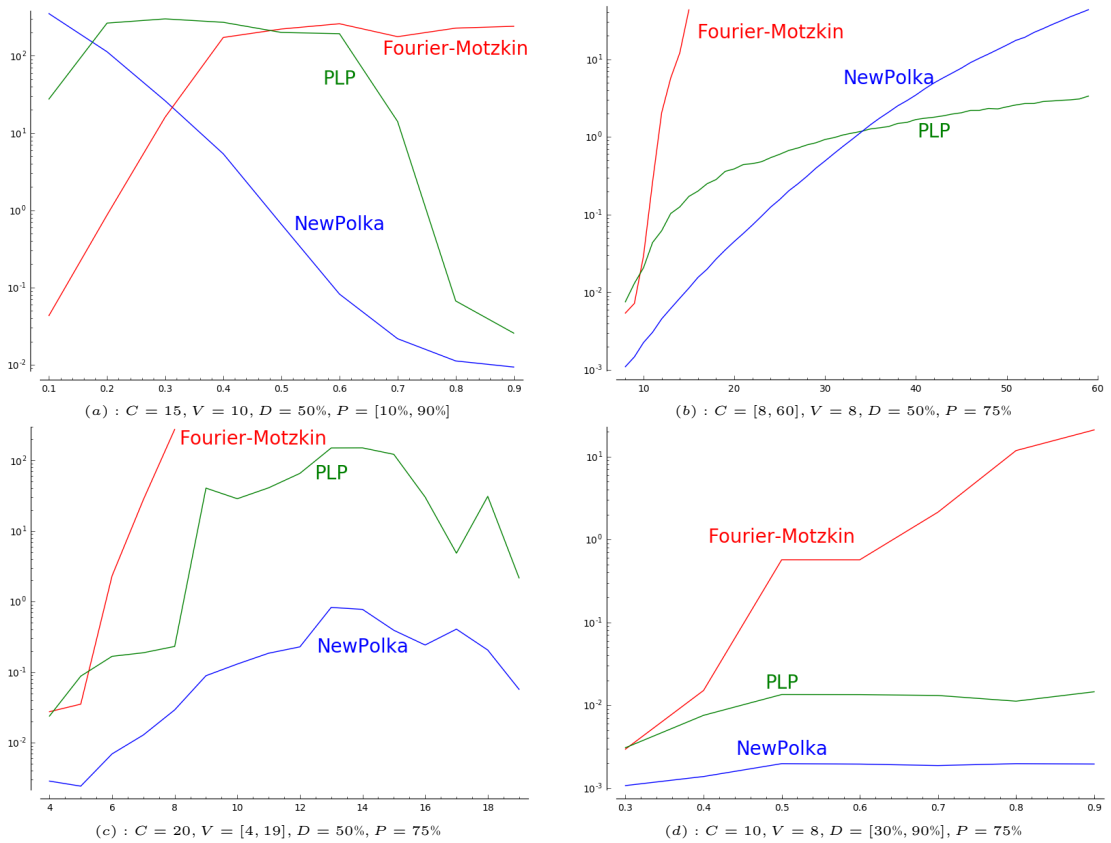


Figure 3.9 – Execution time in seconds of NEWPOLKA (blue), Fourier-Motzkin (red) and PLP (green) depending on respectively (a) projection ratio, (b) number of constraints, (c) number of variables and (d) density.

beyond 20 variables, which takes considerable time since it requires increasing the number of constraints. Indeed, when the dimension is greater than the number of constraints, polyhedra have a really special shape with very few generators and the comparison would be distorted.

Density. The effect of density on execution time is shown on Fig. 3.9(d). NEWPOLKA and PLP are little sensitive to density. The case of Fourier-Motzkin can be explained: Elimination of a variable x with Fourier-Motzkin consists in combining every pair of constraints having opposite signs for x. The more non-zero coefficients within the constraints, the greater the number of possible combinations.

What can we conclude from these experiments? On small problems, our projection is less efficient than that of a double description (DD) library but the shape of the curves of NEWPOLKA and PLP is similar on a logarithmic scale, meaning that there is a proportionality factor between the two algorithms. This is an encouraging result as projection – and the operators encoded as projection – are the Achilles heel of constraints-only representation (exponential complexity with Fourier-Motzkin elimination) whereas it is linear in the number of generators in DD. On the other hand, the conjunction operator, which, in constraints-only representation, consists in the union of two sets followed by a minimization, is less efficient in DD because it triggers one step of Chernikova's algorithm per constraint.

Extracting Intervals from Polyhedra. Let us highlight an interesting property of projection by PLP. Static analyzers often ask for the interval in which lies a variable x_i to check for instance if it exceeds the bounds induced by its type (*i.e.* if it wraps). Even if the expected

result is the range of a variable, polyhedra are sometimes needed to produce a much more precise interval than other abstract domains. When the generators are known, obtaining the range of a variable x_i in a polytope consists in computing the minimum and maximum of the i^{th} component among the vertices. In constraints-only, it requires solving two LP problems:

$$\begin{array}{ll} \textbf{minimize } \mathbf{Z} \stackrel{\text{def}}{=} x_i & \textbf{minimize } \mathbf{Z} \stackrel{\text{def}}{=} -x_i \\ \quad \textbf{subject to } (x_1, \ldots, x_n) \in \mathcal{P} & \quad \textbf{subject to } (x_1, \ldots, x_n) \in \mathcal{P} \end{array}$$

(LP 3.11) (LP 3.12)

Problem (LP 3.11) (resp. (LP 3.12)) returns the minimum (resp. maximum) value reachable by x_i in $\mathcal{P}$. These two bounds give the interval for x_i.
Another way to obtain the interval of variable would be to project every other variables. With Fourier-Motzkin elimination, this would clearly be a bad idea, since this algorithm has an exponential complexity in the number of eliminated variables. In contrast, it is worth noticing that the projection by PLP solves exactly the two standard LP problems. The reason is simple: when eliminating $n-1$ variables from a polyhedron with n variables, the resulting polyhedron has exactly one dimension. The solution of such a PLOP has at most two regions, because there is no way to split a line into three parts that share a point (the normalization point $\mathring{\boldsymbol{x}}$). Hence, the PLP solver only explores two regions.

3.6 Convex Hull via Parametric Linear Programming

3.6.1 Encoding the Convex Hull as a Projection

In constraints-only, the standard way of computing a convex hull is to express points of $\mathcal{P}' \sqcup \mathcal{P}''$ as convex combinations of points of $\mathcal{P}'$ and $\mathcal{P}''$, *i.e.*

$$[\![\mathcal{P}' \sqcup \mathcal{P}'']\!] = \{\boldsymbol{x} \mid \boldsymbol{x}' \in \mathcal{P}',\ \boldsymbol{x}'' \in \mathcal{P}'',\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \alpha' \cdot \boldsymbol{x}' + \alpha'' \cdot \boldsymbol{x}''\}$$

By rewriting the belonging of a point in a polyhedron with matrix notations, we get

$$\{\boldsymbol{x} \mid \boldsymbol{A}'\boldsymbol{x}' \geq \boldsymbol{b}',\ \boldsymbol{A}''\boldsymbol{x}'' \geq \boldsymbol{b}'',\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \alpha' \cdot \boldsymbol{x}' + \alpha'' \cdot \boldsymbol{x}''\} \tag{3.13}$$

where $\mathcal{P}' \stackrel{\text{def}}{=} \boldsymbol{A}'\boldsymbol{x}' \geq \boldsymbol{b}'$ and $\mathcal{P}'' \stackrel{\text{def}}{=} \boldsymbol{A}''\boldsymbol{x}'' \geq \boldsymbol{b}''$. Then, eliminating variables α', α'', $\boldsymbol{x}'$ and $\boldsymbol{x}''$ from this set leads to $\mathcal{P}' \sqcup \mathcal{P}''$. But we cannot use directly the projection operator because the set of points (3.13) is defined with a nonlinear constraint $\boldsymbol{x} = \alpha' \cdot \boldsymbol{x}' + \alpha'' \cdot \boldsymbol{x}''$. To overcome this issue, we apply the changes of variable $\boldsymbol{y}' := \alpha' \cdot \boldsymbol{x}'$ and $\boldsymbol{y}'' := \alpha'' \cdot \boldsymbol{x}''$. By multiplying matrix $\boldsymbol{A}'\boldsymbol{x}' \geq \boldsymbol{b}'$ by α' and $\boldsymbol{A}''\boldsymbol{x}'' \geq \boldsymbol{b}''$ by α'', we obtain equivalent systems $\boldsymbol{A}'\boldsymbol{y}' \geq \alpha' \cdot \boldsymbol{b}'$ and $\boldsymbol{A}''\boldsymbol{y}'' \geq \alpha'' \cdot \boldsymbol{b}''$. The set of points (3.13) is now described as

$$\{\boldsymbol{x} \mid \boldsymbol{A}'\boldsymbol{y}' \geq \alpha' \cdot \boldsymbol{b}',\ \boldsymbol{A}''\boldsymbol{y}'' \geq \alpha'' \cdot \boldsymbol{b}'',\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \boldsymbol{y}' + \boldsymbol{y}''\}$$

We can exploit $\boldsymbol{x} = \boldsymbol{y}' + \boldsymbol{y}''$ to replace $\boldsymbol{y}''$ by $\boldsymbol{x} - \boldsymbol{y}'$. Similarly, equation $\alpha' + \alpha'' = 1$ allows replacing α'' with $1 - \alpha'$. Finally, the convex hull is obtained by eliminating variables $\boldsymbol{y}'$ and α' from the following polyhedron.

$$\{\boldsymbol{x} \mid \boldsymbol{A}'\boldsymbol{y}' \geq \alpha' \cdot \boldsymbol{b}',\ \boldsymbol{A}''(\boldsymbol{x} - \boldsymbol{y}') \geq (1 - \alpha') \cdot \boldsymbol{b}'',\ 1 \geq \alpha' \geq 0\} \tag{3.14}$$

This encoding of convex hull as a projection is due to Benoy et al. (2005). The convex hull operator was implemented that way in VPL 0.1, see (Fouilhé, 2015) for more details.

Once equipped with a PLP solver, we investigated a direct encoding of the convex hull as a PLOP defining the tightest polyhedron $\mathcal{P}$ such that $\mathcal{P}' \sqsubseteq \mathcal{P} \ \wedge \ \mathcal{P}'' \sqsubseteq \mathcal{P}$. For now, we focus on polyhedra with nonempty interior, meaning that they contain no equalities, neither implicit nor explicit. The handling of equalities requires one single extra variable in the PLOP encoding, and is addressed later.

3.6.2 Convex Hull of Two Polyhedra with Nonempty Interior

Consider two polyhedra with nonempty interior $\mathcal{P}' : \boldsymbol{A'x} \geq \boldsymbol{b'}$ and $\mathcal{P}'' : \boldsymbol{A''x} \geq \boldsymbol{b''}$ formed of respectively p and p' constraints. We denote by $\mathcal{P} : \boldsymbol{Ax} \geq \boldsymbol{b}$ the polyhedron equal to $\mathcal{P}' \sqcup \mathcal{P}''$. By Farkas' lemma, the condition $\mathcal{P}' \sqsubseteq \mathcal{P}$ (resp. $\mathcal{P}'' \sqsubseteq \mathcal{P}$) enforces each constraint of $\mathcal{P}$ to be a nonnegative affine combination of constraints of $\mathcal{P}'$ (resp. $\mathcal{P}''$). Henceforth, for each constraint $\boldsymbol{a_i x} \geq b_i$ of $\mathcal{P}$, the following equations must be satisfied

$$\begin{cases} \exists \lambda'_0, \ldots, \lambda'_p \in \mathbb{Q}+, \ \lambda'_0 + \sum_{k=1}^{p'} \lambda'_k (\boldsymbol{a'_k} - b'_k) = \boldsymbol{a_i} - b_i \\ \wedge \\ \exists \lambda''_0, \ldots, \lambda''_p \in \mathbb{Q}+, \ \lambda''_0 + \sum_{k=1}^{p''} \lambda''_k (\boldsymbol{a''_k} - b''_k) = \boldsymbol{a_i} - b_i \end{cases}$$

Exploiting the equalities and separating the linear parts ($\boldsymbol{a_i}$) from the constants (b_i), it can be reformulated using matrix notations as

$$\exists \boldsymbol{\lambda'} = (\lambda'_1, \ldots, \lambda'_{p'}) \in \mathbb{Q}+_{1 \times p'}, \ \boldsymbol{\lambda''} = (\lambda''_1, \ldots, \lambda''_{p''}) \in \mathbb{Q}+_{1 \times p''}, \ \exists \lambda'_0, \lambda''_0 \in \mathbb{Q}+,$$

$$\boldsymbol{\lambda' A'} = \boldsymbol{a}_i = \boldsymbol{\lambda'' A''} \tag{3.15}$$

$$\lambda'_0 + \langle \boldsymbol{\lambda'}, \boldsymbol{b'} \rangle = b_i = \langle \boldsymbol{\lambda''}, \boldsymbol{b''} \rangle + \lambda''_0 \tag{3.16}$$

Equation (3.15) decomposes into n equations, each one focusing on the coefficient $a_{i,j}$ of variable x_j:

$$\forall j \in \{1, \ldots, n\}, \ \sum_{k=1}^{k=p'} \lambda'_k a'_{kj} = a_{ij} = \sum_{k=1}^{k=p''} \lambda''_k a''_{kj}$$

Equation (3.16) ensures that the Farkas combination in terms of $\mathcal{P}'$ has the same constant term as the combination in terms of $\mathcal{P}''$.

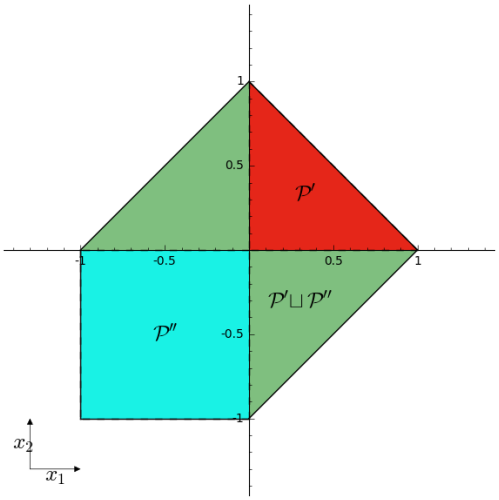
Encoding the Convex Hull as a Parametric Problem. Computing the convex hull comes down to calling a PLP-solver to enumerate the tightest constraints satisfying equations (3.15) and (3.16). As we did for projection, we set as objective function to minimize a bilinear affine form on coefficients of a Farkas combination and on parameters which define constraints of $\mathcal{P}$. Here, there are two input polyhedra $\mathcal{P}'$ and $\mathcal{P}''$, and the result must be an overapproximation of both. We choose arbitrarily to express the objective as a combination of constraints of $\mathcal{P}'$. Equations (3.15) and (3.16) will ensure that it will also be a combination of constraints of $\mathcal{P}''$. The objective function is thus $\lambda'_0 + \sum_{k=1}^{p'} \lambda'_k (\boldsymbol{a'_k}(\boldsymbol{x}) - b'_k)$, where $\boldsymbol{a'_k x} \geq b'_k$ are the constraints of $\mathcal{P}'$.

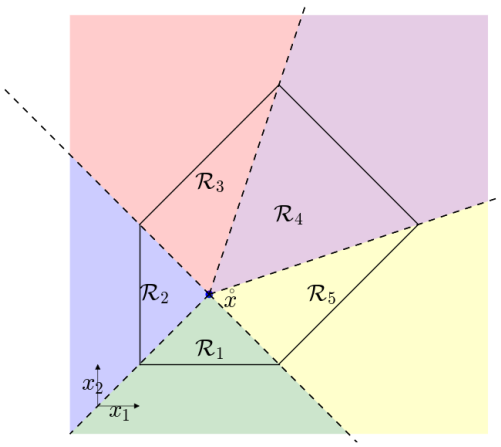
The PLOP encoding needs to be normalized in order to obtain a result free of redundancy. We apply the same normalization principle used for projection: we add constraint $\lambda'_0 + \sum_{k=1}^{p'} \lambda'_k (\boldsymbol{a'_k}(\mathring{\boldsymbol{x}}) - b'_k) = 1$, which specifies that the objective function must evaluate to 1 on a point $\mathring{\boldsymbol{x}}$ lying within $\mathring{\mathcal{P}}$, as explained in §3.4. Since the considered polyhedra have nonempty interiors, $\mathring{\boldsymbol{x}}$ can be chosen as any point of $[\![\mathring{\mathcal{P}'}]\!] \cup [\![\mathring{\mathcal{P}''}]\!]$. If a point in the interior of $\mathcal{P}'$ or $\mathcal{P}''$ is already known, we reuse it. Here is the PLOP encoding the convex hull.

$$
\begin{aligned}
\textbf{minimize } \mathbf{Z}(\boldsymbol{x}) \stackrel{\text{def}}{=}\ & \lambda'_0 + \sum_{k=1}^{p'} \lambda'_k (\boldsymbol{a}'_k(\boldsymbol{x}) - b'_k) \\
\textbf{subject to} & \\
(1)\quad & \boldsymbol{\lambda' A'} = \boldsymbol{\lambda'' A''} \\
(2)\quad & \lambda'_0 + \langle \boldsymbol{\lambda'}, \boldsymbol{b'} \rangle = \langle \boldsymbol{\lambda''}, \boldsymbol{b''} \rangle + \lambda''_0 \\
(\ddagger)\quad & \lambda'_0 + \sum_{k=1}^{p'} \lambda'_k (\boldsymbol{a}'_i(\mathring{\boldsymbol{x}}) - b'_i) = 1 \\
& \lambda'_k \geq 0,\ \forall k = 0, \ldots, p' \\
& \lambda''_k \geq 0,\ \forall k = 0, \ldots, p''
\end{aligned}
\qquad \text{(PLOP 3.17)}
$$

where the λ'_k's, λ''_k's, λ'_0 and λ''_0 are the $(p' + p'' + 2)$ decision variables of the PLOP; $\boldsymbol{x} \stackrel{\text{def}}{=} (x_1, \ldots, x_n)$ are the parameters.

Example 3.10





Let us apply the PLOP encoding of the convex hull on polyhedra $\mathcal{P}' : \{x_1 \geq 0,\ x_2 \geq 0,\ -x_1 - x_2 \geq -1\}$ and $\mathcal{P}'' : \{0 \geq x_1 \geq -1,\ 0 \geq x_2 \geq -1\}$. The associated matrix systems are

$$
\mathcal{P}' : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \quad \text{and} \quad \mathcal{P}'' : \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \geq \begin{pmatrix} -1 \\ 0 \\ -1 \\ 0 \end{pmatrix}
$$

The chosen normalization point $\mathring{\boldsymbol{x}}$ is $(-\frac{1}{2}, -\frac{1}{2})$. The encoding applied on this example is:

$$
\begin{array}{l}
\textbf{minimize } \mathbf{Z}(\boldsymbol{x}) \stackrel{\text{def}}{=} \lambda'_0 + \lambda'_1 x_1 + \lambda'_2 x_2 + \lambda'_3(-x_1 - x_2 + 1) \\
\quad \textbf{subject to} \\
\qquad (1)\ \ \lambda'_1 - \lambda'_3 = \lambda''_1 - \lambda''_2 \\
\qquad (1')\ \ \lambda'_2 - \lambda'_3 = \lambda''_3 - \lambda''_4 \\
\qquad (2)\ \ \lambda'_0 - \lambda'_3 = -\lambda''_1 - \lambda''_3 + \lambda''_0 \\
\qquad (\ddagger)\ \ \lambda'_0 - \frac{1}{2}\lambda'_1 - \frac{1}{2}\lambda'_2 = 1 \\
\qquad \lambda'_i \geq 0,\ \forall i \in \{0,\dots,3\} \\
\qquad \lambda''_i \geq 0,\ \forall i \in \{0,\dots,4\}
\end{array}
\qquad \text{(PLOP 3.18)}
$$

Equation (1) and (1') refer to Equation (1) of (PLOP 3.17). Indeed,

$$
\begin{array}{lccccc}
 & \boldsymbol{\lambda'} & & \boldsymbol{A'} & = & \boldsymbol{\lambda''} \qquad\qquad \boldsymbol{A''} \\
\equiv & \begin{pmatrix} \lambda'_1 & \lambda'_2 & \lambda'_3 \end{pmatrix} & \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix} & = & \begin{pmatrix} \lambda''_1 & \lambda''_2 & \lambda''_3 & \lambda''_4 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \\
\equiv & & \begin{cases} \lambda'_1 - \lambda'_3 & = & \lambda''_1 - \lambda''_2 \\ \lambda'_2 - \lambda'_3 & = & \lambda''_3 - \lambda''_4 \end{cases} &
\end{array}
$$

Similarly, Equation (2) refers to Equation (2) of (PLOP 3.17):

$$
\begin{array}{lccccc}
 & \lambda'_0 + & \langle \boldsymbol{\lambda'}, \boldsymbol{b'} \rangle & = & \langle \boldsymbol{\lambda''}, \boldsymbol{b''} \rangle & + \lambda''_0 \\
\equiv & \lambda'_0 + \begin{pmatrix} \lambda'_1 & \lambda'_2 & \lambda'_3 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} & = & \begin{pmatrix} \lambda''_1 & \lambda''_2 & \lambda''_3 & \lambda''_4 \end{pmatrix} \begin{pmatrix} -1 \\ 0 \\ -1 \\ 0 \end{pmatrix} & + \lambda''_0 \\
\equiv & \lambda'_0 - \lambda'_3 & = & -\lambda''_1 - \lambda''_3 & + \lambda''_0
\end{array}
$$

Finally, Equation (‡) is the normalization constraint applied on $\mathring{\boldsymbol{x}} = \left(-\frac{1}{2}, -\frac{1}{2}\right)$.

The PLP-solver returns the regions $\mathcal{R}_1$ to $\mathcal{R}_5$ given in the right hand side figure. The associated set $\widetilde{\boldsymbol{Z}}^\star \geq 0$ is the expected polyhedron $\{\, x_1 \geq -1,\ x_2 \geq -1,\ x_1 + x_2 \leq 1,\ x_2 - x_1 \leq 1,\ x_1 - x_2 \leq 1\}$.

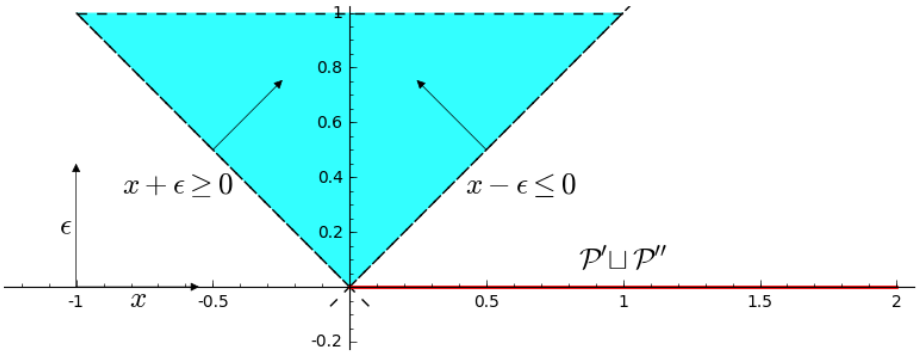
3.6.3 Convex Hull of General Polyhedra

Now, let us extend the previous encoding to the convex hull of two general polyhedra. When one of the two polyhedra contains an equality, their convex hull may have an empty interior. In such a case, it is not possible to choose a point $\mathring{x}$ within the resulting polyhedron to compute the normalization constraint, needed in the PLOP. To solve this issue, we add a variable ϵ to "simulate thickness" in the missing dimension. Each equality $\boldsymbol{a}'_i(\boldsymbol{x}) = b'_i$ of $\mathcal{P}'$ is replaced with the two inequalities $\boldsymbol{a}'_i(\boldsymbol{x}) - \epsilon \leq b'_i \wedge \boldsymbol{a}'_i(\boldsymbol{x}) + \epsilon \geq b'_i$. In the same way, each equality $\boldsymbol{a}''_i(\boldsymbol{x}) = b''_i$ of $\mathcal{P}''$ is replaced with $\boldsymbol{a}''_i(\boldsymbol{x}) - \epsilon \leq b''_i \wedge \boldsymbol{a}''_i(\boldsymbol{x}) + \epsilon \geq b''_i$. Thanks to this trick, it becomes possible to find $\mathring{x}$, build the normalization constraint, and solve the PLOP. The resulting constraints will contain occurrences of ϵ that we discard by setting ϵ to 0.

Updating the PLOP encoding. Adding this additional dimension forces to slightly adapt the PLOP encoding: ϵ is not treated as a standard parameter. In particular, ϵ should not be constrained by Equation (1) of (PLOP 3.17), which states that the coefficient of each parameter x_i should be equal in the Farkas combination of constraints of $\mathcal{P}'$ and $\mathcal{P}''$. As ϵ will be 0 at the end, this constraint would be useless. Even worse, it leads to imprecise results, as

illustrated on the example below.

Example 3.11



Consider the one-dimensional polyhedra $\mathcal{P}' : \{x = 0\}$ and $\mathcal{P}'' : \{x \geq 1\}$. As said above, equality $x = 0$ is rewritten into $\{x - \epsilon \leq 0,\ x + \epsilon \geq 0\}$. This allows to determine a point $\mathring{x}$ in the interior of $[\![\mathcal{P}']\!] \cup [\![\mathcal{P}'']\!]$, for instance we consider $\mathring{x} = (x \leftarrow 0,\ \epsilon \leftarrow 1)$ in the interior of $\mathcal{P}'$. Following encoding (PLOP 3.17), and considering ϵ as any variable, we obtain these constraints:

$$\begin{aligned}
&\textbf{minimize } \mathbf{Z}(\boldsymbol{x}) \stackrel{\text{def}}{=} \lambda'_0 + \lambda'_1(\epsilon - x) + \lambda'_2(x + \epsilon)\\
&\quad\textbf{subject to}\\
&\qquad (1) - \lambda'_1 + \lambda'_2 = \lambda''_1\\
&\qquad (1')\lambda'_1 + \lambda'_2 = 0\\
&\qquad (2)\lambda'_0 = \lambda''_0 - \lambda''_1\\
&\qquad (\ddagger)\lambda'_0 + \lambda'_1 + \lambda'_2 = 1\\
&\qquad\quad \lambda'_i \geq 0,\ \forall i \in \{0, \ldots, 2\}\\
&\qquad\quad \lambda''_i \geq 0,\ \forall i \in \{0, 1\}
\end{aligned}$$

Giving this problem to the PLOP solver, the result will be $\top$, instead of the true convex hull $\{x \geq 0\}$. Looking at the constraints of $\mathcal{P}'$, the only point $\boldsymbol{\lambda}$ leading to $x \geq 0$ is $(\lambda'_0 \leftarrow 0,\ \lambda'_1 \leftarrow 0,\ \lambda'_2 \leftarrow 1)$. The corresponding solution is $x + \epsilon \geq 0$, which is equivalent to $x \geq 0$ when $\epsilon = 0$. However, this point $\boldsymbol{\lambda}$ is unreachable because of constraint $(1')$, which states that the coefficient of ϵ in the combination of $\mathcal{P}'$ (here $\lambda'_1 + \lambda'_2$) must be equal to its coefficient in the combination of $\mathcal{P}''$ (here 0). Removing this constraint from the encoding leads to the correct result.

Projection Versus PLP. We believe that the encoding of convex hull as a PLOP is somehow equivalent to that of Benoy et al. (2005) by projection. Still, our encoding by PLP has the advantage of being more direct and leads to a simpler implementation. Moreover, the inclusion certificates are obtained directly from the decision variables $\boldsymbol{\lambda'}$ and $\boldsymbol{\lambda''}$, while Fouilhé (2015) had to deal with tedious rewritings to retrieve them from the projection result.

3.7 Conclusion on PLP-Based Operators

We have shown how the projection operator can be formulated as a PLOP instance. This approach was made practical by the combination of an efficient PLP solver and a normalization constraint ensuring that the solutions of the PLOP are free of redundancy. Our experiments show that projection via PLP scales better when projecting lots of variables, which happens in particular during the computation of a convex hull thanks to Benoy et al. (2005)'s encoding.

This makes the VPL competitive with other libraries in double description, and much faster on problems that have exponential generator representations.

Fourier-Motzkin elimination stays useful for projecting a small number of variables, *e.g.* in an assignment computation. Ideally, the projection operator should provide heuristics to choose between Fourier-Motzkin and PLP, depending on the size of input polyhedra and the number of variables to eliminate.

More generally, we explained how the solution of a PLOP can be seen as a polyhedron by considering an objective function that corresponds to a Farkas combination of some input constraints. It means that other operators can be expressed in that way; we illustrated this claim by encoding the convex hull operator as such. In the following chapter, we will see a PLOP encoding for computing approximations of semialgebraic sets, that can therefore be used as a linearization operator.

Chapter 4

Linearization

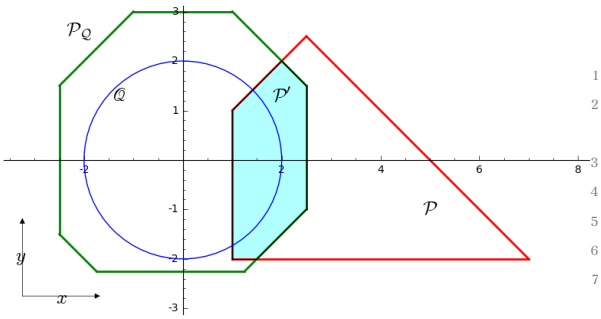
4.1 Polyhedral Approximation of Polynomials

The principle of linearization is to approximate nonlinear relations with linear ones. As for other operators of the abstract domain, the approximation is sound if it contains the original nonlinear set, meaning that a linearization operator must *over-approximates* the nonlinear set. In this work, we consider polynomial expressions formed of $(+, -, \times)$, such as $4-x\times x-y\times y$. More general algebraic expressions, including divisions and root operators, may be reduced to that format; for instance $y=\sqrt{x^2+1}$ is equivalent to $y^2=x^2+1 \wedge y \geq 0$ (Néron, 2013).

In this chapter, symbol Q shall represent a polynomial expression on the variables $x_1, .., x_n$ of a program. We focus on the linearization of inequalities, since equalities and negations can be changed into disjunctions of conjunctions of inequalities, as for affine constraints. For example $\neg(Q_1 = Q_2) \equiv (Q_1 < Q_2 \vee Q_1 > Q_2)$.

Nonlinear expressions occur for instance when addressing computations over matrices, computational geometry, automatic control, distance computations and in programs that approximate transcendental functions (sin, cos, log, etc.) by polynomials (Chevillard et al., 2009, 2010).

Example 4.1




```
1 int x,y,z;
2 if (x ≥ 1 && y ≥ −2 && x ≥ y && x ≤ 5 − y
    ){
3   if(x ∗ x + y ∗ y ≤ 4)
4     z = y * x;
5   else
6     z = 0;
7 }
```

Consider the C code fragment above. The first guard defines the polyhedron $\mathcal{P} = \{x \geq 1,\ y \geq -2,\ x - y \geq 0,\ x + y \leq 5\}$. The disc $\mathcal{Q} = \{(x, y) \mid x^2 + y^2 \leq 4\}$ corresponds to the second guard. The octagon $\mathcal{P}_Q$ that appears on the figure is a polyhedral over-approximation of $\mathcal{Q}$. The polyhedron $\mathcal{P}' \stackrel{\text{def}}{=} \mathcal{P} \sqcap \mathcal{P}_Q$ is a polyhedral over-approximation of $\mathcal{P} \cap \mathcal{Q}$.

To deal with such programs, we need to extend the guard operator of the abstract domain to semialgebraic sets. A *semialgebraic set*, abbreviated as SAS, has the following form

$$\left\{ \boldsymbol{x} \in \mathbb{Q}^n \mid \bigwedge_{i=1}^{q} Q_i(\boldsymbol{x}) \bowtie_i 0 \right\}$$

where $Q_i \in \mathbb{Q}[X]$ and $\bowtie_i \in \{\geq, >\}$. We can see the similitude between a polyhedron and an SAS: affine constraints have been replaced by polynomials. A polyhedron is thus a particular SAS. Actually, many properties of polyhedra extend to SAS. For instance, a finite intersection of SAS is an SAS, and SAS are closed under projection onto linear subspaces. SAS benefit from more properties: they are closed under finite union, and under complement operator.

As said in §1.2, the effect of an assignment $x := e$ on a polyhedron $\mathcal{P}$ can be reduced to a guard using this encoding.

$$\Big((\mathcal{P} \sqcap \{\tilde{x} = e\})_{\setminus \{x\}} \Big) [\tilde{x}/x].$$

where $\tilde{x}$ is a fresh variable. When e is nonlinear, we split the equality $\tilde{x} = e$ into a guard $\tilde{x} \leq e \wedge \tilde{x} \geq e$, hence we can exclusively focus on the linear over-approximation of polynomial guards. Then, the assignment encoding will produce a valid over-approximation.

The effect of a nonlinear guard $Q \geq 0$ on a polyhedron $\mathcal{P}$ consists in the intersection of the set of points of $\mathcal{P}$ with $\mathscr{Q} \stackrel{\text{def}}{=} \{\boldsymbol{x} \in \mathbb{Q}^n \mid Q(\boldsymbol{x}) \geq 0\}$. We approximate $\mathcal{P} \cap \mathscr{Q}$ by a polyhedron $\mathcal{P}'$ such that $\mathcal{P} \cap \mathscr{Q} \subseteq \mathcal{P}'$. Computing, instead, a polyhedral enclosure $\mathcal{P}_Q$ of the set $\mathscr{Q}$ would not be a practical solution: it can be very imprecise, *e.g.* if $\mathscr{Q} = \{(x, y) \,|\, y \leq x^2\}$, then $\mathcal{P}_Q = \mathbb{Q}^2$. Moreover, it is superfluous work: only three of the eight constraints of polyhedron $\mathcal{P}_Q$ on the figure of Example 4.1 are actually useful for $\mathcal{P}'$.

In the following, we shall introduce three linearization techniques. §4.2 details an intervalization process where some variables are replaced with intervals. §4.3 shows a linearization operator based on Bernstein basis: the matrix of coefficients of a polynomial written in the Bernstein basis defines an over-approximating polyhedron. Finally, §4.4 introduces a new linearization technique exploiting Handelman (1988)'s theorem: it cancels nonlinear monomials of Q by using products of constraints of $\mathcal{P}$.

4.2 Intervalization

Linearizing the effect of $0 \leq Q$ on $\mathcal{P}$ by *intervalization* consists in replacing some variables of nonlinear products by intervals of constants. These intervals are infered from the initial polyhedron $\mathcal{P}$. For instance, from $\mathcal{P} = \{x_1 \geq 1,\ x_2 \geq -2,\ x_1 - x_2 \geq 0,\ x_1 + x_2 \leq 5\}$, we can deduce that $x_1 \in [1, 7]$ and $x_2 \in [-2, 3]$, as shown on the figure of Example 4.1. Thus, $x_1^2 + x_2^2 \leq 4$ can be intervalized into $[1, 7]x_1 + [-2, 3]x_2 \leq 4$. If a variable has no upper (resp. lower) bound in $\mathcal{P}$, then its interval has the form $[\alpha, +\infty[$ (resp. $]-\infty, \beta]$).

At this point, we have affine constraints with interval coefficients. A conjunction of such constraints is an interval polyhedron (Chen et al., 2009). To go back to standard convex polyhedra, one must eliminate interval coefficients. This is done by computing an affine interval $[f_1, f_2]$ – where bounds are either constant, affine forms of $\boldsymbol{x}$ or infinite – such that $Q \in [f_1, f_2]$ holds in $\mathcal{P}$, *i.e.* $\forall \boldsymbol{x} \in \mathcal{P},\ f_1(\boldsymbol{x}) \leq Q(\boldsymbol{x}) \leq f_2(\boldsymbol{x})$. Then, a guard $0 \bowtie Q$ is approximated by $0 \bowtie [f_1, f_2]$ which is defined by cases on $\bowtie$. For instance, $0 \leq [f_1, f_2] \Leftrightarrow \exists \boldsymbol{y},\ (f_1 \leq \boldsymbol{y} \leq f_2) \wedge (0 \leq \boldsymbol{y})$, which is itself equivalent to $0 \leq f_2$. Other cases are:

$\bowtie$	$\leq$	$=$	$\neq$
$0 \bowtie [f_1, f_2]$	$0 \leq f_2$	$0 \leq f_2 \wedge f_1 \leq 0$	$0 < f_2 \vee f_1 < 0$

Affine intervals are computed using heuristics inspired from Miné (2006), except that in order to increase precision, we dynamically partition the input polyhedron $\mathcal{P}$ according to the sign

of some affine subterms. For instance,

$$
\begin{array}{lr}
 & [1,7]x_1 + [-2,3]x_2 \leq 4 \\
\Leftrightarrow & 0 \leq 4 - [1,7]x_1 - [-2,3]x_2 \\
\Leftrightarrow & 0 \leq 4 + [-7,-1]x_1 + [-3,2]x_2 \\
\Leftrightarrow & \text{if } x_2 \geq 0 \text{ then } 0 \leq 4 + [-7x_1, -x_1] + [-3x_2, 2x_2] \\
 & \text{if } x_2 < 0 \text{ then } 0 \leq 4 + [-7x_1, -x_1] + [2x_2, -3x_2] \\
\Leftrightarrow & \text{if } x_2 \geq 0 \text{ then } 0 \leq [4 - 7x_1 - 3x_2, 4 - x_1 + 2x_2] \\
 & \text{if } x_2 < 0 \text{ then } 0 \leq [4 - 7x_1 + 2x_2, 4 - x_1 - 3x_2] \\
\Leftrightarrow & \text{if } x_2 \geq 0 \text{ then } 0 \leq 4 - x_1 + 2x_2 \\
 & \text{if } x_2 < 0 \text{ then } 0 \leq 4 - x_1 - 3x_2
\end{array}
$$

Here, to compute an interval approximating $[-3,2]x_2$, we need to split the analysis according to the sign of x_2. If $x_2 \geq 0$, then $[-3,2]x_2$ is bounded by $[-3x_2, 2x_2]$, otherwise it is bounded by $[2x_2, -3x_2]$. We do not need to split on the sign of x_1 to bound $[-7,-1]x_1$ because we know that x_1 is positive (by hypothesis, it belongs to $[1,7]$), hence a bound of $[-7x_1, -x_1]$. This heuristic for sign partitioning is detailed in §4.2.1. At the end, we are only interested in the upper bound of each interval, because $0 \leq [f_1, f_2] \Leftrightarrow 0 \leq f_2$.

Implementation Insights. Our certified linearization is built on a two-tier architecture: an untrusted OCaml oracle uses heuristics to select intervalization strategies and a Coq-certified procedure applies them to build a correct-by-construction result. These strategies are listed below; they finely tune the precision-versus-efficiency trade-off of the linearization.
The sign partitioning procedure is implemented in Coq at the top of the polyhedral domain. This part of the library was designed for VERASCO and works on integers. Thus, contrary to other linearization techniques that come in further sections (Bernstein and Handelman's linearization), our intervalization is available only on integers and does not work on the usual VPL datastructures. It actually manipulates an abstract syntax for expressions and constraints. In particular, it handles conditions defined with symbols $\wedge$, $\vee$, $\leq$, $=$ and $\neq$. Symbol $<$ is not used because it can always be expressed in terms of $\leq$ on integers. Extending this technique to rationals would require some implementation and proof work to integrate that module in the VPL core. In particular, interval arithmetic is a bit more tedious in rationals, since it cannot get rid of open bounds like with integers.

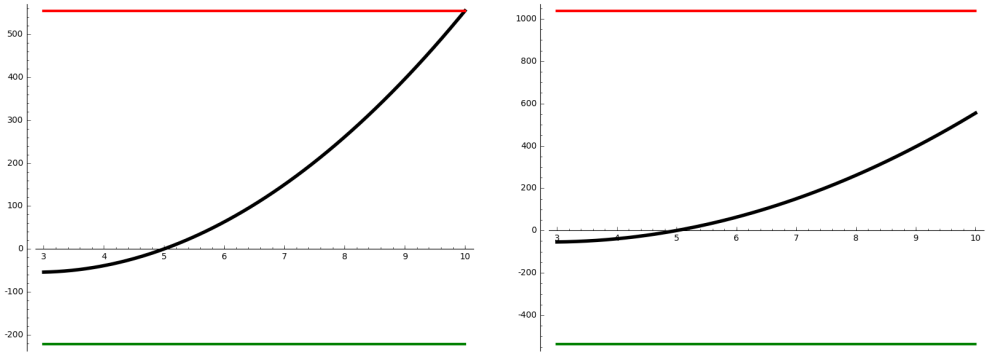
Proof in Coq. The proof of our intervalization was done by Sylvain Boulmé. Based on a refinement technique, the proof ended up to be small (about 2000 lines of code, 1000 of which taken by a certified interval domain). This work was published and presented at the 6^{th} international conference on Interactive Theorem Proving (Boulmé and Maréchal, 2015) in Nanjing, China.

Let us now introduce our list of rewriting heuristics. In the following, Q denotes polynomials and f denotes affine terms.

4.2.1 Our Interval-Based Strategies

Constant Intervalization. Let $Q = f_1 \times \ldots \times f_q$. Our fastest strategy applies a constant intervalization operator π: $\pi(Q)$ over-approximates the polynomial Q by an interval where affine terms are reduced to constants. It means that each occurrence of each variable is intervalized: $\pi(Q) = \pi(f_1 \times \ldots \times f_q) = \pi(f_1) \times \ldots \times \pi(f_q)$. It uses a naive integer interval domain built as an overlay of the polyhedral one. Arithmetic operations $+$ and $\times$ follow the usual definition of intervals (Miné, 2006):
$[\ell, u] + [\ell', u'] \stackrel{\text{def}}{=} [\ell + \ell', u + u']$, and
$[\ell, u] \times [\ell', u'] \stackrel{\text{def}}{=} [\min(E), \max(E)]$ where $E = \{\ell \cdot \ell', \ell \cdot u', u \cdot \ell', u \cdot u'\}$.

Example 4.2



On $x \in [3, 10]$, the constant intervalization of $(3x - 15) \times (4x - 3)$ gives interval $\pi(3x-15) \times \pi(4x-3) = [3 \times 3 - 15, 3 \times 10 - 15] \times [4 \times 3 - 3, 4 \times 10 - 3] = [-6, 15] \times [9, 37] = [-222, 555]$, as shown on the left-hand side figure. The right-hand side figure shows the constant intervalization for the same but expanded polynomial, *i.e.* $\pi(12x^2 - 69x + 45) = [-537, 1038]$, highlighting the great impact of factorization on π.

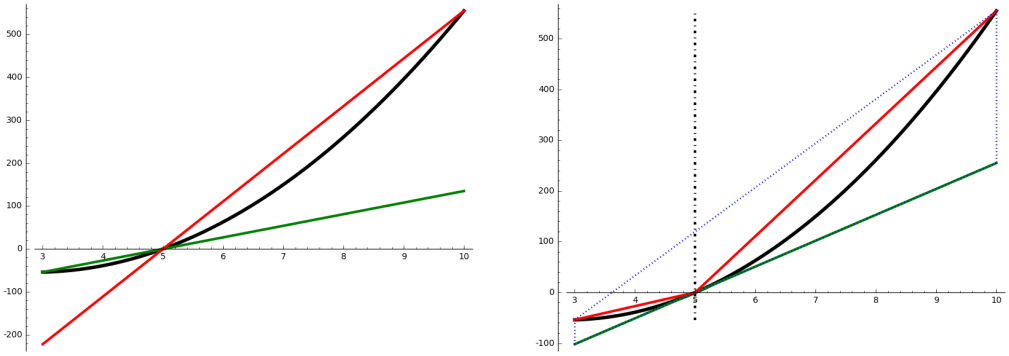
Ring Rewriting. It is difficult to find a factorization minimizing π results: as illustrated in Example 4.2, π is very sensitive to rewritings. For instance, consider a polynomial Q_1 such that $\pi(Q_1) = [0, n]$, with $n \in \mathbb{N}^+$. Then $\pi(Q_1 - Q_1)$ returns $[-n, n]$ instead of the precise result 0. Such imprecision also occurs in barycentric computations like $Q_2 \stackrel{\text{def}}{=} Q_1 \times f_1 + (n - Q_1) \times f_2$: if affine terms f_1, f_2 are bounded by $[\ell, u]$, then $\pi(Q_2)$ returns $2n \cdot [\ell, u]$ instead of $n \cdot [\ell, u]$. Moreover, if we rewrite Q_2 into an equivalent polynomial $Q_1 \times (f_1 - f_2) + n \cdot f_2$, then π returns $n \cdot [2\ell - u, 2u - \ell]$. If $\ell > 0$ or $u < 0$, then this is strictly more precise than $\pi(Q_2)$, but the situation is reversed otherwise. Consequently, our oracle begins with simplifying the polynomial before trying to factorize it conveniently. The following heuristics give more details about the rewriting strategies of the oracle.

Sign Partitioning. In order to find more precise bounds than those given by $\pi(Q)$, we look for an affine interval $[f_l, f_u]$ bounding Q. Assume Q is of the form $Q' \times f$, with f an affine term and Q' a polynomial. Assume $\pi(Q') = [\ell', u']$. Depending on the sign of f, we deduce affine bounds of Q in the following way:

- if $0 \leq f$, then $Q' \times f \in [\ell' \cdot f, u' \cdot f]$
- if $f < 0$, then $Q' \times f \in [u' \cdot f, \ell' \cdot f]$

When the input polyhedron $\mathcal{P}$ imposes the sign of f, we discard one of these two cases and the other is the affine approximation of $Q' \times f$. Otherwise, we split the analysis for each sign of f. The resulting interval of affine terms is computed by convex hull as follows. Considering Q as a function of $\boldsymbol{x}$, we call y the variable associated to $Q(\boldsymbol{x})$, *i.e.* $y = Q(\boldsymbol{x})$. Thus, we are looking for an interval $[f_l, f_u]$ such that $\forall \boldsymbol{x} \in \mathcal{P},\ f_l(\boldsymbol{x}) \leq y \leq f_u(\boldsymbol{x})$. Building upon this, intervals $[\ell' \cdot f, u' \cdot f]$ and $[u' \cdot f, \ell' \cdot f]$ are rewritten as polyhedra over variables $\boldsymbol{x}$ and y, respectively $\mathcal{P} \sqcap \{0 \leq f(\boldsymbol{x})\} \sqcap \{\ell' \cdot f(\boldsymbol{x}) \leq y \leq u' \cdot f(\boldsymbol{x})\}$ and $\mathcal{P} \sqcap \{f(\boldsymbol{x}) < 0\} \sqcap \{u' \cdot f(\boldsymbol{x}) \leq y \leq \ell' \cdot f(\boldsymbol{x})\}$. Then, we compute the convex hull of these two polyhedra, and normalize the resulting constraints so that y have a coefficient equal to 1 in each constraint. By considering only constraints involving y, we obtain two bounds for y that constitute the interval of affine bounds of Q.

Example 4.3 (follows 4.2)



Consider $Q = (3x-15) \times (4x-3)$ with $x \in [3,10]$. Let us illustrate the benefit of sign partitioning. After constant intervalization of the right term $(4x-3)$, which gives $\pi(4x-3) = [9,37]$, we obtain the two affine terms $(3x-15)\cdot 9$ and $(3x-15)\cdot 37$. As shown on the left figure, for $x \in [3,10]$, these two terms are not comparable: the two lines cross when $3x-15=0$. Thus, we cannot conclude that $Q \in [(3x-15)\cdot 9, (3x-15)\times 37]$ nor that $Q \in [(3x-15)\cdot 37, (3x-15)\cdot 9]$, since it is not true on the whole input polyhedron $3 \leq x \leq 10$.

To get an interval of affine terms bounding Q, we need to partition the space at the point where these two terms are equal, *i.e.* at the point where $3x-15=0$ which is $x=5$. Then, by intervalizing in both cells, we get the affine intervals shown on the right figure. Intervalizing $(4x-3)$ on cell $x<5$ gives $[9,17]$, and $[17,37]$ on cell $x \geq 5$. Thus, on $x<5$, $(3x-15)\times\pi(4x-3) = (3x-15)\times[9,17] = [51x-255, 27x-135]$ and on $x \geq 5$, $(3x-15)\times\pi(4x-3) = (3x-15)\times[17,37] = [51x-255, 111x-555]$. Finally, to obtain an interval of affine terms bounding Q, we must compute the convex hull of both sides. The associated polyhedron appears as the dotted polyhedron on the figure, and leads to interval $[51x-255, 87x-315]$.

More generally, we split the polyhedron $\mathcal{P}$ into a partition of polyhedra $(\mathcal{P}_i)_{i\in I}$ according to the sign of some affine subterms of polynomial Q. Then, we recursively call our oracle on each cell $\mathcal{P}_i$ to obtain an affine interval $[f_i, f_i']$ bounding $\mathcal{P}_i$. Finally, $0 \bowtie Q$ is over-approximated by computing the convex hull of all $0 \bowtie [f_i, f_i']$. The main drawback of sign partitioning is a worst-case exponential blow-up if applied systematically.

Let us now illustrate sign partitioning on the previous barycentric-like computation of $Q_2' \stackrel{\text{def}}{=} Q_1 \times (f_1 - f_2) + n \cdot f_2$. Our certified procedure partitions the sign of right affine subterms (here, the sign of $f_1 - f_2$). Recall that $\pi(Q_1) = [0,n]$, we obtain $Q_2' \in [n\cdot f_2, n\cdot f_1]$ in cell $0 \leq f_1 - f_2$, and $Q_2' \in [n\cdot f_1, n\cdot f_2]$ in cell $f_1 - f_2 < 0$. The convex hull of the two affine intervals leads to $n\cdot[\ell,u]$ as the over-approximation of Q_2', as we expect for such a barycentre. Note that sign partitioning is also sensitive to ring rewriting. In particular, the oracle may rewrite a product of affine terms $f_1 \times f_2$ into $f_2 \times f_1$, in order to discard f_1 instead of f_2 by sign partitioning.

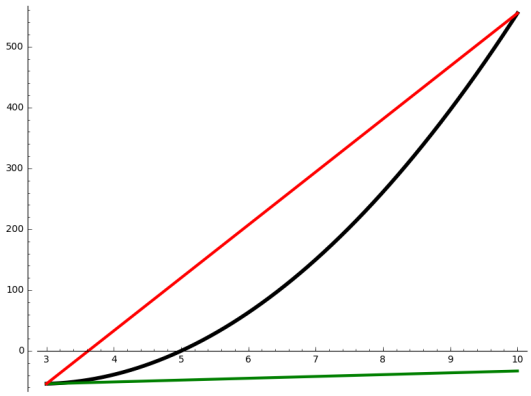
Focusing. Focusing is a ring rewriting heuristic that may increase the precision of sign partitioning. Given a product $Q \stackrel{\text{def}}{=} f_1 \times f_2$, we define the *focusing* of f_2 at center n as the rewriting of Q into $Q' = n\cdot f_1 + f_1 \times (f_2 - n)$. Thanks to this rewriting, the affine term $n \cdot f_1$ still appears after sign partitioning whereas it would have been discarded in Q.

For instance, let $\pi(f_1) = [\ell_1, u_1]$, $\pi(f_2) = [\ell_2, u_2]$, and let $n \in \mathbb{Q}$ such that $0 \leq n \leq \ell_2$. Sign partitioning bounds Q by interval $[\ell_1 \cdot f_2, u_1 \cdot f_2]$ whereas Q' is bounded by $[\ell_1 \cdot f_2 + n \cdot (f_1 - \ell_1), u_1 \cdot f_2 - n \cdot (u_1 - f_1)]$. The former is more precise than the latter:

$$[\ell_1 \cdot f_2 + n \cdot (f_1 - \ell_1), u_1 \cdot f_2 - n \cdot (u_1 - f_1)] \subseteq [\ell_1 \cdot f_2, u_1 \cdot f_2]$$

since n, $(f_1 - \ell_1)$ and $(u_1 - f_1)$ are nonnegative. Under these assumptions, the precision is maximal when $n = \ell_2$.
Applied carelessly, focusing may also decrease the precision. Consequently, on a product $Q'' \times f_2$ where $\pi(f_2) = [\ell_2, u_2]$, our oracle uses the following heuristic, which cannot decrease the precision: if $0 \leq \ell_2$, then focus f_2 at center ℓ_2; if $u_2 \leq 0$, then focus f_2 at center u_2; otherwise, do not change the focus of f_2.

Example 4.4 (follows 4.3)



Consider $Q = (3x-15)\times(4x-3)$ with $x \in [3, 10]$. Here, $\pi(4x-3) = [9, 37]$. As $9 > 0$, the focusing heuristic applies on 9, which rewrites Q into $Q' = 9.(3x-15)+(3x-15)(4x-12)$. Affine intervalization of Q' is done by sign partitioning of $(4x - 12)$. Because we chose to focus on the lower bound of $\pi(4x - 3)$, we obtain a partition where cell $4x - 12 < 0$ is empty, as shown on the figure. As $\pi(3x-15) = [-6, 15]$, we get $9.(3x-15)+[-6, 15](4x-12)$ *i.e.* $Q' \in [3x - 63, 87x - 315]$. Precision comes from the emptiness of cell $4x - 12 < 0$, which avoids the convex hull computation that usually follows sign partitioning.

Intervalizations of this figure and that of Example 4.2 have similar running times, but the latter gives strictly more precise results. The intervalization resulting from sign partitioning (see Example 4.3) is globally more precise than the two others but also more expensive (two constant intervalizations plus one convex-hull instead of one single constant intervalization).

Static vs Dynamic Intervalization During Partitioning. Computing the constant bounds of an affine term inside a given polyhedron invokes a linear programming procedure:

$$[\text{minimize } f(x) \text{ subject to } \boldsymbol{Ax} \leq \boldsymbol{b}, \text{maximize } f(x) \text{ subject to } \boldsymbol{Ax} \leq \boldsymbol{b}]$$

In dynamic mode, operator π exploits cells resulting from sign partitioning to compute bounds of an affine form on demand using LP. For a faster but less precise use of π, one may invoke it in *static* mode, where bounds are obtained using a pre-computed map σ that associates each variable of Q with its range in the initial polyhedron $\boldsymbol{Ax} \leq \boldsymbol{b}$.
For instance, let us consider the sign partitioning of $Q \stackrel{\text{def}}{=} f_1 \times f_2$ in the context $0 < \ell, u$ and $-\ell \leq f_2 \leq f_1 \leq u$. In cell $0 \leq f_2$, static mode bounds Q by $[-\ell.f_2, u.f_2]$, whereas dynamic mode bounds Q by $[0, u.f_2]$. In cell $f_2 < 0$, both modes bound Q by $[u.f_2, -\ell.f_2]$. On the join of these cells, both modes give the same upper bound. But the lower bound is $-\ell.u$ for static mode, whereas it is $\frac{\ell.u}{\ell+u}(f_2 + \ell) - \ell.u$ for dynamic mode, which is strictly more precise.

Combination of Strategies. Two distinct linearization strategies may lead to incomparable polyhedra. For instance, even if strategy of Example 4.3 is *globally* better than those of Example 4.2 and Example 4.4, the result of Example 4.4 is more precise than Example 4.3 around the bottom left corner. Here, we can improve precision by computing the intersection of these polyhedra. Let us remark here that a sequence of two strategies gives more precise

results than intersecting independent runs of these strategies: the second run may benefit from data discovered by the first one. This is illustrated in Example 4.5 below. We use this trick in order to ensure that our linearization necessarily improves and benefits from results of a naive but quick constant intervalization.

4.2.2 Design of Our Implementation

We now describe our procedure in details. Example 4.5 illustrates this description on a concrete guard. For a guard $0 \bowtie Q$, our certified procedure first discovers and extracts the affine part of Q, rewriting it into $Q' + f$ where f is an affine term and Q' a polynomial. The goal of this step is to avoid intervalizing the affine part of Q, and to keep the non-affine part small – in terms of number of monomials. Typically, if Q' is syntactically equal to zero, we simply apply the standard affine guard $0 \bowtie t$. Otherwise, we build the map σ for the variables of Q'. Then, we compute $0 \bowtie [\ell + t, u + t]$ where $[\ell, u]$ is the result of $\pi(Q')$ for static map σ. As mentioned earlier, this ensures that the resulting linearization necessarily improves and benefits from this first constant intervalization. In particular, if this guard is unsatisfiable at this point, the rest of the procedure is skipped. Otherwise, we invoke our external oracle on Q' and σ. This oracle returns a polynomial Q'', which is a rewriting of Q', enriched with tags on subexpressions. We handle three tags to direct the intervalization:

— `AFFINE` expresses that the subexpression is affine;

— `STATIC` expresses that the subexpression has to be intervalized in static mode;

— `INTERV` expresses that intervalization is done using only π (instead of sign partitioning).

At last, a special tag `SKIP_ORACLE` inserted at the root of Q'' indicates that it is not worth attempting to improve naive constant intervalization, *e.g.* because Q' is a too big polynomial and any attempt would be too costly. When this special tag is absent, our certified procedure checks that $Q' = Q''$ using a syntactic equality after normalization of the polynomials with Grégoire and Mahboubi (2005)'s procedure, available in the Coq standard distribution. If $Q' \neq Q''$, the program simply raises an error corresponding to a bug in the oracle. If $Q' = Q''$, the certified procedure applies the intervalization process, guided by the tags added by the oracle.

Design of Our External Oracle. Our external oracle ranks variables according to their priority to be discarded by sign partitioning: The priority rank is mainly computed from the size of intervals in the pre-computed map σ: unbounded variables must not be discarded whereas variables with a single value are always discarded by static intervalization. Our oracle also tries to minimize the number of distinct variables that are discarded: variables appearing in many monomials have a higher priority. Then, it factorizes variables with the highest priority. The oracle also interleaves factorization with focusing. Our oracle is written in 1300 lines of OCaml code.

Example 4.5

Let us consider the effect of our linearization procedure on guard $x \times (y - 2) \leq z$ in a context $\mathcal{P} = (0 \leq x) \wedge (x + 1 \leq y \leq 1000) \wedge (z \leq -2)$. First, note that a constant intervalization of $z - x \times (y - 2)$ would bound it in $]-\infty, 997]$, and thus would not deduce anything useful from this guard.

Instead, our procedure rewrites the guard into $0 \leq Q' + f$ with $Q' \stackrel{\text{def}}{=} -x \times y$ and $f \stackrel{\text{def}}{=} z + 2x$. Then, it computes map $\sigma \stackrel{\text{def}}{=} \{x \mapsto [0, 999], y \mapsto [1, 1000]\}$ and applies constant intervalization on Q', leading to $Q \in]-\infty, 0]$. As you may notice, approximating $0 \leq z - x \times (y - 2)$ requires only an upper-bound on Q', and our procedure does not compute the useless lower bound. From this first approximation of $0 \leq Q' + f$, it deduces $0 \leq f$.

Then, our oracle, invoked on Q' and σ, decides to focus the term y at center 1 and thus rewrites Q' as $Q'' \stackrel{\text{def}}{=} -x + x \times (1 - y)$. Here, it only intervalizes the nonlinear part

$x \times (1-y)$ using sign partitioning on $1-y$. Knowing that $0 \leq z+2x$ (coming from $0 \leq f$) and $z \leq -2$ (by hypothesis), we have $1 \leq x$. We can thus deduce that $1-y \leq -x \leq -1$. Therefore, because $1-y < 0$ and $1 \leq x$, sign partitioning on $1-y$ bounds $x \times (1-y)$ by $]-\infty, 1-y]$. At last, we now approximate $0 \leq Q' + f$ by $0 \leq 1-y-x+f$. In fact, this implies $0 \leq z$ which contradicts $z \leq -2$. Hence, our polyhedral approximation of $x \times (y-2) \leq z$ detects that this guard is unsatisfiable in the given context. Finally, we can reduce guard $x \times (y-z) \leq z$ to false in the context $\mathcal{P}$.

As a conclusion, let us remark that the first approximation leading to $0 \leq f$ is necessary to the full success of the second one.

In the following, we introduce two other linearization techniques. We will compare them with intervalization afterwards, in §4.5.

4.3 Bernstein's Linearization

In this section, we introduce a linearization technique based on the approximation of multivariate polynomials using the Bernstein basis. Given a polyhedron $\mathcal{P}$ and a polynomial Q defined on variables $(x_1, ..., x_l)$, we want to over-approximate $\mathcal{P} \wedge (Q \geq 0)$. As Q is expressed in the canonical basis $\mathfrak{C}$, the method consists in converting Q into a Bernstein basis $\mathfrak{B}$. Then, from the coefficients of Q in $\mathfrak{B}$, we can deduce a polyhedron containing $\mathcal{P} \wedge (Q \geq 0)$.

Note that, because a standard Bernstein basis for multivariate polynomials on $(x_1, \ldots, x_l)$ is defined on $[0,1]^l$, it can only represent polytopes which allow deducing bounded intervals for all variables, meaning that $\mathcal{P}$ can be over-approximated by a product of intervals $[a_1, b_1] \times \ldots \times [a_l, b_l]$. We will see that such products can be scaled to $[0,1]^l$. The Bernstein basis can actually be defined directly on polytopes (instead of an over-approximating box), by expressing points as convex combinations of its vertices (Clauss et al., 2009). This extension will not be treated here.

We begin by giving reminders about the Bernstein basis, and some clues about the conversion from the canonical to the Bernstein basis. Then, we will show how to obtain an over-approximating polyhedron from Bernstein coefficients.

This work was initiated by the following "well known statement": *Bernstein coefficients give a polyhedral approximation of a polynomial.* To our knowledge, in the literature, the result on the univariate case is implicitly extended to the multivariate case. This section provides the missing elements and proofs of Bernstein's linearization: actually, Bernstein coefficients are associated to points distributed along a regular mesh of $[0,1]^l$.

We will compute a Bernstein representation of a polynomial in two steps: Consider a polynomial $Q_{\mathfrak{C}}$, defined in the canonical basis $\mathfrak{C}$ on $\prod_{i=1}^{l}[a_i, b_i]$. First, $Q_{\mathfrak{C}}$ is scaled to $[0,1]^l$. For readability, we will denote the initial polynomial as $Q_{\mathfrak{C}}(\boldsymbol{x})$, where $\boldsymbol{x} \in \prod_{i=1}^{l}[a_i, b_i]$, and the scaled one by $Q'_{\mathfrak{C}}(\boldsymbol{t})$, where $\boldsymbol{t} \in [0,1]^l$. Then, $Q'_{\mathfrak{C}}(\boldsymbol{t})$ is converted into $Q'_{\mathfrak{B}}(\boldsymbol{t})$ expressed in a Bernstein basis $\mathfrak{B}$.

4.3.1 Bernstein Representation of Polynomials

4.3.1.1 The Bernstein Basis

The Univariate Bernstein Basis. We begin by recalling the univariate Bernstein basis, following notations of Farouki (2012)'s survey. The univariate Bernstein basis $\mathfrak{B}$ of degree n is the set of polynomials b_k^n defined on $t \in [0,1]$ as

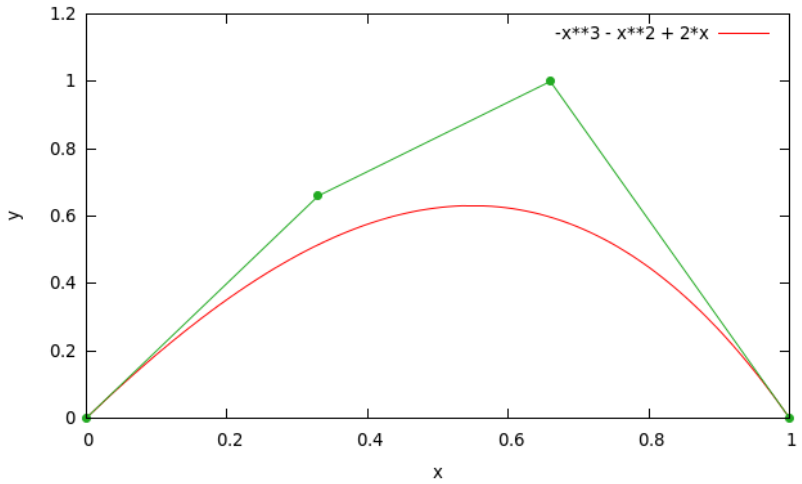
$$b_k^n(t) \stackrel{\text{def}}{=} \binom{n}{k}(1-t)^{n-k}t^k, \qquad k = 0, ..., n$$

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

$$Q(t) \stackrel{\text{def}}{=} \sum_{k=0}^{n} c_k b_k^n(t), \qquad t \in [0,1],\ c_k \in \mathbb{Q}$$

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

Example 4.6



The figure shows the polynomial $Q(t) = -t^3 - t^2 + 2t$. The Bernstein polynomial of degree 3 corresponding to Q is

$$\mathbf{0}\binom{3}{0} \times (1-t)^3 t^0 + \frac{\mathbf{2}}{\mathbf{3}}\binom{3}{1} \times (1-t)^2 t^1 + \mathbf{1}\binom{3}{2} \times (1-t)^1 t^2 + \mathbf{0}\binom{3}{3} \times (1-t)^0 t^3$$

From the Bernstein coefficients $(0, \frac{2}{3}, 1, 0)$, we obtain the four control points (the dots) of the figure: Each coefficient (in the order given above) is placed on a regular mesh of $[0,1]$, *i.e.* $\{0, \frac{1}{3}, \frac{2}{3}, 1\}$. The four control points are then $(0,0)$, $(\frac{1}{3}, \frac{2}{3})$, $(\frac{2}{3}, 1)$ and $(1,0)$.

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

Multi-indices. Following the notations from Ray and Nataraj (2012), let l be the number of variables, let $\boldsymbol{I} = (i_1, ..., i_l) \in \mathbb{N}^l$ be a multi-index and $\boldsymbol{x}^{\boldsymbol{I}} \stackrel{\text{def}}{=} x_1^{i_1} \times ... \times x_l^{i_l}$ be a multi-power. We define a partial order on multi-indices by $\boldsymbol{I} \leq \boldsymbol{J} \Leftrightarrow \forall k \in \{1, \ldots, l\},\ i_k \leq j_k$. We extend the binomial coefficient to multi-indices: $\binom{\boldsymbol{J}}{\boldsymbol{I}} = \binom{j_1}{i_1} \times \ldots \times \binom{j_l}{i_l}$.

The Multivariate Bernstein Basis. The Bernstein basis $\mathfrak{B}$ of degree $\boldsymbol{N} = (n_1, ..., n_l)$ on $\boldsymbol{t} = (t_1, ..., t_l) \in [0,1]^l$ is defined as

$$B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) = b_{i_1}^{n_1}(t_1) \times ... \times b_{i_l}^{n_l}(t_l), \qquad \boldsymbol{I} \leq \boldsymbol{N} \tag{4.1}$$

A multivariate polynomial expressed in this basis is written

$$Q(\boldsymbol{t}) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} c_{\boldsymbol{I}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}), \qquad \boldsymbol{t} \in [0,1]^l,\ c_{\boldsymbol{I}} \in \mathbb{Q}$$

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

$$\forall \boldsymbol{I} \leq \boldsymbol{N},\ \forall \boldsymbol{t} \in [0,1]^l,\ B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) \in [0,1] \tag{4.2}$$

$$\forall \boldsymbol{t} \in [0,1]^l,\ \sum_{\boldsymbol{I} \leq \boldsymbol{N}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) = 1 \tag{4.3}$$

Property (4.3) is called the partition-of-unity property. The two properties allow handling the Bernstein basis elements as coefficients of a convex combination: given points $\boldsymbol{a_0}, ..., \boldsymbol{a_N} \in \mathbb{Q}^l$ and $\boldsymbol{t} \in [0,1]^l$, the point $\boldsymbol{a} = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} \boldsymbol{a_I} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t})$ is a convex combination of $\boldsymbol{a_0}, ..., \boldsymbol{a_N}$. It means that $\boldsymbol{a}$ belongs to the convex hull of $\boldsymbol{a_0}, ..., \boldsymbol{a_N}$.

Rough Approximation. The partition-of-unity property can be used to get a rough approximation of a polynomial Q.

$$\begin{aligned}
& \sum_{\boldsymbol{I} \leq \boldsymbol{N}} min\left\{c_{\boldsymbol{I}} \mid \boldsymbol{I} \leq \boldsymbol{N}\right\} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) \;\leq\; \overbrace{\sum_{\boldsymbol{I} \leq \boldsymbol{N}} c_{\boldsymbol{I}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t})}^{Q(\boldsymbol{t})} \;\leq\; \sum_{\boldsymbol{I} \leq \boldsymbol{N}} max\left\{c_{\boldsymbol{I}} \mid \boldsymbol{I} \leq \boldsymbol{N}\right\} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) \\
\equiv \quad & min\left\{c_{\boldsymbol{I}} \mid \boldsymbol{I} \leq \boldsymbol{N}\right\} \underbrace{\sum_{\boldsymbol{I} \leq \boldsymbol{N}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t})}_{=1} \;\leq\; \sum_{\boldsymbol{I} \leq \boldsymbol{N}} c_{\boldsymbol{I}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) \;\leq\; max\left\{c_{\boldsymbol{I}} \mid \boldsymbol{I} \leq \boldsymbol{N}\right\} \underbrace{\sum_{\boldsymbol{I} \leq \boldsymbol{N}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t})}_{=1} \\
\equiv \quad & min\left\{c_{\boldsymbol{I}} \mid \boldsymbol{I} \leq \boldsymbol{N}\right\} \;\leq\; \sum_{\boldsymbol{I} \leq \boldsymbol{N}} c_{\boldsymbol{I}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) \;\leq\; max\left\{c_{\boldsymbol{I}} \mid \boldsymbol{I} \leq \boldsymbol{N}\right\}
\end{aligned}$$

We obtain two constant bounds:

$$\forall \boldsymbol{t} \in [0,1]^l,\; min\left\{c_{\boldsymbol{I}} \mid \boldsymbol{I} \leq \boldsymbol{N}\right\} \leq Q(\boldsymbol{t}) \leq max\left\{c_{\boldsymbol{I}} \mid \boldsymbol{I} \leq \boldsymbol{N}\right\}$$

This approximation is a first step toward a precise over-approximation. It already has a good precision, but we can do better: in the following, we take advantage of all control points to get tight approximations of the polynomial all around its graph.

4.3.1.2 From Canonical to Bernstein Basis

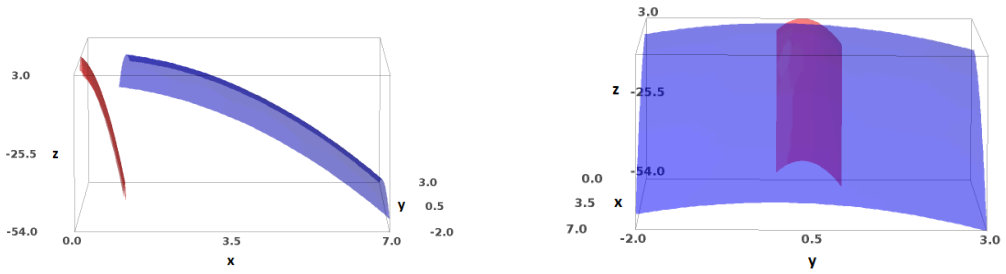
In this section, we show how to convert a polynomial from the canonical basis $\mathfrak{C}$ to a Bernstein basis $\mathfrak{B}$ with the conventional method explained by Ray and Nataraj (2012).[1] As the standard Bernstein basis is defined on $[0,1]^l$, a polynomial $Q_{\mathfrak{C}}$ defined as $Q_{\mathfrak{C}}(\boldsymbol{x}) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} d_{\boldsymbol{I}} \boldsymbol{t}^{\boldsymbol{I}}$ 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 $k \in \{1, \ldots, l\}$, 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

$$\forall (t_1, ..., t_l) \in [0,1]^l,\; Q'_{\mathfrak{C}}(t_1, ...t_l) \stackrel{\text{def}}{=} Q_{\mathfrak{C}}(\sigma_1(t_1), ..., \sigma_l(t_l)) = Q_{\mathfrak{C}}(x_1, \ldots, x_l) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} d'_{\boldsymbol{I}} \boldsymbol{x}^{\boldsymbol{I}}$$

where $x_k = \sigma_k(t_k) \in [a_k, b_k]$. These coefficients $d'_{\boldsymbol{I}}$ can be expressed in terms of the $d_{\boldsymbol{I}}$'s in the following way :

$$d'_{\boldsymbol{I}} = (\boldsymbol{b} - \boldsymbol{a})^{\boldsymbol{I}} \sum_{\boldsymbol{J}=\boldsymbol{I}}^{\boldsymbol{N}} d_{\boldsymbol{J}} \left(\binom{\boldsymbol{J}}{\boldsymbol{I}} \boldsymbol{a}^{\boldsymbol{J}-\boldsymbol{I}} \right),\; \boldsymbol{I} \leq \boldsymbol{N},\; \boldsymbol{a} = (a_1, ..., a_l),\; \boldsymbol{b} = (b_1, ..., b_l) \tag{4.4}$$

Example 4.7



1. This conversion algorithm has been implemented and proved in the Prototype Verification System by Muñoz and Narkawicz (2013).

Let us take the example $Q_{\mathfrak{C}} = -x_1^2 - x_2^2 + 4$ with $x_1 \in [1,7],\ x_2 \in [-2,3]$. The degree of $Q_{\mathfrak{C}}$ is the multi-index $N = (2,2)$, meaning that x_1 and x_2 have maximum degree 2. The coefficients of $Q_{\mathfrak{C}}$ are $d_{(0,0)} = 4$, $d_{(2,0)} = d_{(0,2)} = -1$ and $d_{\boldsymbol{I}} = 0$ for all others multi-indices $\boldsymbol{I}$. To illustrate Equation (4.4), we detail the computation of $d'_{(2,0)}$:

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

The transformation scales and shifts polynomial $Q_{\mathfrak{C}}(x_1,x_2) = -x_1^2 - x_2^2 + 4$ defined on $[1,7]\times[-2,3]$, providing a polynomial $Q'_{\mathfrak{C}}(t_1,t_2) = -36t_1^2 - 25t_2^2 - 12t_1 + 20t_2 - 1$ on $[0,1]^2$. When (t_1,t_2) ranges over $[0,1]^2$, the pink polynomial $Q'_{\mathfrak{C}}$ covers the image of the blue one $Q_{\mathfrak{C}}$ on $[1,7]\times[-2,3]$.

Now, given a polynomial $\sum_{\boldsymbol{I}\leq\boldsymbol{N}} d'_{\boldsymbol{I}}\boldsymbol{t}^{\boldsymbol{I}}$ 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 Ray and Nataraj (2012) shown, the translation from $\mathfrak{C}$ to $\mathfrak{B}$ using this formula has a complexity of $\mathcal{O}(n^{2l})$.

Example 4.8 (follows 4.7)

We can now compute the Bernstein representation of the scaled polynomial $Q'_{\mathfrak{C}} = -36t_1^2 - 25t_2^2 - 12t_1 + 20t_2 - 1$ from Example 4.7:

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

In the following, we will determine a polyhedron over-approximating $Q'_{\mathfrak{B}}$ from its coefficients. Then, by reversing the scaling functions σ_k, this polyhedron will scale so that it over-approximates $Q_{\mathfrak{C}}$.

4.3.2 Polyhedron from Bernstein Coefficients

Now, given a polytope $\mathcal{P}$, let us see how to build a polyhedron over-approximating $\mathcal{P}\wedge(Q \geq 0)$ from the Bernstein coefficients of $Q'_{\mathfrak{B}}$. As said previously, we can deduce intervals for each variable from $\mathcal{P}$. Let us call $[a_i,b_i]$ the interval associated with variable $x_i,\ i = 1,...,l$. Let $\mathcal{P}_{box}$ denotes the product of intervals $[a_1,b_1]\times...\times[a_l,b_l]$ and suppose $Q_{\mathfrak{C}}$ is defined on $\mathcal{P}_{box}$. In the following, we use $\sigma(\boldsymbol{x})$ for the vector $(\sigma_1(x_1),...,\sigma_l(x_l))$, where σ_k is the affine function mapping $[0,1]$ to $[a_k,b_k]$ defined in §4.3.1.2. The transformation of §4.3.1.2 provides a polynomial $Q'_{\mathfrak{C}}$ whose image on $[0,1]^l$ coincides with the image of the original polynomial $Q_{\mathfrak{C}}$ on $\mathcal{P}_{box}$. Since $Q'_{\mathfrak{C}}$ is defined on $[0,1]^l$, there exists a Bernstein representation of $Q'_{\mathfrak{C}}$ as

$$Q'_{\mathfrak{B}}(\boldsymbol{t}) = \sum_{\boldsymbol{I}\leq\boldsymbol{N}} c_{\boldsymbol{I}} B^{\boldsymbol{N}}_{\boldsymbol{K}}(\boldsymbol{t}), \qquad \boldsymbol{t}\in[0,1]^l,\ c_{\boldsymbol{I}}\in\mathbb{Q}$$

from which we obtain the set $\mathcal{V}'$ of control points whose first l dimensions form a regular l-dimensional mesh:

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

Let us define $\mathcal{P}'_{\mathcal{V}}$ as the convex hull of the control points of $\mathcal{V}'$. $\mathcal{P}'_{\mathcal{V}}$ is a polyhedron, and its vertices belong to $\mathcal{V}'$, but not all points of $\mathcal{V}'$ are vertices of $\mathcal{P}'_{\mathcal{V}}$: some can lay in the interior of $\mathcal{P}'_{\mathcal{V}}$. We will now prove that $\mathcal{P}'_{\mathcal{V}}$ is an over-approximation of $Q'_{\mathfrak{B}}$, meaning that $\forall (t_1, \ldots, t_l) \in [0,1]^l$, $(t_1, \ldots, t_l, Q'_{\mathfrak{B}}(t_1, \ldots, t_l)) \in \mathcal{P}'_{\mathcal{V}}$.

We start by giving in the following lemma the Bernstein coefficients of a polynomial reduced to a single variable $R(\boldsymbol{t}) \stackrel{\text{def}}{=} t_k$.

Lemma 4.1 *Let $k \in \mathbb{N}$, $1 \leq k \leq l$ and $\boldsymbol{N} = (n_1, \ldots, n_l)$.*

$$t_k = \sum_{\boldsymbol{I}=(i_1,\ldots,i_l)\leq \boldsymbol{N}} \frac{i_k}{n_k} B^{\boldsymbol{N}}_{\boldsymbol{I}}(\boldsymbol{t}), \quad \boldsymbol{t} \in [0,1]^l$$

Proof. We generalize to multivariate polynomials the proof of Farouki (2012) for the univariate case. Let us define the truncated multi-index $\boldsymbol{I} \setminus k$ as $(i_1, ..., i_{k-1}, i_{k+1}, ..., i_l)$ and $\boldsymbol{t} \setminus k$ as $(t_1, \ldots, t_{k-1}, t_{k+1}, \ldots, t_l)$. Let $\boldsymbol{t} \in [0,1]^l$ and consider the Bernstein polynomial $R(\boldsymbol{t}) \stackrel{\text{def}}{=} \sum_{\boldsymbol{I}\leq \boldsymbol{N}} \frac{i_k}{n_k} B^{\boldsymbol{N}}_{\boldsymbol{I}}(\boldsymbol{t})$. We will show that $R(\boldsymbol{t}) = t_k$.

Recall the definition (4.1) of $B^{\boldsymbol{N}}_{\boldsymbol{I}}$ as the product $b^{n_1}_{i_1}(t_1) \times ... \times b^{n_l}_{i_l}(t_l)$ for $\boldsymbol{I} = (i_1, \ldots, i_l) \leq \boldsymbol{N}$. Let us split $B^{\boldsymbol{N}}_{\boldsymbol{I}}$ into $b^{n_k}_{i_k}(t_k) \times B^{\boldsymbol{N}\setminus k}_{\boldsymbol{I}\setminus k}(\boldsymbol{t} \setminus k)$ to reveal the Bernstein monomial associated to t_k.

$$\begin{aligned} R(\boldsymbol{t}) &= \sum_{\substack{(\boldsymbol{I}\setminus k)\leq(\boldsymbol{N}\setminus k)\\ i_k=0,\ldots,n_k}} \frac{i_k}{n_k}\, b^{n_k}_{i_k}(t_k) \times B^{\boldsymbol{N}\setminus k}_{\boldsymbol{I}\setminus k}(\boldsymbol{t} \setminus k) \\ &= \sum_{\substack{(\boldsymbol{I}\setminus k)\leq(\boldsymbol{N}\setminus k)\\ i_k=0,\ldots,n_k}} \frac{i_k}{n_k}\, \binom{n_k}{i_k} t_k^{i_k}(1-t_k)^{n_k-i_k} \times B^{\boldsymbol{N}\setminus k}_{\boldsymbol{I}\setminus k}(\boldsymbol{t} \setminus k) \qquad \textit{by definition of } b^{n_k}_{i_k}(t_k) \end{aligned}$$

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,

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

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

$$\begin{aligned} R(\boldsymbol{t}) = & \sum_{\substack{(\boldsymbol{I}\setminus k)\leq(\boldsymbol{N}\setminus k)\\ i'_k=0,\ldots,n_k-1}} \binom{n_k-1}{i'_k} t_k^{i'_k+1}(1-t_k)^{n_k-1-i'_k} \times B^{\boldsymbol{N}\setminus k}_{\boldsymbol{I}\setminus k}(\boldsymbol{t} \setminus k) \\ = t_k \times & \sum_{\substack{(\boldsymbol{I}\setminus k)\leq(\boldsymbol{N}\setminus k)\\ i'_k=0,\ldots,n_k-1}} \underbrace{\binom{n_k-1}{i'_k} t_k^{i'_k}(1-t_k)^{n_k-1-i'_k}}_{b^{n_k-1}_{i'_k}(t_k)} \times B^{\boldsymbol{N}\setminus k}_{\boldsymbol{I}\setminus k}(\boldsymbol{t} \setminus k) \qquad \textit{we recognize } b^{n_k-1}_{i'_k} \\ = t_k \times & \sum_{\substack{(\boldsymbol{I}\setminus k)\leq(\boldsymbol{N}\setminus k)\\ i'_k=0,\ldots,n_k-1}} B^{\boldsymbol{N}'}_{\boldsymbol{I}}(\boldsymbol{t} \setminus k) \\ = t_k \times & \sum_{\boldsymbol{I}\leq \boldsymbol{N}'} \end{aligned}$$

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

$$\begin{aligned} R(\boldsymbol{t}) = t_k \quad \times & \underbrace{\sum_{\boldsymbol{I}\leq \boldsymbol{N}'} B^{\boldsymbol{N}'}_{\boldsymbol{I}}(\boldsymbol{t} \setminus k)}_{1} \\ = t_k \quad \times & \end{aligned}$$

□

Thanks to Lemma 4.1, we can relate the set $\mathcal{V}'$ of control points to the Bernstein coefficients of $Q'_{\mathfrak{B}}$. For each indeterminate $t_1, \ldots, t_l$, we associate the Bernstein coefficients given in the lemma. Similarly, we associate to $Q'_{\mathfrak{B}}$ its Bernstein coefficients $c_{\boldsymbol{I}}$ as follows: $\forall \boldsymbol{t} = (t_1, \ldots, t_l) \in$

$[0,1]^l$,

$$\begin{pmatrix} t_1 \\ \dots \\ t_l \\ Q'_{\mathfrak{B}}(\boldsymbol{t}) \end{pmatrix} = \begin{pmatrix} \sum_{\boldsymbol{I}\leq\boldsymbol{N}} \frac{i_1}{n_1} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) \\ \dots \\ \sum_{\boldsymbol{I}\leq\boldsymbol{N}} \frac{i_l}{n_l} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) \\ \sum_{\boldsymbol{I}\leq\boldsymbol{N}} c_{\boldsymbol{I}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) \end{pmatrix} = \sum_{\boldsymbol{I}\leq\boldsymbol{N}} \begin{pmatrix} \frac{i_1}{n_1} \\ \dots \\ \frac{i_l}{n_l} \\ c_{\boldsymbol{I}} \end{pmatrix} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) = \sum_{\boldsymbol{I}\leq\boldsymbol{N}} \boldsymbol{v}'_{\boldsymbol{I}}\ B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) \tag{4.5}$$

Recall that $\sum_{\boldsymbol{I}\leq\boldsymbol{N}} \boldsymbol{v}'_{\boldsymbol{I}}\ B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t})$ is a convex combination of the points $\boldsymbol{v}'_{\boldsymbol{I}}$ (properties (4.3) and (4.2)). Thus, Equation (4.5) means that any point $(t_1,...,t_l,Q'_{\mathfrak{B}}(\boldsymbol{t}))$ of the graph of $Q'_{\mathfrak{B}}$ can be expressed as a convex combination of the control points $\boldsymbol{v}'_{\boldsymbol{I}}$. In other words, $(t_1,...,t_l,Q'_{\mathfrak{B}}(\boldsymbol{t}))$ belongs to $\mathcal{P}'_{\mathcal{V}}$, the convex hull of the points $\boldsymbol{v}'_{\boldsymbol{I}}$. Therefore, $\mathcal{P}'_{\mathcal{V}}$ is an over-approximation of the set $\{(t_1,...,t_l,Q'_{\mathfrak{B}}(\boldsymbol{t}))\mid \boldsymbol{t}\in[0,1]^l\}$.

So far, we defined a polynomial $Q'_{\mathfrak{B}}$ on $[0,1]^l$, which is equivalent to the original $Q_{\mathfrak{C}}$ on $\prod_{i=1}^l [a_i,b_i]$. This gives a polyhedral over-approximation $\mathcal{P}'_{\mathcal{V}}$ using Bernstein coefficients. We now have to scale $\mathcal{P}'_{\mathcal{V}}$ to obtain a polyhedral approximation of $\{(x_1,\dots,x_l,Q_{\mathfrak{C}}(\boldsymbol{x}))\mid \boldsymbol{x}\in \prod_{i=1}^l [a_i,b_i]\}$.

Let us define $\mathcal{P}_{\mathcal{V}}$ as the convex hull of the elements of

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

The l first coordinates of $\boldsymbol{v}_{\boldsymbol{I}}$ are scaled using the mapping functions $\sigma_i:[0,1]\to[a_i,b_k]$. The last coordinate is unchanged since the image of $Q'_{\mathfrak{B}}$ on $[0,1]^l$ coincides with that of Q on $\prod_{i=1}^l [a_i,b_i]$. The following lemma shows that the equation of Lemma 4.1 is preserved by σ.

Lemma 4.2 *Let $k\in\mathbb{N},\ 1\leq k\leq l$. Let $\sigma_k: t\longmapsto \alpha t+\beta,\ \ \alpha,\beta\in\mathbb{Q}$ be an affine function.*

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

Proof. Let $\boldsymbol{t}\in[0,1]^l$ and consider the Bernstein polynomial $R(\boldsymbol{t})\stackrel{\text{def}}{=}\sum_{\boldsymbol{I}\leq\boldsymbol{N}}\sigma_k\left(\frac{i_k}{n_k}\right)B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t})$. We will show that $R(\boldsymbol{t})=\sigma_k(t_k)$.

$$\begin{array}{rcll} R(\boldsymbol{t}) & = & \sum\limits_{\boldsymbol{I}\leq\boldsymbol{N}} \sigma_k\left(\frac{i_k}{n_k}\right) B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) & \\ & = & \sum\limits_{\boldsymbol{I}\leq\boldsymbol{N}} \left(\alpha\frac{i_k}{n_k}+\beta\right) B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) & \\ & = & \alpha\left(\sum\limits_{\boldsymbol{I}\leq\boldsymbol{N}} \frac{i_k}{n_k} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t})\right)+\beta\left(\sum\limits_{\boldsymbol{I}\leq\boldsymbol{N}} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t})\right) & \\ & = & \alpha\left(\sum\limits_{\boldsymbol{I}\leq\boldsymbol{N}} \frac{i_k}{n_k} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t})\right)+\beta & \text{by the Property (4.3)} \\ & = & \alpha t_k+\beta & \text{by Lemma 4.1} \\ & = & \sigma_k(t_k) & \end{array}$$

□

The last step of the construction of our polyhedral over-approximation of $Q_{\mathfrak{C}}$ on $\mathcal{P}_{box}$ reuses the σ-transformation relating $Q_{\mathfrak{C}}$ and $Q'_{\mathfrak{C}}$:

$$\forall \boldsymbol{t}=(t_1,\dots,t_l)\in[0,1]^l,\ \ \sigma(\boldsymbol{t})\stackrel{\text{def}}{=}(\sigma_1(t_1),\dots,\sigma_l(t_l))\in\mathcal{P}_{box}$$

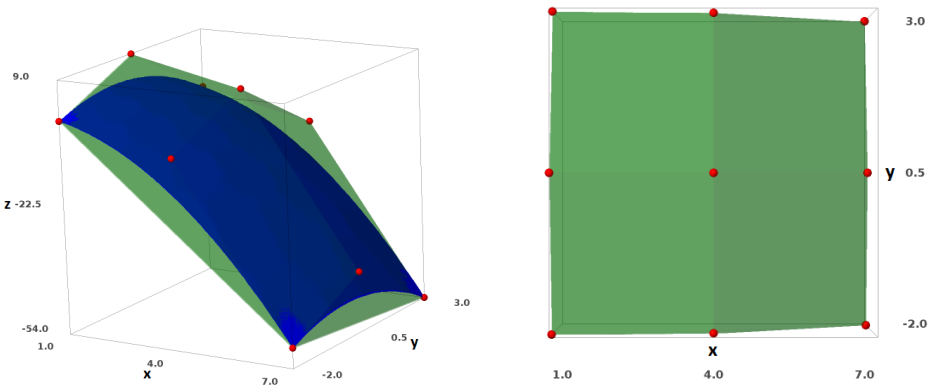
Then,

$$\forall \boldsymbol{x}\in\mathcal{P}_{box},\ \ \exists \boldsymbol{t}\in[0,1]^l,\ \ Q(\boldsymbol{x})=Q(\sigma(\boldsymbol{t}))=Q'_{\mathfrak{B}}(\boldsymbol{t})$$

$$\begin{pmatrix} x_1 \\ \dots \\ x_l \\ Q(\boldsymbol{x}) \end{pmatrix} = \begin{pmatrix} \sigma_1(t_1) \\ \dots \\ \sigma_l(t_l) \\ Q(\sigma(\boldsymbol{t})) \end{pmatrix} = \begin{pmatrix} \sigma_1(t_1) \\ \dots \\ \sigma_l(t_l) \\ Q'_{\mathfrak{B}}(\boldsymbol{t}) \end{pmatrix} = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} \begin{pmatrix} \sigma_1\left(\frac{i_1}{n_1}\right) \\ \dots \\ \sigma_l\left(\frac{i_l}{n_l}\right) \\ c_{\boldsymbol{I}} \end{pmatrix} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} \boldsymbol{v_I} B_{\boldsymbol{I}}^{\boldsymbol{N}}(\boldsymbol{t}) \quad (4.6)$$

Equality (4.6) implies that any point $(x_1, \dots, x_l, Q(\boldsymbol{x}))$ of the graph of Q can be expressed as a convex combination of the $\boldsymbol{v_I}$'s. Thus $(x_1, \dots, x_l, Q(\boldsymbol{x}))$ belongs to $\mathcal{P}_{\mathcal{V}}$, the convex hull of the $\boldsymbol{v_I}$'s. Hence $\mathcal{P}_{\mathcal{V}}$ is an over-approximation of the set $\{(x_1, \dots, x_l, Q(\boldsymbol{x})) \mid \boldsymbol{x} \in \mathcal{P}_{box}\}$.

Example 4.9 (follows 4.8)



With our running example, $\mathcal{P}_{\mathcal{V}}$ is the convex hull of the red points shown on the left figure, where we can see that the surface $\{(x_1, ..., x_l, Q(\boldsymbol{x})) \mid \boldsymbol{x} \in \mathcal{P}_{box}\}$ is clearly inside $\mathcal{P}_{\mathcal{V}}$. Moreover, the right figure shows that the points of $\mathcal{V}$ form a l-dimensional mesh.

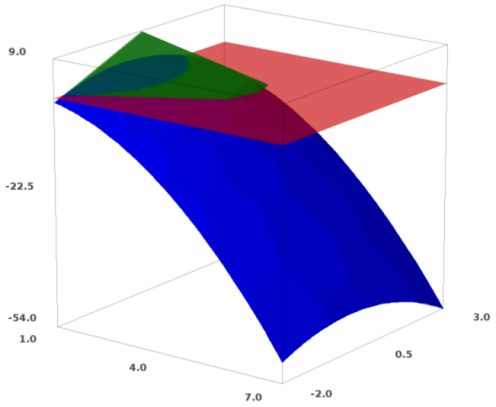
Recall that we are looking for a polyhedron over-approximating $\mathcal{P} \wedge (Q \geq 0)$. We have built the polyhedron $\mathcal{P}_{\mathcal{V}}$ that contains the set $\{(x_1, ..., x_l, Q(\boldsymbol{x})) \in \mathbb{Q}^{l+1} \mid \boldsymbol{x} \in \mathcal{P}_{box}\}$, but we are looking for an over-approximation of $\{\boldsymbol{x} \in P \mid Q(\boldsymbol{x}) \geq 0\}$, *i.e. the subspace of $\mathcal{P}$ where Q is nonnegative*. In order to approximate that subspace of $\mathbb{Q}^l$, we have to eliminate one dimension (that of $Q(\boldsymbol{x})$) by projection. We take into account the constraint $Q \geq 0$ by considering the polyhedron

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

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

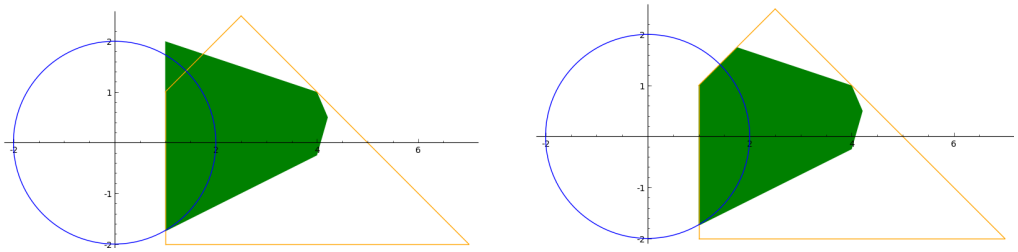
We compute $\mathcal{P}_{\mathcal{V}\setminus\{x_{l+1}\}}^{+}$, the projection of $\mathcal{P}_{\mathcal{V}}^{+}$ onto $\mathbb{Q}^l$. It is an over-approximation of $\{\boldsymbol{x} \in \mathcal{P}_{box} \mid Q(\boldsymbol{x}) \geq 0\}$. Recall that $\mathcal{P} \subseteq \mathcal{P}_{box}$ by construction of $\mathcal{P}_{box}$ as the product $\prod_{i=1}^{l}[a_i, b_i]$ of the ranging intervals of $x_1, \dots, x_l$ in $\mathcal{P}$. Therefore, $\mathcal{P}' \stackrel{\text{def}}{=} \mathcal{P}_{\mathcal{V}\setminus\{x_{l+1}\}}^{+} \sqcap \mathcal{P}$ approximates $\{\boldsymbol{x} \in \mathcal{P} \mid Q(\boldsymbol{x}) \geq 0\}$. Finally, the polyhedron $\mathcal{P}'$ captures the effect of the guard $Q \geq 0$ on the polyhedron $\mathcal{P}$.

Example 4.10 (follows 4.9)



In order to approximate the space where the polynomial (the blue surface) $Q(x_1, x_2) = -x_1^2 - x_2^2 + 4$ is nonnegative on $\mathcal{P}_{box}$ (the enclosing box), we compute the intersection of the polyhedron $\mathcal{P}_{\mathcal{V}}$ (from Example 4.9) with the half space where $x_3 \geq 0$ (above the red plane $x_3 = 0$). We obtain $\mathcal{P}^+_{\mathcal{V}}$, the green polyhedron of the figure.

Example 4.11 (follows 4.10)



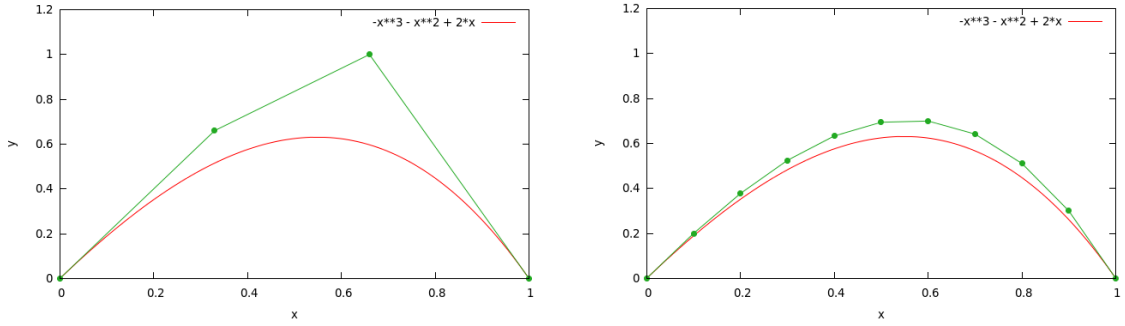
(a) (b)

The starting polyhedron $\mathcal{P} = \{(x_1, x_2) \mid x_1 - 1 \geq 0,\ x_2 + 2 \geq 0,\ x_1 - x_2 \geq 0,\ -x_1 - x_2 + 5 \geq 0\}$ is represented in orange. The circle $\{(x_1, x_2) \mid Q(x_1, x_2) = -x_1^2 - x_2^2 + 4 \geq 0\}$ is drawn in blue. Figure (a) shows $\mathcal{P}^+_{\mathcal{V}_{\setminus\{x_{l+1}\}}}$ in green. Figure (b) shows $\mathcal{P}' = \mathcal{P}^+_{\mathcal{V}_{\setminus\{x_{l+1}\}}} \sqcap \mathcal{P}$ in green. It over-approximates the effect of the guard $x_1^2 + x_2^2 \leq 4$ on $\mathcal{P}$.

4.3.3 Polyhedron Refinement

There exists two methods for improving the precision of Bernstein approximations: degree elevation of the basis and interval splitting. In both cases, the goal is to find control points closer to Q.

- Degree elevation consists in converting $Q_{\mathfrak{C}}$ into a Bernstein basis of higher degree, leading to a finer regular mesh. The number of points in the mesh is the number of Bernstein monomials $B_{\boldsymbol{I}}^{\boldsymbol{N}}$ such that $\boldsymbol{I} \leq \boldsymbol{N}$. Thereby, the set $\mathcal{V}$ contains more control points, hence more adjusted to the graph of $Q_{\mathfrak{C}}$. Thus, their convex hull $\mathcal{P}_{\mathcal{V}}$ is closer to $Q_{\mathfrak{C}}$, as shown on Fig. 4.12.
- The principle of interval splitting is to split the initial box $[a_1, b_1] \times ... \times [a_l, b_l]$ on a point $(x_1, \ldots, x_l)$ in the box (when $a_i \neq x_i \neq b_i,\ \forall i \in \{1, \ldots, l\}$). This process gives k different boxes, and $k = 2^l$ if we split in every dimensions. For each of these boxes, we get an



(a) (b)

Figure 4.12 – Representation of $Q(x) = -x^3 - x^2 + 2x$ in red. (a): $\mathcal{P}_\mathcal{V}$ deduced from a Bernstein basis of degree 3. (b): $\mathcal{P}_\mathcal{V}$ deduced from a Bernstein basis of degree 10.

expression of Q in the basis $\mathfrak{B}$ from which we can build $\mathcal{V}_i$, and we obtain k times more points in $\mathcal{V} = \mathcal{V}_1 \cup \ldots \cup \mathcal{V}_k$. This refinement is efficient as we don't need to convert $Q_\mathfrak{C}$ from $\mathfrak{C}$ to $\mathfrak{B}$ for each box, we can deduce the expression of $Q'_\mathfrak{B}$ on another box directly from its expression in the original one thanks to an algorithm by De Casteljau (Muñoz and Narkawicz, 2013). The result is a disjunction of polyhedra $\mathcal{P}_{\mathcal{V}_1}, \ldots, \mathcal{P}_{\mathcal{V}_k}$. Note that the increase of precision on each box can be lost by the convex hull if we must return only one polyhedron. The difficulty is to find a priori the good point where to split to get an actual gain in precision.

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.

4.3.4 Toward Certification in Coq

Bernstein's linearization was only implemented as a prototype in Sage. It was not added into the VPL because it requires computing the range of each variable, which costs solving two LP problems per variable, and because the result is a set of vertices, the control points of $\mathcal{V}$. To obtain a polyhedron in constraints, it is then necessary to run Chernikova's algorithm to switch to the constraints-only representation. This conversion is exactly what we want to avoid in the VPL. This makes Bernstein's linearization not well-suited for our library.

Besides, certifying Bernstein's linearization in Coq, even with certificate checking, would have required a lot of work. The change of basis can be done externally and provided as a certificate to Coq. It consists in the coefficients $c_\boldsymbol{I}$ of $Q'_\mathfrak{B}$ and the mapping function σ. With this certificate, it is quite straightforward to develop in Coq a checker that decides the equality $Q_\mathfrak{C}(\sigma(\boldsymbol{t})) = \sum_{\boldsymbol{I} \leq \boldsymbol{N}} c_\boldsymbol{I} B^\boldsymbol{N}_\boldsymbol{I}(\boldsymbol{t})$: it can be done by expanding both sides of the equality and compare each monomial coefficient. Actually, for validating Handelman's linearization, we implemented such a checker and proved its correctness in Coq (see Chapter 9). Remark that the polynomial equality must be checked each time the linearization operator is called. On the contrary, the following lemmas in polyhedra and Bernstein basis are proved once for all, but their proof can be a bottomless pit.

(1) The properties (4.2) and (4.3) of the Bernstein basis (p.81).

(2) The convex hull of the control points of $\mathcal{V}$ forms an over-approximation of $\{(x_1, \ldots, x_l, Q(\boldsymbol{x})) \mid \boldsymbol{x} \in \mathcal{P}_{box}\}$.

Yet, even if we overcome these obstacles, the problem of certifying the equivalence between a polyhedron represented as generators and one represented as constraints remains. This was discussed in §1.1.2.3.

4.4 Handelman's Linearization

In this section, we present a linearization algorithm based on Handelman's theorem, which gives a way to express a positive polynomial as a nonnegative combination of products of affine constraints. This work was published and presented at the 17^{th} international conference on Verification, Model Checking, and Abstract Interpretation (Maréchal et al., 2016), in St. Petersburg, Florida.

Consider an input polyhedron $\mathcal{P} = \{C_1 \geq 0, \ldots, C_p \geq 0\}$ defined on variables $(x_1, \ldots, x_n) \in \mathbb{Q}^n$ and a polynomial guard $Q \geq 0$. Our goal is to find an affine term $\alpha_0 + \sum_{i=1}^{n} \alpha_i x_i$ denoted by f such that $\mathcal{P} \Rightarrow f > Q$, meaning that f is an upper bound of Q on $\mathcal{P}$. By transitivity, we will conclude that $\mathcal{P} \wedge (Q \geq 0) \Rightarrow \mathcal{P} \wedge (f > 0)$, which can be expressed in terms of sets as $[\![\mathcal{P} \wedge (Q \geq 0)]\!] \subseteq [\![\mathcal{P} \sqcap (f > 0)]\!]$. This indicates that $\mathcal{P} \sqcap (f > 0)$ is an over-approximation of $\mathcal{P} \wedge (Q \geq 0)$ This linearization based on Handelman's theorem is expressed as a Parametric Linear Optimization Problem (PLOP), and provides several affine constraints $f_1, \ldots, f_k \geq 0$ whose intersection with $\mathcal{P}$ forms the approximation of $\mathcal{P} \wedge (Q \geq 0)$.

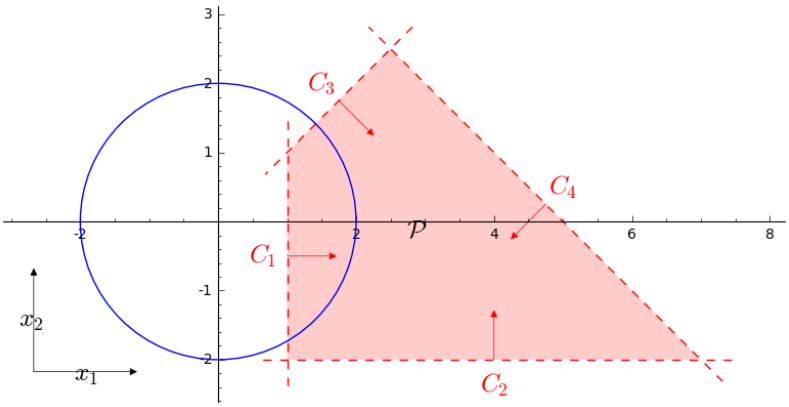
4.4.1 Representation of Positive Polynomials on a Polytope

Notations. A multi-index $\boldsymbol{I} = (i_1, ..., i_n) \in \mathbb{N}^n$ is a vector of exponents, such that $\boldsymbol{x}^{\boldsymbol{I}} \stackrel{\text{def}}{=} x_1^{i_1} \cdots x_n^{i_n}$. We define the set of Handelman products associated to a polyhedron $\mathcal{P} = \{C_1 \geq 0, \ldots, C_p \geq 0\}$ as the set $\mathscr{H}_{\mathcal{P}}$ of all products of constraints C_i of $\mathcal{P}$:

$$\mathscr{H}_{\mathcal{P}} = \{C_1^{i_1} \times \cdots \times C_p^{i_p} \mid (i_1, \ldots, i_p) \in \mathbb{N}^p\}$$

Given a multi-index $\boldsymbol{I} = (i_1, \ldots, i_p)$, $H^{\boldsymbol{I}} \stackrel{\text{def}}{=} C_1^{i_1} \times \ldots \times C_p^{i_p}$ denotes an element of $\mathscr{H}_{\mathcal{P}}$.

Example 4.13



Let us recall the linearization problem $\mathcal{P} \wedge (Q \geq 0)$ of Example 4.1 (p.73): $\mathcal{P} = \{x_1 \geq 1,\ x_2 \geq -2,\ x_1 - x_2 \geq 0,\ x_1 + x_2 \leq 5\}$ and $Q = 4 - x_1^2 - x_2^2$. With this polyhedron, $H^{(0,2,0,0)} = (x_2 + 2)^2$, $H^{(1,0,1,0)} = (x_1 - 1)(x_1 - x_2)$ and $H^{(1,0,0,3)} = (x_1 - 1)(-x_1 - x_2 + 5)^3$ all belong to $\mathscr{H}_{\mathcal{P}}$.

The $H^{\boldsymbol{I}}$'s are nonnegative polynomials on $\mathcal{P}$ as products of nonnegative constraints of $\mathcal{P}$. Handelman's representation of a positive polynomial Q on $\mathcal{P}$ is

$$Q(\boldsymbol{x}) = \sum_{\boldsymbol{I} \in \mathbb{N}^p} \underbrace{\lambda_{\boldsymbol{I}}}_{\geq 0} \underbrace{H^{\boldsymbol{I}}}_{\geq 0}(\boldsymbol{x}) \text{ with } \lambda_{\boldsymbol{I}} \in \mathbb{Q}^+ \tag{4.7}$$

The $\lambda_{\boldsymbol{I}}$'s form a *certificate* that Q is nonnegative on $\mathcal{P}$. Handelman's theorem states the non-trivial opposite implication: any positive polynomial on $\mathcal{P}$ can be expressed in that form (Handelman, 1988)(Schweighofer, 2002, Th. 5.5)(Prestel and Delzell, 2001, Th. 5.4.6)(Lasserre, 2010,

Th. 2.24); a similar result already appeared in Krivine (1964)'s work on decompositions of positive polynomials on semialgebraic sets.

Theorem 4.3 (Handelman) *Let $\mathcal{P} = \{C_1 \geq 0, \ldots, C_p \geq 0\}$ be a polytope where each C_i is an affine form over $\boldsymbol{x} \in \mathbb{Q}^n$. Let Q be a positive polynomial on $\mathcal{P}$, i.e. $Q(\boldsymbol{x}) > 0$ for all $\boldsymbol{x} \in \mathcal{P}$. Then there exists a finite subset $\mathcal{I}$ of $\mathbb{N}^p$ and $\lambda_{\boldsymbol{I}} \in \mathbb{Q}^+$ for all $\boldsymbol{I} \in \mathcal{I}$, such that*

$$Q = \sum_{\boldsymbol{I} \in \mathcal{I}} \lambda_{\boldsymbol{I}} H^{\boldsymbol{I}}$$

This does not necessarily hold if Q is only assumed to be nonnegative. Consider the inequalities $x+1 \geq 0$ and $1-x \geq 0$ and the nonnegative polynomial x^2. Assume the existence of a decomposition (4.7) and apply it at $x = 0$: $Q(0) = 0 = \sum \lambda_{(i_1,i_2)} 1^{i_1} 1^{i_2}$. But, since $1^{i_1} 1^{i_2} > 0$ whatever the value of $(i_1, i_2) \in \mathbb{N}^2$, then $\lambda_{(i_1,i_2)} = 0,\ \forall (i_1, i_2) \in \mathbb{N}^2$. This null decomposition is absurd.

Note that one can look for a Handelman representation of a polynomial even on *unbounded polyhedra*: its positivity will then be ensured. The existence of such representation is not guaranteed though. A sufficient condition for Q is that it is bounded by a concave function. Indeed, any tangent curve of such concave function is an affine function dominating Q.

A common use of Handelman's representation of a polynomial $Q - \Delta$ is to determine a constant bound Δ of Q on $\mathcal{P}$. For instance, Boland and Constantinides (2011) use it to compute an upper bound of the polynomial, in $\boldsymbol{x}$ and the error ϵ, which defines the cascading round-off effects of floating-point calculation. Schweighofer (2002)'s algorithm can iteratively improve such a bound by increasing the degree of the $H^{\boldsymbol{I}}$'s. We present here another use of Handelman's theorem: we are not interested in just one bound but in a whole set of affine constraints dominating the polynomial Q on $\mathcal{P}$.

A similar use of Handelman's theorem appeared in the context of linearization for generating polynomial invariants (Bagnara et al., 2005). Intuitively, as we mention in §4.4.2.2, their approach considers nonlinear products as additional variables. Thereby, an SAS becomes a polyhedron in higher dimension.

The idea of replacing nonlinear expressions by new variables was also applied by Ben Sassi et al. (2012), to find out if a polynomial is positive over a positive semi-definite set, in the context of reachability analysis for discrete-time polynomial dynamical systems. To do so, they compute lower bounds of the polynomial that, if tight enough, ensure the positivity of the polynomial. These bounds are found as optimal solutions of LP problems that are defined as follows. The polynomial is rewritten in the Bernstein basis, and elements $B_{\boldsymbol{I}}^{\boldsymbol{N}}$ of the basis are replaced by fresh variables, leading to an affine form that is used as objective function to minimize. The properties of the Bernstein basis (see (4.2) and (4.3) §4.3.1.1) give affine constraints between these additional variables. Ben Sassi et al. (2015) extend this work to Lyapunov function synthesis, and improve the approximation by considering another property of the Bernstein basis, namely the induction relation between Bernstein polynomials. They also compare this approach (and Handelman-based linearizations, which both produce LP problems) with relaxations by sum-of-squares (SOS) programs.

4.4.2 Linearization as a PLOP

Recall that we are looking for an affine constraint $f \stackrel{\text{def}}{=} \alpha_0 + \sum_{i=1}^{n} \alpha_i x_i$ that approximates a non-linear guard Q, meaning that $f > Q$ on $\mathcal{P}$. According to Theorem 4.3, if $\mathcal{P}$ is bounded, $f - Q$ (which is positive on the polytope $\mathcal{P}$) has a Handelman representation as a nonnegative linear combination of products of constraints of $\mathcal{P}$, *i.e.*

$$\exists \mathcal{I} \subset \mathbb{N}^p,\ f - Q = \sum_{\boldsymbol{I} \in \mathcal{I}} \lambda_{\boldsymbol{I}} H^{\boldsymbol{I}},\ \lambda_{\boldsymbol{I}} \in \mathbb{Q}^+,\ H^{\boldsymbol{I}} \in \mathscr{H}_{\mathcal{P}} \tag{4.8}$$

Relation (4.8) ensures that there exists some nonnegative combinations of Q and some $H^{\boldsymbol{I}} \in \mathscr{H}_{\mathcal{P}}$ that cancel the monomials of degree >1 and lead to affine forms:

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

This decomposition is not unique in general: given a set of Handelman products, there exists several ways of canceling nonlinear monomials of Q, each of which leading to a distinct affine form f_i. The principle of our algorithm is to take advantage of the non-uniqueness of representation to get a precise approximation of the guard: we suppose that a set $\mathcal{I} = \{\boldsymbol{I_1}, \ldots, \boldsymbol{I_q}\}$ of multi-indices is given and we show how to obtain every possible affine form f_i that can be expressed as $Q + \sum_{\ell=1}^{\ell=q} \lambda_\ell H^{\boldsymbol{I_\ell}}$. Each of these f_i bounds Q on $\mathcal{P}$ and their conjunction forms a polyhedron that over-approximates the set $\mathcal{P} \wedge (Q \geq 0)$. A major difference between our work and previous work by Schweighofer (2002) and Boland and Constantinides (2011) is that we are not interested in a constant bound α_0 but an affine bound $\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n$ which still depends on *parameters* $x_1, \ldots, x_n$. We now show that our problem belongs to the class of *parametric linear problems*; §4.4.3 then describes the heuristics used to determine $\mathcal{I}$.

Example 4.14 (follows 4.13)

For $Q = 4 - x_1^2 - x_2^2$ and $\mathcal{P} = \{C_1 : x_1 - 1 \geq 0,\ C_2 : x_2 + 2 \geq,\ C_3 : x_1 - x_2 \geq 0,\ C_4 : -x_1 - x_2 + 5 \geq 0\}$, we choose $\mathcal{I} = \{\boldsymbol{I_1}, \ldots, \boldsymbol{I_{15}}\}$ such that

$$
\begin{array}{ll}
H^{\boldsymbol{I_1}} = H^{(0,0,0,0)} = 1 & H^{\boldsymbol{I_9}} = H^{(0,0,0,2)} = (-x_1 - x_2 + 5)^2 \\
H^{\boldsymbol{I_2}} = H^{(1,0,0,0)} = x_1 - 1 & H^{\boldsymbol{I_{10}}} = H^{(1,1,0,0)} = (x_1 - 1)(x_2 + 2) \\
H^{\boldsymbol{I_3}} = H^{(0,1,0,0)} = x_2 + 2 & H^{\boldsymbol{I_{11}}} = H^{(1,0,1,0)} = (x_1 - 1)(x_1 - x_2) \\
H^{\boldsymbol{I_4}} = H^{(0,0,1,0)} = x_1 - x_2 & H^{\boldsymbol{I_{12}}} = H^{(1,0,0,1)} = (x_1 - 1)(-x_1 - x_2 + 5) \\
H^{\boldsymbol{I_5}} = H^{(0,0,0,1)} = -x_1 - x_2 + 5 & H^{\boldsymbol{I_{13}}} = H^{(0,1,1,0)} = (x_2 + 2)(x_1 - x_2) \\
H^{\boldsymbol{I_6}} = H^{(2,0,0,0)} = (x_1 - 1)^2 & H^{\boldsymbol{I_{14}}} = H^{(0,1,0,1)} = (x_2 + 2)(-x_1 - x_2 + 5) \\
H^{\boldsymbol{I_7}} = H^{(0,2,0,0)} = (x_2 + 2)^2 & H^{\boldsymbol{I_{15}}} = H^{(0,0,1,1)} = (x_1 - x_2)(-x_1 - x_2 + 5) \\
H^{\boldsymbol{I_8}} = H^{(0,0,2,0)} = (x_1 - x_2)^2 &
\end{array}
$$

4.4.2.1 The PLOP Encoding

Considering the products $\{H^{\boldsymbol{I_1}}, \ldots, H^{\boldsymbol{I_q}}\}$, finding a Handelman representation of $f - Q$ can be expressed as a LP problem. Relation (4.8) amounts to finding $\lambda_1, \ldots, \lambda_q \geq 0$ such that

$$
\begin{array}{ccccc}
f & = & 1 \cdot Q + \sum_{\ell=1}^{\ell=q} \lambda_\ell H^{\boldsymbol{I_\ell}} & = & \underbrace{(\lambda_Q, \lambda_1, \ldots, \lambda_q)}_{\boldsymbol{\lambda}^\intercal} \cdot \underbrace{(Q, H^{\boldsymbol{I_1}}, \ldots, H^{\boldsymbol{I_q}})^\intercal}_{\mathcal{H}_Q^\intercal \cdot \boldsymbol{\mathcal{M}}} \\
\| & & \| & & \\
\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n & & \boldsymbol{\lambda}^\intercal \cdot \mathcal{H}_Q^\intercal \cdot \boldsymbol{\mathcal{M}} & & \\
\| & & \| & & \\
\boldsymbol{\mathcal{M}}^\intercal \cdot (\alpha_0, \ldots, \alpha_n, 0, \ldots, 0) & = & \boldsymbol{\mathcal{M}}^\intercal \cdot \mathcal{H}_Q \cdot \boldsymbol{\lambda} & &
\end{array}
$$

where:

(1) $\mathcal{H}_Q$ is the matrix of the coefficients of Q and the $H^{\boldsymbol{I_\ell}}$ organized with respect to $\boldsymbol{\mathcal{M}}$, the sorted list of monomials that appear in the Handelman products generated by $\mathcal{I}$.

(2) the column vector $\boldsymbol{\lambda} = (\lambda_Q, \lambda_1, \ldots, \lambda_q)^\intercal = (1, \lambda_1, \ldots, \lambda_q)^\intercal$ characterizes the combination of Q and the $H^{\boldsymbol{I_\ell}}$. Coefficient λ_Q is set to 1 since we want the affine form f to dominate Q. This is why f is expressed as Q plus something nonnegative.

The product $\mathcal{H}_Q \cdot \boldsymbol{\lambda}$ is a vector $\boldsymbol{\alpha} \stackrel{\text{def}}{=} (\alpha_0, \ldots, \alpha_{|\boldsymbol{\mathcal{M}}|-1})^\intercal$ representing the constraint

$$\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n + \sum_{i=n+1}^{|\boldsymbol{\mathcal{M}}|-1} \alpha_i \cdot (\boldsymbol{\mathcal{M}})_i$$

where $(\boldsymbol{\mathcal{M}})_i$ denotes the i^{th} monomial of $\boldsymbol{\mathcal{M}}$. Since we seek an affine constraint f we are finally interested in finding $\boldsymbol{\lambda} \in \{1\} \times (\mathbb{Q}^+)^q$ such that $\mathcal{H}_Q \cdot \boldsymbol{\lambda} = (\alpha_0, \ldots, \alpha_n, 0, \ldots, 0)^\intercal$. By construction, each such $\boldsymbol{\lambda}$ gives an affine constraint f that bounds Q on $\mathcal{P}$.

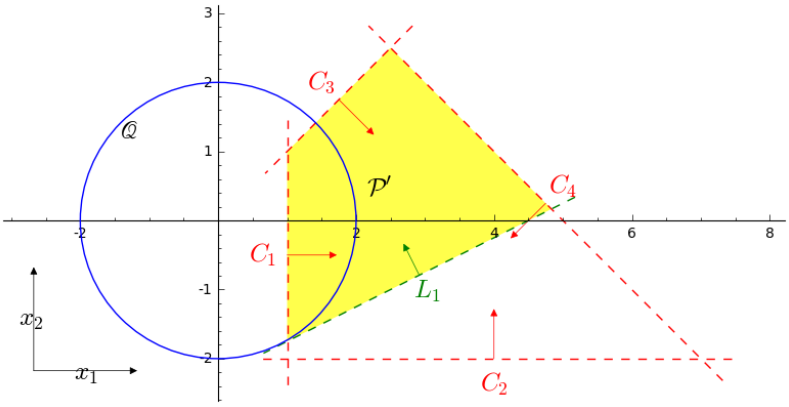
Example 4.15 (follows 4.14)

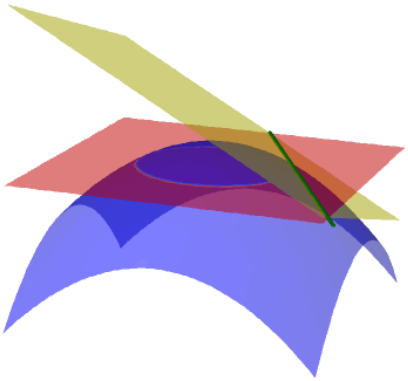
Here is the matrix $\mathcal{H}_Q$ associated to $Q \stackrel{\text{def}}{=} 4 - x_1^2 - x_2^2$ and the Handelman products from Example 4.14 with respect to $\boldsymbol{\mathcal{M}} = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2]$.

	Q	H^{I_1}	H^{I_2}	H^{I_3}	H^{I_4}	H^{I_5}	H^{I_6}	H^{I_7}	H^{I_8}	H^{I_9}	$H^{I_{10}}$	$H^{I_{11}}$	$H^{I_{12}}$	$H^{I_{13}}$	$H^{I_{14}}$	$H^{I_{15}}$
1	4	1	-1	2	0	5	1	4	0	25	-2	0	-5	0	10	0
x_1	0	0	1	0	1	-1	-2	0	0	-10	2	-1	6	2	-2	5
x_2	0	0	0	1	-1	-1	0	4	0	-10	-1	1	1	-2	3	-5
x_1x_2	0	0	0	0	0	0	0	0	-2	2	1	-1	-1	1	-1	0
x_1^2	-1	0	0	0	0	0	1	0	1	1	0	1	-1	0	0	-1
x_2^2	-1	0	0	0	0	0	0	1	1	1	0	0	0	-1	-1	1

The choices $\lambda_Q = \lambda_6 = \lambda_7 = 1$ and every other $\lambda_\ell = 0$ are a solution to the problem $\mathcal{H}_Q \cdot \boldsymbol{\lambda} = (\alpha_0, \alpha_1, \alpha_2, 0, 0, 0)^\intercal$. We obtain $\mathcal{H}_Q \cdot \boldsymbol{\lambda} = (9, -2, 4, 0, 0, 0)^\intercal$ that corresponds to $9 - 2x_1 + 4x_2 + 0 \times x_1x_2 + 0 \times x_1^2 + 0 \times x_2^2$. Thus, $f = 9 - 2x_1 + 4x_2$ is a constraint that bounds Q on $\mathcal{P}$, as shown on Example 4.16.

Example 4.16 (follows 4.15)





The figure above represents the cut at $z = 0$ of the figure on the left hand side, where z is the vertical axis, in which we added the polyhedron $\mathcal{P} = \{x_1 - 1 \geq 0,\ x_2 + 2 \geq 0,\ x_1 - x_2 \geq 0,\ -x_1 - x_2 + 5 \geq 0\}$. The circle $\mathcal{Q}$ appears in 3D as the intersection of the surface $z = Q(x_1, x_2) \stackrel{\text{def}}{=} 4 - x_1^2 - x_2^2$ (the blue curve) with the plane $z = 0$ (the red one). The polyhedral approximation of Q is the inclined yellow plane $z = f(x_1, x_2) \stackrel{\text{def}}{=} -2x_1 + 4x_2 + 9$ that dominates Q. It cuts the plane $z = 0$ along the green line L_1, which is reported in 3D. The line L_1 is the frontier of the affine constraint $-2x_1 + 4x_2 + 9 \geq 0$. The filled area is the polyhedron $\mathcal{P} \wedge (-2x_1 + 4x_2 + 9 \geq 0)$ that over-approximates $\mathcal{P} \wedge (Q(x_1, x_2) \geq 0)$.

Any solution $\boldsymbol{\lambda}$ of the problem $\mathcal{H}_Q \cdot \boldsymbol{\lambda} = (\alpha_0, \ldots, \alpha_n, 0, \ldots, 0)^\intercal$ is a polyhedral constraint f that bounds Q on $\mathcal{P}$. Among all these solutions we are only interested in the best approximations. One constraint $f_i > Q$ is better than another $f_j > Q$ at point $\boldsymbol{x}$ if $f_i(\boldsymbol{x}) < f_j(\boldsymbol{x})$. It then appears that for a given point $\boldsymbol{x}$ we are looking for the polyhedral constraint $f > Q$ that minimizes its value on that point. Therefore, we define a linear minimization problem that depends on some parameters: the point $\boldsymbol{x}$ of evaluation.

Finally, finding the tightest affine forms f_i that bound Q on $\mathcal{P}$ with respect to a given set of indices $\mathcal{I}$ can be expressed as the PLOP given below. As said in §3.3, the solution of a PLOP is a function mapping parameters to optimal values for decision variables $\boldsymbol{\lambda}$. For (PLOP 4.9), the solution is a function associating an affine form f_i (corresponding to a point $\boldsymbol{\lambda}$) to the region of the parameter space where f_i is optimal. The over-approximation of $\mathcal{P} \wedge (Q \geq 0)$ that we return is then $\mathcal{P} \bigcap_i f_i(\boldsymbol{x}) \geq 0$.

$$\begin{aligned} \textbf{minimize } \mathbf{Z}(\boldsymbol{x}) &\stackrel{\text{def}}{=} \alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n \stackrel{\text{def}}{=} Q + \sum_{\ell=1}^{q} \lambda_\ell H^{\boldsymbol{I_\ell}} \\ \textbf{subject to } & \mathcal{H}_Q \cdot (\lambda_Q, \lambda_1, \ldots, \lambda_q)^\intercal = (\alpha_0, \ldots, \alpha_n, 0, \ldots, 0)^\intercal \\ & \lambda_Q = 1, \lambda_\ell \geq 0, \ \forall \ell \in \{1, \ldots, q\} \end{aligned} \tag{PLOP 4.9}$$

where

(1) $\lambda_1, \ldots, \lambda_q$ are the decision variables of the PLOP

(2) $x_1, \ldots, x_n$ are the parameters

(3) $\alpha_0, \ldots, \alpha_n$ are kept for the sake of presentation, in practice they are substituted by their expression issued from $\mathcal{H}_Q \cdot \boldsymbol{\lambda}$.

Example 4.17 (follows 4.16)

In our running example, the objective f, *i.e.* $Q + \sum_{\ell=1}^{\ell=15} \lambda_\ell H^{\boldsymbol{I_\ell}}$, is

$$4 + \lambda_1 + \lambda_2(x_1 - 1) + \lambda_3(2 + x_2) + \lambda_4(x_1 - x_2) + \lambda_5(5 - x_1 - x_2) + \lambda_6(1 - 2x_1) + \lambda_7(4 + 4x_2)$$

$$+\lambda_9(25 - 10x_1 - 10x_2) + \lambda_{10}(2x_1 - x_2 - 2) + \lambda_{11}(x_2 - x_1) + \lambda_{12}(6x_1 + x_2 - 5) + \lambda_{13}(2x_1 - 2x_2)$$

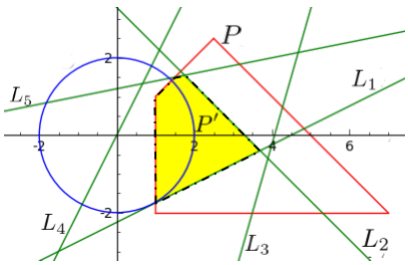
$$+\lambda_{14}(10 - 2x_1 + 3x_2) + \lambda_{15}(5x_1 - 5x_2).$$

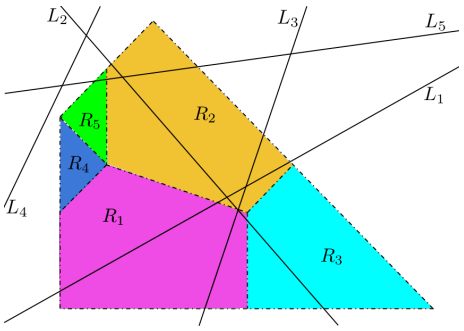
In practice we use this presentation (without α) which exhibits the parametric coefficients in x_1, x_2 of each variable λ_ℓ. Nonlinear monomials do not appear since the problem imposes the non-linear part of $Q + \sum_{\ell=1}^{\ell=15} \lambda_\ell H^{\boldsymbol{I_\ell}}$ to be canceled, *i.e.*

$$\begin{aligned} & x_1 x_2(-2\lambda_8 + 2\lambda_9 + \lambda_{10} - \lambda_{11} - \lambda_{12} + \lambda_{13} - \lambda_{14}) \\ + \ & x_1^2(-1 + \lambda_6 + \lambda_8 + \lambda_9 + \lambda_{11} - \lambda_{12} - \lambda_{15}) \\ + \ & x_2^2(-1 + \lambda_7 + \lambda_8 + \lambda_9 - \lambda_{13} - \lambda_{14} + \lambda_{15}) \end{aligned}$$

The solutions of the problem are the vectors $\boldsymbol{\lambda}$ that minimize the objective and cancel the coefficients of $x_1 x_2$, x_1^2 and x_2^2.

Example 4.18 (follows 4.17)





The solution of (PLOP 4.9) instantiated on our running example is a decision tree

with five optimal solutions $\boldsymbol{\lambda}$ at leaves:

$$z^\star \stackrel{\text{def}}{=} (x_1, x_2) \to \begin{cases} \boldsymbol{Z}_1^\star : & 4x_1 & -2x_2 & +9 & \textit{on } \mathcal{R}_1 \\ \boldsymbol{Z}_2^\star : & -5x_1 & -5x_2 & +\frac{33}{2} & \textit{on } \mathcal{R}_2 \\ \boldsymbol{Z}_3^\star : & 4x_1 & -14x_2 & +57 & \textit{on } \mathcal{R}_3 \\ \boldsymbol{Z}_4^\star : & -2x_1 & +4x_2 & & \textit{on } \mathcal{R}_4 \\ \boldsymbol{Z}_5^\star : & -5x_1 & +x_2 & +6 & \textit{on } \mathcal{R}_5 \end{cases}$$

where

$$\begin{array}{lllllllllll} \mathcal{R}_1 = & x_2 > -2 & \wedge & x_1 > 1 & \wedge & x_1 < 4 & \wedge & 2x_1 + 6x_2 < 5 & \wedge & 2x_1 - 2x_2 > 3 \\ \mathcal{R}_2 = & 2x_1 + 6x_2 > 5 & \wedge & -2x_1 + 2x_2 > -9 & \wedge & x_1 + x_2 < 5 & \wedge & x_1 - x_2 > 0 & \wedge & 4x_1 > 7 \\ \mathcal{R}_3 = & 2x_1 - 2x_2 > 9 & \wedge & x_1 + x_2 < 5 & \wedge & x_2 > -2 & \wedge & x_1 > 4 & & \\ \mathcal{R}_4 = & x_1 + x_2 < 2 & \wedge & -2x_2 + 2x_2 > -3 & \wedge & x_1 > 1 & & & & \\ \mathcal{R}_5 = & x_1 + x_2 > 2 & \wedge & x_1 - x_2 > 0 & \wedge & 4x_1 < 7 & & & & \end{array}$$

Remark that all regions are open (*i.e.* defined only with strict inequalities), for the reasons given in §3.5. Each of the five solutions $\boldsymbol{Z}_i^\star$ is interpreted as constraint $\boldsymbol{Z}_i^\star \geq 0$. These five constraints appear as the lines L_1 to L_5 in the figures above, where $L_i \stackrel{\text{def}}{=} [\boldsymbol{Z}_i^\star = 0]$. Their conjunction with $\mathcal{P}$ forms the polyhedron $\mathcal{P}'$ which over-approximates $\mathcal{P} \wedge (Q \geq 0)$. $\mathcal{P}'$ is delimited by $\mathcal{P}$ and the constraints $\boldsymbol{Z}_1^\star, \boldsymbol{Z}_2^\star$ and $\boldsymbol{Z}_5^\star$ returned by the parametric simplex; $\boldsymbol{Z}_3^\star$ and $\boldsymbol{Z}_4^\star$ are redundant.

4.4.2.2 Normalizing the PLOP

To avoid generating redundant constraints like $\boldsymbol{Z}_3^\star$ and $\boldsymbol{Z}_4^\star$ in Example 4.18, we can add a normalization constraint in the PLOP encoding, as we did for projection and convex hull in Chapter 3. The normalization constraint is defined by forcing the objective function to evaluate to 1 at a point $\mathring{\boldsymbol{x}}$ in the interior of the result, *i.e.*

$$Q(\mathring{\boldsymbol{x}}) + \sum_{\ell=1}^{q} \lambda_\ell H^{\boldsymbol{I_\ell}}(\mathring{\boldsymbol{x}}) = 1, \text{ for some } \mathring{\boldsymbol{x}} \in [\![\mathring{\mathcal{P}}']\!] \qquad (\maltese)$$

where $\mathcal{P}'$ is the approximation of $\mathcal{P} \wedge (Q \geq 0)$, *i.e.* $\mathcal{P}' = \mathcal{P} \bigsqcap_i (\boldsymbol{Z}_i^\star \geq 0)$. However, we meet two obstacles in normalizing the PLOP of Handelman's linearization:

(1) It is not that easy to find a point within $[\![\mathring{\mathcal{P}}']\!]$

(2) The constant term 1 of the normalization constraint is not suitable for Handelman's linearization.

Finding a Normalization Point. Finding a point within $[\![\mathring{\mathcal{P}}']\!]$ is not as trivial as it was for projection or convex hull. Here, $\mathcal{P}'$ is not an over-approximation of the input one $\mathcal{P}$: it is precisely the opposite. Thus, a point inside $\mathring{\mathcal{P}}$ has no guarantee to lie in $[\![\mathring{\mathcal{P}}']\!]$. The trick to determine a normalization point is to notice that the PLOP encoding of Handelman's linearization actually corresponds to a projection. To see that, let us call $H_{lin}^{\boldsymbol{I_k}}$ the Handelman product $H^{\boldsymbol{I_k}}$ where all nonlinear monomials have been renamed by a new variable. For instance, renaming x_1^2 as x_3 and x_1x_2 as x_4, the Handelman product $H^{\boldsymbol{I_{12}}} = (x_1-1)(-x_1-x_2+5)$ from Example 4.14 becomes $H_{lin}^{\boldsymbol{I_{12}}} = -x_3-x_4+6x_1+x_2-5$. We claim that eliminating these new variables by projection leads to the same result as (PLOP 4.9). Indeed, this PLOP is designed to cancel the coefficient of nonlinear monomials, which can be seen as extra variables to eliminate by projection. The polyhedron we are projecting here is $\mathcal{P} \bigsqcap_{\ell=1}^{q} H_{lin}^{\boldsymbol{I_\ell}} \geq 0$. As for projection (see §3.4.1 p.60), we can find a point of the result $[\![\mathring{\mathcal{P}}']\!]$ by picking up $\boldsymbol{x} \in [\![\mathring{\mathcal{P}}]\!] \bigcap_{\ell=1}^{q} H_{lin}^{\boldsymbol{I_\ell}} > 0$ and delete the non-pertinent coefficients (from variables corresponding to nonlinear monomials).

Finding a Suitable Constant. The constant 1 of the normalization constraint was chosen arbitrarily. Any positive value would have been correct for normalizing projection or convex-hull, since any constraint of the result can be scaled accordingly. This does not work for Handelman's linearization, because variable λ_Q is fixed to 1 instead of being only specified as nonnegative like others λ_i's. Indeed, encoding (PLOP 4.9) is designed to produce affine forms f_i such that

$$f_i(\boldsymbol{x}) > Q(\boldsymbol{x}),\ \forall \boldsymbol{x} \in \mathcal{P} \qquad (\star)$$

But if $Q(\mathring{\boldsymbol{x}}) > 1$, then there exists no affine form that fulfills both (‡) and ($\star$).
To fix this issue, we could replace the constraint $\lambda_Q = 1$ by $\lambda_Q > 0$ in (PLOP 4.9). Then, f would no longer dominate Q on $\mathcal{P}$. Instead, we would get $f > \lambda_Q Q$ for some $\lambda_Q > 0$. In this case, $f \geq 0$ would still give a correct over-approximation of $Q \geq 0$. Note that it is wrong if $\lambda_Q = 0$, hence the strict positivity required for λ_Q.

4.4.3 Heuristics and Certificates

We previously assumed a given set of Handelman products to be considered in (PLOP 4.9); our implementation actually uses Schweighofer products ($S^{\boldsymbol{I}}$), which generalize Handelman's ones as shown by Theorem 4.4 below. We shall now describe the oracle that generates the products together with a certificate of nonnegativity, then the heuristics it uses.

Theorem 4.4 (Schweighofer, 2001) *Let $\mathcal{P} = \{C_1 \geq 0, \ldots, C_p \geq 0\}$ be a polytope where each C_i is an affine form over $\boldsymbol{x} \in \mathbb{Q}^n$. Let $Q_{p+1}, \ldots, Q_q \in \mathbb{Q}[X]$. Then $Q > 0$ on $\mathcal{P} \wedge \{Q_{p+1} \geq 0, \ldots, Q_q \geq 0\}$ if and only if*

$$Q = \lambda_0 + \sum_{\boldsymbol{I} \in \mathbb{N}^q} \lambda_{\boldsymbol{I}} \cdot S^{\boldsymbol{I}}, \quad \lambda_0 \in \mathbb{Q}^{*+},\ \lambda_{\boldsymbol{I}} \in \mathbb{Q}^+$$

where $S^{(i_1,\ldots,i_q)} = C_1^{i_1} \cdots C_p^{i_p} \cdot Q_{p+1}^{i_{p+1}} \cdots Q_q^{i_q}$.

Schweighofer products are products of polyhedral constraints of $\mathcal{P}$ and polynomials $(Q_i)_{i=p+1}^{i=q}$. They are obviously nonnegative on the set $\mathcal{P} \wedge \{Q_{p+1} \geq 0, \ldots, Q_q \geq 0\}$. From a certification point of view, the key property of the polynomials resulting from Handelman or Schweighofer products is their nonnegativity on the input polyhedron. Therefore, heuristics must attach to each product a nonnegativity certificate as its representation in the OCaml/Coq type nonNegCert given below. The Coq checker, detailed in Chapter 9, contains the proof that this type only yields nonnegative polynomials by construction.

`type nonNegCert = C of` $\mathbb{N}$	*with*	$[\![C(i)]\!]$	$=$	$C_i \geq 0$ *of* $\mathcal{P}$
| Square of polynomial		$[\![\mathsf{Square}\,(p)]\!]$	$=$	$p^2 \geq 0 \quad \forall p \in \mathbb{Q}[X]$
| Power of $\mathbb{N} *$ nonNegCert		$[\![\mathsf{Power}\,(n, S)]\!]$	$=$	S^n *with* $S \geq 0$
| Product of nonNegCert list		$[\![\mathsf{Product}\,(L)]\!]$	$=$	$\Pi_{S \in L}\ S \geq 0$

Design of the Oracle. The oracle treats the input polynomial Q as the set $\mathscr{M}$ of its nonlinear monomials and maintains a set $\mathscr{M}_C$ of already-canceled monomials. Each heuristic looks for a monomial m in $\mathscr{M}$ it can apply to, checks that it doesn't belong to $\mathscr{M}_C$ and generates a product S or H for it. Monomial m is then added to $\mathscr{M}_C$ and the nonlinear monomials of S that are different from $-m$ are added to $\mathscr{M}$. The oracle finally returns a list of couples formed of a product H or S. The heuristics are applied according to their priority. The most basic of them consists in taking every Handelman product whose degree is smaller than or equal to that of Q. If solving (PLOP 4.9) fails with these products, we increase the maximum degree up to which all the products are considered. Theorem 4.3 ensures eventual success. However, the number of products quickly becomes so large that this heuristic is used as a last resort.

Targeted Heuristics. The following heuristics aim at finding either Handelman products $H^{\boldsymbol{I}}$ or Schweighofer products $S^{\boldsymbol{I}}$ which cancel a given nonlinear monomial m. Besides a monomial canceling m, a product may contain nonlinear monomials which need to be eliminated. The heuristics guarantee that these monomials are of smaller degree than m when the polyhedron is bounded, thereby ensuring termination. Otherwise, they try to limit the degree of these additional monomials as much as possible, so as to make them easier to cancel. As before, we consider an input polyhedron $\{C_1 \geq 0, \ldots, C_p \geq 0\}$ with $C_i : \boldsymbol{a_i x} \geq b_i$. In the following, we wish to cancel monomial $m \stackrel{\text{def}}{=} c_m \times x_1^{e_1} \cdots x_n^{e_n}$, with $c_m \in \mathbb{Q}$.

Extraction of Even Powers. This heuristic builds on squares being always nonnegative to apply Schweighofer's theorem in an attempt to simplify the problem. The idea is to rewrite m into $m = m' \times (x_1^{\epsilon_1} \ldots x_n^{\epsilon_n})^2$ where $m' \stackrel{\text{def}}{=} c_m \times x_1^{\delta_1} \ldots x_n^{\delta_n}$, with $\delta_j \in \{0, 1\}$. The heuristic recursively calls the oracle in order to find a product S canceling m'. Then, $S \times (x_1^{\epsilon_1} \ldots x_n^{\epsilon_n})^2$ cancels the monomial m. If W_S is the nonnegativity certificate for S, then $\mathsf{Product}\,[W_S; \mathsf{Square}\,(x_1^{\epsilon_1} \ldots x_n^{\epsilon_n})]$ is that of the product.

Simple Products. Consider a monomial $m = c_m \times x_1 \cdots x_n$ where $c_m \in \mathbb{Q}$, as can be produced by the previous heuristic. We aim at finding a Schweighofer product S that cancels m, and such that every other monomial of S has a degree strictly smaller than that of m. We propose an analysis based on intervals, expressing S as a product of *variable bounds*, *i.e.* $x_j \in [l_j, u_j]$ where $l_j,\ u_j \in \mathbb{Q}$. For each variable x_j, we may choose either constraint $x_j + l_j \geq 0$ or $-x_j + u_j \geq 0$, so that the product of the chosen constraints contains $x_1 \cdots x_n$ with the appropriate sign. Moreover, other monomials of this product are ensured to have a degree smaller than that of m. The construction of a product of bounds is guided by the following concerns (for the full details, see Appendix A).

- The sign of the canceling monomial must be opposite to that of m.
- The bounds that are available in the input constraints are used in priority. It is possible to call the VPL to deduce additional bounds on any variable from the input constraints. However, finding a new bound requires solving a LP problem.
- The selected bounds should exist, which is not necessarily the case if the input polyhedron is not a polytope. If too many bounds don't exist, the heuristic fails.

Thanks to Farkas' lemma, each implied bound on a variable ($x_j + l_j$ or $-x_j + u_j$) can be expressed as a nonnegative linear combination of the input constraints, *i.e.* $\sum_{i=1}^{p} \beta_{ij} C_i$ for some $\beta_{ij} \geq 0$ solutions of a linear problem. The combination reduces to C_i if C_i is already a constraint of the input polyhedron P. The resulting product of bounds can then be expressed as follows.

$$\prod_{j \in L} (x_j + l_j) \times \prod_{j \in U} (-x_j + u_j) = \prod_{j \in L \cup U = \{1,\ldots,n\}} \Big(\sum_{i=1}^{p} \beta_{ij} \cdot C_i\Big), \qquad \beta_{ij} \geq 0$$

The right-hand side expression is then refactorized with the C_i's kept symbolic, so that the Handelman products appear. This case is illustrated in the following example.

Example 4.19

We illustrate the behavior of the oracle on the polynomial $Q = x_2^2 - x_1^2 x_2 + x_1 x_2 - 85$ and still the same polytope

$$\mathcal{P} = \{C_1 : x_1 - 1 \geq 0,\ C_2 : x_2 + 2 \geq 0,\ C_3 : x_1 - x_2 \geq 0,\ C_4 : 5 - x_1 - x_2 \geq 0\}.$$

The oracle starts with $\mathscr{M} = \{x_1 x_2, -x_1^2 x_2, x_2^2\}$ and processes the monomials in order.

($x_1 x_2$) For eliminating $x_1 x_2$, the simple product heuristic uses constraint $C_1 : x_1 - 1 \geq 0$ and the combination $C_1 + C_4 = (x_1 - 1) + (-x_1 - x_2 + 5)$ which entails the upper bound $-x_2 + 4 \geq 0$ on x_2. Their product $(x_1 - 1)(-x_2 + 4) = -x_1 x_2 + 4x_1 + x_2 - 4$ cancels $x_1 x_2$

and the expansion $C_1 \cdot (C_1 + C_4) = C_1^2 + C_1 C_4$ reveals the useful Handelman products: $H_1 \stackrel{\text{def}}{=} C_1^2 = x_1^2 - 2x_1 + 1$ and $H_2 \stackrel{\text{def}}{=} C_1 C_4 = -x_1^2 - x_1 x_2 + 6x_1 + x_2 - 5$. They are returned with their certificates of nonnegativity: $\mathsf{Power}\,(2, C_1)$ and $\mathsf{Product}\,[C_1; C_4]$. Then, $x_1 x_2$ is added to $\mathscr{M}_C$ as well as the new monomials x_1^2 and $-x_1^2$, which are not placed in $\mathscr{M}$ since opposite monomials cancel each other.

($-x_1^2 x_2$) The heuristic for squares splits the term $-x_1^2 x_2$ into $m' \times x_1^2$ and lets the oracle deal with $m' \stackrel{\text{def}}{=} -x_2$. The simple product heuristic reacts by looking for a constraint with the term $+x_2$ and as few variables as possible: $C_2 : x_2 + 2 \geq 0$ fulfills these criteria. The calling heuristic builds the Schweighofer product $S_3 \stackrel{\text{def}}{=} x_1^2 \cdot C_2 = x_1^2 x_2 + 2x_1^2$ that cancels $-x_1^2 x_2$, and returns S_3 with its certificate of nonnegativity $\mathsf{Product}\,[\mathsf{Square}\,(x_1); C_2]$. Then, the oracle removes $x_1^2 x_2$ from the working set and places it into the set of cancelled monomials.

(x_2^2) The heuristic on squares cannot produce $x_2^2 \times (-1)$ with a certificate of nonnegativity for -1. The last heuristic is then triggered and finds two Handelman's products that generate $(-x_2^2)$: $H_4 \stackrel{\text{def}}{=} C_2 C_3 = (x_2 + 2)(x_1 - x_2) = x_1 x_2 - x_2^2 + 2x_1 - 2x_2$ and $H_5 \stackrel{\text{def}}{=} C_2 C_4 = (x_2 + 2)(5 - x_1 - x_2) = 5x_2 - x_1 x_2 - x_2^2 + 10 - 2x_1 - 2x_2$. H_4 is preferred since it does not introduce a new monomial – indeed $x_1 x_2 \in \mathscr{M}_C$ – whereas H_5 would add $-x_2^2$ to the working set $\mathscr{M}$.

Finally the oracle returns the four polynomials with their certificates. The expanded forms of H_1, H_2, S_3, H_4 are installed in the matrix $\mathcal{H}_Q$ and each of them is associated with a decision variable $\lambda_1, \ldots, \lambda_4$. The parametric simplex computes all the nonnegative, minimal, affine constraints f of the form $1 \cdot Q + \lambda_1 \cdot H_1 + \lambda_2 \cdot H_2 + \lambda_3 \cdot S_3 + \lambda_4 \cdot H_4$. With such few products, it returns only one affine constraint $f = Q + 2H_2 + S_3 + H_4 = 13x_1 + x_2 - 95$ from which we build a polyhedral over-approximation of the set $\mathcal{P} \wedge (Q \geq 0)$ as $\mathcal{P} \sqcap f \geq 0$. The VPL reveals that this polyhedron is empty, meaning that $\mathcal{P} \wedge (Q \geq 0)$ is unsatisfiable.

4.4.4 Experiments in Satisfiability Modulo Theory

Handelman's linearization is now part of the VPL. It is split into two parts: an OCaml oracle, defined in the previous section, uses heuristics to select the most promising Handelman-Schweighofer products $S_1, \ldots, S_q$, then it runs the parametric simplex to find coefficients $\lambda_1, \ldots, \lambda_q$ such that $Q + \sum \lambda_i S_i$ is affine. The result is fed into a checker implemented and proved correct in Coq. The architecture of the checker is detailed in Chapter 9.

Increasing precision. We show on Fig. 4.20 the results of Handelman's linearization on the running example. We chose the subset $\{H^{I_1}, \ldots, H^{I_{15}}\}$ from Example 4.14, meaning that we are faced with a 15-variables LP problem. Precision can be increased without degree elevation by iterating Handelman's linearization (HL): $\mathcal{P}_0 = \mathcal{P}$, $\mathcal{P}_{i+1} = \mathsf{HL}\,(\mathcal{P}_i, Q \geq 0)$. The linearization operator of the VPL computes this sequence until reaching a fixpoint, *i.e.* $\mathcal{P}_{k+1} = \mathcal{P}_k$, or a time limit. The sequence is decreasing with respect to inclusion since $\mathsf{HL}\,(\mathcal{P}_i, Q \geq 0) = \mathcal{P}_i \bigsqcap_i f_i \geq 0$ is by construction included in $\mathcal{P}_i$.

Evaluation on Satisfiability Modulo Theory. Although our contribution applies to static analysis, we met troubles in finding real programs where precise nonlinear approximations play a decisive role in analyses. Therefore, we performed our experimental evaluation on Satisfiability Modulo Theory (SMT) solvers, because the SMT community has a standard set of nonlinear benchmarks from SMT-LIB, which the static analysis community lacks. The satisfiability of a quantifier-free formula of first-order linear arithmetic over the reals is usually decided by a "DPLL(T)" (Ganzinger et al., 2004) combination of a propositional solver and a decision procedure for conjunctions of linear inequalities based on the simplex algorithm (Dutertre and de Moura, 2006a,b). Nonlinear formulas are more challenging; some solvers implement a variant of cylindrical algebraic decomposition, a very complex and costly approach (Jovanovic and de Moura, 2012); some replace the propositional abstraction of DPLL(T) by a direct search for a model (de Moura and Jovanovic, 2013).

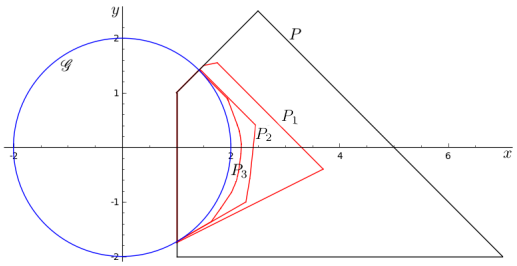


Figure 4.20 – The polytopes resulting from three iterations of Handelman's linearization: $\mathcal{P}_0 = \mathcal{P},\ \mathcal{P}_i = \mathsf{HL}\,(\mathcal{P}_{i-1}, 4 - x^2 - y^2 \geq 0)$. $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ are respectively composed of 5, 9 and 36 constraints.

Showing emptiness of semialgebraic sets. A SMT solver for nonlinear real arithmetic using the DPLL(T) architecture enumerates conjunctions of nonlinear inequalities, each of which having to be tested for satisfiability. We show the unfeasibility of the conjunction of affine constraints $C_1 \geq 0, \ldots, C_p \geq 0$ and nonlinear ones $Q_1 \geq 0, \ldots, Q_q \geq 0$ by computing the sequence of approximations: $\mathcal{P}_0 = \{C_1 \geq 0, \ldots, C_q \geq 0\},\ \mathcal{P}_{i+1} = \mathsf{HL}\,(\mathcal{P}_i, Q_i \geq 0)$. Polynomials are added one after the other, meaning that Q_{i+1} is linearized with respect to the previous polyhedral approximation $\mathcal{P}_i$. If at some point $\mathcal{P}_k = \emptyset$, it means that the conjunction is unsatisfiable, as our approximation is sound. Otherwise, as it is not complete, we cannot state on the satisfiability. Such a procedure can thus be used to soundly prune branches in DPLL(T) search. Furthermore, the subset of constraints appearing in the products used in the emptiness proof is unsatisfiable, and thus the negation of its conjunction may be used as a learned clause.

We evaluated Handelman's linearization with conjunctions arising from deciding formulas from the Quantifier-Free Nonlinear Real Arithmetic (QF_NRA) benchmark, from SMT-LIB 2014 (Barrett et al., 2010). These conjunctions, that we know to be unsatisfiable, are mostly coming from approximations of transcendental functions as polynomial expressions. We added Handelman's linearization as a theory solver for the SMT solver CVC4 (Deters et al., 2014). The calls to our linearization follow a factorization step, where for instance polynomial guards such as $x_1^2 - x_2^2 \geq 0$ are split into two cases ($x_1 + x_2 \geq 0 \wedge x_1 - x_2 \geq 0$ and $x_1 + x_2 \leq 0 \wedge x_1 - x_2 \leq 0$), in order to provide more constraints to the input polyhedron.

The comparison of our contribution with the state of the art SMT solvers Z3 (de Moura and Bjørner, 2008), Yices2 (Dutertre, 2014), SMT-RAT (Corzilius et al., 2012) and raSat (Khanh et al., 2014) was done on the online infrastructure StarExec (Stump et al., 2014). Fig. 4.21 is a cactus plot showing the number of benchmarks proved unsatisfiable depending on time. Table 4.22 shows the summary for each SMT solver on the same benchmarks. Both illustrate that linearization based on Handelman's representation, implemented as a non-optimized prototype, gives fast answers and that its results are precise enough in many cases. Note that our approach also provides an easy-to-verify certificate, as opposed to the cylindrical algebraic decomposition implemented in Z3 for example. Indeed, if the answer of the VPL is that the final polyhedral approximation is empty, then the nonzero coefficients in the solution $\boldsymbol{\lambda}$ of (PLOP 4.9) give a list of sufficient Schweighofer products. Together with the nonlinear guards, the conjunctions of the original constraints involved in these products are actually sufficient for emptiness. As mentioned above, in a SMT solver the negation of this conjunction may be used as a *learned theory lemma*. However, due to engineering issues we have not been able to fully integrate this procedure into CVC4 by sending back minimized learned lemmas. Over a total of 4898 benchmarks, adding our method (represented in the figure as curve *cvc4+vpl*) allows CVC4 to show the unsatisfiability of 1030 more problems. Failure in showing emptiness may come from strict constraints since up to now, our solver considers each inequality as nonstrict.

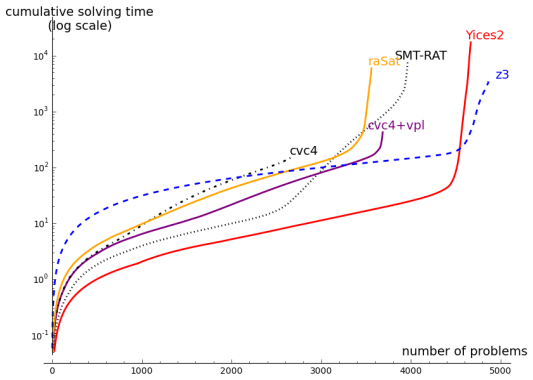


Figure 4.21 – Comparison between CVC4+VPL and other SMT solvers on Quantifier-Free Nonlinear Real Arithmetic benchmarks.

Table 4.22 – Summary of SMT solvers results

SMT solver	cvc4	cvc4 +VPL	raSat	SMT-RAT	Yices2	z3
Number of unsat found	2657	3687	3561	3965	4669	4864
total execution time (s)	148	425	5936	7639	17356	3463

4.5 Linearization in the VPL

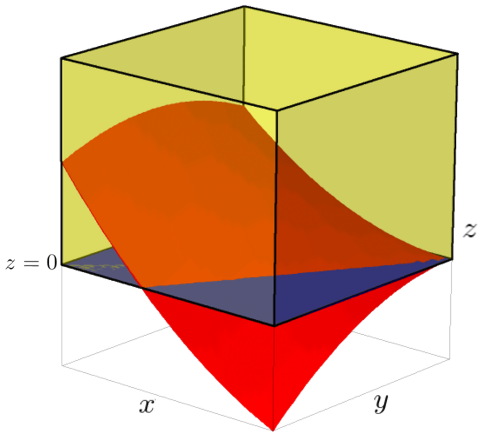
We presented three operators for approximating the effect of a polynomial guard on a polyhedron:

1. A variant of *intervalization*, that replaces some variables of nonlinear products by intervals, themselves eliminated by sign partitioning;
2. *Bernstein's linearization*, where the coefficients of the polynomial expressed in the Bernstein basis gives a bounding polyhedron;
3. *Handelman's linearization*, finding dominating affine forms by looking at a Handelman representation of the polynomial.

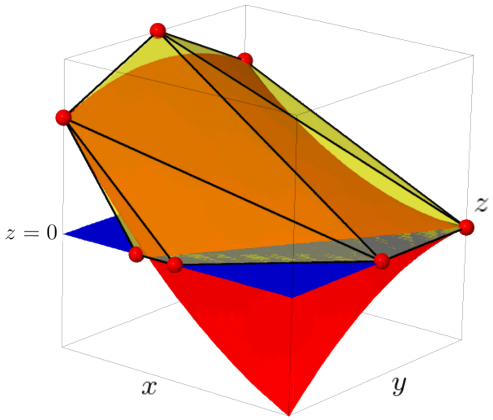
Only approaches (1) and (3) are implemented in the VPL, but the three of them were initially tested as prototypes in Sage. They give results with different shape, as illustrated on Fig. 4.23. Figure (a) shows a constant intervalization that bounds Q in a box. Figure (b) shows a part of the regular mesh on which Bernstein coefficients are placed. Figure (c) shows the concave piecewise affine shape of the approximation, resulting from the use of a PLP solver.

Our linearization operators are directly usable in abstract interpretation: besides linear expressions, the VPL now accepts polynomials as well. Apart from handmade examples, we actually did not find programs manipulating integers where the linearization improves the global analysis result: nonlinearity is too sparse in such programs. We believe that linearization could have an impact on the analysis of floating-point computations where polynomials appear more naturally in programs for approximating transcendental functions and in the analysis of the round-off errors (Boland and Constantinides, 2011).

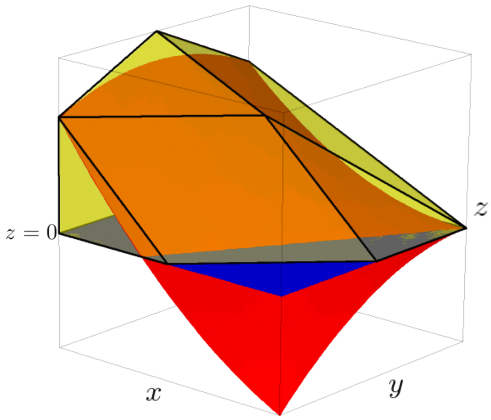
Handelman's linearization already proved to be useful in satisfiability modulo theory solving. A simple coupling of the VPL with the competitive SMT solver cvc4 improved notably the performance of that solver on nonlinear arithmetic. In contrast to cylindrical algebraic decomposition, which is a complete approach, our method may fail to prove a true property. However, it provides easy-to-check certificates for its results.



(a) Intervalization



(b) Bernstein's linearization



(c) Handelman's linearization

Figure 4.23 – Approximations using the three linearization techniques applied on the polynomial guard $x^2 + xy + y^2 \geq 0$.

Even if it is less precise than Handelman's linearization, intervalization is still useful. Thanks to its fast execution time, it can be used as a first linearization step. If lucky, it could discard by itself an unsatisfiable nonlinear guard. Otherwise, it can still provide additional affine constraints for Handelman's linearization.

Chapter 5

Parametric Linear Programming Problem Solving

In §1.3, we introduced Linear Programming (LP), that focuses on problems of the form

$$\begin{aligned} &\textbf{minimize } \mathbf{Z} \stackrel{\text{def}}{=} \boldsymbol{c}^{\intercal}\boldsymbol{\lambda} \\ &\quad\textbf{subject to } \quad \boldsymbol{A\lambda} \geq \boldsymbol{b} \end{aligned}$$

It consists in minimizing the linear objective function $\boldsymbol{c}^{\intercal}\boldsymbol{\lambda} = \sum_{i=1}^{p} c_i\lambda_i$ over polyhedron $\boldsymbol{A\lambda} \geq \boldsymbol{b}$. In §1.3.2, we detailed the simplex algorithm for solving LP problems. Further, in Chapter 3 and §4.4, we encoded some polyhedral operators as Parametric Linear Optimization Problems (PLOP). For its part, a PLOP manipulates two kinds of indeterminates: *decision variables* denoted by $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_p)$ and *parameters* denoted by $\boldsymbol{x} = (x_1, \ldots, x_n)$, following our notations of Chapter 3. In a PLOP, parameters can appear in the objective function or in the right hand side of constraints, but not in both.

(1) When parameters are in the right hand side of constraints, the PLOP has the form

$$\begin{aligned} &\textbf{minimize } \mathbf{Z}(\boldsymbol{x}) \stackrel{\text{def}}{=} \boldsymbol{c}^{\intercal}\boldsymbol{\lambda} \\ &\quad\textbf{subject to } \quad \boldsymbol{A\lambda} \geq \boldsymbol{b} + \boldsymbol{Dx} \end{aligned}$$

The right hand side of the system of constraints now contains the product $\boldsymbol{Dx}$, meaning that each constraint $\boldsymbol{a_i\lambda} \geq b_i$ has been changed into $\boldsymbol{a_i\lambda} \geq b_i + \boldsymbol{d_ix}$, where the bound depends on parameters. This form of PLOP called the *polytope model* (Feautrier, 1996) is used in compilation for instance to determine reachable values for indices in nested loops, depending on a parameter `N` that typically represents the size of an array. For instance, consider the following code fragment:

```
for(i=0 ; i < N ; i++){
  for(j=0 ; j < i ; j++){
    ...
  }
}
```

The set of reachable values for `(i,j)` can be represented as a $\mathbb{Z}$-polytope, that is the set of integers points of a polytope $\mathcal{P}$, *i.e.* $\mathbb{Z}^n \cap \mathcal{P}$. Some variables fixed at the beginning of the execution, such as `N`, can be considered as parameters. One could ask for the reachable values of `(i,j)` as a function of `N`. The PIP solver (for Parametric Integer Programming) of Feautrier (1988) was designed to solve such integer PLP problems.

(2) When parameters are in the objective function, the PLOP has the form

$$\begin{aligned} &\textbf{minimize } \mathbf{Z}(\boldsymbol{x}) \stackrel{\text{def}}{=} \boldsymbol{c}(\boldsymbol{x})^{\intercal}\boldsymbol{\lambda} \\ &\quad\textbf{subject to } \quad \boldsymbol{A\lambda} \geq \boldsymbol{b} \end{aligned}$$

Each coefficient c_i of the objective function has been replaced by an affine function $\boldsymbol{c_i}(\boldsymbol{x})$ of some parameters $\boldsymbol{x} = (x_1, \ldots, x_n)$. The objective function is therefore bilinear in the decision variables and in the parameters. Constraints are unchanged in this setting.

We will focus on the second form of PLOP where parameters appear in the objective function. There are two ways of extending the standard simplex algorithm to deal with parameters:

— either adapt the choice of the entering variable in the basis;

— or instantiate the parameters in the objective function with a parametric point, and generalize the optimal value to a region in space.

The first method is implemented in PIPlib; we will detail it in the next section. It gives a good introduction to PLP because it is only a small variation of the standard simplex algorithm. In this setting, the result of a PLOP is given as a decision tree, where edges are labelled with conditions on the parameters $\boldsymbol{x}$ and leaves are valuations of the decision variables $\boldsymbol{\lambda}$. So, we will call this method the *tree-exploration* parametric simplex. The PLP solver of the VPL follows the second method, that we will present in §5.2.

5.1 Tree-Exploration Parametric Simplex

Let us briefly recall how the optimization phase of the simplex algorithm works. To minimize the objective value, the algorithm looks for a variable with a negative coefficient in the objective. Since variables are assumed nonnegative, increasing this variable will decrease the objective value. In the parametric version, coefficients in the objective are affine forms of parameters, *i.e.* $\boldsymbol{c_i}(\boldsymbol{x})$. Thus the sign of these coefficients generally depends on $\boldsymbol{x}$. The solver must explore two cases for each coefficient, depending on their sign.

For instance, consider the following parametric problem:

$$\begin{aligned} \textbf{minimize } \mathbf{Z}(\boldsymbol{x}) \stackrel{\text{def}}{=} & \; x_1\lambda_1 + x_2\lambda_2 \quad \textit{i.e. } \boldsymbol{c_1}(x_1, x_2) = x_1 \text{ and } \boldsymbol{c_2}(x_1, x_2) = x_2 \\ \textbf{subject to} & \; \lambda_1 + \lambda_2 \leq 5 \\ & -\lambda_1 \leq 1 \\ & -\lambda_2 \leq 2 \\ & -\lambda_1 + \lambda_2 \leq 0 \\ & \lambda_1, \lambda_2 \geq 0 \end{aligned} \tag{PLOP 5.1}$$

Like in the standard simplex algorithm, the first step is to transform inequalities into equalities by adding slack variables. We obtain the following dictionary, which is equivalent to (PLOP 5.1) provided that slack variables $s_1, \ldots, s_4$ are nonnegative.

$$\begin{aligned} s_1 &= -\lambda_1 - \lambda_2 + 5 \\ s_2 &= \lambda_1 + 1 \\ s_3 &= \lambda_2 + 2 \\ s_4 &= \lambda_1 - \lambda_2 \\ \boldsymbol{Z} &= x_1\lambda_1 + x_2\lambda_2 \end{aligned} \tag{Dict. 5.2}$$

The basis $\mathcal{B} = \{s_1, s_2, s_3, s_4\}$ is feasible and gives the solution $(\lambda_1 = 0,\ \lambda_2 = 0,\ s_1 = 5,\ s_2 = 1,\ s_3 = 2,\ s_4 = 0)$. The value of the objective associated to this point is $x_1 \times 0 + x_2 \times 0 = 0$.

To minimize the objective value, the idea is to pick a variable that has a negative coefficient in the current objective $x_1\lambda_1 + x_2\lambda_2$ and to increase it as much as possible, such that the dictionary remains satisfiable. Thus, we look for the sign of the coefficients $\boldsymbol{c_i}(\boldsymbol{x})$. Due to its parametric nature, in general the sign of $\boldsymbol{c_i}(\boldsymbol{x})$ cannot be decided. So, we build an exploration tree which considers two cases $\boldsymbol{c_i}(\boldsymbol{x}) < 0$ and $\boldsymbol{c_i}(\boldsymbol{x}) \geq 0$. Since in our example $\boldsymbol{c_1}(\boldsymbol{x}) \stackrel{\text{def}}{=} x_1$ and $\boldsymbol{c_2}(\boldsymbol{x}) \stackrel{\text{def}}{=} x_2$, we first look at the coefficient x_1: we assume $x_1 < 0$ and obtain a branch leading to one or several optimal solutions depending on the sign of x_2. Later, we will assume $x_1 \geq 0$ and explore a second branch.

Let us execute the algorithm for the branch $x_1 < 0$ to clearly see what type of results appears. Variable λ_1 is entering the basis. As in the standard simplex algorithm, we must find the more restricting equation for the increase of λ_1; this will determine the variable leaving the basis. The limiting equation is $s_1 = 5 - \lambda_1 - \lambda_2$ which prevents λ_1 from exceeding 5. The pivot operation is the same as in standard LP: exchanging $\lambda_1 \leftrightarrow s_1$ leads to $\mathcal{B} = \{\lambda_1, s_2, s_3, s_4\}$, associated to the solution $(\lambda_1 = 5,\ \lambda_2 = 0,\ s_1 = 0,\ s_2 = 6,\ s_3 = 2,\ s_4 = 5)$ and the objective value becomes $5x_1$.

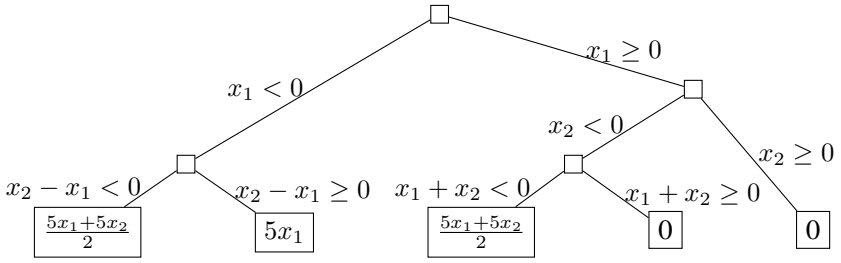
$$\begin{aligned} \lambda_1 &= -s_1 - \lambda_2 + 5 \\ s_2 &= -s_1 - \lambda_2 + 6 \\ s_3 &= \lambda_2 + 2 \\ s_4 &= -2\lambda_2 - s_1 + 5 \\ \boldsymbol{Z} &= -x_1 s_1 + (x_2 - x_1)\lambda_2 + 5x_1 \end{aligned} \qquad \text{(Dict. 5.3)}$$

For the next iteration, we look again for a variable whose coefficient is negative in the objective: $-x_1$ cannot be negative because we made the assumption $x_1 < 0$. However, none of our assumptions prevents $x_2 - x_1$ from being negative. Again, we explore two branches where the first one shall assume $x_1 < 0 \wedge x_2 - x_1 < 0$ whereas the other one shall assume $x_1 < 0 \wedge x_2 - x_1 \geq 0$. In the first of these two branches, the variable entering the basis is λ_2, and the most limiting equation is $s_4 = -2\lambda_2 - s_1 + 5$ – it imposes the upper bound $\frac{5}{2}$ on λ_2 – hence s_4 is the leaving variable. The new feasible solution is $\big(\lambda_1 = \frac{5}{2},\ \lambda_2 = \frac{5}{2},\ s_1 = 0,\ s_2 = \frac{7}{2},\ s_3 = \frac{9}{2},\ s_4 = 0\big)$, $\mathcal{B}$ becomes $\{\lambda_1, \lambda_2, s_2, s_3\}$, the objective value is $\frac{5x_1+5x_2}{2}$ and we end up with the following dictionary.

$$\begin{aligned} \lambda_1 &= -\frac{s_1}{2} + \frac{s_4}{2} + \frac{5}{2} \\ s_2 &= \frac{s_4}{2} - \frac{s_1}{2} + \frac{7}{2} \\ s_3 &= -\frac{s_4}{2} - \frac{s_1}{2} + \frac{9}{2} \\ \lambda_2 &= -\frac{s_4}{2} - \frac{s_1}{2} + \frac{5}{2} \\ \boldsymbol{Z} &= \frac{-x_1 - x_2}{2} s_1 + \frac{x_1 - x_2}{2} s_4 + \frac{5x_1 + 5x_2}{2} \end{aligned}$$

Recall that we are examining a region of the space of parameters defined by $\mathcal{R} = x_1 < 0 \wedge x_2 - x_1 < 0$. With these assumptions, neither $\frac{-x_1-x_2}{2}$ nor $\frac{x_1-x_2}{2}$ can be negative. This can be assessed by checking the unsatisfiability of $\mathcal{R} \sqcap \frac{-x_1-x_2}{2} < 0$ and $\mathcal{R} \sqcap \frac{x_1-x_2}{2} < 0$. Thus, there is no more way to improve the current objective, meaning that the optimum has been found *for this region of the parametric space*. Going back to (Dict. 5.3), the second branch assumes $x_1 < 0 \wedge x_2 - x_1 \geq 0$, thus there is no remaining decision variable with a negative coefficient in the objective $-x_1 s_1 + (x_2 - x_1)\lambda_2 + 5x_1$, and the optimum for this region is $5x_1$. The region where $x_1 < 0$ has been fully explored and leads to two different optimal solutions depending on x_2. At this point, the space where $x_1 \geq 0$ still needs to be analyzed, so the next iteration starts from (Dict. 5.2) with the assumption $x_1 \geq 0$, and computations are conducted in the same way.

Solution Shape. The parametric simplex returns a function $\boldsymbol{Z}^\star$ of the parameters $x_1, \ldots, x_n$ that associates an optimum to each region of the parametric space. The tree-exploration version of the parametric simplex explores the parametric space by building an exploration tree where edges are assumptions on the sign of parametric coefficients. The solution $\boldsymbol{Z}^\star$ can therefore be represented as a decision tree. For instance, the solution of (PLOP 5.1) is the following:



The piecewise affine solution $z^\star$ can be summarized as

$$z^*(x_1, x_2) = \begin{cases} \frac{5x_1+5x_2}{2} & \text{if } x_1 < 0 \wedge x_2 - x_1 < 0 \\ 5x_1 & \text{if } x_1 < 0 \wedge x_2 - x_1 \geq 0 \\ \frac{5x_1+5x_2}{2} & \text{if } x_1 \geq 0 \wedge x_2 < 0 \wedge x_1 + x_2 < 0 \\ 0 & \text{if } x_1 \geq 0 \wedge x_2 < 0 \wedge x_1 + x_2 \geq 0 \\ 0 & \text{if } x_1 \geq 0 \wedge x_2 \geq 0 \end{cases}$$

Our experiments tend to show that this algorithm produces many branches that lead to the same solution. It means that some regions of the parametric space are unnecessarily split on non-significant conditions. This observation suggested another exploration scheme, already noticed by Jones et al. (2007).

5.2 Algorithm by Generalization

In this section, we present the PLP solver implemented in the VPL. We will focus on normalized problems (as defined in §3.4), since this is what we need to solve in the VPL. Recall that normalization induces a particular geometry of regions: they become polyhedral cones, pointed in the normalization point $\mathring{\boldsymbol{x}}$.

Our PLP solver extends the standard simplex algorithm using an approach inspired from Jones et al. (2007). Instead of splitting the exploration at each parametric coefficient like the tree-exploration simplex does, it builds upon the fact that, by instantiating the objective function on a point $\boldsymbol{x}$ to obtain a non-parametric objective and by solving the corresponding standard LP problem, the optimal solution can be generalized to a whole region of the parametric space. Actually, the objective is kept symbolic in the dictionary, but when looking for a variable with a negative coefficient in the objective, we instantiate parametric coefficients on $\boldsymbol{x}$ to determine their sign.

For instance in (Dict. 5.2) that we recall here, we instantiate the objective $\boldsymbol{Z}(\boldsymbol{x}) = x_1\lambda_1 + x_2\lambda_2$ with $\boldsymbol{x} \stackrel{\text{def}}{=} (x_1 = -1, x_2 = 1)$, that covers the case where coefficient x_1 is negative and x_2 is positive.

$$\begin{aligned} s_1 &= -\lambda_1 - \lambda_2 + 5 \\ s_2 &= \lambda_1 + 1 \\ s_3 &= \lambda_2 + 2 \\ s_4 &= \lambda_1 - \lambda_2 \\ \boldsymbol{Z} &= x_1\lambda_1 + x_2\lambda_2 \end{aligned}$$

The coefficient of λ_1 in the instantiated objective is negative, thus the standard LP resolution

will pivot on λ_1 to obtain (Dict. 5.3):

$$\begin{aligned}
\lambda_1 &= -s_1 - \lambda_2 + 5 \\
s_2 &= -s_1 - \lambda_2 + 6 \\
s_3 &= \lambda_2 + 2 \\
s_4 &= -2\lambda_2 - s_1 + 5 \\
\boldsymbol{Z} &= -x_1 s_1 + (x_2 - x_1)\lambda_2 + 5x_1
\end{aligned}$$

Again, we instantiate parametric coefficients on $(x_1 = -1, x_2 = 1)$ and we see that none of them is negative. It means that we have reached an optimum for this particular $\boldsymbol{x}$, associated to the basis $\{\lambda_1, s_2, s_3, s_4\}$; the optimum is obtained by replacing nonbasic variables by 0 in the parametric objective. We obtain an objective $\boldsymbol{Z}^\star(\boldsymbol{x}) = 5x_1$ which is optimal at $\boldsymbol{x} = (x_1 = -1, x_2 = 1)$. In fact, the current objective value $5x_1$ is optimal for all points where its parametric coefficients are nonnegative, *i.e.* on $-x_1 \geq 0 \wedge x_2 - x_1 \geq 0$. Hence, those constraints define a whole region of the parametric space that share the same optimum $5x_1$.

In general, once the instantiated LP problem is solved, we generalize it to a new region by taking the conjunction of the sign conditions on the parameterized coefficients. If the objective function is denoted by $\sum_{i=1}^{m} c_i(\boldsymbol{x})\lambda_i$, the associated region $\mathcal{R}$ is

$$\mathcal{R} = \bigwedge_{i=1}^{m} (\boldsymbol{c_i} \geq 0)$$

This set of constraints is a polyhedron, and it can contain redundancies that we eliminate using our raytracing algorithm presented in Chapter 2.

Algorithm 5.1: PLP solver of the VPL.

Input : A parametric LP problem sx
Output: A set of regions and their associated optimal solution $Solutions = \{(\mathcal{R}_i, \boldsymbol{Z}_i^\star)\}$
Data : $instantiation\_points$: a list of tuples $(\mathcal{R}_i, \boldsymbol{f} \geq 0, \boldsymbol{w})$, where $\boldsymbol{w}$ is a witness point of $\boldsymbol{f} \geq 0$, which is a frontier of $\mathcal{R}_i$
`adjustPoint` $(\mathcal{R}, \boldsymbol{w}, regions)$: returns either ADJACENT$(\mathcal{R}, \mathcal{R}_i)$, meaning that we already know a region $\mathcal{R}_i \in regions$ such that $\mathcal{R}$ and $\mathcal{R}_i$ are adjacent; or ADJUSTED$(\boldsymbol{w'})$ where $\boldsymbol{w'}$ is a point along the ray RAY$(\mathring{\boldsymbol{x}}, \boldsymbol{w})$ – with $\mathring{\boldsymbol{x}} \in \mathring{\mathcal{R}}$ – that is not contained in any known region.

```
((R_0, Z*_0), witness_points) ← instantiateAndGeneralize (sx, 0)
instantiation_points ← witness_points
Solutions ← {(R_0, Z*_0)}
while instantiation_points ≠ [] do
    (R, f ≥ 0, w) ← head (instantiation_points)
    switch adjustPoint (R, w, regions(Solutions)) do
        case ADJACENT(R, R_i) do
            instantiation_points ←tail (instantiation_points)  /* crossing frontier f
             leads to a known region */
        case ADJUSTED(w') do
            ((R', Z*'), witness_points) ← instantiateAndGeneralize (sx, w')
            instantiation_points ←concat (instantiation_points, witness_points)
            Solutions ← Solutions ∪ {(R', Z*')}
return Solutions
```

To get the full PLP solution, we must find several regions, so that their union covers the whole parametric space. To do so, we look for instantiation points that are outside of the regions we already know. Actually, the minimization process provides such points. As

detailed in §2.3, raytracing attaches to each irredundant constraint $\boldsymbol{c_i} \geq 0$ a witness point $\boldsymbol{w_i}$ such that

$$\boldsymbol{c_i}(\boldsymbol{w_i}) < 0 \text{ and } \forall j \neq i,\ \boldsymbol{c_j}(\boldsymbol{w_i}) \geq 0$$

In other words, $\boldsymbol{w_i}$ is a point outside of $\mathcal{R}$ that violates only $\boldsymbol{c_i} \geq 0$. Such points are perfect to pursue our exploration of the parametric space by crossing one frontier of $\mathcal{R}$. Because regions are all polyhedral cones pointed on $\mathring{\boldsymbol{x}}$, any frontier of any region separates exactly two regions. Thus, starting from a region $\mathcal{R}_0$, *crossing* once each frontier $\boldsymbol{c_i} \geq 0$ of $\mathcal{R}_0$ gives us all neighbouring regions of $\mathcal{R}_0$. There is a technicality here: crossing a frontier a bit too far can miss a neighbouring region. The adjacency test of §5.2.1 detects such cases and adjusts the witness point. Then, we go on by exploring those neighbours and step by step, this process leads to the full partition of the parametric space into regions. Picking a point $\boldsymbol{w_i}$ across the frontier $\boldsymbol{c_i} \geq 0$ but close to it gives a good chance to discover a neighbouring region by instantiating the PLOP with $\boldsymbol{w_i}$, optimizing and generalizing to a region.

Algorithm Overview. Based on the previous intuition, let us sketch the algorithm of our PLP solver, given in Algorithm 5.1:

(1) Instantiate the PLOP on a parametric point (any point is suitable, since we will ultimately cover the whole space).

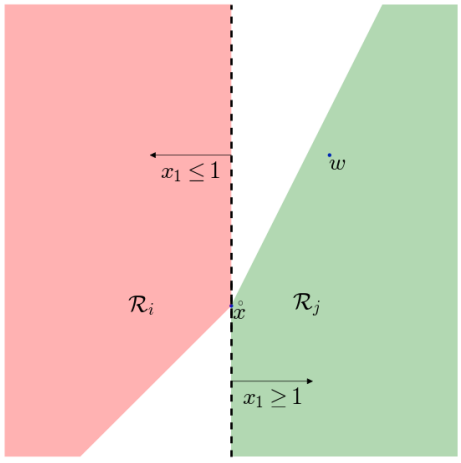
(2) Generalize the solution to a region $\mathcal{R}$

(3) Minimize $\mathcal{R}$ to obtain a list of witness points. Go back to (1), this time instantiating the PLOP with these new witness points.

(4) Stop when all regions are discovered.

5.2.1 The Halting Condition: Adjacency Test

Example 5.2



The figure shows two regions $\mathcal{R}_i$ and $\mathcal{R}_j$ that share a common frontier $\boldsymbol{f} \stackrel{\text{def}}{=} 1 - x_1$. $\mathcal{R}_j$ has been obtained by instantiating the PLOP on $\boldsymbol{w}$, which is the witness point of $\boldsymbol{f} \geq 0$ *i.e.* $x_1 \leq 1$ in $\mathcal{R}_i$. Constraint $-\boldsymbol{f} \geq 0$, *i.e.* $x_1 \geq 1$, is present in $\mathcal{R}_j$, but $\mathcal{R}_i$ and $\mathcal{R}_j$ are not adjacent.

We said that the algorithm stops when all regions are discovered. This needs some extra explanations. Consider a region $\mathcal{R}_i$, whose minimization led to a witness point $\boldsymbol{w}$ associated to a frontier $\boldsymbol{f} \geq 0$ of $\mathcal{R}_i$. Suppose the instantiation of the PLOP with $\boldsymbol{w}$ gave a new region $\mathcal{R}_j$. To ensure that we do not miss any region, we must prove that for each frontier $\boldsymbol{f}$ of each region $\mathcal{R}_i$, the set of discovered regions contains a region $\mathcal{R}_j$ which is adjacent to $\mathcal{R}_i$ on $\boldsymbol{f}$.

First, we can look at their constraints: if $\mathcal{R}_i$ and $\mathcal{R}_j$ are adjacent, they share the crossed frontier $\boldsymbol{f}$, meaning that $\boldsymbol{f} \geq 0$ is a frontier of $\mathcal{R}_i$ while $-\boldsymbol{f} \geq 0$ is a frontier of $\mathcal{R}_j$. If there is no such common frontier $\boldsymbol{f}$, then they are not adjacent and there is a region in between. But, sharing a common frontier is not a sufficient condition, as illustrated by Example 5.2.

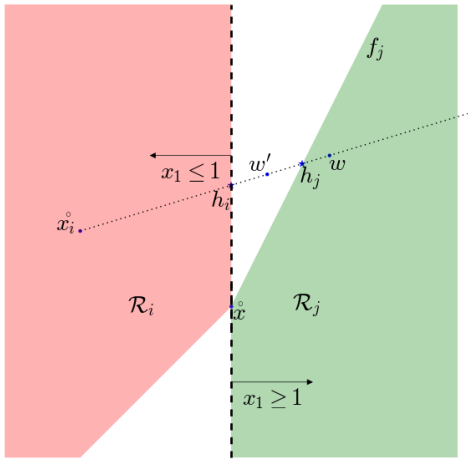
Efficient Adjacency Test. To check adjacency between $\mathcal{R}_i$ and $\mathcal{R}_j$, one could look at the dimension of $\mathcal{R}_i \sqcap \mathcal{R}_j \sqcap (\boldsymbol{f} = 0)$. If the dimension is 0, it means that this intersection is reduced to the normalization point $\mathring{\boldsymbol{x}}$ (which is shared by all regions, by construction due to the normalization) and thus $\mathcal{R}_i$ and $\mathcal{R}_j$ are not adjacent. Otherwise, the two regions intersect on a frontier and are therefore adjacent. However, this adjacency test is expensive, since it involves the intersection operator.
Adjacency can be checked in a much cheaper way by exploiting the optimal solutions $\boldsymbol{Z}_i^\star$ and $\boldsymbol{Z}_j^\star$ associated respectively to $\mathcal{R}_i$ and $\mathcal{R}_j$. Indeed, the intersection between $\mathcal{R}_i$ and $\mathcal{R}_j$ is the space where $\boldsymbol{Z}_i^\star$ and $\boldsymbol{Z}_j^\star$ are equal. Thus, if $\mathcal{R}_i$ and $\mathcal{R}_j$ are adjacent on $\boldsymbol{f}$, then any point $\boldsymbol{x} \in [\![\boldsymbol{f} = 0]\!]$ must fulfill $\boldsymbol{Z}_i^\star(\boldsymbol{x}) = \boldsymbol{Z}_j^\star(\boldsymbol{x})$.
Finally, our adjacency test is the following. We pick any point $\boldsymbol{x} \in [\![\boldsymbol{f} = 0]\!]$ such that $\boldsymbol{x} \neq \mathring{\boldsymbol{x}}$. To do so, we simply pick $n-1$ random coordinates $x_1, \ldots, x_{n-1}$ and adjust the last one x_n accordingly. Then, $\mathcal{R}_i$ and $\mathcal{R}_j$ are adjacent if and only if $\boldsymbol{Z}_i^\star(\boldsymbol{x}) = \boldsymbol{Z}_j^\star(\boldsymbol{x})$.

5.2.2 Adjusting Points Before Instantiation

Like before, consider a region $\mathcal{R}_i$, the minimization of which led to a witness point $\boldsymbol{w}$, associated to a constraint $\boldsymbol{f} \geq 0$ of $\mathcal{R}_i$. Before instantiating the PLOP with a witness point, we must be sure that this point does not lie within any already discovered region. Let *Regions* be the set of regions already discovered by the algorithm. By construction, $\boldsymbol{w}$ is outside of $\mathcal{R}_i$, but it could be in an already known region $\mathcal{R}_j$ (see Example 5.3). To obtain a point $\boldsymbol{w'}$ outside of every discovered regions, we reuse the raytracing algorithm along a ray $\textsc{ray}(\mathring{\boldsymbol{x}}_i, \boldsymbol{w})$ where $\mathring{\boldsymbol{x}}_i$ is the interior point of $\mathcal{R}_i$. If $\mathcal{R}_i$ and $\mathcal{R}_j$ are not adjacent, it will provide a witness point $\boldsymbol{w'}$ satisfying

$$\begin{cases} \boldsymbol{f}(\boldsymbol{w'}) < 0 \\ \exists \text{ constraint } (\boldsymbol{f_j} \geq 0) \in \mathcal{R},\ \boldsymbol{f_j}(\boldsymbol{w'}) < 0 \end{cases}$$

Example 5.3 (follows 5.2)



The figure shows two non-adjacent regions $\mathcal{R}_i$ and $\mathcal{R}_j$. Point $\boldsymbol{w}$ is a witness point of $x_1 \leq 1$ in $\mathcal{R}_i$, obtained by minimization of $\mathcal{R}_i$. The interior point of $\mathcal{R}_i$ that was used during the minimization is $\mathring{\boldsymbol{x}}_i$. We wish to cross frontier $x_1 \leq 1$, that is exploring the parametric space beyond constraint $x_1 \leq 1$, but the associated witness point $\boldsymbol{w}$ lies in region $\mathcal{R}_j$ that we already know. We must therefore adjust $\boldsymbol{w}$: we launch a ray $\textsc{ray}(\mathring{\boldsymbol{x}}_i, \boldsymbol{w})$; it hits $\boldsymbol{f} \overset{\text{def}}{=} 1 - x_1$ at point $\boldsymbol{h_i}$ and encounters a first frontier of $\mathcal{R}_j$ at point $\boldsymbol{h_j}$. Then, we take he point $\boldsymbol{w'}$ as the middle of the segment $[\boldsymbol{h_i}, \boldsymbol{h_j}]$. This point will then be used to instantiate the PLOP, and to find the missing region between $\mathcal{R}_i$ and $\mathcal{R}_j$.

5.3 Degeneracy

In standard linear programming, when the value of all basic variables are strictly positive, a pivot will necessarily lead to a strictly better solution. But, when a basic variable λ_B has value 0, we say that the basis is degenerate: by pivoting on λ_B, we will obtain another dictionary that represents the same feasible solution, hence associated to the same objective value.

Example 5.4

The following dictionary contains a basis degeneracy, because the basic variable λ_4 has value 0.

$$\begin{aligned} \lambda_1 &= \lambda_2 + \lambda_3 + 2 \\ \lambda_4 &= 2\lambda_2 - \lambda_3 \\ \boldsymbol{Z} &= 2\lambda_2 + \lambda_3 + 1 \end{aligned} \qquad \text{(Dict. 5.4)}$$

The current objective value is 1, reached on the feasible point $(\lambda_1 = 2, \lambda_2 = 0, \lambda_3 = 0, \lambda_4 = 0)$. By pivoting $\lambda_3 \leftrightarrow \lambda_4$, we obtain

$$\begin{aligned} \lambda_1 &= 3\lambda_2 - \lambda_4 + 2 \\ \lambda_3 &= 2\lambda_2 - \lambda_4 \\ \boldsymbol{Z} &= 4\lambda_2 - \lambda_4 + 1 \end{aligned} \qquad \text{(Dict. 5.5)}$$

The feasible point associated with this dictionary is still $(\lambda_1 = 2, \lambda_2 = 0, \lambda_3 = 0, \lambda_4 = 0)$, and the objective value remains 1.

Cycling. Starting from a degenerate dictionary D, when several degenerate pivots follow each other, there is a risk that the simplex algorithm may find D again. At this point, if the pivoting heuristic is deterministic, the algorithm is cycling. As we mentioned in §1.3.2.2 (p.32), some heuristics such as Bland's rule avoid this phenomenon.

Degeneracy is particularly problematic in PLP solving: basis degeneracy induces false frontiers that divide regions into subregions that share the same optimal solution. For instance, consider the dictionary of a region with the following objective function:

$$\boldsymbol{Z} = \sum_{i \in \mathcal{N}} \boldsymbol{c_i} \lambda_i$$

This objective is optimal as long as the parametric coefficients $\boldsymbol{c_i}(\boldsymbol{x})$ are nonnegative. Hence, the region associated with this dictionary is $\bigwedge_{i \in \mathcal{N}} \boldsymbol{c_i} \geq 0$. Suppose that this basis is degenerate, meaning that a basic variable λ_B has value 0 in the dictionary. Then, by pivoting λ_B with any nonbasic variable λ_k, we would cross frontier $\boldsymbol{c_k} \geq 0$ and reach another region where $\boldsymbol{c_k} < 0$. But, since λ_B has value 0, the objective value would not be changed by this pivot. Thus, the new region would share the same optimal solution as the previous one. This is a big issue: a subdivision brings nothing more than useless computations. In our experiments, we found that the encoding of convex hull as a PLOP generates many degenerate bases.

To avoid degeneracy, Jones et al. (2007) propose to define a lexicographic order on decision variables, and to adapt the pivoting rule so that degenerate pivots become forbidden. We have not implemented this optimization in our PLP solver yet.

Part II

Certification in Coq

Chapter 6

Introduction to Coq Certification

> One does not simply prove an imperative program. Its conditionals are guarded by more than just Boolean expressions. There are loops there that do not terminate, and Coq is ever watchful. It is a tricky paradigm, riddled with mallocs, corruptions and deadlocks. The very pointer you deallocate is a potential exception. Not with ten thousand PhD students could you do this. It is folly.

As we mentioned many times, the VPL is certified in the proof assistant Coq, using some untrusted OCaml oracles. In this chapter, we introduce useful concepts related to Coq. We will also discuss the exploitation of OCaml code into a Coq development.

6.1 The Struggle of Certification

Venturing into the formal certification of a program is a big decision to make. It requires a lot of efforts and guides the software design towards unusual directions: it must be written while thinking of its proof. For instance, it is often helpful to implement two versions `f_1` and `f_2` of the same function, `f_1` being efficient and `f_2` easy to prove. Then, by showing that both functions are equal for all inputs, properties proved on `f_2` propagate to `f_1`. Similarly, this duplication of functions can be applied on datastructures. For instance, the Coq standard library provides two modules for integers: module `nat` represents Peano integers that are convenient for proofs, whereas module `N` implements binary integers.

All these investments are sometimes rewarded. Yang et al. (2011) looked for bugs in C compilers by generating random tests that cover a wide subset of C. Among the compilers they experimented, there was COMPCERT, which is proved correct in Coq (Leroy, 2009): it preserves semantics between source code and compiled code. Yang et al. could find some bugs in COMPCERT, but only in the uncertified frontend part, and all the middle-end bugs they found in all other compilers were absent in COMPCERT. This is a remarkable success, proportional to the efforts required to reach it: The certified part of the compiler (that excludes for instance parsing processes) uses 11 intermediate languages from C to the executable code. Each intermediate language is given formal semantics and handles a particular compilation step (type elimination, loop simplifications, CFG construction, etc.). The transformation from one language to the following is proved to preserve semantics.

CompCert is marketed commercially by Absint GmbH. Its users most notably include Airbus Avionics and Simulation Products (Bedin França et al., 2012).

Interactive Theorem Provers. There are several theorem provers, among which the famous Isabelle/HOL and Coq. Both are based on powerful higher-order logics, which differ on some aspects. Coq logic is a dependent type theory known as the calculus of inductive constructions. A *dependent type* is a type that depends on a value. For instance, one can define the type

of pairs of integers that are relatively prime. Isabelle/HOL logic does not provide such rich types.
Although both provers encourage proof automation, Isabelle/HOL offers more decisive tactics to do so. It is designed for an intensive use of the powerful `sledgehammer` tactic, that invokes several external solvers trying to discharge the goal in one shot. Coq also provides some automatic tactics, such as `omega` or `ring`, but they are usable only in specific cases. The following sections gives more details on Coq, its proof language, and its ability to use external oracles.

6.2 The Coq Proof Assistant

Coq is a tool designed to verify theorem proofs. These theorems can come from very different thematics, from pure mathematical theory – *e.g.* algebra or arithmetic – to properties satisfied by a program.

Coq provides its own programming language named Gallina. The user can interactively prove that the program satisfies a specification he has written in the same language. During the redaction of a proof, the user manipulates a set of goals to prove, and a set of premises. Then, applying *tactics* – defined in the language Vernacular – transforms premises and goals until there is no goal left to prove.

6.2.1 A Basic Example of Handmade Coq Proof

Knowing the basics and vocabulary of Coq is recommended to read Part II. Hence, this section gives a really simple example of Coq proof and introduces basic concepts, such as lemmas, goals, subgoals, tactics, etc. If you never saw a Coq proof, this part should be helpful. For a complete overview, please refer to (Chlipala, 2013), which is available online for free, or to (Bertot and Castéran, 2004). Otherwise, simply skip to the next section.

As mentioned earlier, most proofs needed in the VPL are polyhedral inclusions. So, as an introduction, let us prove in Coq that the polyhedral inclusion $(2 \leq x) \wedge (0 \leq x + 2y) \sqsubseteq (4 \leq 3x + 2y)$ holds for integers. The interface of COQIDE is split into three parts: the writing panel, where the user writes code and proofs, the feedback panel, displaying the current state of the proof and a last panel showing error messages and replies to queries.

First, let us define the lemma that we will prove. Note that `Z` is the Coq type for integers ($\mathbb{Z}$).

```
Lemma ex_incl : ∀ (x y : Z),
  2 ≤ x → 0 ≤ x + 2*y → 4 ≤ 3*x + 2*y.
```

This lemma, named `ex_incl`, is a currified version of the polyhedral inclusion given above. The right arrow $\rightarrow$ stands here for implication. But, following Curry-Howard correspondence, $\rightarrow$ is also the usual type constructor of functional programming. Therefore, a lemma can be seen and used as a function, the parameters of which are the quantified variables.

The feedback panel shows:

```
1 subgoal
______________________________________(1/1)
∀ x y : Z, 2 ≤ x → 0 ≤ x + 2 * y → 4 ≤ 3 * x + 2 * y
```

It informs that there is currently one *subgoal* to prove, with no *hypothesis* (or *premise*). The first step of the proof is to "declare" our two integer variables `x` and `y`:

```
intros x y.
```

```
1 subgoal
x, y : Z
________________________(1/1)
2 <= x -> 0 <= x + 2 * y -> 4 <= 3 *
     x + 2 * y
```

`intros` is a *tactic* and as such, it transforms the current state of premises and goals. The variables are now part of the hypotheses set, and the universal quantifier has vanished from the goal. Then, thanks to `intros`, we assume $2 \leq x$ and $0 \leq x + 2y$ by naming the two first terms `Hx` and `Hxy`.

```
intros Hx Hxy.
```

```
1 subgoal
x, y : Z
Hx : 2 <= x
Hxy : 0 <= x + 2 * y
______________________(1/1)
4 <= 3 * x + 2 * y
```

The key idea of this inclusion proof is the Farkas combination $(4 \leq 3x + 2y) = 2 \times (2 \leq x) + 1 \times (0 \leq x + 2y)$. To prove it, we will use two lemmas taken from the module ZArith of the standard Coq library (The Coq Development Team, 2016): `Zplus_le_compat` (addition of two constraints) and `Zmult_le_compat_l` (product of a constraint with a nonnegative scalar).

```
Zplus_le_compat : ∀ n m p q : Z,
  n ≤ m → p ≤ q → n + p ≤ m + q

Zmult_le_compat_l : ∀ n m p : Z,
  n ≤ m → 0 ≤ p → p * n ≤ p * m
```

Let us begin by applying the first one. We cannot use it directly since Coq awaits a goal of the form `n + p <= m + q`, and our current one is `4 <= 3 * x + 2 * y`. Note that in the lemma, `n`, `m`, `p` and `q` do not necessarily correspond to scalars or variables, but more generally to terms in `Z`. For example, `p` can be instantiated with `3 * x`. Still, to apply the lemma, each side of the inequality must be a sum. Thus, let us rewrite `4` as `4 + 0` and `3*x + 2*y` as `(2*x)+ (x+2*y)`, exhibiting the Farkas combination.

```
replace 4 with (4 + 0) by auto.
replace (3 * x + 2 * y) with
((2 * x) + (x + 2 * y)) by ring.
```

```
1 subgoal
x, y : Z
Hx : 2 <= x
Hxy : 0 <= x + 2 * y
__________________________(1/1)
4 + 0 <= 2 * x + (x + 2 * y)
```

Tactic `replace` replaces a term with another in the goal, provided a proof of equality between them. Proving that `4` equals `4 + 0` can be done immediately with the tactic `auto`. The second rewriting is a bit more tedious, so we apply the powerful tactic `ring` that solves any ring equation. The goal has now the good shape to apply lemma `Zplus_le_compat`. A lemma can be called thanks to tactic `apply`, and is treated like a function which parameters are the bound terms. Coq can infer universally quantified parameters from the goal. Here, all instantiations `n = 4`, `m = 2 * x`, `p = 0` and `q = x + 2 * y` can be guessed by Coq. Thus, we can simply type `apply Zplus_le_compat` instead of `apply (Zplus_le_compat 4 (2*x)0 (x+2*y))`.

```
apply Zplus_le_compat.
```

```
2 subgoals
x, y : Z
Hx : 2 <= x
Hxy : 0 <= x + 2 * y
______________________(1/2)
4 <= 2 * x
______________________(2/2)
0 <= x + 2 * y
```

When applying a lemma, Coq requires to prove that each premise of the lemma holds. Here, we end up with two subgoals, one for each premise of lemma `Zplus_le_compat`. The second one is trivial, as it is exactly assumption `Hxy`. Let us discard it.

```
(* Switch to the subgoal 2 *)
Focus 2.
assumption.
```

```
1 subgoal
x, y : Z
Hx : 2 <= x
Hxy : 0 <= x + 2 * y
------------------------(1/1)
4 <= 2 * x
```

The last step is to prove that $2 \leq x \Rightarrow 4 \leq 2x$. Lemma `Zmult_le_compat_l` given above will do the job. Again, we need our goal to be syntactically of the form `p*n <= p*m`. Let us rewrite `4` as `2*2`, and apply the lemma.

```
replace 4 with (2*2) by ring.
apply Zmult_le_compat_l.
```

```
2 subgoals
x, y : Z
Hx : 2 <= x
Hxy : 0 <= x + 2 * y
------------------------(1/2)
2 <= x
------------------------(2/2)
0 <= 2
```

Again, two subgoals appear when applying `Zmult_le_compat_l`, one for each premise of the lemma. Subgoal 1 corresponds to assumption `Hx`, while subgoal 2 is discharged by the `intuition` tactic, which ends the proof.

6.2.2 Toward Automation

Actually, experienced users try to avoid such fragile proof scripts. Indeed, changing one scalar value of the lemma breaks the proof, which makes it quite weak. For instance, to show that $3 \leq 3x + 2y$ instead of $4 \leq 3x + 2y$ the Farkas combination $2 \times (2 \leq x) + 1 \times (0 \leq x + 2y)$ gives $4 \leq 3x + 2y$, and the final argument is that $4 \leq 3x + 2y \Rightarrow 3 \leq 3x + 2y$. This proof thus involves the transitivity of $\leq$, which was not needed in the proof of lemma `ex_incl`

Instead of making such "handmade proofs", it is highly preferable to automatize them as much as possible. Coq offers several tools to do so. One can build lemma stacks, and tell Coq to try to apply them when using tactic `auto`. It is also possible to declare user-defined tactics that pattern matches the goal and applies different tactics depending on its shape. For our example, the following tactic does the job, but the goal needs to be already in the form `4 <= (2*x)+ (x + 2*y)`, *i.e.* rewritten so that the Farkas combination syntactically appears.

```
Ltac tac_incl :=
  repeat match goal with
  | [H : ?a ≤ ?b |- ?a ≤ ?b] ⇒ assumption
  | [|- ?a + ?b ≤ ?c + ?d] ⇒ apply Zplus_le_compat
  | [H : ?a ≤ ?b |- ?n * ?a ≤ ?n * ?b] ⇒ apply Zmult_le_compat_l
  | [H : ?a ≤ ?b |- ?c ≤ ?n * ?b] ⇒ replace c with (a * n) by ring
  | [|- ?a ≤ ?c + ?d] ⇒ replace a with (a + 0) by ring
  | [|- 0 ≤ ?a] ⇒ intuition
end.
```

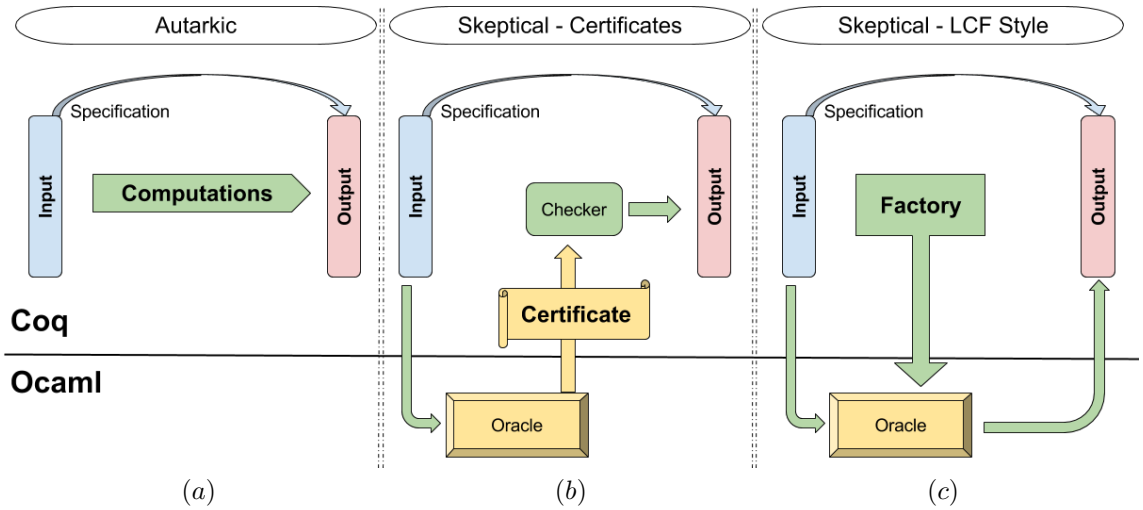


Figure 6.1 – Three methods of certification: certified computation, certificate generation and Logical Consequences Factories.

```
Lemma ex_incl2 : forall (x y : Z),
  2 <= x -> 0 <= x + 2*y -> 4 <= (2*
      x) + (x + 2*y).
Proof.
  intros.
  tac_incl.
```

```
No more subgoals.
```

This tactic provides a proof that is a bit more robust than the previous one. For instance, it allows proving $(2 \leq x) \wedge (0 \leq x+2y) \sqsubseteq (4 \leq 5x+6y)$, still provided that the Farkas combination syntactically appears in the goal.

There exist powerful decision procedures for some specific cases. For instance, tactic `omega` can solve any system of inequalities in $\mathbb{Z}$. Actually, lemmas `ex_incl` and `ex_incl2` can all be solved directly by this tactic.

When the good Farkas combination is known, inclusion proofs involve only simple results on inequality, such as lemmas `Zplus_le_compat` and `Zmult_le_compat_l`, or the transitivity of $\leq$. The hard part is to find the combination, which requires a LP solver as we saw in §1.2. We could write a LP solver in Coq and provide the Farkas combination to an automatic proof. But writing programs in Coq restricts us in several points. For instance, we are limited to Coq datastructures (that were designed in the first place for proofs, not efficiency) and we must prove that any recursive function that we define terminates. Instead, we prefer to use OCaml and C oracles to perform complex computations and determine Farkas combinations. The next section details how to embed uncertified programs within Coq code.

6.3 Three Certification Approaches

In this section, we will introduce several ways to prove a program correct using Coq. First, let us make a distinction between two paradigms of certification: *static* versus *dynamic*.

– **Static certification**: Basically, it corresponds to proving that a program satisfies a specification, once for all. It means that it requires no runtime verification. In Coq, the proof system roughly boils down to type-checking: by the Curry-Howard correspondence, a proof of a proposition `A` is a term of type `A`. Thus, we could say that a program proved by static certification is verified at compilation.

– **Dynamic (or a posteriori) certification**: The program generates certificates that are a posteriori verified – during the execution – by a checker which is statically verified. Note that dynamic certification does not necessarily involve any non-Coq oracle: a Coq function can itself produce certificates checked at runtime!

These two paradigms are two distinct ways of thinking certification. This choice affects the whole program design. On the one hand, the static point of view brings *total correctness*: the program is certified for any input and there is no runtime verification. On the other hand, a posteriori certification provides simpler and shorter proofs, which can be desirable for tricky algorithms. Moreover, Coq proofs can be hard to maintain: it is not rare to see a correct proof become invalid in a new version of Coq. This encourages to make proofs as simple as possible. However, a posteriori certification is restricted to *partial correctness* only. The property proved on the program is ensured provided that the oracle terminates withouth any crash or exception, and that the certificate is correct.

The polyhedral operators of the VPL are a posteriori certified: following an idea from Besson et al. (2007), an oracle computes the Farkas combinations proving the needed inclusions. In the VPL, Fouilhé et al. (2013) improved this technique: instead of computing Farkas combinations afterwards, they are gathered directly along the operators.

Now, let us study three certification approaches and compare their Trusted Computing Base (TCB), which represents the amount of code that must be trusted to consider the certification correct. We will take as example the polyhedral operator `is_empty`, that checks for the existence of a point of $\mathbb{Q}^n$ satisfying the constraints of an input polyhedron. In the following, we describe a polyhedron as a list of constraints of type `Cstr.t`. The type for `is_empty` is thus `Cstr.t list -> bool`. We compare three ways of certifying in Coq that a `true` answer from `is_empty` ensures the emptiness of its input polyhedron. The three certification approaches are schematized on Figure 6.1.

6.3.1 Autarkic Approach

The whole software could be directly implemented and proved correct in Coq following an *autarkic* approach (Barendregt and Barendsen, 2002). While dynamic certification is available in the three approaches, autartkic approach is the only one that allows static certification. It also offers the smallest TCB: an autarkic program involves only the Coq proof-checker, which is alone to be trusted. Most tactics and theorems are proved this way.

However, autarkic certification is very development-time consuming. Coq is complex to use and to learn, writing proofs is difficult, and takes a lot of time. Also, a good understanding of type theory is necessary. This is in my opinion the main obstacle: the initial effort needed to enter the Coq world is significant, as it is hard to build non-trivial proofs without knowing much about the underlying theory. Despite this, the Coq community does not stop growing, as well as the Coq core.

The autarkic approach is restrictive: it forbids the use of efficient C libraries like GMP (for multi-precision arithmetic) or GLPK (for linear programming). It enforces using exclusively Coq datastructures, which significantly slows down executions. This is why in practice, Coq programs are often exported and executed in OCaml by *extraction*.

Extraction. Extraction is a built-in Coq process that allows translating Coq code into OCaml, Haskell or Scheme (Pierre Letouzey, 2008, 2004). The interest of extraction is that it somehow preserves properties proved on Coq functions. Basically, it gets rid of all proof terms and keeps the computational part. Running an extracted program has several advantages. First, the user can use certified software without installing Coq. Second, it allows exploiting uncertified oracles, as the next section will highlight. Third, an extracted program can be used more easily within a large non-Coq development. Indeed, Coq being purely functional, it can be hard to link with other tools. Still, it makes the TCB grow, adding the extraction process and the OCaml compiler into the trusted core.

Several Coq developments are used after extraction, such as the C compiler COMPCERT (Leroy, 2009), or its dedicated static analyzer VERASCO (Jourdan et al., 2015).

6.3.2 Skeptical Approach with Certificates.

The VPL uses uncertified code through a *skeptical* approach. Unlike most polyhedra libraries, VPL uses the constraints-only representation of polyhedra in order to ease its certification in Coq. Certifying a double-representation library would require to prove the correctness of Chernikova's conversion algorithm. Instead, the initial VPL developer A. Fouilhé looked for efficient polyhedra operators in constraint-only representation. Its abstract domain is thus designed in a two-tier architecture:

1. A backend, combining OCaml and C code, provides a set of untrusted oracles that perform efficient but unproved computations;
2. A Coq frontend uses the backend oracles to provide the certified operators of the abstract domain.

Each of these oracles outputs a *certificate* allowing a certified *checker* – developed in Coq – to easily compute a certified result (see Figure 6.1(b)). Linking the frontend to the backend is done by translating the former from Coq to OCaml thanks to extraction. The software is finally compiled by OCaml into binaries. Therefore, the Coq extraction process is part of the TCB, in addition to the Coq proof-checker and the ML compiler.

Implementing this skeptical approach requires first to introduce a certificate format that captures the information needed to prove the correctness of the polyhedral operators. Fortunately, proving their correctness reduces to verifying implications between polyhedra, in conjunction with other simple verifications that depend on the operator. For instance, testing whether a polyhedron $\mathcal{P}$ is empty is done by showing that $\mathcal{P} \sqsubseteq \mathcal{P}_\emptyset$, where $\mathcal{P}_\emptyset$ is a single contradictory constant constraint, such as $0 \geq 1$. The emptiness of $\mathcal{P}_\emptyset$ is thus itself checkable by a simple rational comparison.

Recall that such inclusions can be proved by exhibiting a Farkas combination in terms of $\mathcal{P}$ that yields $\mathcal{P}_\emptyset$. In the skeptical approach, a OCaml oracle for `is_empty` returns a certificate as the list of coefficients that represents this combination. The OCaml type of the oracle is thus

```
Back.is_empty: Cstr.t list -> Cert.t option
```

where the `None` answer means that the input polyhedron is not empty, and a `Some` answer gives a certificate of type `Cert.t` allowing to establish the polyhedron emptiness. From `Cert.t`, the frontend computes the result of the combination with its own certified Coq datastructures and obtains $0 \geq 1$.

Certificate generation does not guarantee the absence of bugs in the oracle. For instance, the backend may not terminate normally on some inputs. This approach ensures actually a *partial correctness* property: when the oracle terminates and provides a certificate, the frontend uses the certificate to compute a certified result satisfying the *formal specification* of the operator. It detects if the backend went wrong and can then fail or return a trivially correct – but weak – result.

In an informal discussion, A. Fouilhé stated that the VPL certificates were more complex than sketched above and that the code generating them was particularly difficult to develop and debug. He concluded that simplifying this process would be helpful.

6.3.3 Skeptical Method with LCF Style

In order to completely avoid the handling of certificates, one could be tempted by another style of skeptical certification, called LCF style. The name LCF stands for "Logic for Computable Functions", a prover at the origin of ML where theorems were handled through an abstract datatype (Gordon et al., 1978). This LCF style is still at the heart of HOL provers.

This style is much lighter than the preceding one, because it avoids the introduction of a certificate format – *i.e.* an abstract syntax – in order to represent the certified computations. Instead, the OCaml oracle uses a factory of certified operators (*i.e.* the "Factory" of Fig. 6.1(c)) to perform trusted computations. The key idea is that such a factory can only build logical consequences of some given set of axioms. Thus, in our use, LCF style also means *Logical Consequences Factories* style.

For `is_empty`, the oracle manipulates two versions of each constraint: the untrusted one, named `BackCstr.t`, manipulated by the oracle, and the one extracted from Coq, named `FrontCstr.t`, on which the oracle can only apply factory operators. Given an empty polyhedron, the backend uses an untrusted solver to find the contradictory Farkas combination and then builds a certified combination of type `FrontCstr.t` using the factory operators extracted from Coq. Then, the frontend only has to check that the resulting constraint is $0 \geq 1$.

```
Back.is_empty: (BackCstr.t * FrontCstr.t) list -> FrontCstr.t option
```

This style of certification relies on one assumption: the frontend can trust results produced by an external oracle that uses its certified operators. This is not true in general: making `FrontCstr.t` an abstract datatype is not sufficient to forbid an imperative OCaml program to cheat by returning a contradictory constraint from a previous run. Hence, this naive LCF style is unsound for certification.

Chapter 7

Certification by Polymorphic Factories

In the previous chapter, we introduced *skeptical certification* that allows embedding uncertified code into a Coq development. We have seen that the two existing skeptical approaches have weaknesses:

- The skeptical approach with certificates is sound, but it requires a certificate format that can be hard to implement and debug.
- The skeptical approach with LCF style is easier to develop but can be unsound.

To get advantages of both approaches, we introduce in this chapter the *Polymorphic LCF Style*, that we abbreviate as PFS (for *Polymorphic Factories Style*) for convenience, for developing correct-by-construction oracles in a skeptical approach. More precisely, a PFS oracle can be trusted to preserve some invariant without the need for an intermediate certificate. This *design pattern* was introduced by Sylvain Boulmé, and we experimented it by reimplementing the certification features of the VPL: it simplifies both Coq and OCaml parts.

Let us illustrate PFS on operator `is_empty`, on which we introduced the three certification approaches in §6.3, that checks for the existence of a point of $\mathbb{Q}^n$ satisfying the constraints of an input polyhedron. Recall that, in LCF style, the type of `is_empty` in the OCaml oracle (*i.e.* the backend) would be

```
Back.is_empty: (BackCstr.t * FrontCstr.t) list -> FrontCstr.t option
```

where `BackCstr.t` and `FrontCstr.t` are respectively the type of backend and frontend constraints.

PFS consists in abstracting the certified datatype `FrontCstr.t` by a polymorphic type `'c` in the oracle, and providing the oracle with operators on this datatype grouped in a factory of type `'c lcf`. Polymorphism ensures that the oracle can only produce correct results by combining its inputs using the operators of the factory. In other words, the type of the `is_empty` oracle becomes

```
Back.is_empty: 'c lcf -> (BackCstr.t * 'c) list -> 'c option
```

Polymorphism of PFS brings the soundness that was missing in the naive LCF style. Results produced by oracles are correct by construction provided that the operators of the factory preserve the desired correctness property. We will explain how such correctness proofs are elegantly expressed in Coq from the type of PFS oracles (see §7.2). Furthermore, PFS makes our oracles flexible. In particular, we can easily profile or debug oracles, simply by changing the factory. If necessary, we can still produce certificates as in the original VPL using an adequate factory, or even disable the certification for efficiency using a "do-nothing" factory.

The performances of the original version of the VPL were analyzed by Fouilhé et al. (2013). They are comparable with those of PPL (Bagnara et al., 2008) and NEWPOLKA (Jeannet and Miné, 2009), two state-of-the-art – but unverified – polyhedra libraries. Our new design seems to have a little benefit on VPL performances. More significantly, it simplifies the development

while keeping the same TCB than the original version. Thanks to PFS, the number of lines of code in the VPL modules at the interface of OCaml and Coq was divided by two, both for OCaml and Coq sides. And it gives simpler and more readable code.

Currently, the soundness proof of our approach is partial. On the one hand, we are able to prove that our reasonings on PFS oracles – which are actually parametricity reasonings (Wadler, 1989) – are correct. Indeed, they apply a weak *parametricity* property of polymorphic types, that we call *parametric invariance*. This property had been formalized on system F with higher-order references *a la* ML (Birkedal et al., 2011; Ahmed et al., 2002; Appel et al., 2007). S. Boulmé adapted this proof on a subset of imperative ML. This is well outside of my area of expertise, and I refer the interested reader to (Boulmé and Maréchal, 2017a).

On the other hand, we have not yet proved that our particular way to invoke Coq extraction is perfectly sound w.r.t. the actual OCaml compiler. We only conjecture that it is. However, to our best knowledge, none of the real world developments that mix Coq and OCaml code, including the certified compiler CompCert (Leroy, 2009), come with such a proof; they rely on similar conjectures.

Actually, we are not the first to relate Coq extraction with parametricity reasoning. In a sense, Bernardy and Moulin (2012); Bernardy and Guilhem (2013) already looked for a generalization of Coq extraction in order to internalize some parametricity reasonings within dependent type theory. The novelty of our proposal is to use parametricity as a very cheap approach to reason about imperative ML code in Coq. To our knowledge, since the proposal to get "theorems for free" from parametricity by Wadler (1989), this paper describes its first application to the certification of realistic software, indeed implemented within widespread tools like Coq and OCaml.

In the next section, we incrementally detail PFS on a slightly more complex example: operator `proj`. It also illustrates why the original LCF style is unsound in this context. §7.2 shows how to use PFS in Coq proofs. §7.3 reveals the flexible power of polymorphic factories thanks to operator `join`. Benoy et al. (2005) have shown how to derive a simple implementation of operator `join` from `proj`. We show that the certification of Benoy's `join` boils down to defining a well-chosen instance of the factory expected by operator `proj`. With this approach, the certification of Benoy's `join` becomes elegant and straightforward, whereas the one of Fouilhé et al. (2013) was cumbersome because of many certificate rewritings.

7.1 PFS Oracle Explained on Projection

This section gives a tutorial on PFS oracles, illustrated on operator `proj` of the abstract domain of polyhedra. This operator was presented in §1.2.1 with its standard algorithm (Fourier-Motzkin elimination), and we gave another encoding as a PLOP in Chapter 3. But in our two-tier approach, the correctness proof of `proj` does not need to consider these implementation details.

Let us consider the example of Figure 7.1. Predicate $\mathcal{P}_0$ expresses that q is the result of the Euclidean division of x by 3, with r as remainder. Predicate $\mathcal{P}_1$ "instantiates" $\mathcal{P}_0$ with $x = 15$. Then, predicate $\mathcal{P}'_1$ corresponds to the computation of $\exists r, \mathcal{P}_1$ (as a polyhedron on $\mathbb{Q}$).

$$\mathcal{P}_0 \stackrel{\text{def}}{=} \begin{cases} & x = 3 \cdot q + r & [C_1] \\ \wedge & r \geq 0 & [C_2] \\ \wedge & r < 3 & [C_3] \end{cases} \qquad \mathcal{P}'_1 \stackrel{\text{def}}{=} \begin{cases} & x - 15 = 0 & [C'_1] \\ \wedge & q - 4 > 0 & [C'_2] \\ \wedge & 5 - q \geq 0 & [C'_3] \end{cases}$$

$$\mathcal{P}_1 \stackrel{\text{def}}{=} \mathcal{P}_0 \wedge \; x = 15 \qquad [C_4]$$

Figure 7.1 – Computation of $\mathcal{P}'_1$ as "`proj` $\mathcal{P}_1\, r$"

In the following, we assume that for proving the correctness of our surrounding software (typically, a static analyzer), we do not need to prove $\mathcal{P}' \Leftrightarrow \exists x, \mathcal{P}$ but only $(\exists x, \mathcal{P}) \Rightarrow \mathcal{P}'$. Thus, we only want to prove the correctness of `proj` as defined below.

Definition 7.1 (Correctness of `proj`) *Function `proj` is correct iff any result $\mathcal{P}'$ for a computation $(\texttt{proj}\,\mathcal{P}\,x)$ satisfies $(\mathcal{P} \Rightarrow \mathcal{P}') \ \wedge \ x \notin V(\mathcal{P}')$ where $V(\mathcal{P}')$ is the set of variables appearing in $\mathcal{P}'$ with a non-null coefficient.*

The condition $x \notin V(P')$ ensures that variable x is no longer bound in $\mathcal{P}'$. As dynamically checking this condition is fast and easy, we only look for a way to build P' from P that ensures by construction that $P \Rightarrow P'$. For this purpose, we exploit Farkas' lemma as follows. Internally, in the frontend, we handle constraints in the form "$t \bowtie 0$" where t is a linear term and $\bowtie \in \{=, \geq, >\}$. Hence, each input constraint "$t_1 \bowtie t_2$" is first normalized as "$t_1 - t_2 \bowtie 0$". Then, we generate new constraints using only the two operations of Definition 7.2. Obviously, such constraints are necessarily implied by $\mathcal{P}$.

Definition 7.2 (Linear Combinations of Constraints) *We define operations $+$ and $\cdot$ on normalized constraints by*

- $(t_1 \bowtie_1 0) + (t_2 \bowtie_2 0) \stackrel{\text{def}}{=} (t_1 + t_2) \bowtie 0$
  where $\bowtie \stackrel{\text{def}}{=} \max(\bowtie_1, \bowtie_2)$ for the total increasing order induced by the sequence $=, \geq, >$.
- $n \cdot (t \bowtie 0) \stackrel{\text{def}}{=} (n \cdot t) \bowtie 0$
  under preconditions $n \in \mathbb{Q}$ and, if $\bowtie \in \{\geq, >\}$ then $n \geq 0$.

For example, $\mathcal{P}'_1$ is generated from $\mathcal{P}_1$ by the script on the right hand-side. Here `tmp` is an auxiliary constraint, where variable x has been eliminated from C_1 by rewriting using equality C_4.

$$\begin{array}{rcl} \texttt{tmp} & \leftarrow & C_4 + -1 \cdot C_1 \\ C'_1 & \leftarrow & C_4 \\ C'_2 & \leftarrow & \frac{1}{3} \cdot (C_3 + \texttt{tmp}) \\ C'_3 & \leftarrow & \frac{1}{3} \cdot (C_2 + -1 \cdot \texttt{tmp}) \end{array}$$

In the following, we study how to design – in OCaml– a certified frontend `Front.proj` that monitors Farkas' combinations produced by an untrusted backend `Back.proj`. §7.2 will then formalize `Front.proj` in Coq.

7.1.1 Simple (but Unsound) LCF Style

In a first step, we follow the LCF style introduced in §6.3. We thus consider two datatypes for constraints: modules `BackCstr` and `FrontCstr` define respectively the representation of constraints for the backend and the frontend.
Each module is accessed both in the backend and in the frontend, but the frontend representation is abstract for the backend. Hence, the visible interface of `FrontCstr` for the backend is given on the right-hand side. Type `Rat.t` represents $\mathbb{Q}$, and `add` and `mul` represent respectively operators $+$ and $\cdot$ on constraints.

```
module FrontCstr: sig
  type t
  val add: t -> t -> t
  val mul: Rat.t -> t -> t
end
```

Going back to our example, $\mathcal{P}'_1$ is firstly computed from $\mathcal{P}_1$ using backend constraints. This representation allows finding the solution by efficient computations, combining complex datastructures, GMP rationals and even floating-point numbers. On the contrary, the frontend representation of constraints is based on certified code extracted from Coq. In particular, it uses internally the certified rationals of the Coq standard library, where integers are represented as lists of bits. Once a solution is found, the backend thus rebuilds this solution in the frontend representation. For example, the following function builds the certified constraints of $\mathcal{P}'_1$ from constraints of $\mathcal{P}_1$, according to the previous Farkas combinations. Here, rational constants are written with an informal notation.

```
let build_P'1 (l: FrontCstr.t list): FrontCstr.t list =
  match l with
  | c1::c2::c3::c4::_ ->
    let coeff = 1/3 and tmp = FrontCstr.add c4 (FrontCstr.mul −1 c1) in
    [ c4;
      FrontCstr.mul coeff (FrontCstr.add c3 tmp);
      FrontCstr.mul coeff (FrontCstr.add c2 (FrontCstr.mul −1 tmp)) ]
  | _ -> failwith "unexpected input"
```

But making `Back.proj` return such a function is not so convenient. It is simpler to make `Back.proj` compute the certified constraints (of type `FrontCstr.t`), in parallel to its own computations. Hence, we propose a first version of `Back.proj`, called `Back.proj0`, with the following type.

```
Back.proj0: (BackCstr.t * FrontCstr.t) list -> Var.t -> FrontCstr.t list
```

Let us define two certified functions:

— `occurs: Var.t -> FrontCstr.t -> bool` such that `occurs x c` tests whether $\mathtt{x} \in V(\mathtt{c})$;

— `export: FrontCstr.t -> BackCstr.t` that converts a frontend constraint into a backend one.

Then, we implement `Front.proj` as follows:

```
let Front.proj (p: FrontCstr.t list) (x: Var.t): FrontCstr.t list =
  let bp = List.map (fun c -> (export c, c)) p in
  let p' = List.map snd (Back.proj0 bp x) in
  if List.exists (occurs x) p'
  then failwith "oracle error"
  else p'
```

Ideally – mimicking a LCF-style prover – function `Back.proj0` uses type `FrontCstr.t` as a type of theorems. It derives logical consequences of a list of constraints (of type `FrontCstr.t`) by combining them with `FrontCstr.mul` and `FrontCstr.add`. Like in a LCF-style prover, there is no explicit "proof object" as value of this theorem type.

Unfortunately, this approach is unsound. We now provide an example which only involves two input polyhedra that are reduced to a single constant constraint. Let us imagine an oracle wrapping function `memofst` given below. Assuming that it is first applied to the unsatisfiable constraint $0 \geq 1$, this first call returns $0 \geq 1$, which is a correct answer. However, when it is then applied to the satisfiable constraint $2 \geq 0$, this second call still returns $0 \geq 1$, which is now incorrect! This unsoundness is severe, because even a faithful programmer could, by mistake, implement such a behavior while handling mutable datastructures.

```
let memofst:FrontCstr.t -> FrontCstr.t =
  let first = ref None in
  fun c ->
    match !first with
    | None -> (first := Some c); c
    | Some c' -> c'
```

7.1.2 Generating an Intermediate Certificate

To be protected against lying backends, we could introduce an intermediate datastructure representing a trace of the backend computation. Then, the frontend would use this trace to rebuild the certified result using its own certified datastructures. Such a trace has the form of an Abstract Syntax Tree (AST) and is called a *certificate*. This approach was used by Fouilhé et al. (2013) to design the first version of the VPL. In the following, we detail the process of certificate generation and why we prefer avoiding it.

We define below a certificate type named `pexp`. It represents a type of polyhedral computations, and depends on type `fexp` that corresponds to Farkas combinations. Constraints are identified by an integer. In `pexp`, we provide a `Bind` construct for computing auxiliary constraints like `tmp` in the example of P_1'.

```
type fexp =
  | Ident of int
  | Add of fexp * fexp
  | Mul of Rat.t * fexp
```

```
type pexp =
  | Bind of int * fexp * pexp
  | Return of fexp list
```

Example 7.2

Here is an example of certificate for P'_1, where each input constraint C_i is represented by "`Ident` i":

```
Bind (5, Add (Ident 4, Mul (−1, Ident 1)),
  Return [ Ident 4;
   Mul (1/3, Add (Ident 3, Ident 5));
   Mul (1/3, Add (Ident 2, Mul (−1, Ident 5))) ])
```

The intermediate constraint `tmp` is bound to identifier 5.

Next, we easily implement in Coq a `Front.run` interpreter of `pexp` certificates (corresponding to the "checker" part of Fig. 6.1, p.115) and prove that it only outputs a logical consequence of its input polyhedron.

```
Front.run: pexp -> (FrontCstr.t list) -> (FrontCstr.t list)
```

Recall that when a `pexp` uses certificate identifiers that have no meaning w.r.t. `Front.run`, the latter fails. For the following, we do not need to specify how identifiers are generated and attached to constraints. We let this implementation detail under-specified.

Now, we need to turn our `Back.proj0` into a function `Back.proj1` where each `BackCstr.t` constraint in input is associated to a unique identifier.

```
Back.proj1: (BackCstr.t * int) list -> Var.t -> pexp
```

However, `Back.proj1` is more complex to program and debug than `Back.proj0`. Indeed, in LCF-style, certified operations run "in parallel" to the oracle. On an oracle bug (for instance, if the oracle multiplies an inequality by a negative scalar), the LCF-checker raises an error right at the point where the bug appears in the oracle: this makes debugging of oracles much easier. On the contrary, in presence of an ill-formed certificate, the developer has to understand where the ill-formness comes from in its oracle. Moreover, an oracle like `Back.proj1` needs to handle constraint identifiers for `Bind` according to their semantics in `Front.run`. This is particularly painful in operations like the `join` operator of §7.3, which involve several spaces of constraint names (one for each "implication proof"). This consideration motivates the introduction of our new design pattern, where incorrect handling of constraint names is *statically* forbidden, thanks to the OCaml typechecker.

7.1.3 Polymorphic LCF Style

We generalize LCF style in order to solve its soundness issue. We also *conjecture* that the approach of §7.2 provides a sound link between the backend and a Coq extracted frontend, without the need for an intermediate AST. Moreover, an AST can still be generated if needed for another purpose (see Chapter 8).

Our idea is very simple: instead of abstracting the "type of theorems" (*i.e.* `FrontCstr.t`) using an ML abstract datatype, we abstract it using ML polymorphism. Intuitively, the lying function `memofst` from Section 7.1.1 exploits the fact that we have a *static* type of theorems, defined once and for all. But, when we interpret constraints of the result $\mathcal{P}'$ as theorems, they are relative to a given set of axioms: the input constraints of $\mathcal{P}$. Hence, we need to have a *dynamic* type, generated at each call to the oracle. Using ML polymorphism, we actually express that our oracle is parameterized by any of such dynamic type of theorems.

In practice, the type `FrontCstr.t` used in backend oracles, such as `Back.proj`, is replaced by `'c`. In order to allow the backend to build new "theorems" – *i.e.* Farkas combinations – we introduce a polymorphic record type `lcf` (acronym of *Logical Consequences Factory*).

```
type 'c lcf = {
  add: 'c -> 'c -> 'c;
  mul: Rat.t -> 'c -> 'c
}
```

Then, the previous oracle `Back.proj0` that we defined for the simple LCF style is generalized into

```
Back.proj: 'c lcf -> (BackCstr.t * 'c) list -> Var.t -> 'c list
```

Intuitively, function `Back.proj0` could now be redefined as `(Back.proj {add=FrontCstr.add; mul =FrontCstr.mul})`.

The type of the `Back.proj` implementation must generalize the above signature, and not simply unify with it. This directly forbids the `memofst` trick. Indeed, if we remove the type coercion from the preceding code of `memofst`, the type system infers that `memofst: '_a -> '_a` where `'_a` is an existential type variable introduced for a sound typing of references, as described in Wright (1995) and Garrigue (2002). Hence, a cheating usage of `memofst` would prevent the `Back.proj` implementation from having an acceptable type.

7.2 Formalizing the Frontend in Coq

In order to program and prove `Front.proj` in Coq, we need to declare `Back.proj` and its type in Coq. This is achieved by turning `Back.proj` into a Coq axiom, itself replaced by the actual OCaml function at extraction. However, such an axiom may be unsound w.r.t. a runtime execution. In particular, a Coq function f satisfies $\forall x, (f\,x) = (f\,x)$. But, an OCaml function may not satisfy this property, because of side-effects or because of low-level constructs distinguishing values considered equal in the Coq logic. §7.2.1 recalls the may-return monad introduced by Fouilhé and Boulmé (2014) to overcome this issue. §7.2.2 explains how PFS oracles are embedded in this approach.

7.2.1 Coq Axioms for External OCaml Functions

Let us consider the Coq example on the right hand-side. It first defines a constant `one` as the Peano natural number representing 1. Then, it declares an axiom `test` replaced at extraction with a function `oracle`. At last, a lemma `congr` is proved, using the fact that `test` is a function. The following OCaml implementation of `oracle` makes the lemma `congr` false at runtime:

```
Definition one: nat := (S O).

Axiom test: nat → bool.
Extract Constant test ⇒ "oracle".

Lemma congr: test one = test (S O).
  auto.
Qed.
```

```
let oracle x = (x == one)
```

Indeed `(oracle one)` returns `true` whereas `(oracle (S O))` returns `false`, because `==` tests the equality between *pointers*. Hence, the Coq axiom is unsound w.r.t this implementation. A similar unsoundness can be obtained if `oracle` uses a reference in order to return `true` at the first call, and `false` at the following ones.

Fouilhé and Boulmé (2014) solve this problem by axiomatizing OCaml functions using a notion of non-deterministic computations. For example, if the result of `test` is declared to be non-deterministic, then the property `congr` is no more provable. For a given type A, type $?A$ represents the type of non-deterministic computations returning values of type A: type $?A$ can be interpreted as $\mathscr{P}(A)$. Formally, the type transformer "$?\,.$" is axiomatized as a monad that provides a *may-return* relation $\rightsquigarrow_A : ?A \to A \to \text{Prop}$. Intuitively, when "$k : ?A$" is seen as "$k \in \mathscr{P}(A)$", then "$k \rightsquigarrow a$" means that "$a \in k$". At extraction, $?A$ is extracted like A, and its binding operator is efficiently extracted as an OCaml let-in. See (Fouilhé and Boulmé, 2014) for more details.

For example, replacing the `test` axiom by "`Axiom test: nat → ?bool`" avoids the above unsoundness w.r.t the OCaml `oracle`. The lemma `congr` can still be expressed as below, but it is no longer provable.

```
∀ b b', (test one)⇝b → (test (S O))⇝b' → b=b'.
```

7.2.2 Reasoning on PFS Oracles in Coq

Let us now sketch how the frontend is formalized in Coq. We define the type `Var.t` as `positive` – the Coq type for binary positive integers. We build the module `FrontCstr` of constraints encoded as radix trees over `positive` with values in `Qc`, which is the Coq type for $\mathbb{Q}$. Besides operations `add` and `mul`, module `FrontCstr` provides two predicates: `(sat c m)` expresses that a model `m` satisfies the constraint `c` ; and `(noccurs x c)` expresses that variable `x` does not occur in constraint `c`.

```
sat: t → (Var.t → Qc) → Prop.
noccurs: Var.t → t → Prop.
```

We also prove that `sat` is preserved by functions `add` and `mul`. Then, these predicates are lifted to polyhedra `p` of type `(list FrontCstr.t)`.

```
Definition sat p m := List.Forall (fun c ⇒ FrontCstr.sat c m) p.
Definition noccurs x p := List.Forall (FrontCstr.noccurs x) p.
```

Because `front_proj` invokes a non-deterministic computation (the external oracle as detailed below), it is itself a non-deterministic computation. Here is its type and its specification:

```
front_proj: list FrontCstr.t → Var.t → ?(list FrontCstr.t).
Lemma front_proj_correctness: ∀ p x p',
  (front_proj p x) ⇝ p' → (∀ m, sat p m → sat p' m) ∧ noccurs x p'.
```

We implement `front_proj` in PFS, as explained in §7.1.3. First, we declare a `lcf` record type containing operations for frontend constraints. These operations do not need to be declared as non-deterministic: in the Coq frontend, they will be only instantiated by pure Coq functions. Then, `back_proj` is defined as a non-deterministic computation. The type of `back_proj` is given uncurried in order to avoid nested "?" type transformers. At extraction, this axiom is replaced by a wrapper of `Back.proj` from §7.1.3.

```
Record lcf A := { add: A → A → A; mul: Qc → A → A }.
Axiom back_proj: ∀ {A},
  ((lcf A) * (list (FrontCstr.t * A))) * Var.t → ?(list A).
```

Like in §7.1.3, `back_proj` receives each constraint in two representations: an opaque one of polymorphic type `A` and a clear one of another type. For simplicity, we consider that type `FrontCstr.t` is used as the clear representation.[1]

Now, let us sketch how we exploit our polymorphic `back_proj` to implement `front_proj` and prove its correctness. For a given `p: (list FrontCstr.t)`, parameter `A` of `back_proj` is instantiated with `wcstr(sat p)` where `wcstr(s)` is the type of constraints satisfied by any model satisfying `s`. In other words, `wcstr(sat p)` is the type of logical consequences of `p`, *i.e.* the type of its Farkas combination. Hence, instantiating parameter `A` of `back_proj` by this dependent type expresses that combinations from the input `p` and from the `lcf` operations are satisfied by models of `p`. Concretely, `(front_proj p x)` binds the result of `(back_proj ((mkInput p), x))` to a polyhedron `p'` and checks that `x` does not occur in `p'`.

```
Record wcstr(s: (Var.t → Qc) → Prop) :=
  { rep: FrontCstr.t; rep_sat: ∀ m, s m → FrontCstr.sat rep m }.

mkInput: ∀ p, lcf(wcstr(sat p)) * list(FrontCstr.t * wcstr(sat p)).
```

Actually, we can see `rep_sat` above as a data-invariant attached to a `rep` value. This invariant is trivially satisfied on the input values, *i.e.* the constraints of `p`. And, it is preserved by `lcf` operations. These two properties are reflected in the type of `mkInput`. The polymorphism of `back_proj` is a way to ensure that `back_proj` preserves any data-invariant like this one, on the output values.

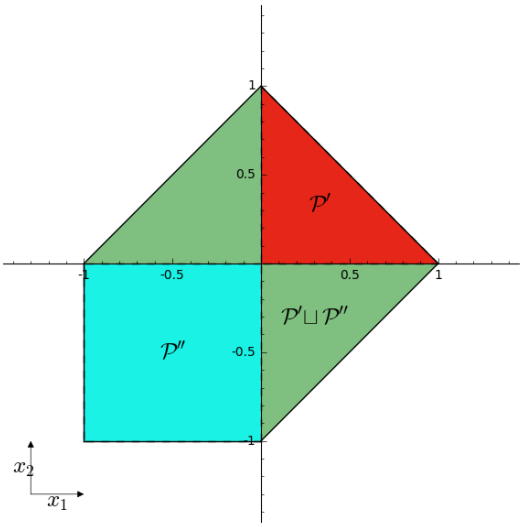
1. In order to avoid unnecessary conversions from `FrontCstr.t` to `BackCstr.t` (that would be hidden in `back_proj` wrapper), our actual implementation uses instead an axiomatized type which is replaced by "`BackCstr.t`" at extraction: this is similar to the implementation of Fouilhé and Boulmé (2014).

7.3 The Flexible Power of PFS Illustrated on Convex Hull

This section provides an advanced usage of polymorphic factories through the `join` operator. It illustrates the flexible power of PFS, by deriving `join` from the projection operator of §7.1.3. On this `join` oracle, PFS induces a drastic simplification by removing many cumbersome rewritings on certificates. Indeed, we simply derive the certification of the `join` operator by invoking the projection operator on a *direct product* of factories. As we detail below, such a product computes two independent polyhedral inclusions, in parallel.

We presented operator `join` in §1.2.1. To summarize, it approximates the union of two polyhedra $\mathcal{P}' \cup \mathcal{P}''$. But in general, such a union is not a convex polyhedron. Operator `join` thus overapproximates this union by the convex hull $\mathcal{P}' \sqcup \mathcal{P}''$ that *we define* as the smallest convex polyhedron containing $\mathcal{P}' \cup \mathcal{P}''$.[2]

Example 7.3



For instance, given $\mathcal{P}' \stackrel{\text{def}}{=} \{x_1 \geq 0,\ x_2 \geq 0, x_1+x_2 \leq 1\}$ and $\mathcal{P}'' \stackrel{\text{def}}{=} \{x_1 \leq 0,\ x_2 \leq 0, x_1 \geq -1,\ x_2 \geq -1\}$ then, as illustrated on the figure, $\mathcal{P}' \sqcup \mathcal{P}'' \stackrel{\text{def}}{=} \{x_1 \geq -1,\ x_2 \geq -1,\ x_1+x_2 \leq 1,\ x_2-x_1 \geq -1,\ x_2-x_1 \leq 1\}$.

The correctness of `join`, given in Definition 7.3, is reduced to two implications themselves proved by Farkas' lemma. More precisely, on a computation $(\texttt{join}\ \mathcal{P}'\ \mathcal{P}'')$, the oracle produces internally two lists of Farkas combinations that build a pair of polyhedra $(\mathcal{P}_1, \mathcal{P}_2)$ satisfying $\mathcal{P}' \Rightarrow \mathcal{P}_1$ and $\mathcal{P}'' \Rightarrow \mathcal{P}_2$. Then, the front-end checks that $\mathcal{P}_1$ and $\mathcal{P}_2$ are syntactically equal. If the check is successful, it returns polyhedron $\mathcal{P}_1$.

Definition 7.3 (Correctness of `join`) *Function `join` is correct iff any result $\mathcal{P}$ for a computation $(\texttt{join}\ \mathcal{P}'\ \mathcal{P}'')$ satisfies* $(\mathcal{P}' \Rightarrow \mathcal{P}) \ \wedge\ (\mathcal{P}'' \Rightarrow \mathcal{P})$.

7.3.1 Extended Farkas Factories

The factory operations of Definition 7.2 are sufficient to compute any result of a projection, but they do not suffice for the convex-hull and more generally for proving all kinds of polyhedra inclusions. The definition 7.4 given here completes this set of operations, to handle polyhedra with equalities and strict inequalities.

Definition 7.4 (Extended Farkas Combination) *Besides operations $+$ and $\cdot$ of Definition 7.2, an extended Farkas combination may invoke one of the three operations:*

2. Actually, there exists cases where the convex hull of two polyhedra is *not* a polyhedron. See (Fouilhé, 2015) for more details.

— *weaken:* $\Downarrow (t \bowtie 0) \stackrel{\text{def}}{=} t \geq 0$, *for all linear term* t *and* $\bowtie \in \{=, \geq, >\}$.

— *cte*$(n, \bowtie) \stackrel{\text{def}}{=} n \bowtie 0$ *assuming* $n \in \mathbb{Q}$ *and* $n \bowtie 0$.

— *merge:* $(t \geq 0)$ & $(-t \geq 0) \stackrel{\text{def}}{=} (t = 0)$, *for all linear term* t.

From now on, we only consider extended Farkas combinations and omit the adjective "extended". Definition 7.4 leads to extend our factory type as given on the right hand-side.
Here, constant `top` of the factory encodes constraint $0 = 0$ and is a shortcut for `cte`$(0, =)$. Hence, this equality is neutral for operations $+$ and $\cdot$ on constraints. It is thus a very convenient default value in our PFS oracles.
Fields `weaken` and `merge` correspond respectively to operators $\Downarrow$ and &. Type `cmpT` is our enumerated type of comparisons representing $\{\geq, >, =\}$.

```
type 'c lcf = {
  top: 'c;
  add: 'c -> 'c -> 'c;
  mul: Rat.t -> 'c -> 'c;
  weaken: 'c -> 'c;
  cte: Rat.t -> cmpT -> 'c;
  merge: 'c -> 'c -> 'c;
}
```

7.3.2 Encoding Join as a Projection

Following an encoding of Benoy et al. (2005), `proj` can be expressed as a projection problem. We already presented this process in §3.6.1. We saw that computing the convex hull of $\mathcal{P}' : \boldsymbol{A'x} \leq \boldsymbol{b'}$ and $\mathcal{P}'' : \boldsymbol{A''x} \leq \boldsymbol{b''}$ is equivalent to eliminating variables $\boldsymbol{y'}, \boldsymbol{y''}, \alpha'$ and α'' from the following polyhedron $\mathcal{H}$:

$$\mathcal{H} \stackrel{\text{def}}{=} \{\boldsymbol{x} \mid \boldsymbol{A'y'} \geq \alpha' \cdot \boldsymbol{b'},\ \boldsymbol{A''y''} \geq \alpha'' \cdot \boldsymbol{b''},\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \boldsymbol{y'} + \boldsymbol{y''}\} \qquad (7.1)$$

Example 7.4 (follows 7.3)

For our previous example, $\mathcal{H}$ is the set of points $\boldsymbol{x} \stackrel{\text{def}}{=} (x_1, x_2)$ that satisfy

$$\begin{cases} y'_1 \geq 0,\ y'_2 \geq 0,\ -y'_1 - y'_2 \geq -\alpha' \\ -y''_1 \geq 0,\ -y''_2 \geq 0,\ y''_1 \geq -\alpha'',\ y''_2 \geq -\alpha'' \\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1 \\ x_1 = y'_1 + y''_1,\ x_2 = y'_2 + y''_2 \end{cases}$$

The presence of equalities or strict inequalities requires an additional pass that follows the projection, involving operators `weaken` and `merge` of the factory. We will omit this step here in order to keep our explanations simple. Moreover, as mentioned in §3.6.1, encoding (7.1) can be done more efficiently by considering fewer variables, exploiting the fact that $\alpha'' = 1 - \alpha'$ and $\boldsymbol{y''} = \boldsymbol{x} - \boldsymbol{y'}$. But as this complicates the understanding and does not affect much the certification, we will not consider this improvement here.

In the following, we compare certificate style to PFS for proving `join` from the results of `proj`. In order to have a simpler presentation, we limit ourselves here to the case where polyhedra contain only non strict inequalities.

7.3.3 Proving Join with Certificates

As previously explained about Definition 7.3, the correctness of `join` is ensured by building $\mathcal{P}$ from two Farkas combinations, one of $\mathcal{P}'$ and one of $\mathcal{P}''$. Fouilhé (2015) described how to extract such combinations from the result of the projection of $\mathcal{H}$. Built in a skeptical way with certificates, Fouilhé's version of `join` has the following type:

```
Back.join1 : (BackCstr.t * int) list -> (BackCstr.t * int) list ->
             pexp * pexp
```

It takes the two polyhedra $\mathcal{P}'$ and $\mathcal{P}''$ as input, and each of their constraint is attached to a unique identifier, as explained in Section 7.1.2. It returns two certificates of type `pexp`, one for each inclusion $\mathcal{P}' \Rightarrow \mathcal{P}$ and $\mathcal{P}'' \Rightarrow \mathcal{P}$ of Definition 7.3.

Let us now detail how Fouilhé retrieves such certificates from the projection of $\mathcal{H}$. Consider operator `Back.proj1_list` that extends `Back.proj1` from §7.1.2 by projecting several variables one after the other instead of a single one. Assume that `Back.proj1_list` $\mathcal{H}$ $[x_1, \ldots, x_q]$ returns $(\mathcal{P},\ \mathbf{\Lambda})$ where $\mathbf{\Lambda}$ is a certificate of type `pexp` showing that $\mathcal{P}$ is a logical consequence of $\mathcal{H}$. Actually, as explained in §1.2.2, $\mathbf{\Lambda}$ can be viewed as a matrix where each line contains the coefficients of a Farkas combination of $\mathcal{H}$, and it fulfills

$$\mathbf{\Lambda} \cdot \mathcal{H} = \mathcal{P} \tag{7.2}$$

Fouilhé showed that $\mathbf{\Lambda}$ can be decomposed into three parts: $\mathbf{\Lambda}_1$ speaking about constraints of $\mathcal{P}'$, $\mathbf{\Lambda}_2$ speaking about constraints of $\mathcal{P}''$ and $\mathbf{\Lambda}_3$ speaking about remaining constraints.

$$\{\boldsymbol{x} \mid \underbrace{\boldsymbol{A}'\boldsymbol{y}' \geq \alpha' \cdot \boldsymbol{b}'}_{\mathbf{\Lambda}_1},\ \underbrace{\boldsymbol{A}''\boldsymbol{y}'' \geq \alpha'' \cdot \boldsymbol{b}''}_{\mathbf{\Lambda}_2},\ \underbrace{\alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \boldsymbol{y}' + \boldsymbol{y}''}_{\mathbf{\Lambda}_3}\}$$

Note that equation (7.2) holds whatever the value of variables α', α'', $\boldsymbol{y}'$ and $\boldsymbol{y}''$. The key idea is to assign values to the projected variables α', α'', $\boldsymbol{y}'$ and $\boldsymbol{y}''$ in encoding (7.1). Considering assignment $\sigma_1 \stackrel{\text{def}}{=} (\alpha' = 1,\ \alpha'' = 0,\ \boldsymbol{y}'' = \mathbf{0})$, it becomes

$$\{\boldsymbol{x} \mid \underbrace{\boldsymbol{A}'\boldsymbol{y}' \geq \boldsymbol{b}'}_{\mathbf{\Lambda}_1},\ \underbrace{\mathbf{0} \geq \mathbf{0}}_{\mathbf{\Lambda}_2},\ \underbrace{1 \geq 0,\ 0 \geq 0,\ 1 + 0 = 1,\ \boldsymbol{x} = \boldsymbol{y}'}_{\mathbf{\Lambda}_3}\}$$

that simplifies into

$$\{\boldsymbol{x} \mid \underbrace{\boldsymbol{A}'\boldsymbol{x} \geq \boldsymbol{b}'}_{\mathbf{\Lambda}_1},\ \underbrace{1 \geq 0}_{\mathbf{\Lambda}_3}\} \tag{7.3}$$

which is equivalent to $\mathcal{P}'$. Let us call λ the coefficient of $1 \geq 0$ in $\mathbf{\Lambda}_3$. Then, we deduce from $(\mathbf{\Lambda}_1 \cdot \boldsymbol{A}')\boldsymbol{x} \geq \boldsymbol{b}' + \lambda \cdot (1 \geq 0) = \mathcal{P}$ that $\mathcal{P}' \Rightarrow \mathcal{P}$. The same reasoning applied with assignment $\sigma_2 \stackrel{\text{def}}{=} (\alpha' = 0,\ \alpha'' = 1,\ \boldsymbol{y}' = \mathbf{0})$ leads to $\mathcal{P}'' \Rightarrow \mathcal{P}$.

7.3.4 Proving Join with a Direct Product of Polymorphic Factories

In PFS, the oracle of `join` has the following type :

```
Back.join : 'c1 lcf -> (BackCstr.t * 'c1) list ->
            'c2 lcf -> (BackCstr.t * 'c2) list -> 'c1 list * 'c2 list
```

Polyhedra $\mathcal{P}'$ and $\mathcal{P}''$ come with their own polymorphic type, respectively `'c1` and `'c2`. The polymorphic type of `Back.join` ensures that it returns a pair of polyhedra $(\mathcal{P}_1, \mathcal{P}_2)$ of type `'c1 list * 'c2 list` such that $\mathcal{P}' \Rightarrow \mathcal{P}_1$ and $\mathcal{P}'' \Rightarrow \mathcal{P}_2$. In practice, $\mathcal{P}_1$ and $\mathcal{P}_2$ should represent the same polyhedron. As a consequence, `Back.join` must take as parameters two factories, one for each polymorphic type.

We said that to compute the convex hull, `join` eliminates variables α', α'', $\boldsymbol{y}'$ and $\boldsymbol{y}''$ from $\mathcal{H}$. Recall that the projection operator that we defined for PFS in Section 7.1.3 has type

```
Back.proj: 'c lcf -> (BackCstr.t * 'c) list -> Var.t -> 'c list
```

As we did for the certificate approach, let us define `Back.proj_list` that extends `Back.proj` by eliminating a list of variables.

```
Back.proj_list: 'c lcf -> (BackCstr.t * 'c) list -> Var.t list -> 'c list
```

At this point, we instantiate the factory of `Back.proj_list` in order to produce the pair of polyhedra with their two distinct polymorphic types. Indeed, although the parameter `'c lcf` of `Back.proj_list` is designed to be provided by the frontend, nothing prevents it from being tuned by the backend. This is where the flexibility of PFS comes into play! We combine the two factories of types `'c1 lcf` and `'c2 lcf` into a new one of type `('c1*'c2)lcf` as follows.

```
let factory_product (lcf1: 'c1 lcf) (lcf2: 'c2 lcf) : ('c1 * 'c2) lcf =
{
  top = (lcf1.top, lcf2.top);
  add  = (fun (c1,c2) (c1',c2') -> lcf1.add c1 c1', lcf2.add c2 c2');
  mul = (fun r (c,c') -> lcf1.mul r c, lcf2.mul r c');
  weaken = (fun (c,c') -> lcf1.weaken c, lcf2.weaken c');
  cte = (fun r cmp (c,c') -> lcf1.cte r cmp c, lcf2.cte r cmp c');
  merge = (fun (c1,c1') (c2,c2') ->  lcf1.merge c1 c1', lcf2.merge c2 c2');
}
```

This new factory computes with frontend constraints from $\mathcal{P}'$ and $\mathcal{P}''$ in parallel: it corresponds to the *direct product* of the two Farkas factories. Still, to be able to use such a factory, each backend constraint must be attached to a frontend constraint of type `'c1 * 'c2`.

Constraints of $\mathcal{P}'$ – that have type `(BackCstr.t * 'c1)` – are converted into type `(BackCstr.t * ('c1 * 'c2))` by being attached to constraint `lcf2.top` of type `'c2`. Similarly, constraints of type `(BackCstr.t * 'c2)` are attached to constraint `lcf1.top` of type `'c1`. Then, we apply the changes of variable $\boldsymbol{y}' := \alpha' \cdot \boldsymbol{x}'$ and $\boldsymbol{y}'' := \alpha'' \cdot \boldsymbol{x}''$ that occur in the encoding of `join` as a projection, explained in §3.6.1. But actually, we *do not need* to apply these changes of variables on frontend constraints. As mentioned earlier, the two Farkas combinations of `join` are found by evaluating the result of the projection of $\mathcal{H}$ on two assignments, respectively σ_1 and σ_2. This evaluation makes variables $\boldsymbol{y}'$ and $\boldsymbol{y}''$ both vanish, as in Equation (7.3). Thus, to build the frontend version of constraints of $\mathcal{H}$, we evaluate them directly on each assignment as follows:

$$
\begin{array}{lcccccc}
 & \texttt{BackCstr.t} & \to & \texttt{BackCstr.t} & * & (\ \texttt{'c1} & * \ \texttt{'c2} \ ) \\
\boldsymbol{A}'\boldsymbol{x}' \geq \boldsymbol{b}' \left\{ \begin{array}{l} \\ \\ \end{array} \right. & \boldsymbol{A}'_1\boldsymbol{x}' \geq b'_1 & \to & \boldsymbol{A}'_1\boldsymbol{y}' \geq b'_1 & , & (\ \underbrace{[\boldsymbol{A}'_1\boldsymbol{y}' \geq \alpha' b'_1]_{\sigma_1}}_{\boldsymbol{A}'_1\boldsymbol{x}' \geq b'_1} & , \ [0=0]_{\sigma_2} \ ) \\
 & \overset{\cdots}{\boldsymbol{A}'_p\boldsymbol{x}'' \geq b'_p} & \to & \boldsymbol{A}'_p\boldsymbol{y}' \geq b'_p & , & (\ \underbrace{[\boldsymbol{A}'_p\boldsymbol{y}' \geq \alpha' b'_p]_{\sigma_1}}_{\boldsymbol{A}'_p\boldsymbol{x}' \geq b'_p} & , \ [0=0]_{\sigma_2} \ ) \\
\boldsymbol{A}''\boldsymbol{x}'' \geq \boldsymbol{b}'' \left\{ \begin{array}{l} \\ \\ \end{array} \right. & \boldsymbol{A}''_1\boldsymbol{x}'' \geq b''_1 & \to & \boldsymbol{A}''_1\boldsymbol{y}'' \geq b''_1 & , & (\ [0=0]_{\sigma_1} & , \ \underbrace{[\boldsymbol{A}''_1\boldsymbol{y}'' \geq \alpha'' b''_1]_{\sigma_2}}_{\boldsymbol{A}''_1\boldsymbol{x}'' \geq b''_1} \ ) \\
 & \overset{\cdots}{\boldsymbol{A}''_q\boldsymbol{x}' \geq b''_q} & \to & \boldsymbol{A}''_q\boldsymbol{y}'' \geq b''_q & , & (\ [0=0]_{\sigma_1} & , \ \underbrace{[\boldsymbol{A}''_q\boldsymbol{y}'' \geq \alpha'' b''_q]_{\sigma_2}}_{\boldsymbol{A}''_q\boldsymbol{x}'' \geq b''_q} \ )
\end{array}
$$

Finally, we add constraints $\alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1$. As all the others, these constraints need to have type `BackCstr.t * ('c1 * 'c2)`. However, they contain variables α' and α'' that were not present in the input polyhedra $\mathcal{P}'$ and $\mathcal{P}''$. Here again, we build directly their evaluation in σ_1 and σ_2. Constraints $1 \geq 0$ and $0 \geq 0$ are built in types `'c1` or `'c2` thanks to operator `cte` from factories `lcf1` and `lcf2`. Note that $\alpha' + \alpha'' = 1$ is not given here because it evaluates to $(0=0, 0=0)$, and can therefore be discarded.

$$
\begin{array}{lcccccc}
\texttt{BackCstr.t} & \to & \texttt{BackCstr.t} & * & (\ \texttt{'c1} & * \ \texttt{'c2} \ ) \\
\alpha' \geq 0 & \to & \alpha' \geq 0 & , & (\ \underbrace{[\alpha' \geq 0]_{\sigma_1}}_{1 \geq 0} & , \ \underbrace{[\alpha' \geq 0]_{\sigma_2}}_{0 \geq 0} \ ) \\
\alpha'' \geq 0 & \to & \alpha'' \geq 0 & , & (\ \underbrace{[\alpha'' \geq 0]_{\sigma_1}}_{0 \geq 0} & , \ \underbrace{[\alpha'' \geq 0]_{\sigma_2}}_{1 \geq 0} \ )
\end{array}
$$

Example 7.5 (follows 7.4)

Let us focus on the proof that $\mathcal{P}'$ and $\mathcal{P}''$ both imply $-x_1 - x_2 \geq -1$, which is a constraint of $\mathcal{P}' \sqcup \mathcal{P}''$. We build $\mathcal{H}$ as described above, and obtain from its projection a frontend

constraint, that is

$$(-x_1 \geq 0,\ 0 \geq 0) + (-x_2 \geq 0,\ 0 \geq 0) + (1 \geq 0,\ 0 \geq 0) + (0 \geq 0,\ -x_1 - x_2 \geq -1)$$
$$= (-x_1 - x_2 \geq -1,\ -x_1 - x_2 \geq -1)$$

The left hand side of each term is the frontend constraint of type `'c1`, and the one on the right hand side is of type `'c2`. From $\mathcal{P}'$ point of view, we obtain $-x_1 - x_2 \geq -1$ as the combination of $-x_1 \geq 0$, $-x_2 \geq 0$ and the constant constraint $1 \geq 0$ that comes from $\alpha' \geq 0$. On the other hand, $-x_1 - x_2 \geq -1$ is a constraint of $\mathcal{P}''$ and is directly returned as a frontend constraint of type `'c2`. The projection returns such results for each constraint of the convex hull $\mathcal{P}' \sqcup \mathcal{P}''$.

In conclusion, with a well chosen factory, we define our PFS `join` as a simple call to `proj_list`. This makes our implementation much simpler than Fouilhé's one, where the two certificates of `join` are obtained from the one of `proj_list` by tedious rewritings that perform on-the-fly renamings of constraint identifiers.

7.4 Related Works

The skeptical approach has been pioneered in the design of two interactive provers, Automath (de Bruijn, 1968) and LCF (Gordon et al., 1979). Both provers reduce the soundness of a rich mathematical framework to the correctness of a small automatic proof checker called the kernel. But, their style is very different. LCF is written as a library in a functional programming language (ML) which provides the type of theorems as an abstract datatype. Its safety relies on the fact that objects of this type can only be defined from a few primitives (*i.e.* the kernel). Each of them corresponds to an inference rule of Higher-Order Logic in natural deduction. On the contrary, Automath introduces a notion of *proof object* and implements the kernel itself as a typechecker, thanks to Curry-Howard isomorphism. LCF style is more lightweight – both for the development and the execution of proof tactics – whereas the proof object style allows a richer logic (*e.g.* with *dependent types*). Nowadays, the kernel of skeptical interactive provers is still designed according to one of this style: Coq has proof objects whereas HOL provers are in LCF style.

Since the 90's, the skeptical approach is also applied in two kinds of slightly different contexts: making interactive provers communicate with external solvers like Maple (Harrison and Théry, 1998), and verifying the safety of untrusted code, like in "*Proof Carrying Code*" (Necula, 1997). In Coq, it is also applied to the design of proof tactics communicating with external solvers (Besson, 2006; Grégoire et al., 2008; Armand et al., 2010; Blech and Grégoire, 2011; Armand et al., 2011; Magron et al., 2015), and to certify stand-alone programs like compilers or static analyzers which embed some untrusted code (Tristan and Leroy, 2008; Besson et al., 2010; Jourdan et al., 2015; Blazy et al., 2015).

Beyond interactive provers, producing certificates of unsatisfiability has become mandatory for state-of-the-art Boolean SAT-solvers. Indeed, certificates of unsatisfiability have been required for the UNSAT tracks since SAT Competition 2013. In 2016, they were required – in DRAT format (Wetzler et al., 2014) – for all solvers in the *Main* track of the SAT Competition.

Actually, there are now so many works related to the skeptical approach that it seems impossible to be exhaustive. With respect to all these works, we propose a design pattern, the Polymorphic LCF Style (abbreviated as PFS), in order to certify in Coq the results of an untrusted ML oracle. We illustrated this degisn on the new implementation of the VPL. In PFS, oracles produce these witnesses as ordinary ML values (*e.g.* linear constraints). In other words, instead of building an AST that the Coq frontend uses to compute the certified value, the oracle directly generates this value by using certified operators of the Coq frontend. This provides several advantages over AST style. First, it makes the oracle development easier. Ill-handling of certified operators is much straightforward to debug. Without an AST to build, it naturally removes cumbersome details such as handling of binders. Second, polymorphism ensures that oracle results are sound by construction. In the polyhedra library, it means that

oracles can only produce logical consequences of their input. This property is proved for free from the types of the oracles, in the spirit of the "theorems for free" coined by Wadler (1989). At last, polymorphism makes witness generation very flexible and modular. Generating a compact AST is still possible if necessary. The next chapter applies this process for the embedding of an oracle within a Coq tactic.

We strongly believe that PFS could be used with other applications. For instance, the nonlinear support based on Handelman's theorem that we explained in §4.4 could be easily certified using a factory that provides nonlinear multiplications. Actually, we provide a proof in Coq of this linearization in Chapter 9. This proof is anterior to our PFS development and uses a standard skeptical approach by certificates. Chapter 9 will show that the handling of certificates and their associated proofs are much more complex than the building of a factory.

Also, certifying UNSAT answers of a Boolean SAT-Solver can be achieved by a similar approach, by using resolution proofs instead of Farkas certificates (Keller, 2013). This seems to indicate than our approach is also relevant to a large class of coNP-hard problems.

Chapter 8

A Coq Tactic for Equality Learning

Several Coq tactics solve goals containing linear inequalities: `omega` and `lia` on integers; `fourier` or `lra` on reals and rationals (The Coq Development Team, 2016; Besson, 2006). In this chapter, we provide yet another tactic for proving such goals. This tactic – called `vpl`[1] – is currently limited to rationals. It is built on the top of the VPL and its main feature appears when it *cannot prove* the goal. In this case, whereas above tactics fail, our tactic "simplifies" the goal. In particular, it injects as hypotheses a *complete* set of linear equalities that are deduced from the linear inequalities in the context. Then, many Coq tactics – such as `congruence`, `field` or even `auto` – can exploit these equalities, even if they cannot deduce them from the initial context by themselves. By simplifying the goal, our tactic both improves the user experience and proof automation.

Let us illustrate this feature on the following – almost trivial – Coq goal, where `Qc` is the type of rationals on which our tactic applies.

```
Lemma ex1 (x:Qc) (f:Qc → Qc):
  x≤1 → (f x)<(f 1) → x<1.
```

This goal is valid on `Qc` and `Z`, but both `omega` and `lia` fail on the `Z` instance without providing any help to the user. Indeed, since this goal contains an uninterpreted function `f`, it does not fit into the pure linear arithmetic fragment. On the contrary, this goal is proved by two successive calls to the `vpl` tactic. As detailed below, equality learning plays a crucial role in this proof: the rewriting of a learned equality inside a non-linear term (because under symbol `f`) is interleaved between deduction steps in linear arithmetic. Of course, such a goal is also provable in `Z` by SMT solving tactics, such as the `verit` tactic of SMTCoq (Armand et al., 2011) or the one of Besson et al. (2011). However, such SMT tactics are also "*prove-or-fail*": they do not simplify the goal when they cannot prove it. Conversely, our tactic may help users in their interactive proofs, by simplifying goals that do not fully fit into the scope of existing SMT solving procedures.

The tactic learns equalities from conflicts between strict inequalities detected by a LP solver. This algorithm can be viewed as a special but optimized case of "*conflict driven clause learning*" – at the heart of modern DPLL procedures (Silva et al., 2009). On most cases, it is strictly more efficient than the naive equality learning algorithm previously implemented in the VPL. In particular, our algorithm is cheap when there is no equality to learn. We have implemented this algorithm in an OCaml oracle, able to produce *proof witnesses* for these equalities. We will detail the generation of witnesses. Their embedding in Coq proofs was done by Sylvain Boulmé, and we will not address this here. In particular, it exploits the PFS framework of Chapter 7 to produce optimized certificates. For more details, refer to (Boulmé and Maréchal, 2017b).

1. Available on `http://github.com/VERIMAG-Polyhedra/VplTactic`.

8.1 Specification of the VPL Tactic

Let us now introduce the specification of the `vpl` tactic. As mentioned above, the core of the tactic is performed by an oracle programmed in OCaml, and called `reduce`. This oracles takes as input a polyhedron $\mathcal{P}$ and outputs a *reduced polyhedron* $\mathcal{P}'$ such that $\mathcal{P}' \Leftrightarrow \mathcal{P}$ and such that the *number of constraints* in $\mathcal{P}'$ is lower or equal to that of $\mathcal{P}$.

As we saw in Chapter 2, a polyhedron may be suboptimally written. In particular, some of its constraints may be implied by the others. Moreover, a set of inequalities can imply *implicit* equalities, such as $x = 0$ that can be deduced from $x \geq 0 \wedge -x \geq 0$. Definition 8.2 characterizes polyhedra without *implicit* equalities.

Definition 8.1 (Complete set of linear equalities) *Let E be a set of linear equalities and I be a set of linear inequalities. E is said* complete *w.r.t. I if any linear equality deduced from the conjunction $E \wedge I$ can also be deduced from E alone, meaning that I contains no equality, neither implicit nor explicit. Formally, E is complete if and only if for all linear terms t_1 t_2,*

$$(E \wedge I \Rightarrow t_1 = t_2) \textit{ implies } (E \Rightarrow t_1 = t_2) \tag{8.1}$$

Definition 8.2 (Reduced Polyhedron) *A polyhedron $\mathcal{P}$ is* reduced *if and only if it satisfies the following condition.*

- *If $\mathcal{P}$ is unsatisfiable, then $\mathcal{P}$ is a* single constant constraint *like $0 > 0$ or $0 \geq 1$. In other words, its unsatisfiability is checked by one comparison on $\mathbb{Q}$.*
- *Otherwise, $\mathcal{P}$ contains no* redundant *constraint and is syntactically given as a conjunction $E \wedge I$ where polyhedron I contains only inequalities and where polyhedron E is a* complete *set of equalities w.r.t. I.*

Having a reduced polyhedron ensures that any provable linear equality admits a pure equational proof which ignores the remaining inequalities.

Specification of the Tactic. Roughly speaking, a Coq goal corresponds to a sequent $\Gamma \vdash T$ where context Γ represents a conjunction of hypotheses and T a conclusion. In other words, this goal is logically interpreted as the meta-implication $\Gamma \Rightarrow T$. The tactic transforms the current goal $\Gamma \vdash T$ through three successive steps.

- **First Step.** The goal is equivalently rewritten into

  $$\Gamma', [\![\mathcal{P}]\!](m) \vdash T'$$

  where $\mathcal{P}$ is a polyhedron and m an assignment of $\mathcal{P}$ variables. For example, the `ex1` goal is rewritten as $[\![\mathcal{P}_1]\!](m_1) \vdash$ `False`, where

  $$\mathcal{P}_1 \stackrel{\text{def}}{=} \{x_1 \leq 1,\ x_2 < x_3,\ x_1 \geq 1\}$$
  $$m_1 \stackrel{\text{def}}{=} \{\ x_1 \mapsto \texttt{x};\ x_2 \mapsto \texttt{(f x)};\ x_3 \mapsto \texttt{(f 1)}\ \}$$

  Hence, $[\![\mathcal{P}]\!](m)$ corresponds to a conjunction of inequalities on $\mathbb{Q}$ that *are not necessarily* linear, because m may assign arbitrary Coq terms on $\mathbb{Q}$ to variables of $\mathcal{P}$. Actually, $[\![\mathcal{P}]\!](m)$ contains at least all (in)equalities on $\mathbb{Q}$ that appear as hypotheses of Γ. Moreover, if T is an inequality on $\mathbb{Q}$, then an inequality equivalent to $\neg T$ appears in $[\![\mathcal{P}]\!](m)$ and T' is proposition `False`.[2] This step is traditionally called *reification* in Coq tactics.
- **Second Step.** The goal is equivalently rewritten into

  $$\Gamma', [\![\mathcal{P}']\!](m) \vdash T'$$

  where $\mathcal{P}'$ is the *reduced polyhedron* computed from $\mathcal{P}$ by our `reduce` oracle. For instance, polyhedron $\mathcal{P}_1$ found above is reduced into

  $$\mathcal{P}'_1 \stackrel{\text{def}}{=} \{x_1 = 1\} \wedge \{x_2 < x_3\}$$

2. Here, $T \Leftrightarrow (\neg T \Rightarrow$ `False`$)$ because comparisons on $\mathbb{Q}$ are decidable.

– **Third Step.** If $\mathcal{P}'$ is unsatisfiable, then so is $[\![\mathcal{P}']\!](m)$, and the goal is finally discharged. Otherwise, given E the complete set of equalities in $\mathcal{P}'$, equalities of $[\![E]\!](m)$ are rewritten in the goal. For example, on the `ex1` goal, our tactic rewrites the learned equality "`x=1`" into the remaining hypothesis. In summary, a first call to the `vpl` tactic transforms the `ex1` goal into

```
x=1, (f 1)<(f 1) ⊢ False
```

A second call to `vpl` detects that hypothesis `(f 1)<(f 1)` is unsatisfiable and finally proves the goal.

In the description above, we claim that our transformations on the goals are equivalences. This provides a guarantee to the user: the tactic can always be applied on the goal, without loss of information. However, in order to make the Coq proof checker accept our transformations, we only need to prove implications, as detailed in the next paragraph.

The Proof Built by the Tactic. The tactic mainly proves the two following implications which are verified by the Coq kernel:

$$\Gamma', [\![\mathcal{P}]\!](m) \vdash T' \;\Rightarrow\; \Gamma \vdash T \tag{8.2}$$

$$\forall m, [\![\mathcal{P}]\!](m) \;\Rightarrow\; [\![\mathcal{P}']\!](m) \tag{8.3}$$

Semantics of polyhedra $[\![.]\!]$ is encoded as a Coq function, using binary integers to encode variables of polyhedra. After simple propositional rewritings in the initial goal $\Gamma \vdash T$, an OCaml oracle provides m and $\mathcal{P}$ to the Coq kernel, which simply computes $[\![\mathcal{P}]\!](m)$ and checks that it is syntactically equal to the expected part of the context. Hence, verifying implication (8.2) is mainly syntactical.

For implication (8.3), our `reduce` oracle actually produces a Coq AST, that represents a *proof witness* that allows expressing each constraint of $\mathcal{P}'$ as a Farkas combination of $\mathcal{P}$ constraints. In practice, this proof witness is a value of a Coq inductive type. A Coq function called `reduceRun` takes as input a polyhedron $\mathcal{P}$ and its associated witness, and computes $\mathcal{P}'$. A Coq theorem ensures that any result of `reduceRun` satisfies implication (8.3). Thus, this implication is ensured by construction, while – for the last step of the tactic described above – the Coq kernel computes $\mathcal{P}'$ by applying `reduceRun`.

8.2 Using the Tactic

Combining solvers by exchanging equalities is one of the basis of modern SMT solving, as pioneered by approaches of Nelson-Oppen (Oppen, 1980) and Shostak (1984). This section illustrates how equality learning in a interactive prover mimics such equality exchange, in order to combine independent tactics. While much less automatic than standard SMT solving, our approach provides opportunities for the user to compensate "by hand" for the weaknesses of a given tactic.

The main aspects of the `vpl` tactic are illustrated on the following goal. It contains two uninterpreted functions `f` and `g` such that `f`'s domain and `g`'s codomain are the same uninterpreted type `A`. As we will see below, in order to prove this goal, we need to use its last hypothesis – of the form "`g(..) <> g(13)`" – by combining equational reasoning on `g` and on `Qc` field. Of course, we also need linear arithmetic on `Qc` order.

```
Lemma ex2 (A:Type) (f:A → Qc) (g:Qc → A)
  (v1 v2 v3 v4:Qc) :
  6*v1 - v2 - 10*v3 + 7*(f(g v1)+1) ≤ -1
  → 3*(f(g v1)-2*v3)+4 ≥ v2-4*v1
  → 8*v1 - 3*v2 - 4*v3 - f(g v1) ≤ 2
  → 11*v1 - 4*v2 > 3
  → v3 > -1
  → v4 ≥ 0
  → g((11-v2+13*v4)/(v3+v4)) <> g(13)
  → 3 + 4*v2 + 5*v3 + f(g v1) > 11*v1.
```

The vpl tactic reduces this goal to the equivalent one given below (where typing of variables is omitted).

```
H5: g((11-(11-13*v3)+13*v4)/(v3+v4))=(g 13)
    → False
vpl: v1 = 4-4*v3
vpl0: v2 = 11-13*v3
vpl1: f(g(4-4*v3)) = -3+3*v3
______________________________________(1/1)
0 ≤ v4 → (3#8) < v3 → False
```

Here, three equations vpl, vpl0 and vpl1 have been learned from the goal. Two irredundant inequalities remain in the hypotheses of the conclusion – where (3#8) is the Coq notation for $\frac{3}{8}$. The bound $v3 > -1$ had disappeared because it is implied by $(3\#8) < v3$. By taking $v3 = 1$, we can build a model satisfying all the hypotheses of the goal – including $(3\#8) < v3$ – except H5. Thus, using H5 is necessary to prove False.

Actually, we provide another tactic which automatically proves the remaining goal. This tactic, called vpl_post, combines equational reasoning on Qc field with a bit of congruence.[3] Let us detail how it works on this example. First, in backward reasoning, hypothesis H5 is applied to eliminate False from the conclusion. We get the following conclusion (where previous hypotheses have been omitted).

```
______________________________________(1/1)
g((11-(11-13*v3)+13*v4)/(v3+v4))=(g 13)
```

Here, backward congruence reasoning reduces this conclusion to

```
______________________________________(1/1)
(11-(11-13*v3)+13*v4)/(v3+v4)=13
```

Now, the field tactic reduces the conclusion to

```
______________________________________(1/1)
v3+v4 <> 0
```

Indeed, the field tactic mainly applies ring rewritings on Qc while generating subgoals for checking that denominators are not zero. Here, because we have a linear denominator, we discharge the remaining goal using the vpl tactic again. Indeed, it gets the following unsatisfiable polyhedron in hypotheses.

$$v4 \geq 0 \quad \wedge \quad v3 > \frac{3}{8} \quad \wedge \quad v3 + v4 = 0$$

Let us remark that lemma ex2 is also valid when the codomain of f and types of variables v1,...,v4 are restricted to $\mathbb{Z}$ and operator "/" means the Euclidean division. However, both omega and lia fail on this goal without providing any help to the user. This is also the case of the verit tactic of SMTCoq because it deals with "/" as a non-interpreted symbol and can only deal with uninterpreted types A providing a decidable equality. By assuming a decidable equality on type A and by turning the hypothesis involving "/" into "g((11-v2+13*v4)) <> g(13*(v3+v4))", we get a slightly weaker version of ex2 goal which is proved by verit.

This illustrates that our approach is complementary to SMT solving: it provides less automation than SMT solving, but it may still help to progress in an interactive proof when SMT solvers fail.

8.3 The Reduction Algorithm

The specification of the reduce oracle is given in §8.1: it transforms a polyhedron $\mathcal{P}$ into a reduced polyhedron $\mathcal{P}'$ with a smaller number of constraints and such that $\mathcal{P}' \Leftrightarrow \mathcal{P}$. §8.3.3 and §8.3.4 describe our implementation. In preliminaries, §8.3.1 gives a sufficient condition,

3. It is currently implemented on the top of auto with a dedicated basis of lemma.

through Lemma 8.6, for a polyhedron to be reduced. This condition leads to learn equalities from conflicts between strict inequalities as detailed in §8.3.2 and §8.3.3. In our proofs and algorithms, we only handle linear constraints in the restricted form $t \bowtie 0$. But, for readability, our examples use the arbitrary form $t_1 \bowtie t_2$.

8.3.1 A Refined Specification of the Reduction

Definition 8.3 (Echelon Polyhedron) *An echelon polyhedron is written as a conjunction $E \wedge I$ where polyhedron I contains only inequalities and where polyhedron E is written "$\bigwedge_{i\in\{1,\ldots,k\}} x_i - t_i = 0$" such that each x_i is a variable and each t_i is a linear term, and such that the two following conditions are satisfied. First, no variable x_i appears in polyhedron I. Second, for all integers $i, j \in \{1, \ldots, k\}$ with $i < j$ then x_i does not appear in t_j.*

Intuitively, in such a polyhedron, each equation $x_i - t_i = 0$ actually *defines* variable x_i as t_i. As a consequence, $E \wedge I$ is satisfiable if and only if I is satisfiable.

Definition 8.4 (Strict Version of Inequalities) *Let I be a polyhedron containing only inequalities. We note $I^>$ the polyhedron obtained from I by replacing each nonstrict inequality "$t \geq 0$" by its strict version "$t > 0$". Strict inequalities of I remain unchanged in $I^>$.*

Geometrically, polyhedron $I^>$ is the interior of polyhedron I. Hence if $I^>$ is satisfiable (*i.e.* the interior of I is non empty), then polyhedron I does not fit inside an hyperplane. The following Lemma 8.6 is only a reformulation of this trivial geometrical fact. Let us first introduce another corollary of Farkas' lemma that will be useful for the proof of Lemma 8.6.

Corollary 8.5 *Let us consider a* satisfiable *polyhedron I written $\bigwedge_{j=1}^{k} t_j \bowtie_j 0$ with $\bowtie_j \in \{\geq, >\}$. Then, $I^>$ is unsatisfiable if and only if there exists k nonnegative rationals $(\lambda_j)_{j\in\{1,\ldots,k\}} \in \mathbb{Q}+$ such that $\sum_{j=1}^{k} \lambda_j t_j = 0$ and $\exists i \in \{1, \ldots, k\},\ \lambda_i > 0$.*

Proof.
($\Leftarrow$): Suppose k nonnegative rationals $(\lambda_j)_{j\in\{1,\ldots,k\}}$ such that $\sum_{j=1}^{k} \lambda_j t_j = 0$ and some index $i \in \{1, \ldots, k\}$ such that $\lambda_i > 0$. It means that there is a Farkas combination of $0 > 0$ in terms of $I^>$. Thus by Farkas' lemma, $I^>$ is unsatisfiable.
($\Rightarrow$): Let us assume that $I^>$ is unsatisfiable. Then there exists k nonnegative rationals $(\lambda_j)_{j\in\{1,\ldots,k\}}$ such that at least one of them is positive and $\sum_{j=1}^{k} \lambda_j t_j = -\lambda$, with $\lambda \geq 0$. This result is an extension of Corollary 1.6 (p.23). Let $\boldsymbol{x}$ be a point of I. By definition, we have $\sum_{j=1}^{k} \lambda_j t_j(\boldsymbol{x}) = \lambda'$ with $\lambda' \in \mathbb{Q}+$. But since I is satisfiable, any nonnegative combination of constraints of I is feasible, *i.e.* $\sum_{j=1}^{k} \lambda_j t_j = \lambda'$ with $\lambda' \in \mathbb{Q}+$. Thus, $-\lambda = \lambda' = 0$. □

Lemma 8.6 (Completeness from Strict Satisfiability) *Let us assume an echelon polyhedron $E \wedge I$ without redundant constraints, and such that $I^>$ is satisfiable.*
Then, $E \wedge I$ is a reduced polyhedron.

Proof. Let us prove property (8.1) of Definition 8.1, *i.e.* that E is complete w.r.t. I. Because $t_1 = t_2 \Leftrightarrow t_1 - t_2 = 0$, without loss of generality, we only prove property (8.1) in the case where $t_2 = 0$ and t_1 is an arbitrary linear term t.

Let t be a linear term such that $E \wedge I \Rightarrow t = 0$. In particular, $E \wedge I \Rightarrow t \geq 0$. By Farkas' lemma, there are $k + 1$ nonnegative rationals $(\lambda_j)_{j\in\{0,\ldots,k\}}$ such that $t = \lambda_0 + \sum_{j=1}^{k} \lambda_j t_j$. Moreover, since $I^>$ is satisfiable, then by Corollary 8.5, forall $(\lambda'_j)_{j=1}^{k} \in \mathbb{Q}+$, $\sum_{j=1}^{k} \lambda'_j t_j > 0$.

Suppose by contradiction that a constraint of I appears in the Farkas combination of t in terms of $E \wedge I$. Then, the Farkas combination $t = \lambda_0 + \sum_{j=1}^{k} \lambda_j t_j$ is positive, which contradicts the initial hypothesis $t = 0$. Thus, $E \Rightarrow t \geq 0$.

A similar reasoning with $E \wedge I \Rightarrow t \leq 0$ finishes the proof that $E \Rightarrow t = 0$. □

Lemma 8.6 gives a strategy to implement the `reduce` oracle. If the input polyhedron P is satisfiable, then try to rewrite P as an echelon polyhedron $E \wedge I$ where $I^>$ is satisfiable. The next step is to see that from an echelon polyhedron $E \wedge I$ where $I^>$ is unsatisfiable, we can learn new equalities from a minimal subset of $I^>$ inequalities that is unsatisfiable. The inequalities in such a minimal subset are said *in conflict*.

8.3.2 Conflict Driven Equality Learning

Conflict Driven Clause Learning (CDCL) is a standard framework of modern DPLL SAT solving (Silva et al., 2009). Given a set of nonstrict inequalities I, we reformulate the satisfiability of I into this framework by considering each nonstrict constraint $t \geq 0$ of I as a clause $(t > 0) \vee (t = 0)$. Hence, our literals are either strict inequalities or equalities.

Let us run a CDCL SAT solver on such a set of clauses I. This "thought experiment" will simply help to interpret our equality learning algorithm – presented in the next sections – as a particular optimization of the generic clause learning algorithm. First, let us imagine that the SAT solver assumes all literals of $I^>$. Then, an oracle decides whether $I^>$ is satisfiable. If so, then we are done. Otherwise, by Corollary 8.5, the oracle returns the unsatisfiable constant constraint $0 > 0$ that is written $\sum_{j\in J} \lambda_j t_j$ where for all $j \in J$, $\lambda_j > 0$ and $(t_j > 0) \in I^>$. The CDCL solver learns the new clause $\bigvee_{j\in J} t_j = 0$ equivalent to $\neg I^>$ under hypothesis I.

In fact, a simple arithmetic argument improves this naive CDCL algorithm by learning directly the conjunction of literals $\bigwedge_{j\in J} t_j = 0$ instead of the clause $\bigvee_{j\in J} t_j = 0$. Indeed, since $\sum_{j\in J} \lambda_j t_j = 0$ (by Corollary 8.5) and $\forall j \in J,\ \lambda_j > 0$, then each term t_j of this sum must be 0. Thus, $\forall j \in J,\ t_j = 0$.

In the following, we learn equalities from conflicts between strict inequalities in an approach inspired from this naive CDCL algorithm. Whereas the number of oracle calls for learning n equalities in the naive CDCL algorithm is $\Omega(n)$, our additional arithmetic argument limits this number to $\mathcal{O}(1)$ in the best cases.

8.3.3 Building Equality Witnesses from Conflicts

Let us now detail our algorithm to compute equality witnesses. Let I be a satisfiable inequality set such that $I^>$ is unsatisfiable. The oracle returns a witness combining $n + 1$ constraints of $I^>$ (for $n \geq 1$) that implies a contradiction:

$$\sum_{i=1}^{n+1} \lambda_i \cdot I_i^> \quad \text{where} \quad \lambda_i > 0$$

By Corollary 8.5, this witness represents a contradictory constraint $0 > 0$ and each inequality I_i is nonstrict. Each inequality I_i is turned into an equality written $I_i^=$, which is proved by

$$I_i \ \&\ \frac{1}{\lambda_i} \cdot \sum_{j=1,\ j\neq i}^{n+1} \lambda_j \cdot I_j$$

where & is the `merge` operator that we defined in §7.3.1. This operator takes two inequalities $t \geq 0$ and $-t \geq 0$ to prove $t = 0$. Here, $\frac{1}{\lambda_i} \cdot \sum_{j=1,\ j\neq i}^{n+1} \lambda_j \cdot I_j = -I_i$. Hence, each equality $I_i^=$ is proved by combining $n + 1$ constraints. Proving $(I_i^=)_{i\in\{1,\ldots,n+1\}}$ in this naive approach combines $\Theta(n^2)$ constraints.

We rather propose a more symmetric way to build equality witnesses which leads to a simple linear algorithm. Actually, we build a system of n equalities noted $(E_i)_{i\in\{1,\ldots,n\}}$ such that, for $i \in \{1, \ldots, n\}$, E_i corresponds to the unsatisfiability witness where the i-th "+" has been replaced with a "&":

$$\left(\textstyle\sum_{j=1}^{i} \lambda_j \cdot I_j\right)\ \&\ \left(\textstyle\sum_{j=i+1}^{n+1} \lambda_j \cdot I_j\right)$$

This system of equations is proved equivalent to system $(I_i^=)_{i\in\{1,\ldots,n+1\}}$ thanks to the following correspondence.

$$\begin{cases} I_1^= = \frac{1}{\lambda_1} \cdot E_1 \\ I_{n+1}^= = -\frac{1}{\lambda_n} \cdot E_n \\ \forall i \in \{2, \ldots, n\}, I_i^= = \frac{1}{\lambda_i} \cdot (E_i - E_{i-1}) \end{cases}$$

This also shows that one equality $I_i^=$ is redundant, because $(I_i^=)_{i\in\{1,\ldots,n+1\}}$ contains one more equality than $(E_i)_{i\in\{1,\ldots,n\}}$.

In order to use a linear number of combinations, we build $(E_i)_{i\in\{1,\dots,n\}}$ thanks to two lists of intermediate constraints $(A_i)_{i\in\{1,\dots,n\}}$ and $(B_i)_{i\in\{2,\dots,n+1\}}$ defined by

$$\begin{cases} A_1 \stackrel{\text{def}}{=} \lambda_1 \cdot I_1 \\ \text{for } i \text{ from 2 up to } n,\ A_i \stackrel{\text{def}}{=} A_{i-1} + \lambda_i \cdot I_i \end{cases}$$
$$\begin{cases} B_{n+1} \stackrel{\text{def}}{=} \lambda_{n+1} \cdot I_{n+1} \\ \text{for } i \text{ from } n \text{ down to 2},\ B_i \stackrel{\text{def}}{=} B_{i+1} + \lambda_i \cdot I_i \end{cases}$$

Then, we build $E_i \stackrel{\text{def}}{=} A_i \ \&\ B_{i+1}$ for $i \in \{1,\dots,n\}$.

Example 8.1

Let us detail the computation of the reduced form of the following polyhedron P.

$$P \stackrel{\text{def}}{=} \begin{cases} I_1 : x_1 + x_2 \geq x_3, \\ I_2 : x_1 \geq -10, \\ I_3 : 3x_1 \geq x_2, \\ I_4 : 2x_3 \geq x_2, \\ I_5 : -\frac{1}{2}x_2 \geq x_1 \end{cases}$$

P is a satisfiable set of inequalities. Thus, we first extract a complete set of equalities E from constraints of P by applying the previous ideas. We ask a LP solver for a point satisfying $P^{>}$, the strict version of P. Because there is no such point, the solver returns the unsatisfiability witness $I_1^{>} + \frac{1}{2}\cdot I_4^{>} + I_5^{>}$ (which reduces to $0 > 0$). By building the two sequences (A_i) and (B_i) defined previously, we obtain the two equalities

$$E_1 : x_1 + x_2 = x_3 \quad \textbf{proved by} \quad \underbrace{(x_1 + x_2 \geq x_3)}_{A_1:\ I_1} \ \&\ \underbrace{(x_3 \geq x_1 + x_2)}_{B_2:\ \frac{1}{2}\cdot I_4 + I_5}$$
$$E_2 : x_1 = -\tfrac{1}{2}x_2 \quad \textbf{proved by} \quad \underbrace{(x_1 \geq -\tfrac{1}{2}x_2)}_{A_2:\ I_1 + \frac{1}{2}\cdot I_4} \ \&\ \underbrace{(-\tfrac{1}{2}x_2 \geq x_1)}_{B_3:\ I_5}$$

Thus, P is rewritten into $E \wedge I$ with

$$E \stackrel{\text{def}}{=} \{E_1 : x_1 + x_2 = x_3,\ E_2 : x_1 = -\tfrac{1}{2}x_2\}$$
$$I \stackrel{\text{def}}{=} \{I_2 : x_1 \geq 10,\ I_3 : 3x_1 \geq x_2\}$$

To be reduced, the polyhedron must be in echelon form, as explained in Definition 8.3. This implies that each equality of E must have the form $x_i - t_i = 0$, and each such x_i must not appear in I. Here, let us consider that E_1 defines x_2. To be in the form $t = 0$, E_1 is rewritten into $x_2 - (x_3 - x_1) = 0$. Then, x_2 is eliminated from E_2, leading to $E_2' : x_1 + x_3 = 0$. In practice, our oracle goes one step further by rewriting x_1 (using its definition in E_2') into E_1 in order to get a reduced echelon system E' of equalities:

$$E' \stackrel{\text{def}}{=} \{E_1' : x_2 - 2x_3 = 0,\ E_2' : x_1 + x_3 = 0\}$$

Then, variables defined in E' (*i.e.* x_1 and x_2) are eliminated from I, which is rewritten into

$$I' \stackrel{\text{def}}{=} \{I_2' : -x_3 \geq -10,\ I_3' : -x_3 \geq 0\}$$

The last step is to detect that I_2' is redundant w.r.t. I_3' with a process indicated in the next section.

8.3.4 Description of the Algorithm

The pseudo-code of Algorithm 8.2 describes the `reduce` algorithm. For simplicity, the construction of proof witnesses is omitted. To summarize, the result of `reduce` is

Algorithm 8.2: Pseudo-code of the `reduce` oracle

```
Function reduce (E ∧ I)
    (E, I) ← echelon(E, I);
    switch isSat (I) do
        case UNSAT (λ) do return CONTRAD(λ^T · I) ;
        case SAT (_) do
            while TRUE do
                switch isSat (I^>) do
                    case UNSAT (λ) do
                        (E', I') ← learn(I, λ);
                        (E, I) ← echelon(E ∧ E', I');
                    case SAT (x) do
                        I ← minimize(I, x);
                        return REDUCED(E, I);
```

— either "`Contrad`(c)" where c is a contradictory constraint

— or "`Reduced`(P')" where P' is a satisfiable reduced polyhedron.

The input polyhedron is assumed to be given in the form $E \wedge I$, where E contains only equalities and I contains only inequalities. First, polyhedron $E \wedge I$ is echeloned: function `echelon` returns a new system $E \wedge I$ where E is an echelon system of equalities without redundancies (they have been detected as $0 = 0$ during echeloning and removed) and without contradiction (they have been detected as $1 = 0$ during echeloning and inserted as a contradictory constraint $-1 \geq 0$ in I). Second, the satisfiability of I is tested by function `is_sat`. If `is_sat` returns `Unsat` $(\boldsymbol{\lambda})$, then $\boldsymbol{\lambda}$ is a Farkas combination yielding a contradictory constant constraint written $\boldsymbol{\lambda}^\intercal \cdot I$. Otherwise, I is satisfiable and `reduce` enters into a loop to learn all implicit equalities.

At each step of the loop, the satisfiability of $I^>$ is tested. If `is_sat` returns `Unsat` $(\boldsymbol{\lambda})$, then a new set E' of equalities is learned from $\boldsymbol{\lambda}$ and I' contains the inequalities of I that do not appear in the conflict. After echeloning the new system, the loop continues.

Otherwise, `is_sat` returns `Sat`$(\boldsymbol{x})$ where $\boldsymbol{x}$ is a model of $I^>$. Geometrically, $\boldsymbol{x}$ is a point in the interior of polyhedron I. Point $\boldsymbol{x}$ helps function `minimize` detect and remove redundant constraints of I: it can be used as an interior point for the raytracing algorithm described in Chapter 2. At last, `reduce` returns $E \wedge I$, which is a satisfiable reduced polyhedron because of Lemma 8.6.

Variant. In a variant of this algorithm, we avoid testing the satisfiability of I before entering the loop (*i.e.* we avoid the first step of the algorithm). Indeed, the satisfiability of I can be directly deduced from the witness returned by `is_sat`$(I^>)$. If the combination of the linear terms induced by the witness gives a negative number instead of 0, it means that I is unsatisfiable. However, we could make several loop executions before finding a witness showing that I is unsatisfiable: I can contain several implicit equalities which do not imply the unsatisfiability of I and that may be discovered first. We do not know which version is the more efficient. It probably depends on the kind of polyhedra the user is upon to use.

8.4 Conclusion

We described a Coq tactic that learns equalities from a set of linear rational inequalities. It is less powerful than Coq SMT tactics (Armand et al., 2011; Besson et al., 2011) and than the famous `sledgehammer` of ISABELLE (Böhme and Nipkow, 2010; Blanchette et al., 2013).

But, it may help users to progress on goals that do not exactly fit into the scope of existing SMT solving procedures.

This tactic uses a simple algorithm that follows a kind of conflict driven clause learning. This equality learning algorithm only relies on an efficient SAT solver on inequalities able to generate nonnegativity witnesses. Hence, it seems generalizable to arbitrary polynomials. We may also hope to generalize it to totally ordered rings like $\mathbb{Z}$.

In the previous version of the VPL, Fouilhé also reduced polyhedra as defined in Definition 8.2. It implemented equality learning in a more naive way: for each inequality $t \geq 0$ of the current (satisfiable) inequalities I, the algorithm checks whether $I \wedge t > 0$ is satisfiable. If not, equality $t = 0$ is learned. In other words, each learned equality derives from one satisfiability test. Our new algorithm is more efficient, since it may learn several equalities from a single satisfiability test. Moreover, when there is no equality to learn, the new algorithm performs only one satisfiability test, whereas the previous version checks all inequalities one by one.

Our tactic is still a prototype. Additional works are required to make it really robust in interactive proofs. For example, the user may need to stop the tactic before the rewritings of the learned equalities are performed, for instance when some rewriting interferes with dependent types. The user can invoke instead a subtactic `vpl_reduce`, and apply these rewritings "by hand". The maintainability of such user scripts thus depends on the stability of the generated equalities and their order w.r.t. small changes in the input goal. A first step toward stability would be to make our tactic idempotent by keeping the goal unchanged on a already reduced polyhedron. However, we have not yet investigated these stability issues.

Chapter 9

Certification of Handelman's Linearization

In §4.4, we presented Handelman's linearization that aims at over-approximating the effect of a polynomial guard $Q \geq 0$ on a polyhedron $\mathcal{P} = \{C_1 \geq 0, \ldots, C_p \geq 0\}$. Let us summarize the whole process.

First, an oracle chooses a set of Schweighofer products $(S^{\boldsymbol{I}})_{\boldsymbol{I}\in\mathcal{I}}$. Recall that such products have general form $C_1^{i_1} \cdots C_p^{i_p} \cdot Q_{p+1}^{i_{p+1}} \cdots Q_q^{i_q}$: they are composed of affine forms $C_1 \geq 0, \ldots, C_p \geq 0$ and polynomials $Q_{p+1} \geq 0, \ldots, Q_q \geq 0$. Then, the chosen $S^{\boldsymbol{I}}$'s are used to define a $PLOP$ encoding that looks for affine functions of the form

$$f(\boldsymbol{x}) = Q(\boldsymbol{x}) + \sum_{\boldsymbol{I}\in\mathcal{I}} \lambda_{\boldsymbol{I}} S^{\boldsymbol{I}}(\boldsymbol{x}) \tag{9.1}$$

By construction, each product $S^{\boldsymbol{I}}$ is nonnegative on $\mathcal{P} \wedge \{Q_{p+1} \geq 0, \ldots, Q_q \geq 0\}$. This entails $f \geq Q$ on $\mathcal{P}$, hence $\mathcal{P} \wedge (Q \geq 0) \subseteq \mathcal{P} \sqcap (f \geq 0)$.

In practice, our heuristics for choosing $\mathcal{I}$ (detailed in §4.4.3) can only produce Schweighofer products of two forms:

(1) $H^{\boldsymbol{J}} \times (x_1^{\epsilon_1} \cdots x_n^{\epsilon_n})^2$, where $H^{\boldsymbol{J}}$ is a Handelman product, *i.e.* of the form $C_1^{j_1} \times \ldots \times C_p^{j_p}$. The right part $(x_1^{\epsilon_1} \cdots x_n^{\epsilon_n})^2$ results from a heuristic that extracts even powers of polynomials.

(2) $(\prod F_{\boldsymbol{\lambda}}(\mathcal{P})) \times (x_1^{\epsilon_1} \cdots x_n^{\epsilon_n})^2$, where $F_{\boldsymbol{\lambda}}(\mathcal{P})$ is the Farkas combination $\sum_{j=1}^{p} \lambda_j C_j$. This form results from the "Simple Products" heuristic, that cancels nonlinear monomials by finding products of variable bounds (*i.e.* $x_j + l_j \geq 0$ or $-x_j + u_j \geq 0$). A variable bound is obtained from a Farkas combination $F_{\boldsymbol{\lambda}}(\mathcal{P})$, found by solving a LP problem.

Like the rest of the VPL, Handelman's linearization is certified a posteriori. An OCaml oracle chooses the set of Schweighofer products, and provides nonnegativity certificates used by a Coq checker. The certificate format `schweighofer` that we use covers both cases (1) and (2):

```
Definition natIndex : Type := list nat.

Definition squares : Type := list (PExpr * N).

Definition QIndex : Type := list Q.

Definition schweighofer : Type := natIndex * squares * (list QIndex.t).
```

`natIndex` is the type for multi-indices of natural integers. An element of this type represents an index $\boldsymbol{J}$ associated to a Handelman product, *i.e.* $H^{\boldsymbol{J}} = C_1^{j_1} \times \ldots \times C_p^{j_p}$. Type `squares` represents a product of polynomials with even exponents. A pair `(`p`,`ϵ`) : (PExpr * N)` represents $p^{2\times\epsilon}$. Here, `PExpr` is the type of polynomial expressions over rationals, that comes from the setoid theory of Coq. Type `QIndex` is a list of rationals that represents the vector $\boldsymbol{\lambda}$ of a Farkas

combination $F_{\boldsymbol{\lambda}}(\mathcal{P})$. A list of `QIndex` represents a product of such Farkas combinations, *i.e.* $\prod F_{\boldsymbol{\lambda}}(\mathcal{P})$. Finally, type `schweighofer` represents a product of the three other types of products: `natIndex`, `squares` or `list QIndex`.

The following section gives the design of our Coq checker for such certificates.

9.1 Design of the Coq Checker

Basically, the proof of Handelman's linearization boils down to proving that "products and sums of nonnegative elements are nonnegative". But, as often in Coq, some subtleties appear. In particular, we will manipulate two types for polynomials: type `QTerm.t` that implements an abstract syntax for general arithmetic expressions in the VPL, and type `PExpr`, that we will use to efficiently test polynomial equality. These two types will induce conversions from one to the other, that must be proved in Coq.

When generating a Schweighofer product $S^{\boldsymbol{I}}$, the oracle attaches to it a nonnegativity certificate of type `schweighofer` given above. To build the affine form f, the oracle provides the set (represented as a list) of Schweighofer products $(S^{\boldsymbol{I}})_{\boldsymbol{I}\in\mathcal{I}}$ and their associated coefficients $\lambda_{\boldsymbol{I}}$ that give the expression of f in Equation (9.1). The Coq type for this certificate is the following:

```
Definition certificate : Type := list (Q * schweighofer)
```

In addition to a `certificate`, the oracle provides two other elements:

— The affine form f resulting from the linearization. It is given in type `QTerm.t`, the type of general terms over `Qc` in the VPL. `Qc` is the type of irreducible rationals of Coq, it is a dependent type associating a fraction with a proof of irreducibility, contrary to `Q` that are arbitrary fractions.

— A map τ indexed by `natIndex` that guides Coq into the computation of Handelman products. The creation of this map is detailed in the next section.

The result returned by the oracle is finally given in the following type:

```
Record oracle_result : Type := mk{
    f : QTerm.t;
    certif : certificate;
    τ : Map(natIndex →list natIndex)}.
```

In Coq, we implement a function `checker` for verifying an `oracle_result`. Roughly, it checks that `f` is actually affine and that it is nonnegative on $\mathcal{P}$. If it is the case, `checker` returns an affine form (of type `QAffTerm.t`) equal to `f`.

```
Definition checker (P : list Cstr.t) (Q : QTerm.t) (r : oracle_result)
  : QAffTerm.t :=
  let f' := compute_solution P Q r in
  if f' =^{PExpr} to_PExpr r.(f)
  then let (te,aft) := QTerm.affineDecompose r.(f) in
    if te =^{PExpr} 0_{PExpr}
    then aft
    else failwith CERT "eq_witness : f is not linear" trivial
  else failwith CERT "eq_witness : f and f' differ" trivial.
```

This function takes the input polyhedron `P` (given as a list of constraints), the input polynomial `Q`, and the oracle result `r`. Function `compute_solution` builds `f'` $\stackrel{\text{def}}{=} Q+\sum_{\boldsymbol{I}\in\mathcal{I}}\lambda_{\boldsymbol{I}}S^{\boldsymbol{I}}$ from the oracle result. When computing `f'`, `compute_solution` checks that each coefficient $\lambda_{\boldsymbol{I}}$ provided by the oracle is actually nonnegative. We prove a lemma ensuring that any result of `compute_solution` is nonnegative on $\mathcal{P}$. Then, the checker compares `f'` with the solution `r.(f)` returned by the oracle. If `f'` $\neq$ `r.(f)`, it means that the certificate provided by the oracle is wrong, and an alarm informing on an oracle bug is raised. In that case, the checker returns `trivial`, which is a constant of type `QAffTerm.t` representing the trivial constraint $1 \geq 0$. If `f' = r.(f)`, `r.(f)` is

ensured to be nonnegative on $\mathcal{P}$, since `f'` is. After that, a certified procedure `affineDecompose` splits `r.(f)` into two terms `(te,aft)`, `aft` being the affine part of `r.(f)`. If `te` $\neq 0_{PExpr}$, another alarm is raised: the result is not affine as claimed by the oracle! Otherwise, it means that the nonlinear part `te` of `r.(f)` is null, hence `r.(f)` is proved affine.

In fact, the function `r.(f)` returned by the oracle is not really necessary for the proof. The checker could simply prove that the function `f'` it has built is affine, and return it. The equality test performed by the checker between `f'` and `r.(f)` is an additional verification (both of the certificate generation and the Coq functions) ensuring that the computation of `f'` went as expected. Indeed, this computation involves many datastructures, conversions and certificates, and even if `f'` is proved nonnegative, it is not straightforward that it ends up equal to `r.(f)`.

To summarize, the checker verifies that `r.(f)` is nonnegative by building a polynomial `f'` (that should be equal to `r.(f)`) from the certificates provided by the oracle. Then, it checks that `r.(f)` is affine by testing if its nonlinear part is null. If either of these two conditions is false, the checker returns a trivial (but weak) constraint $1 \geq 0$ and raises an alarm. Thus, in any case, the following theorem holds:

```
Lemma checker_pos (P : list Cstr.t) (Q : QTerm.t) (r : oracle_result) :
  ∀ (x : Mem.t),
  x ∈ P
  → 0_PExpr ≤(PExpr) eval_PExpr (to_PExpr Q) x
  → 0_QAffTerm ≤(QAffTerm) eval_QAffTerm (checker P Q r) x
```

This theorem states that, given a memory `x` (representing a point mapping variables of $\mathcal{P}$ to rational coefficients), if `x` lies in the input polyhedron $\mathcal{P}$ and satisfies guard $0 \leq Q$, then it also satisfies guard $0 \leq f$, where f is the affine function returned by `checker`.

As mentioned above, the proof of this theorem boils down to proving that products and sums of nonnegative elements are nonnegative. The difficulty comes from the multiple datastructures that we use to represent terms:

- Constraints of $\mathcal{P}$ have type `Cstr.t`, which are radix trees mapping variables to their coefficient in the constraint.
- The input polynomial `Q` is given in type `QTerm.t`, which is the type of general arithmetic expressions in the VPL.
- The oracle result `r.(f)` is also given in type `QTerm.t`, to be easily proved affine thanks to function `affineDecompose`.
- The term `f'` built by `compute_solution` is in type `PExpr`, which is more efficient to test equality between polynomials.

Manipulating all those types requires writing conversion functions from one type to another. Moreover, expressing theorem `checker_pos` needs the definition of semantics for each type (to evaluate a polynomial on a point). Therefore, an important part of this Coq development consists in proofs of semantics preservation during the conversion between the different types for polynomials.

The following section summarizes the implementation of function `compute_solution`, that builds the affine form `f'` from the certificate provided by the oracle.

9.2 Computing Handelman Products in Coq

As we saw, a Schweighofer product can be defined from a Handelman product, *i.e.* a product of the form

$$H^{\boldsymbol{J}} \stackrel{\text{def}}{=} C_1^{j_1} \cdots C_p^{j_p}$$

Let us call $(H^{\boldsymbol{J}})_{\boldsymbol{J}\in\mathcal{J}}$ the set of Handelman products involved in all chosen Schweighofer products $(S^{\boldsymbol{I}})_{\boldsymbol{I}\in\mathcal{I}}$. Blindly computing each Handelman product $H^{\boldsymbol{J}}$ from scratch would be too costly, especially with Coq datastructures. Among all the chosen products $H^{\boldsymbol{J}}$'s, lots of

them share a common subproduct. For instance, with three constraints, $H^{(2,1,3)} = C_1^2 \cdot C_2 \cdot C_3^3$ and $H^{(1,2,1)} = C_1 \cdot C_2^2 \cdot C_3$ have a common subproduct which is $H^{(1,1,1)}$. Indeed, $H^{(2,1,3)} = H^{(1,0,2)} \cdot H^{(1,1,1)}$ and $H^{(1,2,1)} = H^{(0,1,0)} \cdot H^{(1,1,1)}$. A nonnull multi-index $\boldsymbol{J}$ is a subproduct of $\boldsymbol{J'}$ and $\boldsymbol{J''}$ if and only if $\boldsymbol{J} \leq \boldsymbol{J'}$ and $\boldsymbol{J} \leq \boldsymbol{J''}$ where $\leq$ denotes the element-wise comparison of multi-indices that we defined in §4.3.1.1:

$$\boldsymbol{J} \leq \boldsymbol{J'} \Leftrightarrow \forall k,\ j_k \leq j'_k$$

Multiplying polynomials in Coq representation is costly. Therefore, we will try to limit the number of multiplications by exploiting common subproducts as much as possible. To do so, the type `oracle_result` defined above provides a map τ associating a multi-index $\boldsymbol{J}$ (representing a Handelman product $H^{\boldsymbol{J}}$) to a decomposition in subproducts, given as a list of multi-indices. For instance, to compute products $H^{(2,1,3)}, H^{(1,2,1)}$ and $H^{(1,1,3)}$, one possible map τ would be:

$$\begin{array}{l}(2,1,3) \rightarrow (1,0,2),\ (1,1,1)\\ (1,2,1) \rightarrow (1,1,1),\ (0,1,0)\\ (1,1,3) \rightarrow (1,1,1),\ (0,0,2)\\ (0,0,2) \rightarrow (0,0,1),\ (0,0,1)\\ (1,0,2) \rightarrow (1,0,0),\ (0,0,2)\\ (1,1,1) \rightarrow (1,0,0),\ (0,1,0),\ (0,0,1)\end{array}$$

When the index $\boldsymbol{J}$ of a product is *unitary*, meaning that it contains one coefficient 1 and other coefficients are 0, this product cannot be decomposed: it corresponds to a single constraint C_i, and we denote this index as $\boldsymbol{U_i}$.

To compute Handelman products, function `compute_solution` will build another map σ in Coq, associating a multi-index $\boldsymbol{J}$ to its Handelman product $H^{\boldsymbol{J}}$, which is a polynomial of type `PExpr`. To prove theorem `checker_pos`, we will exploit an invariant of σ saying that any polynomial it contains is nonnegative. To ensure this, σ must be entirely built in Coq in the following way:

- First, each unitary multi-index $\boldsymbol{U_i}$ is associated in σ with the input constraint C_i, which is nonnegative by hypothesis.
- Then, Handelman products are computed by multiplying polynomials that are already present in σ, and therefore nonnegative.

To avoid redundant calculus, each polynomial of σ will be computed following decompositions given by τ. For instance, with $C_1 : x_1 - 1 \geq 0,\ C_2 : x_2 + 2 \geq 0$ and $C_3 : 3 - x_1 - x_2 \geq 0$, suppose we wish to compute Handelman products $H^{(2,1,3)}, H^{(1,2,1)}$ and $H^{(1,1,3)}$. The map σ, built following instructions from τ, would be:

$$\begin{array}{lcl}(1,0,0) & \rightarrow & H^{(1,0,0)} = x_1 - 1\\ (0,1,0) & \rightarrow & H^{(0,1,0)} = x_2 + 2\\ (0,0,1) & \rightarrow & H^{(0,0,1)} = 3 - x_1 - x_2\\ (1,1,1) & \rightarrow & H^{(1,0,0)} \cdot H^{(0,1,0)} \cdot H^{(0,0,1)}\\ & & = -x_1^2x_2 - x_1x_2^2 - 2x_1^2 + 2x_1x_2 + x_2^2 + 8x_1 - x_2 - 6\\ (0,0,2) & \rightarrow & H^{(0,0,1)} \cdot H^{(0,0,1)}\\ & & = x_1^2 + 2x_1x_2 + x_2^2 - 6x_1 - 6x_2 + 9\\ (1,0,2) & \rightarrow & H^{(1,0,0)} \cdot H^{(0,0,2)}\\ & & = x_1^3 + 2x_1^2x_2 + x_1x_2^2 - 7x_1^2 - 8x_1x_2 - x_2^2 + 15x_1 + 6x_2 - 9\\ (1,1,3) & \rightarrow & H^{(1,1,1)} \cdot H^{(0,0,2)} = ...\\ (1,2,1) & \rightarrow & H^{(1,1,1)} \cdot H^{(0,1,0)} = ...\\ (2,1,3) & \rightarrow & H^{(1,0,2)} \cdot H^{(1,1,1)} = ...\end{array}$$

Thanks to τ, we computed $H^{(2,1,3)}, H^{(1,2,1)}$ and $H^{(1,1,3)}$ with 7 multiplications. Computing a Handelman product $H^{\boldsymbol{J}}$ from scratch requires $\sum_k j_k - 1$ multiplications. Hence, computing naively these three products would involve 12 multiplications.

Now, let us see how τ is built by the OCaml oracle. Then, we will show how σ is represented in Coq, and how it is built from τ.

Algorithm 9.1: The building of map τ.

Input : A set $\mathcal{J}$ of multi-indices to decompose
Output: A map τ containing decompositions of multi-indices of $\mathcal{J}$ into subproducts

$\tau \leftarrow empty\_map$
while $\mathcal{J} \neq \emptyset$ **do**
  $\boldsymbol{K} \leftarrow \underset{\boldsymbol{J'}}{argmax} \left\{ \|\boldsymbol{J'}\| \middle| \#\{ \boldsymbol{J} \in \mathcal{J} \mid \boldsymbol{J'} \leq \boldsymbol{J}\} \geq max\left(2, \frac{|\mathcal{J}|}{2}\right) \right\}$
  if $\boldsymbol{K} = None$ **then**
    for $\boldsymbol{J} \in \mathcal{J}$ **do**
      τ.addAssociation $\left(H^{\boldsymbol{J}} \rightarrow \underbrace{H^{(1,0,\ldots,0)} \cdot H^{(1,0,\ldots,0)}}_{j_1 \text{ times}} \cdots \underbrace{H^{(0,\ldots,0,1)} \cdot H^{(0,\ldots,0,1)}}_{j_m \text{ times}} \right)$
    $\mathcal{J} \leftarrow \emptyset$
  else
    for $\boldsymbol{J} \in \mathcal{J},\ \boldsymbol{K} \leq \boldsymbol{J}$ **do**
      τ.addAssociation $\left(H^{\boldsymbol{J}} \rightarrow H^{\boldsymbol{K}} \cdot H^{\boldsymbol{J}-\boldsymbol{K}}\right)$
      $\mathcal{J} \leftarrow \mathcal{J} \setminus \{\boldsymbol{J}\}$
      $\mathcal{J} \leftarrow \mathcal{J} \cup \{\boldsymbol{J} - \boldsymbol{K}\}$
return τ

9.2.1 Construction of the Map τ.

The algorithm used by the OCaml oracle to build the map τ is given in Algorithm 9.1. Let us explain the idea. Given a list $\mathcal{J}$ of Handelman products to compute, we try to find the "highest" multi-index $\boldsymbol{K}$ such that $H^{\boldsymbol{K}}$ is a subproduct of "many" $H^{\boldsymbol{J}}$'s, where "highest" and "many" are left to specify:

- To define "highest", we need to be able to compare two multi-indices. To do so, we define a preorder among multi-indices, induced by the Euclidian norm $\|\boldsymbol{J}\| \stackrel{\text{def}}{=} \sqrt{\left(\sum_i {j_i}^2\right)}$. Another norm could have been chosen, but the Euclidian one appears to behave better on the cases we tested, in the sense that it builds a smaller map, hence less multiplications to compute.
- We will consider indices $\boldsymbol{K}$ such that $H^{\boldsymbol{K}}$ is a subproduct of at least half of the $H^{\boldsymbol{J}}$'s, with a minimum of 2.

More formally, we look for a multi-index $\boldsymbol{K}$ defined as

$$\boldsymbol{K} \stackrel{\text{def}}{=} \underset{\boldsymbol{J'}}{argmax} \left\{ \|\boldsymbol{J'}\| \middle| \#\{ \boldsymbol{J} \in \mathcal{J} \mid \boldsymbol{J'} \leq \boldsymbol{J}\} \geq max\left(2, \frac{|\mathcal{J}|}{2}\right) \right\} \tag{9.2}$$

where $\underset{\boldsymbol{J'}}{argmax}$ designates the index $\boldsymbol{J'}$ on which the maximal norm is reached.

Then, $\mathcal{J}$ and τ are updated as follows. Let us call $\boldsymbol{J_1}, \ldots, \boldsymbol{J_m}$ the multi-indices of $\mathcal{J}$ from which $\boldsymbol{K}$ is a subproduct. Then, $\boldsymbol{J_1}, \ldots, \boldsymbol{J_m}$ are removed from $\mathcal{J}$, and they are decomposed in τ as

$$H^{\boldsymbol{J_i}} \rightarrow H^{\boldsymbol{K}} \cdot H^{\boldsymbol{J_i}-\boldsymbol{K}},\ \forall i \in \{1, \ldots, m\}$$

where $\boldsymbol{J_i} - \boldsymbol{K}$ is the element-wise substraction. Then, $\mathcal{J}$ is augmented with $\boldsymbol{K}$ and $\boldsymbol{K} - \boldsymbol{J_1}, \ldots, \boldsymbol{K} - \boldsymbol{J_m}$, which are the new Handelman products to decompose. The algorithm then iterates until $\mathcal{J}$ becomes empty.

If at some point, no multi-index $\boldsymbol{K}$ can be found by (9.2), it means that among all products $(H^{\boldsymbol{J}})_{\boldsymbol{J} \in \mathcal{J}}$, none of them share any common subproduct. All these $H^{\boldsymbol{J}}$'s are thus decomposed

into unitary products:

$$H^{\boldsymbol{J}} \rightarrow \underbrace{H^{(1,0,\ldots,0)} \cdot H^{(1,0,\ldots,0)}}_{j_1 \text{ times}} \cdots \underbrace{H^{(0,\ldots,0,1)} \cdot H^{(0,\ldots,0,1)}}_{j_m \text{ times}}$$

9.2.2 Making Maps Indexed by Multi-Indices in Coq

The Coq standard library provides a functor `FMapAVL` that takes as input a module `M` of totally ordered elements, and builds a map indexed by elements from `M`. A module `M` is totally ordered if it fulfills the following properties:

```
Parameter t : Type.

Parameter eq : t → t → Prop.
Parameter lt : t → t → Prop.

Axiom eq_refl : ∀ x : t, eq x x.
Axiom eq_sym : ∀ x y : t, eq x y → eq y x.
Axiom eq_trans : ∀ x y z : t, eq x y → eq y z → eq x z.

Axiom lt_trans : ∀ x y z : t, lt x y → lt y z → lt x z.
Axiom lt_not_eq : ∀ x y : t, lt x y → ¬ eq x y.

Inductive Compare (X : Type) (lt eq : X → X → Prop) (x y : X) : Type :=
  | LT : lt x y → Compare lt eq x y
  | EQ : eq x y → Compare lt eq x y
  | GT : lt y x → Compare lt eq x y.

Parameter compare : ∀ x y : t, Compare lt eq x y.

Definition eq_dec : ∀ x y : t, {eq x y} + {¬ eq x y}.
```

In our case, `M.t` is type `natIndex` that represents an index $\boldsymbol{J}$, which is implemented as a list of integers – type `nat` of Coq.

We define the usual lexicographic order over indices. In the following, $\boldsymbol{J}[n]$ designates the n^{th} element of list $\boldsymbol{J}$, and $\boldsymbol{J}[:k]$ is the sublist $[\boldsymbol{J}[0],\ldots,\boldsymbol{J}[k-1]]$. Let $\boldsymbol{J_1}$ and $\boldsymbol{J_2}$ be two indices. They are equal if they have the same length, and if their coefficients are equal:

$$\boldsymbol{J_1} = \boldsymbol{J_2} \Leftrightarrow (length(\boldsymbol{J_1}) = length(\boldsymbol{J_2})) \wedge (\forall i \in \{1,\ldots,length(\boldsymbol{J_1})\},\ \boldsymbol{J_1}[i] = \boldsymbol{J_2}[i])$$

$\boldsymbol{J_1} < \boldsymbol{J_2}$ either if $\boldsymbol{J_1}$ is a sublist of $\boldsymbol{J_2}$, or if they are equal until the $(k-1)^{th}$ element, and $\boldsymbol{J_1}[k] < \boldsymbol{J_2}[k]$:

$$\boldsymbol{J_1} < \boldsymbol{J_2} \Leftrightarrow \begin{cases} (length(\boldsymbol{J_1}) < length(\boldsymbol{J_2}) \wedge \boldsymbol{J_1} = \boldsymbol{J_2}[:length(\boldsymbol{J_1})]) \\ \vee \\ (\exists k \in \mathbb{N},\ k < length(\boldsymbol{J_1}),\ k < length(\boldsymbol{J_2}),\ \boldsymbol{J_1}[k] < \boldsymbol{J_2}[k] \wedge \boldsymbol{J_1}[:k] = \boldsymbol{J_2}[:k]) \end{cases} \tag{9.3}$$

Let us give these operators in Coq. Note that `nth : i l d` returns the `i`th element of list `l`; `d` is a default element returned by `nth` if `l` does not have `i`+1 elements.

```
Definition eq (I1 I2 : t): Prop :=
length I1 = length I2 ∧
∀ i:nat, ∀ d:N.t, (i < length I1 → nth i I1 d = nth i I2 d).

Definition lt (I1 I2 : t) : Prop :=
∀ d:N.t,
(length I1 < length I2 ∧ ∀ i:nat, i < length I1 → nth i I1 d = nth i I2 d)
∨ (∃ k:nat, k < length I1 ∧ k < length I2
   ∧ N.lt (nth k I1 d) (nth k I2 d)
   ∧ ∀ i:nat, i < k → nth i I1 d = nth i I2 d).
```

With these two definitions, we can prove each property of an ordered type. `eq_refl`, `eq_sym` and `eq_trans` can be proved by induction on `i`. As `lt` is defined with a disjunction, `lt_trans` is a bit more tedious to prove: we must prove transitivity for all cases of (9.3).

After proving that `natIndex` is an ordered type, we can now build maps indexed by `natIndex` in Coq. The map τ, built by the OCaml oracle, has type `Map(natIndex -> list natIndex)`. The map σ that we shall build now has type `Map(natIndex -> PExpr)`. Let us see how to compute σ from τ.

9.2.3 Construction of Map σ

Function `cons_map` takes as input τ, and the map σ initialized with input constraints C_i's of $\mathcal{P}$. `cons_map` will fill σ with the Handelman products that have a decomposition in τ. First, τ is changed into a list of association `L` of type `list (natIndex * (list natIndex))` thanks to function `elements`. We make sure that `L` is sorted according to the dependency between multi-indices: elements that are subproducts of others come first.

Then, we iterate on `L`: each multi-index `J` is computed by multiplying the subproducts coming from its decomposition in τ. If everything went as expected, those subproducts should exist in σ: they have been computed in a previous iteration. If one of them is missing, it means that the oracle is buggy and has produced a wrong map τ. In that case, an alarm is raised, and the missing subproduct is replaced by 1_{PExpr} to maintain the nonnegativity invariant of σ.

```
Definition find (J : natIndex) (σ : Map(natIndex →PExpr)) : PExpr :=
  match find J σ with
  | Some p ⇒ p
  | _ ⇒ failwith CERT "find : a subproduct is missing in σ" 1_PExpr
  end.

Definition compute_product (subproducts : list natIndex)
  (σ : Map(natIndex →PExpr)) : PExpr :=
  fold_right
  (fun J p ⇒ (find J σ) ×^PExpr p)
  1_PExpr
  subproducts.

Definition cons_map (τ : Map(natIndex →list natIndex))
  (σ : Map(natIndex →PExpr)) : Map(natIndex → PExpr) :=
  let L = elements τ in
  fold_right
  (fun (J, subproducts) σ' ⇒
    let polynomial = compute_product subproducts σ' in
    add_association J polynomial σ')
  σ L.
```

Finally, given an `oracle_result`, function `compute_solution` uses `cons_map` to build map σ. It can then exploit σ to compute $Q + \sum_{\boldsymbol{I}\in\mathcal{I}} \lambda_{\boldsymbol{I}} S^{\boldsymbol{I}}$.

9.3 Coq Code of Handelman's Linearization

The certification of Handelman's linearization is implemented in about 2000 lines of Coq code:

- 600 lines for the definition of `natIndex` as an ordered type, to be able to declare a map type indexed by `natIndex`.
- 250 lines of semantics preservation between `Q` and `Qc`.
- 650 lines to build map τ from the oracle results, to define and prove the function `compute_solution` that computes $f' = Q + \sum_{\boldsymbol{I}\in\mathcal{I}} \lambda_{\boldsymbol{I}} S^{\boldsymbol{I}}$, and to prove theorem `checker_pos`.

— 400 lines to define type `PExpr` of polynomials over `Qc`, its semantics, and some conversions between `PExpr` and `QTerm.t`.

Most tedious parts came from the multiple types for representing polynomials, that require lemmas ensuring semantics preservation. Manipulating elements from `Q` and `Qc` at the same time in proofs is particularly painful.

We presented the certification of the linearization of a polynomial guard $Q \geq 0$ when the approximation is a single affine form f. But, as presented in §4.4, Handelman's linearization produces several such affine forms f_i. Actually, our proof covers this general case and handle a list of `oracle_result`.

This Coq development would have been greatly simplified if realised in PFS (see Chapter 7). Indeed, we could make a factory for computing Schweighofer products, ensuring that only nonnegative polynomials are generated. This would avoid the need of maps τ and σ: products would be computed in Coq datastructures directly by the OCaml oracle using operations from the factory. In particular, the type `natIndex` that was used to build these maps would no longer be necessary, as well as the proof that it is an ordered type. Roughly, PFS would make obselete half the number of lines of code in the certification of Handelman's linearization. Actually, the use of PFS in the VPL came when the proof of Handelman's linearization was already finished, and we did not found the time to adapt it yet.

Conclusion

The goal of this work was to improve the scalability of the VPL, without jeopardizing the certification. The most expensive operators of the library were projection and convex hull, hence we looked for new algorithms to compute them.

Now, projection can be encoded as a PLP problem. As a consequence, several variables can be eliminated at the same time, and the result is free of redundancies thanks to a normalization constraint. Our comparison between the two algorithms for projection, namely Fourier-Motzkin elimination and PLP, shows that our new encoding scales better (see §3.5).

In addition, I extended the VPL to the handling of nonlinear constraints with two approaches: intervalization and Handelman's linearization. This latter showed interesting performances in proving the emptiness of semialgebraic sets from the SMT community (see §4.4.4).

The certification process of the VPL in Coq has been reworked, following an innovative framework named Polymorphic Factory Style (PFS) initiated by Sylvain Boulmé. By combining LCF style with type polymorphism, PFS provides a lightweight approach of certification that avoids cumbersome handling of certificates.

In this chapter, I present the current status of the VPL, and the testing framework that I implemented for evaluating its performances. Then, we will see several research directions that could follow this work.

Current Status of the VPL Implementation

Distribution. The source code of the VPL is available on GitHub.[1] Compilation and installation instructions are given on the GitHub webpage. It can also be installed via an OPAM package, distributed via GitHub. The library currently contains about 25K lines of OCaml, 14K lines of Coq (mostly written by Sylvain Boulmé, extracted into 11.5K lines of OCaml), and 1600 lines of C++ (written by Hang Yu).
With the help of David Bühler, we made a binding of the VPL for the static analyzer FRAMA-C . It is for now available only on a developer branch of the FRAMA-C development.

Functors. In the VPL, vectors (which represent the linear part of constraints) are implemented as radix trees indexed by positive variables, encoded as binary integers. These datastructures date back to the initial version of the library. They were chosen with the belief that polyhedra are sparse data; yet they are not sparse in intermediate computations. Simpler and more adaptable structures could be beneficial, based for instance on OCaml maps indexed by standard integers.
Following this idea, I started functorizing the VPL modules, so that the constraint type could be changed. This work is not finished yet; it was not possible to adapt the simplex algorithm written by A. Fouilhé, because its implementation heavily exploits the current tree structure of vectors. Completing the functorization could allow changing the constraint type and introducing hash-consing to optimize the involved memory space. We could also adapt the datastructures for further needs:

— scalars could be directly implemented as Flint rationals, and polyhedra as Flint matrices on which the C++ algorithms can operate;

1. https://github.com/VERIMAG-Polyhedra/VPL

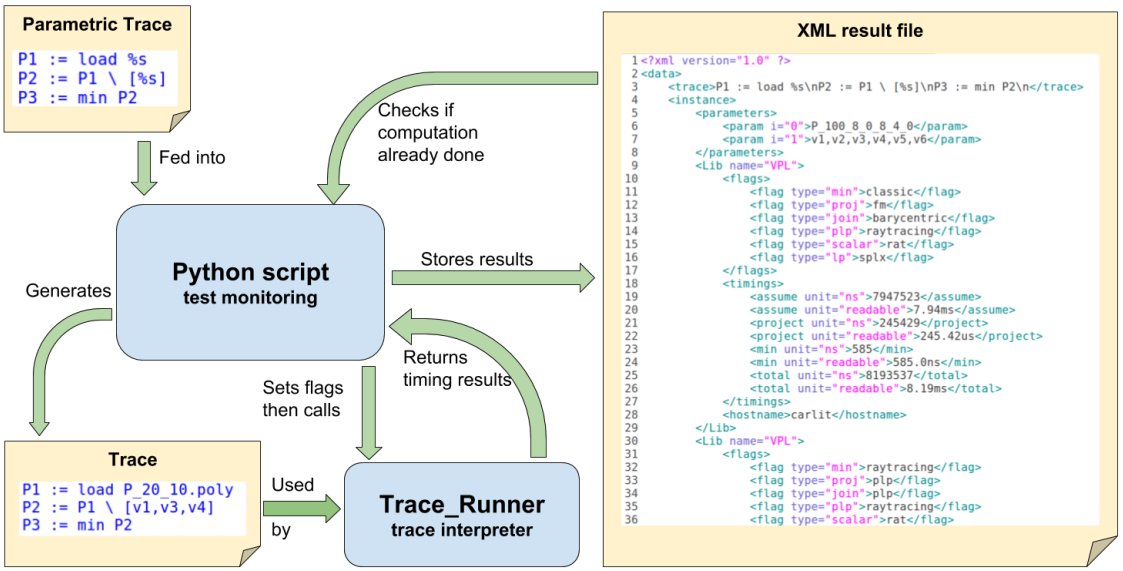


Figure 9.2 – Scheme of the test infrastructure.

— constraints could be encoded with integers coefficients and a single rational number representing their gcd.

Flags. The VPL has a module `Flags` containing several mutable variables for customizing operators. By changing them, the user can for example choose the algorithm used for projection (between Fourier-Motzkin elimination or PLP), or for the convex hull (by projection or directly by PLP). It is also possible to choose the type of scalar values used internally by the PLP solver, between floats, GMP rationals or rationals with symbolic error, as defined in §1.3.2.2 (p.32).

Regression Testing. The VPL comes with a battery of regression tests that cover a large part of the library. Most of them were implemented by A. Fouilhé in the previous version, and were adapted to the new datastructures. In particular, operators are tested with different values for the flags detailed above. In total, about 3300 tests are executed.

Testing Infrastructure

To evaluate the VPL performances, we developed a testing infrastructure, sketched on Fig. 9.2. It was used for the evaluation of minimization by raytracing (in §2.4) and for projection via PLP (in §3.5). The idea behind this infrastructure is the following: it is hard to compare different abstract domain implementations while running an analysis. It requires a lot of engineering work to connect these libraries into a single analyzer, which is even more unpleasant when they are written in different languages. Moreover, analyzers are often not designed for an optimal use of the polyhedra abstract domain: they make an intensive use of the interval abstract domain, and ask for instance a lot of intervals bounding expressions and subexpressions, resulting in useless and punishing computations.

Trace Format. We propose to evaluate libraries outside of analyzers. To do so, we define a *trace format* (given in Fig. 9.3) for storing and replaying calls to the abstract domain. It associates a unique identifier to store each abstract value, in order to avoid recomputing

it during subsequent operations. For instance, the call `P3 := P1 || P2` reuses the internal representation of `P1` and `P2` in the abstract domain to compute their convex hull `P3`.

Trace Interpreter. The testing infrastructure comes with an interpreter of traces, written in OCaml and named `Trace_Runner`. Given a trace file, the interpreter runs it on a given library. For now, two libraries can be used: NEWPOLKA (via the OCaml interface of APRON) and VPL. `Trace_Runner` is callable by command line and offers several parameters to customize the execution. In addition to the library choice, these parameters allow changing flag values for the VPL.
For instance, the trace used for the experiments on projection via PLP (in §3.5) is the following:

```
P1 := load P_20_10.poly
P2 := P1 \ [v1,v3,v4]
P3 := min P2
```

It computes the minimized representation of the polyhedron resulting from the elimination of a list of variables from an input polyhedron loaded from file `P_20_10.poly`. This file contains a polyhedron given as a matrix of rational coefficients.

Trace Generation. To obtain traces, two options are available: either write them by hand (or by script), or ask the VPL used as an abstract domain in an analyzer to generate them. Indeed, the VPL module `UserInterface`, which is the abstract domain interface of the VPL, can keep track of each operation that has been called during an execution. Thus, by using the VPL within a static analyzer such as FRAMA-C , we can record the calls to the abstract domain made by a real analyzer. Then, this sequence of operations can be replayed on several libraries to compare their performance, regardless of the analyzer.

Test Monitoring. The testing infrastructure provides a Python script to monitor a whole set of tests. The script is fed with a *parametric trace*, that is, a trace where some data is missing, such as the input file where to load a polyhedron. For instance, a parametric version of the previous script could be:

```
P1 := load %s
P2 := P1 \ [%s]
P3 := min P2
```

Here, the polyhedron to load and the list of variables to eliminate are left to be defined by the test campaign, which instantiates the parametric trace with actual values, and launches `Trace_runner`. An instantiation of the parametric trace form an *instance* of test.

XML Result File. `Trace_runner` provides timing results and stores them into an XML file. It also associates additional information to these time measurements:

- the instance, *i.e.* parameters with which the parametric trace was instantiated;
- the name of the machine that ran the tests;
- the library used;
- the flags given to `Trace_runner`.

Before running a test instance, the Python script that manages the test campaign looks if the exact same test already appears in the XML result file. If it is not the case, it launches `Trace_runner` and adds the results in the XML file. An example of such XML file is given in Appendix B.

Curve Generation. The testing infrastructure provides a tool to finally generate curves from the XML result file. The user specifies what he wants in abscissa (*e.g.* the number of constraints), in ordinate (*e.g.* the execution time for the projection operator) and the machine on which the tests have been done. Then, a python script will generate a plot with one curve per different combination (library$\times$flag set).

```
trace ::=
    Pident := operation
  | load filename                          (* load polyhedron from file *)
  | trace \n trace                         (* sequence of traces *)

operation ::=
    polyhedron && condition                (* assume *)
  | polyhedron && polyhedron               (* meet *)
  | is_bottom polyhedron                   (* is_bottom *)
  | assert condition in polyhedron         (* assert *)
  | assignments in polyhedron              (* assignments *)
  | polyhedron || polyhedron               (* join *)
  | polyhedron + polyhedron                (* Minkowski sum *)
  | polyhedron \ variables                 (* projection *)
  | polyhedron widen polyhedron            (* widening *)
  | upper_bound term in polyhedron         (* get upper bound *)
  | lower_bound term in polyhedron         (* get lower bound *)
  | itv term in polyhedron                 (* get interval *)
  | min polyhedron                         (* minimization *)

polyhedron ::=
    Pident
  | top                                    (* unbounded polyhedron *)

Pident ::= {P} (0..9)+                     (* unique identifier *)

assignments ::=
    []
  | (var := term) :: assignments

condition ::=
    true
  | false
  | condition /\ condition                  (* conjunction *)
  | condition \/ condition                  (* disjunction *)
  | not (condition)                        (* negation *)
  | term <= term
  | term >= term
  | term = term

term ::=
    var                                    (* variable *)
  | [-] (0..9)+ / (1..9)+                  (* rational scalar *)
  | term + term                            (* sum *)
  | term * term                            (* product *)

variables ::=
    []
  | var :: variables

var ::= {v} (0..9)+
```

Figure 9.3 – Trace format of the testing infrastructure.

Benchmark Generators. Aside from the testing infrastructure, we also provide a series of SageMath scripts to generate polyhedra benchmarks. Each script focuses on a particular shape of polyhedra: sphere approximations, rotated hypercubes, cylinder approximations, etc. These scripts generate a set of random polyhedra fulfilling several user-defined parameters, such as the number of variables, constraints, density, etc. They output polyhedra as matrices of rational coefficients, loadable by the trace interpreter with the `load` command defined in the trace format.

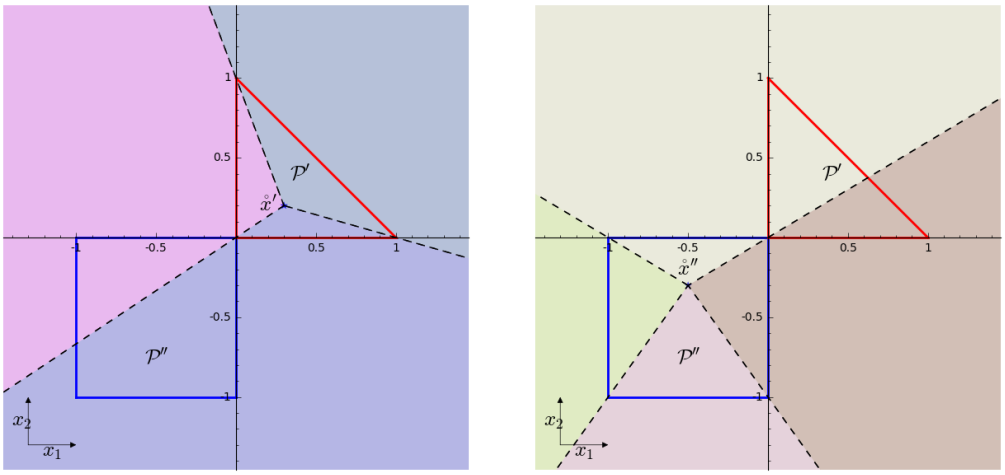
Distribution. We plan to distribute the whole testing infrastructure on GitHub, with our benchmark generators. Before that, we need to adapt our trace format, to make it as easy as possible to parse. We also plan to extend the format to make it more powerful, *e.g.* by adding a loop structure.

Toward a PLP Abstract Domain

Two operators are now encoded as PLP problems in the VPL: projection and convex hull. The library scalability has been greatly improved by this new encoding of the projection, mainly by avoiding the generation of redundant constraints. Yet, the convex hull suffers a lot from basis degeneracy (defined in §5.3), which still makes it a costly operation.

To overcome this issue, we propose to keep and exploit the partition in regions – resulting from a PLP solving – of a polyhedron. We believe that the partitioning could be exploited to speed up further operations based on PLP, especially the convex hull. For instance, consider two polyhedra $\mathcal{P}'$ and $\mathcal{P}''$, and their region partitioning, respectively $(\mathcal{R}'_i)_{i\in I}$ and $(\mathcal{R}''_j)_{j\in J}$. Assume we want to compute the convex hull $\mathcal{P} = \mathcal{P}' \sqcup \mathcal{P}''$ using the PLP encoding presented in §3.6. We believe that this computation could be significantly faster by reusing regions $\mathcal{R}'_i$'s and $\mathcal{R}''_j$'s.

Example 9.4



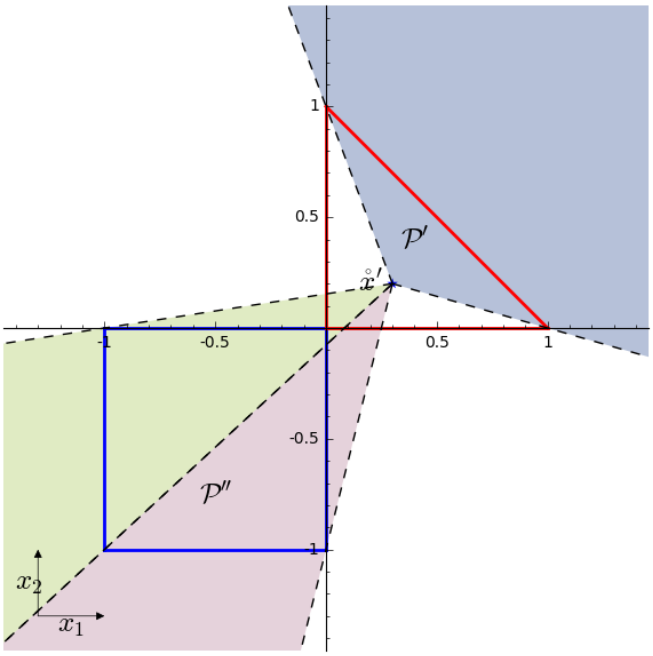
Consider $\mathcal{P}' \stackrel{\text{def}}{=} \{x_1 \geq 0,\ x_2 \geq 0, x_1 + x_2 \leq 1\}$ and $\mathcal{P}'' \stackrel{\text{def}}{=} \{x_1 \leq 0,\ x_2 \leq 0, x_1 \geq -1,\ x_2 \geq -1\}$. The left figure shows the region partitioning of $\mathcal{P}'$ into three regions, normalized in $\mathring{\boldsymbol{x}}'$. The right figure shows the region partitioning of $\mathcal{P}''$ into four regions, normalized in $\mathring{\boldsymbol{x}}''$.

Several constraints of $\mathcal{P}$ are constraints of $\mathcal{P}'$ or $\mathcal{P}''$. To decide if a constraint $\boldsymbol{c'} \geq 0$ of $\mathcal{P}'$ will still appear in the convex hull $\mathcal{P}$, it suffices to check if $\mathcal{P}' \sqsubseteq (\boldsymbol{c'} \geq 0)$. Similarly, a constraint $\boldsymbol{c''} \geq 0$ of $\mathcal{P}''$ will be kept into $\mathcal{P}$ if $\mathcal{P}'' \sqsubseteq (\boldsymbol{c''} \geq 0)$. Building upon this remark, our idea is to keep each region $\mathcal{R}'_i$ or $\mathcal{R}''_j$ that is the support of a constraint that is not discarded by the convex hull. Technically, the regions should be renormalized on the same point inside

the resulting polyhedron $\mathcal{P}$. We can reuse the normalization point $\mathring{\boldsymbol{x}}'$ of $\mathcal{P}'$, hence limiting the renormalization to regions of $\mathcal{R}''_j$. Indeed, by construction $\mathring{\boldsymbol{x}}' \in [\![\mathring{\mathcal{P}}']\!]$, thus $\mathring{\boldsymbol{x}}' \in [\![\mathring{\mathcal{P}}]\!]$. So, $\mathring{\boldsymbol{x}}'$ is a convenient normalization point for the PLOP encoding of the convex hull $\mathcal{P}' \sqcup \mathcal{P}''$. Renormalizing a region is straightforward: each frontier is transformed so that it crosses the new normalization point $\mathring{\boldsymbol{x}}'$ instead of $\mathring{\boldsymbol{x}}''$.

At this point, we have a set of regions (some from $\mathcal{R}'_i$'s, some others from renormalized $\mathcal{R}''_j$'s) all normalized on the same point $\mathring{\boldsymbol{x}}'$. They form an incomplete partition of $\mathcal{P}$ into regions. Finally, all we have left to do is to compute the last constraints of $\mathcal{P}$ (that are not constraints of $\mathcal{P}'$ or $\mathcal{P}''$), by finding the missing regions of our incomplete partition. To do so, we initialize our PLP solver with the regions that we already know. The solver will then finish the job and compute only the missing regions.

Example 9.5 (follows 9.4)



The figure shows the incomplete partition of region that we obtain after renormalization of useful regions of $\mathcal{P}''$. The only constraint of $\mathcal{P}'$ that will not be discarded for $\mathcal{P}' \sqcup \mathcal{P}''$ is $x_1 + x_2 \leq 1$. Thus, the region associated to $x_1 + x_2 \leq 1$ is kept. Two constraints of $\mathcal{P}''$ will be kept for the convex hull $\mathcal{P}' \sqcup \mathcal{P}''$: $x_1 \geq -1$ and $x_2 \geq -1$. The figure shows the renormalization of their corresponding regions on the point $\mathring{\boldsymbol{x}}'$.

This idea encourages working into a new direction, that is trying to maintain the region partitioning associated to a polyhedron. This is a kind of new double description, where the generators are replaced with a partition into regions. Most operators on polyhedra could be adapted to maintain the region partitioning without difficulty.

Future Work

In addition to the new representation of polyhedra discussed in the previous section, our work on the VPL can be extended in several directions.

VPL Implementation. The current implementation already provides all the operators required to use the VPL as an abstract domain. Several flags, mentioned earlier, let the user choose between the available algorithms, *e.g.* between Fourier-Motzkin elimination and PLP

for the projection operator. Still, some work is needed to provide a user-friendly library, most of it related to documentation. Even if the user-level interface is documented, many modules still need a precise documentation, such as the debug module that allows printing a lot of data about the execution of the operators.

Distribution. Some engineering work is needed on the VPL to touch a wider used community. First, we need to finish the functorization process mentioned earlier. Then, we plan to build a user interface compliant with that of Apron, which is widely spread among polyhedra users. We would like also to integrate the VPL into SageMath.

Use of Polyhedra in Static Analyzers. The integration of the VPL as an abstract domain in the FRAMA-C static analyzer (EVA) reveals that it was not meant to use polyhedra. Replacing calls to the interval abstract domain by requests on polyhedra results in inefficiency, and not in a great increase in precision since many heuristics are guided by the range of variables. We started some discussion with David Bühler and Boris Yakobowski from the FRAMA-C team in order to use polyhedra only where the interval abstract domain failed to prove an assertion.

Testing Infrastructure. The testing infrastructure must be generalized to be usable with libraries in C, Java, C++ and OCaml. It will provide a platform for a real comparison of several implementation of polyhedra including the NEWPOLKA and PPL (in C), the Fast Polyhedra library (in C++) and the VPL (in OCaml). We started discussing with Martin Vechev's team at ETH Zurich on this subject.

Factorization in Handelman's Linearization. Handelman's linearization can be used to decide the satisfiability of a conjunction of polynomial constraints but requires to delineate the exploration space as a polyhedron. In our experiments (§4.4.4), we failed to show unsatisfiability when the problem provided no initial polyhedron. Then, we started an investigation with Bruno Grenet, from LIRM, on extracting linear factors from polynomial constraints in order to get an initial polyhedron.

Constraint Programming. In Constraint Programming, solvers try to cover the solution with a finite union of elements, in order to obtain an approximation to an arbitrary precision. To find new disjunctions, solvers rely on a *split* operator to divide elements into several pieces. Then, each piece is recursively split so that useless elements (that do not contain any point of the solution) are discarded, and the remaining ones are close enough to the solution.
Polytopes are usually split into two parts. The cutting direction can be chosen following several heuristics (Pelleau et al., 2013), for instance

- splitting along the variable with the widest range;
- or splitting the segment joining the two farthest vertices.

To get a split operator that gives more than two pieces, I propose to use the partition into regions resulting from a PLP operator, which gives one piece per constraint of the initial polytope. By changing the position of the normalization point, the precision of the mesh could be finely tuned.

Integer Programming. We received many demands for a VPL domain on integers, meaning that polyhedra are implicitly intersected with $\mathbb{Z}^n$, as it is done in the polyhedron model for compiler optimization. This is a topic that we did not consider yet but that could be interesting and challenging.

PLP Encodings. The PLP solver is a generic tool, which is the dual version of the PIP solver by Feautrier (1988). There are probably some problems in other domains which can expressed as a PLP problem, for instance in controller synthesis and model predictive control design (Rubagotti et al., 2014).

Precision Certificates by PLP. Finally, the PLP solver could help us certifying the precision of the VPL operators. Let us take the example of variable elimination. On the one hand, it is not clear how to certify the precision of a projection computed by Fourier-Motzkin elimination: one should probably prove that all possible combinations of constraints that eliminate the variable have been tried. This boils down to proving directly Fourier-Motzkin in Coq. On the other hand, PLP provides some kind of precision certificates: the optimal dictionary obtained in each region. Such a dictionary proves that the associated solution is optimal as long as the parametric coefficients of its objective are nonnegative, meaning that we are in the region. Then, to show that no constraint is missing in the result, we need to prove that the regions we have discovered form a partition of the whole space of parameters. We have ideas on how to address this challenge but this requires some extra-work.

Appendix A

Handelman's Heuristics: LP Problem Definition

Here, we show how we implement the "simple products" heuristic, defined in §4.4.3. The goal is to find a product of variables bounds (of the form $x_j + l_j \geq 0$ or $-x_j + u_j \geq 0$) able to cancel a monomial $m = c_m \times x_1 \cdot x_n$. But, finding a variable bound requires solving a LP problem. Thus, we wish to minimize the use of variable bounds that we do not already know. In the following, we define a LP problem to choose such variable bounds.

Constants. Let us define constant values.

$$hasSup_i = \begin{cases} 0 \text{ if } x_i \text{ has no upper bound in } \mathcal{P} \\ 1 \text{ otherwise} \end{cases}$$

$$hasInf_i = \begin{cases} 0 \text{ if } x_i \text{ has no lower bound in } \mathcal{P} \\ 1 \text{ otherwise} \end{cases}$$

$$knownSup_i = \begin{cases} 1 \text{ if the upper bound of } x_i \text{ is known} \\ 0 \text{ otherwise} \end{cases}$$

$$knownInf_i = \begin{cases} 1 \text{ if the lower bound of } x_i \text{ is known} \\ 0 \text{ otherwise} \end{cases}$$

$$s = \begin{cases} 1 \text{ if } c_m \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

This last variable encodes the sign of coefficient c_m.

Decision Variables. Let us now define the decision variables of the LP problem. For each variable x_i, we must choose either an upper or a lower bound to cancel x_i. The product of all bounds must have the good sign to cancel m. The following variable y_i encodes that choice:

$$y_i = \begin{cases} 1 \text{ if we select the upper bound of } x_i \\ 0 \text{ if we select the lower bound of } x_i \end{cases}$$

We will also need a decision variable $k \in \mathbb{N}$ to encode the sign restriction of the product.

Objective Function. The objective is to minimize the number of times we choose a bound that is unknown. If $y_i = 1$ and $knownSup_i = 0$, it means that we choose the upper bound of x_i, but it is unknown. Thus, we count 1. Similarly, we count 1 if $y_i = 0$ and $knownInf_i = 0$. The sum to minimize is therefore

$$\sum_i y_i(1 - knownSup_i) + (1 - y_i)(1 - knownInf_i)$$

$$= \sum_i y_i(knownInf_i - knownSup_i) + knwonInf_i + 1$$

The constant part $knwonInf_i + 1$ is not important for us: only the values for y_i's do matter. Finally, the objective to minimize is

$$\sum_i y_i(knownInf_i - knownSup_i)$$

Constraints. We specify that, if x_i has no upper bound, then we must choose the lower bound, *i.e.* $\neg hasSup_i \Rightarrow \neg y_i$:

$$hasSup_i - y_i \geq 0, \ \forall i$$

Similarly, if x_i has no lower bound, then we must choose the upper bound, *i.e.* $\neg hasInf_i \Rightarrow y_i$:

$$hasInf_i + y_i \geq 1, \ \forall i$$

Finally, we specify that the coefficient of the product of interval bounds we are building, say m', has opposite sign to m. In m', each upper bound brings a negative coefficient ($-x_i + u_i \geq 0$), while lower bounds bring positive coefficients ($x_i + l_j \geq 0$). Thus, the sign of m''s coefficient is positive if the number of upper bounds chosen is even, and negative otherwise. Thus, the following constraint have the wanted effect:

$$\sum_i y_i = 2k + s$$

If $s = 1$, *i.e.* if c_m is positive, then $\sum_i y_i$ must be odd, and the coefficient of m' is negative. On the contrary, if $s = 0$, *i.e.* if c_m is negative, then $\sum_i y_i$ must be even and the coefficient of m' is positive.

Finally, the LP problem that we solve is the following:

$$\begin{aligned} \textbf{minimize } \mathbf{Z} \stackrel{\text{def}}{=} & \sum_i y_i(knownSup_i - knownInf_i) \\ \textbf{subject to} \quad & hasSup_i - y_i \geq 0, \ \forall i \\ & hasInf_i + y_i \geq 1, \ \forall i \\ & \sum_i y_i = 2k + s \end{aligned} \tag{LP A.1}$$

Appendix B

Example of XML Result File

The parametric trace is given in tag `trace`. An instance is defined by an instantiation of all parameters. Then, for each instance, we store the time measurements of all operators for each execution of the instantiated trace, depending on the library, the values of flags, and the machine used.

```
<?xml version="1.0" ?>
<data>
    <trace>P1 := load %s\nP2 := P1 \ [%s]\nP3 := min P2\n</trace>
    <instance>
        <parameters>
            <param i="0">P_100_8_0_8_4_0</param>
            <param i="1">v1,v2,v3,v4,v5,v6</param>
        </parameters>
        <Lib name="VPL">
            <flags>
                <flag type="min">classic</flag>
                <flag type="proj">fm</flag>
                <flag type="join">barycentric</flag>
                <flag type="plp">raytracing</flag>
                <flag type="scalar">rat</flag>
                <flag type="lp">splx</flag>
            </flags>
            <timings>
                <assume unit="ns">7947523</assume>
                <assume unit="readable">7.94ms</assume>
                <project unit="ns">245429</project>
                <project unit="readable">245.42us</project>
                <min unit="ns">585</min>
                <min unit="readable">585.0ns</min>
                <total unit="ns">8193537</total>
                <total unit="readable">8.19ms</total>
            </timings>
            <hostname>carlit</hostname>
        </Lib>
        <Lib name="VPL">
            <flags>
                <flag type="min">raytracing</flag>
                <flag type="proj">plp</flag>
                <flag type="join">plp</flag>
                <flag type="plp">raytracing</flag>
                <flag type="scalar">rat</flag>
                <flag type="lp">splx</flag>
            </flags>
            <timings>
                <assume unit="ns">7074286</assume>
```

```
41                  <assume unit="readable">7.7ms</assume>
42                  <project unit="ns">3418744</project>
43                  <project unit="readable">3.41ms</project>
44                  <min unit="ns">416</min>
45                  <min unit="readable">416.0ns</min>
46                  <total unit="ns">10493446</total>
47                  <total unit="readable">10.49ms</total>
48              </timings>
49              <hostname>carlit</hostname>
50          </Lib>
51          <Lib name="Apron">
52              <timings>
53                  <assume unit="ns">1148569</assume>
54                  <assume unit="readable">1.14ms</assume>
55                  <project unit="ns">35549</project>
56                  <project unit="readable">35.54us</project>
57                  <min unit="ns">287367</min>
58                  <min unit="readable">287.36us</min>
59                  <total unit="ns">1471485</total>
60                  <total unit="readable">1.47ms</total>
61              </timings>
62              <hostname>carlit</hostname>
63          </Lib>
64      </instance>
65      <instance>
66          <parameters>
67              <param i="0">P_100_8_0_8_4_1</param>
68              <param i="1">v1,v2,v3,v4,v5,v6</param>
69          </parameters>
70          <Lib name="VPL">
71              <flags>
72                  <flag type="min">classic</flag>
73                  <flag type="proj">fm</flag>
74                  <flag type="join">barycentric</flag>
75                  <flag type="plp">raytracing</flag>
76                  <flag type="scalar">rat</flag>
77                  <flag type="lp">splx</flag>
78              </flags>
79              <timings>
80                  <assume unit="ns">6601215</assume>
81                  <assume unit="readable">6.60ms</assume>
82                  <project unit="ns">119739</project>
83                  <project unit="readable">119.73us</project>
84                  <min unit="ns">542</min>
85                  <min unit="readable">542.0ns</min>
86                  <total unit="ns">6721496</total>
87                  <total unit="readable">6.72ms</total>
88              </timings>
89              <hostname>carlit</hostname>
90          </Lib>
91          <Lib name="VPL">
92              <flags>
93                  <flag type="min">raytracing</flag>
94                  <flag type="proj">plp</flag>
95                  <flag type="join">plp</flag>
96                  <flag type="plp">raytracing</flag>
97                  <flag type="scalar">rat</flag>
98                  <flag type="lp">splx</flag>
99              </flags>
100             <timings>
101                 <assume unit="ns">6908881</assume>
```

```
                <assume unit="readable">6.90ms</assume>
                <project unit="ns">3324538</project>
                <project unit="readable">3.32ms</project>
                <min unit="ns">448</min>
                <min unit="readable">448.0ns</min>
                <total unit="ns">10233867</total>
                <total unit="readable">10.23ms</total>
            </timings>
            <hostname>carlit</hostname>
        </Lib>
        <Lib name="Apron">
            <timings>
                <assume unit="ns">1041667</assume>
                <assume unit="readable">1.4ms</assume>
                <project unit="ns">31039</project>
                <project unit="readable">31.3us</project>
                <min unit="ns">266202</min>
                <min unit="readable">266.20us</min>
                <total unit="ns">1338908</total>
                <total unit="readable">1.33ms</total>
            </timings>
            <hostname>carlit</hostname>
        </Lib>
    </instance>
</data>
```

Bibliography

Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *Symposium on Logic in Computer Science (LICS)*, page 75. IEEE Computer Society, 2002.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *Principles of Programming Languages (POPL)*, pages 109–122. ACM Press, 2007.

Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending coq with imperative features and its application to SAT verification. In *Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 83–98. Springer, 2010.

Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to coq through proof witnesses. In *Certified Programs and Proofs (CPP)*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011.

Roberto Bagnara, Enric Rodríguez-Carbonell, and Enea Zaffanella. *Generation of Basic Semi-algebraic Invariants Using Convex Polyhedra*, pages 19–34. Springer, 2005.

Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.3551`.

Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2010. URL `http://www.SMT-LIB.org`.

Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *Embedded Real Time Software and Systems (ERTS2)*, 2012.

Mohamed Amin Ben Sassi, Romain Testylier, Thao Dang, and Antoine Girard. *Reachability Analysis of Polynomial Systems Using Linear Programming Relaxations*, pages 137–151. Springer, 2012. URL `http://www-verimag.imag.fr/~tdang/Papers/ATVA2012.pdf`.

Mohamed Amin Ben Sassi, Sriram Sankaranarayanan, Xin Chen, and Erika Ábrahám. Linear relaxations of polynomial positivity for polynomial lyapunov function synthesis. *IMA Journal of Mathematical Control and Information*, 33(3):723–756, 2015.

Florence Benoy, Andy King, and Frédéric Mesnard. Computing convex hulls with a linear solver. *Theory and Practice of Logic Programming*, 5(1-2), 2005. URL `https://arxiv.org/abs/cs/0311002`.

Jean-Philippe Bernardy and Moulin Guilhem. Type-theory in color. In *International Conference on Functional Programming (ICFP)*, ICFP '13, pages 61–72. ACM Press, 2013.

Jean-Philippe Bernardy and Guilhem Moulin. A computational interpretation of parametricity. In *Symposium on Logic in Computer Science (LICS)*, LICS '12, pages 135–144. IEEE Computer Society, 2012.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. An EATCS Series. 2004. URL `https://www.labri.fr/perso/casteran/CoqArt/`.

Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Types for Proofs and Programs (TYPES)*, volume 4502 of *LNCS*, pages 48–62. Springer, 2006.

Frédéric Besson, Thomas Jensen, David Pichardie, and Tiphaine Turpin. Result certification for relational program analysis. Research report, 2007. URL `https://hal.inria.fr/inria-00166930`.

Frédéric Besson, Thomas P. Jensen, David Pichardie, and Tiphaine Turpin. Certified result checking for polyhedral analysis of bytecode programs. In *Trustworthy Global Computing (TGC)*, volume 6084 of *LNCS*, pages 253–267. Springer, 2010.

Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular SMT proofs for fast reflexive checking inside coq. In *Certified Programs and Proofs (CPP)*, volume 7086 of *LNCS*, pages 151–166. Springer, 2011.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed kripke models over recursive worlds. In *Principles of Programming Languages (POPL)*, pages 119–132. ACM Press, 2011.

Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.

Robert G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2(2):103–107, 1977. URL `http://www.ohio.edu/people/melkonia/math4620/bland.pdf`.

Sandrine Blazy, Delphine Demange, and David Pichardie. Validating dominator trees for a fast, verified dominance test. In *Interactive Theorem Proving (ITP)*, volume 9236 of *LNCS*, pages 84–99. Springer, 2015.

Jan Olaf Blech and Benjamin Grégoire. Certifying compilers using higher-order theorem provers as certificate checkers. *Formal Methods in System Design*, 38(1):33–61, 2011.

Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010.

David Boland and George A. Constantinides. Bounding Variable Values and Round-Off Effects Using Handelman Representations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(11):1691–1704, 2011.

Sylvain Boulmé and Alexandre Maréchal. Refinement to certify abstract interpretations, illustrated on linearization for polyhedra. In *Interactive Theorem Proving (ITP)*, volume 9236 of *LNCS*, pages 100–116. Springer, 2015. URL `https://hal.archives-ouvertes.fr/hal-01133865v2/`.

Sylvain Boulmé and Alexandre Maréchal. Toward Certification for Free! preprint, 2017a. URL `https://hal.archives-ouvertes.fr/hal-01558252`.

Sylvain Boulmé and Alexandre Maréchal. A Coq Tactic for Equality Learning in Linear Arithmetic. preprint, 2017b. URL `https://hal.archives-ouvertes.fr/hal-01505598`.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. URL `http://stanford.edu/~boyd/cvxbook/`.

Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *Static Analysis Symposium (SAS)*, LNCS, pages 309–325. Springer, 2009.

N. V. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 1968.

S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. In *International Congress on Mathematical Software (ICMS)*, volume 6327 of *LNCS*, pages 28–31. Springer, 2010.

Sylvain Chevillard, Mioara Joldeş, and Christoph Lauter. Certified and fast computation of supremum norms of approximation errors. In *Computer Arithmetic (ARITH)*, pages 169–176. IEEE Computer Society, 2009.

Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. URL `http://adam.chlipala.net/cpdt/`.

Vašek Chvátal. *Linear Programming*. Series of books in the Mathematical Sciences. W. H. Freeman, 1983.

Philippe Clauss, Federico Javier Fernández, Diego Garbervetsky, and Sven Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *Transactions on Very Large Scale Integration (VLSI) Systems*, 17(8):983–996, 2009. URL `https://hal.inria.fr/inria-00504617`.

William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. John Wiley & Sons, Inc., 1998.

Florian Corzilius, Ulrich Loup, Sebastian Junges, and Erika Ábrahám. SMT-RAT: An SMT-Compliant Nonlinear Real Arithmetic Toolbox. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *LNCS*, pages 442–448. Springer, 2012.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977. URL `http://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml`.

N.G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*, volume 125 of *LNM*, pages 29–61. Springer, 1968.

Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

Leonardo de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7737 of *LNCS*, pages 1–12. Springer, 2013.

Morgan Deters, Andrew Reynolds, Tim King, Clark Barrett, and Cesare Tinelli. A tour of cvc4: How it works, and how to use it. In *Formal Methods in Computer-Aided Design (FMCAD)*, FMCAD '14, pages 4:7–4:7. FMCAD Inc, 2014.

Bruno Dutertre. Yices 2.2. In *Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.

Bruno Dutertre and Leonardo de Moura. Integrating simplex with DPLL(T). Technical Report SRI-CSL-06-01, SRI International, computer science laboratory, 2006a.

Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006b.

Julius Farkas. Theorie der einfachen ungleichungen. *Journal für die reine und angewandte Mathematik*, 124:1–27, 1902. URL https://eudml.org/doc/149129.

Rida T. Farouki. The Bernstein polynomial basis: A centennial retrospective. *Computer Aided Geometric Design*, 29(6):379–419, 2012.

Paul Feautrier. Parametric integer programming. *RAIRO - Operations Research - Recherche Opérationnelle*, 22(3):243–268, 1988.

Paul Feautrier. *Automatic parallelization in the polytope model*, pages 79–103. Springer Berlin Heidelberg, 1996.

Alexis Fouilhé. *Revisiting the abstract domain of polyhedra: constraints-only representation and formal proof*. PhD thesis, Université de Grenoble, 2015.

Alexis Fouilhé and Sylvain Boulmé. A certifying frontend for (sub)polyhedral abstract domains. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8471 of *LNCS*, pages 200–215. Springer, 2014. URL https://hal.archives-ouvertes.fr/hal-00991853.

Alexis Fouilhé, David Monniaux, and Michaël Périn. Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra. In *Static Analysis Symposium (SAS)*, volume 7935 of *LNCS*, pages 345–365. Springer, 2013. URL http://hal.archives-ouvertes.fr/hal-00806990.

Komei Fukuda. How hard is it to verify that an H-polyhedron and a V-polyhedron are equal?, 2004. URL https://www.inf.ethz.ch/personal/fukudak/polyfaq/node21.html.

Komei Fukuda. *CDD Homepage*, 2016. URL https://www.inf.ethz.ch/personal/fukudak/cdd_home/.

Komei Fukuda and Alain Prodon. Double description method revisited. In *Combinatorics and Computer Science*, LNCS, pages 91–111. Springer, 1996.

Tomas Gal and Josef Nedoma. Multiparametric linear programming. *Management Science*, 18 (7):406–422, 1972.

Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

Jacques Garrigue. Relaxing the value restriction. In *Asian Programming Languages and Systems Symposium (APLAS)*, volume 2998 of *LNCS*, pages 31–45. Springer, 2002.

Alan J. Goldman and Albert W. Tucker. Polyhedral convex cones. In *Linear inequalities and related systems*, volume 38 of *Annals of Mathematics Studies*, pages 19–40. Princeton University Press, 1956.

Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In *Principles of Programming Languages (POPL)*, pages 119–130. ACM Press, 1978. URL http://www-public.tem-tsp.eu/~gibson/Teaching/CSC4504/ReadingMaterial/GordonMMNW78.pdf.

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *LNCS*. Springer, 1979.

Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Theorem Proving in Higher Order Logics (TPHOL)*, volume 3603 of *LNCS*, pages 98–113, 2005. URL http://www.cs.ru.nl/~freek/courses/tt-2014/read/10.1.1.61.3041.pdf.

Benjamin Grégoire, Loïc Pottier, and Laurent Théry. Proof certificates for algebra and their application to automatic geometry theorem proving. In *Automated Deduction in Geometry (ADG)*, volume 6301 of *LNCS*. Springer, 2008.

Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Université scientifique et médicale de Grenoble, 1979. (in french).

David Handelman. Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific Journal of Mathematics*, 132(1):35–62, 1988. URL `http://msp.org/pjm/1988/132-1/pjm-v132-n1-p04-s.pdf`.

John Harrison and Laurent Théry. A skeptic's approach to combining HOL and maple. *Journal of Automated Reasoning*, 21(3):279–294, 1998.

Jacob M. Howe and Andy King. Polyhedral analysis using parametric objectives. In *Static Analysis Symposium (SAS)*, volume 7460 of *LNCS*, pages 41–57. Springer, 2012.

Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification (CAV)*, 2009. URL `https://hal.inria.fr/hal-00786354`.

Colin. Jones, N., Eric C. Kerrigan, and Jan M. Maciejowski. On polyhedral projections and parametric programming. *Journal of Optimization Theory and Applications*, 138(2):207–220, 2008. URL `https://spiral.imperial.ac.uk/bitstream/10044/1/4344/1/proj_mplp.pdf`.

Colin N. Jones, Eric C. Kerrigan, and Jan M. Maciejowski. Lexicographic perturbation for multiparametric linear programming with applications to control. *Automatica*, 43(10):1808–1816, 2007. URL `https://pdfs.semanticscholar.org/891f/f644ab4ccd970a5fce881a02972686a68299.pdf`.

Jacques-Herni Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *Principles of Programming Languages (POPL)*, pages 247–259. ACM Press, 2015. URL `http://gallium.inria.fr/~xleroy/publi/verasco-popl2015.pdf`.

Dejan Jovanovic and Leonardo de Moura. Solving non-linear arithmetic. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *LNCS*, pages 339–354. Springer, 2012.

Chantal Keller. Extended resolution as certificates for propositional logic. In *Proof Exchange for Theorem Proving*, volume 14 of *EPiC Series in Computing*, pages 96–109. EasyChair, 2013.

To Van Khanh, Xuan-Tung Vu, and Mizuhito Ogawa. rasat: SMT for polynomial inequality. In *Satisfiability Modulo Theories (SMT)*, page 67, 2014. URL `http://ceur-ws.org/Vol-1163/paper-11.pdf`.

Jean-Louis Krivine. Anneaux préordonnés. *Journal d'analyse mathématique*, 12:307–326, 1964. URL `https://hal.archives-ouvertes.fr/hal-00165658/`.

Jean Bernard Lasserre. *Moments, Positive Polynomials and Their Applications*, volume 1 of *Imperial College Optimization Series*. Imperial College Press, 2010.

Jean-Louis Lassez, Tien Huynh, and Ken McAloon. Simplification and elimination of redundant linear arithmetic constraints. In *Constraint Logic Programming*, pages 73–87. MIT Press, 1993.

Hervé Le Verge. A Note on Chernikova's algorithm. Research Report RR-1662, INRIA, 1992. URL `https://hal.inria.fr/inria-00074895`.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. URL `http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf`.

Vincent Loechner and Philippe Clauss. *Polylib*, 2010. URL `http://icps.u-strasbg.fr/PolyLib`.

Victor Magron, Xavier Allamigeon, Stéphane Gaubert, and Benjamin Werner. Formal proofs for nonlinear optimization. *Journal of Formalized Reasoning*, 8(1):1–24, 2015.

Andrew Makhorin. GNU Linear Programming Kit, 2000–2017. URL `http://www.gnu.org/software/glpk/glpk.html`.

Alexandre Maréchal and Michaël Périn. Efficient elimination of redundancies in polyhedra by raytracing. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 10145 of *LNCS*, pages 367–385. Springer, 2017. URL `https://hal.archives-ouvertes.fr/hal-01385653`.

Alexandre Maréchal, Alexis Fouilhé, Tim King, David Monniaux, and Michaël Périn. Polyhedral approximation of multivariate polynomials using Handelman's theorem. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS, pages 166–184. Springer, 2016. URL `https://hal.archives-ouvertes.fr/hal-01223362`.

Alexandre Maréchal, David Monniaux, and Michaël Périn. Scalable minimizing-operators on polyhedra via parametric linear programming. In *Static Analysis Symposium (SAS)*, volume 10422 of *LNCS*. Springer, 2017. URL `https://hal.archives-ouvertes.fr/hal-01555998`.

Peter McMullen. The maximum numbers of faces of a convex polytope. *Mathematika*, 17: 179–184, 1970.

Peter McMullen and Geoffrey C. Shepard. *Convex polytopes and the upper bound conjecture*, volume 3 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1971.

Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *LNCS*, pages 348–363. Springer, 2006. URL `https://hal.inria.fr/hal-00136661`.

Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1): 31–100, 2006. URL `https://www-apr.lip6.fr/~mine/publi/article-mine-HOSC06.pdf`.

David Monniaux. Quantifier elimination by lazy model enumeration. In *Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 585–599. Springer, 2010.

T.S. Motzkin, H. Raiffa, G.L. Thompson, and R.M. Thrall. The double description method. *Contributions to the Theory of Games*, 2:51–74, 1953.

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, 2013.

George C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL)*, pages 106–119. ACM Press, 1997.

Pierre Néron. *A Quest for Exactness: Program Transformation for Reliable Real Numbers*. PhD thesis, École Polytechnique, Palaiseau, France, 2013. URL `https://tel.archives-ouvertes.fr/tel-00924379`.

Derek C. Oppen. Complexity, convexity and combinations of theories. *Theor. Comput. Sci.*, 12: 291–302, 1980.

Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7737 of *LNCS*, pages 434–454. Springer, 2013.

Pierre Letouzey. *Certified functional programming : Program extraction within Coq proof assistant*. PhD thesis, Paris Diderot, 2004. URL `https://www.researchgate.net/publication/280790704_Certified_functional_programming_Program_extraction_within_Coq_proof_assistant`.

Pierre Letouzey. Extraction in Coq: An Overview. In *Computability in Europe (CiE)*, volume 5028 of *LNCS*, pages 359–369. Springer, 2008. URL `https://www.irif.fr/~letouzey/download/letouzey_extr_cie08.pdf`.

Alexandre Prestel and Charles N. Delzell. *Positive Polynomials: From Hilbert's 17th Problem to Real Algebra*. Springer-Verlag, 2001.

Shashwati Ray and P. S. V. Nataraj. A matrix method for efficient computation of bernstein coefficients. *Reliable Computing*, 17(1):40–71, 2012. URL `http://interval.louisiana.edu/reliable-computing-journal/volume-17/reliable-computing-17-pp-40-71.pdf`.

Matteo Rubagotti, Davide Barcelli, and Alberto Bemporad. Robust explicit model predictive control via regular piecewise-affine approximation. *International Journal of Control*, 87(12): 2583–2593, 2014. URL `http://cse.lab.imtlucca.it/~bemporad/publications/papers/ijc-rmpc-pwas.pdf`.

Markus Schweighofer. An algorithmic approach to Schmüdgen's Positivstellensatz. *Journal of Pure and Applied Algebra*, 166(3):307–319, 2002. URL `http://www.sciencedirect.com/science/article/pii/S002240490100041X`.

Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

Axel Simon and Andy King. Exploiting sparsity in polyhedral analysis. In *Static Analysis Symposium (SAS)*, volume 3672 of *LNCS*, pages 336–351. Springer, 2005.

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Making numerical program analysis fast. In *Programming Language Design and Implementation (PLDI))*, pages 303–313. ACM Press, 2015. URL `http://elina.ethz.ch/papers/PLDI15-Octagon.pdf`.

Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In *Principles of Programming Languages (POPL)*, pages 46–59. ACM Press, 2017.

Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: a cross-community infrastructure for logic solving. In *International Joint Conference on Automated Reasoning (IJCAR)*, LNIA. Springer, 2014.

The Coq Development Team. *The Coq proof assistant reference manual – version 8.6*. INRIA, 2016. URL `https://coq.inria.fr/refman/`.

Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Principles of Programming Languages (POPL)*, pages 17–27. ACM Press, 2008.

Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359. ACM Press, 1989.

Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.

Doran K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, 1993. URL `https://hal.inria.fr/inria-00074515/document`. Also published as IRISA Technical Report PI 785, Rennes, France (1993).

Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4): 343–355, 1995.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation (PLDI))*, pages 283–294. ACM Press, 2011. URL `https://www.flux.utah.edu/download?uid=114`.

N. Yu. Zolotykh. New modification of the double description method for constructing the skeleton of a polyhedral cone. *Computational Mathematics and Mathematical Physics*, 52(1): 146–156, 2012. URL `http://www.uic.unn.ru/~zny/papers/skeleton.pdf`.

Index

abstract interpretation, 11

Bernstein, 80
  basis
    multivariate, 81
    univariate, 80
  certification, 88
  precision refinement, 87

certificates, 117
  Bernstein's linearization, 88
  Handelman's linearization, 95, 143
  inclusion, 24
  irredundancy, 44
  join, 126
  minimization, 38
  projection, 120
  witness points, 40
certification
  a posteriori, 116
  autarkic, 116
  dynamic, 116
  factories, 117
  skeptical, 117
  static, 115
Chernikova's algorithm, 14
convex optimization, 25
Coq
  extraction, 116
  lemma, 112
  proof, 112
  proof assistant, 112
  tactic, 112, 133

double description
  minimization, 50

Farkas' lemma, 19
floating point
  minimization, 45
Fourier-Motzkin elimination, 20

generators
  minimization, 46

Handelman, 89
  certification, 95, 143
  representation of polynomials, 89
  theorem, 90
  via PLP, 90

intersection, 18
irredundancy, 35, 44

linear programming, 25
  degeneracy, 107
  dictionary, 28
  echelon form, 27
  pivot, 30
linearization, 18, 73
  Bernstein, 80
  Handelman, 89
  intervalization, 74

multivariate polynomials, 73

normalization constraint, 60
  irredundancy, 62

operators, 15
  assignment, 18
  complexity, 19
  emptiness, 16
  guard, 18
    nonlinear, 73
  inclusion, 16, 24
  join, 17, 126
    via PLP, 67
  linearization, 18, 73
  meet, 18
  minimization, 18, 35, 40
  PLP, 53
  projection, 16, 120
    via PLP, 54
  reduction, 138
  widening, 18

parametric linear programming, 53
  algorithm by generalization, 104
  convex hull, 67
  degeneracy, 107
  Handelman's linearization, 90
  normalization, 60
  polyhedral solution, 56

projection, 54
regions of optimality, 56
solver, 59
tree-exploration, 102
polyhedra, 12
constraints, 12
dimension, 13
double description, 14
generators, 14
operators, 15
polyhedral cones, 36

raytracing, 35
redundancy, 35
regions of optimality, 56

satisfiability modulo theory, 97
Schweighofer's theorem, 95
simplex algorithm, 27
static analysis, 11