Revisiting the abstract domain of polyhedra: constraints-only representation and formal proof

Alexis Fouilhé

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Alexis Fouilhé

Thèse dirigée par **David Monniaux**
et coencadrée par **Michaël Périn**

préparée au sein **du laboratoire VERIMAG**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Le domaine abstrait des polyèdres revisité : représentation par contraintes et preuve formelle

Revisiting the abstract domain of polyhedra: constraints-only representation and formal proof

Thèse soutenue publiquement le **16 octobre 2015**,
devant le jury composé de :

Pr. Paul Feautrier
ÉNS de Lyon, Président
Pr. Andy King
University of Kent, Rapporteur
Pr. Enea Zaffanella
Università degli Studi di Parma, Rapporteur
Pr. Sandrine Blazy
Université de Rennes 1, Examinatrice
Dr. Chantal Keller
Université Paris-Sud, Examinatrice
Dr. David Monniaux
CNRS, Directeur de thèse
Dr. Michaël Périn
Université Joseph Fourier, Co-Encadrant de thèse

Contents

The validation of safety-critical software, such as fly-by-wire systems in aircrafts, revolves around testing and manual code review. Besides functional requirements, such as a fly-by-wire system preserving the stability of the aircraft, safety-critical executable code must verify more technical safety properties. For example, it must never divide numbers by zero, or try to access memory locations which don't exist. Formal methods provide tools to check that these properties are verified. However, in today's validation process of safety-critical software, these tools can't replace testing for the same reason the assembly code produced by a compiler is manually checked to match its source code: the tools are complex pieces of software. They are implemented by human beings and might have bugs. This thesis reports on work aiming at making the result of formal method tools more trustworthy.

Trust. Our focus will be on abstract interpreters [16]. These are *sound* code analysis tools, which means that they consider all possible executions of the program they analyse. They check that none of these executions can violate the required safety properties. For state-of-the-art abstract interpreters, such as Astrée [7], the mathematical proof that, if the analyser doesn't report possible violations, then no execution of the analysed program can violate safety properties, exists on paper. The distance between this proof and the actual code of the analyser can't be measured, which presents a problem of trust.

In order to bridge the gap between proof and implementation, we will use a proof assistant, called Coq [50]. A proof assistant—this isn't specific to Coq—combines

- a language for writing programs,
- a language for expressing properties they should verify and
- a language for proving that programs indeed verify the properties.

The key benefit is that the proof assistant checks that the proofs are consistent.

The Compcert compiler. One of the most successful undertakings using machine-checked proofs is the Compcert compiler [40] which compiles most of the ISO C99 programming language to efficient code for PowerPC, ARM and x86 processors. It comes with a proof that the compiler preserves the semantics of the source program, which guarantees that the behaviour of the compiled program is one of the possible behaviours of the source program.

One step further: Verasco. This guarantee only holds as long as the source program doesn't behave in a way which is undefined by the C standard. Undefined behaviours include divisions by zero and out-of-bounds array accesses and their absence can be checked by an abstract interpreter. My work is part of project Verasco [37], which aims at building an abstract interpreter and writing its soundness proof using Coq. The resulting tool, Verasco, can discharge the assumption made in Compcert proof that the behaviour of the source program is always well-defined. My work within Verasco focuses on capturing relations between the numerical variables of a program.

```
1  Module M.
2    Inductive list (A : Type) : Type :=
3      nil : list A | cons : A → list A → list A.
4
5    Arguments nil {A}.
6    Arguments cons {A} a l'.
7    Infix "::" := cons (at level 60, right associativity).
8
9    Fixpoint app {A : Type} (l m : list A) {struct l} : list A :=
10     match l with
11     | nil ⇒ m
12     | a :: l' ⇒ a :: app l' m
13     end.
14
15   Lemma app_nil : ∀ (A : Type) (l : list A), app l nil = l.
16   Proof.
17     intros A l.
18     induction l as [|a l' ih].
19     − reflexivity.
20     − simpl.
21       rewrite ih.
22       reflexivity.
23   Qed.
24 End M.
25
26 Extraction M.
```

Listing 1 – an example of Coq code

Using a proof assistant

Before we see the main ideas behind the static analysis performed by Verasco, I'll try to give a flavour of what proof assistants do, through two example code fragments. The first fragment is written in the Coq input language and is shown on listing 1. The second fragment is the Ocaml code equivalent to listing 1 and is shown on listing 2. It is generated by Coq *extraction* mechanism, which strips all the proof-related parts of a Coq development so as to build a usual Ocaml program. We will refer to it in order to shed light on Coq constructs.

An example of Coq code

The Coq code of module M shown on listing 1, illustrates the three activities which using a proof assistant consists in.

Programs. Coq features a purely functional language for writing programs. Our example defines a data type list of polymorphic lists, on line 2, and a function app operating on this type, on line 9, which appends a list to another. The extracted list and app, on lines 3 and 13 of listing 2, should convince you that these definitions are mostly straightforward.

```
1  module M =
2   struct
3    type 'a list =
4    | Coq_nil
5    | Coq_cons of 'a * 'a list
6
7    (** val list_rec : 'a2 → ('a1 → 'a1 list → 'a2 → 'a2) → 'a1 list → 'a2 **)
8    let rec list_rec f f0 = function
9    | Coq_nil → f
10   | Coq_cons (y, l0) → f0 y l0 (list_rec f f0 l0)
11
12   (** val app : 'a1 list → 'a1 list → 'a1 list **)
13   let rec app l m =
14     match l with
15     | Coq_nil → m
16     | Coq_cons (a, l') → Coq_cons (a, (app l' m))
17  end
```

Listing 2 – the OCaml code extracted from listing 1

Specifications. Then comes what we usually do on paper. Lemma app_nil, on line 15, states one specification which function app is supposed to meet: appending the empty list to a list l yields list l. Note that there is no corresponding definition to app_nil in the extracted code of listing 2. Removing the proof-related material is precisely the reason the extractor exists.

Proofs. The fragment between lines 16 and 23 is the *proof* that specification app_nil is met. A Coq proof is built using *tactics*. Each tactic represents a step of reasoning. Understanding a proof often requires replaying within Coq, in order to see how each step transforms the proof goal. I'll only detail a few tactics here. More details can be found in Coq reference manual [50]. The main step in the proof of lemma app_nil is induction l [|a l' ih], on line 18. It states "We now perform an induction on the structure of list l". Naming instructions for each of the two cases—they are introduced by a dash—of the induction are given within square brackets. The base case, on line 19, assumes that the list l is nil. The inductive case, on line 20, assumes that the list l is of the form a :: l'. The induction hypothesis, ih, assumes that app l' nil = l'. Another example of a tactic is rewrite ih, on line 21. It is used for the inductive case, where we need to prove a :: app l' nil = a :: l'. The induction hypothesis ih is used to rewrite app l' nil on the left-hand side, into l'.

Realistic Coq

Besides the general principle of a proof assistant, the code fragments on listings 1 and 2 show some features of a realistic Coq development.

{**struct** l}**.** Every function defined in Coq must terminate and you need to explain why when Coq isn't convinced. In our example, function app operates

by deconstructing list l: each recursive call applies to the tail of the list, which eventually becomes the empty list. Function app is said to perform a *structural descent* on list l, written {**struct** l}.

Arguments nil {A}. Parametric polymorphism is a lot more powerful in Coq than it is in OCaml. As the number of parameters grow, writing them down each time obfuscates the code. What's more, Coq can often infer them from the context. It is therefore possible to declare them as implicit. For example, constructor nil of type list has type ∀ A, list A. The **Arguments** command above declares that the type parameter A can be omitted. This permits to write app l nil instead of app l (nil A). Similarly, {A : **Type**} declares an implicit type parameter A in the definition of function app.

Infix "::" = cons. A Coq development is also made more readable using *notations*, much like notations are introduced in mathematics. The one defined here makes a :: l' mean cons a l'. Notations in Coq are two-edged. They make the code a lot more concise and convey the big picture to the newcomer more easily. However, they can redefine most of the syntax of the language, which makes detailed understanding a lot harder.

list_rec. The extracted code in listing 2 defines a function which doesn't appear in the Coq code: list_rec. It is a generalised folding function for type list. Coq generates such a function automatically when an inductive data type is defined: it is defined implicitly in Coq module M. Function list_rec is a form of induction principle for type list, although it is beyond the scope of this overview to explain how this relates to proofs being eliminated by the extractor. A thorough explanation can be found in the book "Certified Programming with Dependent Types" by Adam Chlipala [13].

Trusted computing base

Proof assistants such as Coq are among the tools which provide the highest level of confidence in software. Trusting the result of a program written in Coq along with its proof of correctness is reduced to trusting three well-defined elements.

The specification. As with any specification, formal or written in prose, you need to make sure that the Coq specification captures the informal idea of correctness of the program. This may require some work. For example, formalised semantics of the C programming language are part of Verasco specification. Although the project reused the formalised semantics written for the CompCert compiler, they had to be tested extensively against the C standard in the early stages of the development of CompCert.

The proof assistant. Once we have good faith in the specification capturing our intention, we should turn our attention to the proof assistant itself. Using a proof assistant amounts to transferring one's trust from the program being developed—in our case, an abstract interpreter—to the proof assistant. Both CompCert and Verasco use the Coq proof assistant. Coq is designed so

that all the proofs are checked by a single and well-defined component: a type-checking kernel. This kernel is what you need to trust, keeping in mind that it has been tested over several decades in a wide variety of settings.

The extractor. It is possible to execute a Coq program inside Coq. However, the execution environment provided by Coq is a closed world. For example, Verasco couldn't read a file containing the source program we wish to analyse from within Coq. The program extractor generates a standalone Ocaml program, equivalent to the Coq program but without the proofs, as we saw on listing 2. The extract program can be linked with hand-written Ocaml code and therefore interact with the rest of the world. As a side note, the generated Ocaml program needs to be compiled into a regular binary program, which brings the Ocaml compiler among the trusted components.

Static analysis of computer programs

Now that we settled on what guarantees using Coq enables us to claim, we may look further at what Verasco actually does. From a user perspective, it analyses a C program, checking whether some of its executions may behave in way which is specified as undefined by the C standard. "Absence of undefined behaviours" makes an implicit specification, which can be enriched by adding `assert()` statements to the program under analysis. If the analyser is able to find that all the behaviours of the program are well defined and that no assertion can be violated, then the analyser declares the program correct. An analyser such as Verasco also has the following features.

- It is a *static* analyser: it analyses programs without a computer actually running them.
- It is also a *source code* analyser: it takes C source code as input. Work has been done to analyse binary programs [4], on the ground that all sorts of errors may be introduced by the compiler. Our setting, which includes the Compcert compiler, rids us of the concern.
- It is an *automatic* analyser: it requires no user input beyond the source code of the input program.
- Last, it is a *sound* analyser: it is built on the theory of abstract interpretation [16] and discovers properties of the program under analysis which hold for all of its executions.

Under the hood, Verasco analyses a program by computing an overapproximation of the reachable memory states after each of its instructions. *Over*approximation is key to ensure soundness: the analyser is allowed to consider memory states which can't be reached by an actual execution, but it mustn't neglect a state which the program can be in. The latter would be an underapproximation. The analysis starts from an overapproximation of the reachable memory states at the entry point, usually considering that nothing is known on the contents of memory. From there, it performs a line-by-line exploration of the source code, similarly to an actual execution, except that all of its possible states are considered at once, hence the name "abstract interpretation".

Abstract domains

Static analysis would be intractable without overapproximation. It is called "abstraction", capturing the idea that the analyser cannot keep track of all the fine details of every execution of the program being analysed: it needs to infer some sort of overview.

The most common abstractions capture information about the possible values of the numerical variables of a program. It would be very cumbersome and memory hungry to keep this information as value sets. Consider for example parameter `x` of function `f` in listing 3. Assuming no information is provided on the calling context, it may take any value of type **int**, which has 2^{32} possible values. A more compact representation of the same information is the interval $[-2^{31}, 2^{31} - 1]$, which is what analysers usually use. This representation has a drawback, though. Consider the possible values of variable `i` in listing 3 when the execution reaches the **return** statement on line 15. You will convince yourself that the value of variable `i` is either 5 or 100, which an interval would record as $[5, 100]$. While saying that the value of variable `i` is between 5 and 100 is correct—in no execution of function `f` can variable `i` have a value outside of this interval—, it is also an overapproximate statement.

In the theory of abstract interpretation, a type of abstraction, such as intervals, combined with a set of operators to compute over abstractions of this type, is called an "abstract domain".

Example analysis

In order to make this discussion more concrete, we may now observe how the abstract interpretation of function `f` on listing 3 proceeds, using the abstract domain of intervals. We will overlook the fact that the numbers are machine integers and just consider them mathematical integers in $\mathbb{Z}$. The analysis determines an interval for the possible value of each variable, that is a product of intervals, after each instruction. The results are summarised in the comments on listing 3. Right after the entry point of the function, all the analyser knows is that there are two variables `x` and `i` and they may have any value. Line 5 through 10 are executed under the condition that the value of `x` is no more than 3. This information is used to restrict the subset of the reachable states to those where $i \in]-\infty, +\infty[$ and $x \in]-\infty, 3]$. A guard such as `x <= 3` in the program code results in the analyser performing an intersection of the reachable states before the guard with the states satisfying the guard. The assignment on line 5 determines the value of variable `i` in all of these executions.

Next in the program comes a loop. Loops are the major difficulty of static analysis: they preserve a loop invariant which doesn't appear explicitly in the source code. Therefore, the analyser needs to be creative and infer an invariant which can be expressed in the abstract domain. The loop of function `f` is bounded trivially: it would be easy to unroll it entirely to find out that variable `i` has value 100 when the loop exits. However, unrolling would be slow on this example and it is not possible in general. Instead, the analyser guesses a possible invariant and checks whether it is inductive, meaning that it holds when the loop is entered and, if it holds at the beginning of an iteration of the loop, then it also holds at the beginning of the next interaction. If the guess of the analyser isn't inductive, a coarser guess is tried and checked for inductiveness, until the process stabilises:

```
1  int f(int x) {
2    int i;
3    /* x ∈]−∞, +∞[, i ∈]−∞, +∞[ */
4    if(x <= 3) { /* x ∈]−∞, 3], i ∈]−∞, +∞[ */
5      i = 1; /* x ∈]−∞, 3], i ∈ [1, 1] */
6      while(i <= 99) /* x ∈]−∞, 3], i ∈ [1, 100] */ {
7        /* x ∈]−∞, 3], i ∈ [1, 99] */
8        i = i + 1; /* x ∈]−∞, 3], i ∈ [2, 100] */
9      }
10     /* x ∈]−∞, 3], i ∈ [100, 100] */
11   } else { /* x ∈ [4, +∞[, i ∈]−∞, +∞[ */
12     i = 5; /* x ∈ [4, +∞[, i ∈ [5, 5] */
13   }
14   /* x ∈]−∞, +∞[, i ∈ [5, 100] */
15   return i;
16 }
```

Listing 3 – Program fragment illustrating abstract interpretation over intervals

the sequence of guesses is designed so that the analysis eventually stops. In the worst case, the analyser considers that the possible values of all the variables lie in interval $]-\infty, +\infty[$, which is trivially inductive, but makes a useless invariant. It is called "*top*".

In the example at hand, let the guess be $i \in [1, 100]$, $x \in]-\infty, 3]$. It holds initially, when the loop is entered: $i = 1$ and $x \leq 3$. To check that it is preserved by the loop, notice that the loop body is executed under the condition $i \leq 99$. This restricts the possible values of variable i in the same way an **if** statement would. The assignment i = i + 1, on line 8, simply offsets the interval of possible values by 1. At the end of the loop body, the interval for variable i is included in that of the candidate invariant and variable x was left untouched. The candidate invariant is thus inductive.

When the loop exits, the condition $i \leq 99$ is false. The reachable states after the loop are in the intersection of the loop invariant and the negated condition $i > 99$, which is equivalent to $i \geq 100$ for integers. It results that $x \in]-\infty, 3[$ and $i \in [100, 100]$. The **else** branch of the test on line 4 is executed in the subset of the reachable states where $x \in [4, +\infty[$.

What happens next introduces the last abstract domain operator: *join*. When the analyser reaches the **return** statement on line 15, it has to consider two possible scenarios. An execution path through function f comes from either line 10 or line 12. This alternative can't be represented in the abstract domain without introducing a disjunction. The join operator of the domain of intervals summarises what happened in both paths by computing the union of the intervals for each variable. Without information on the calling context, the analysis of function f infers that it returns an integer between 5 and 100.

Making guesses and widening

The example analysis we just followed skipped over the details of guessing a candidate loop invariant. The accuracy of the guess has a big impact on the

precision of the properties discovered during the analysis, the worst case being guessing no better than the trivial property *top*, which always holds. Using *top* as a loop invariant in the analysis of function `f` yields "function `f` returns an integer no smaller than 5".

Making good guesses is inherently heuristic and is a research problem [1] which I won't cover here beyond the most common solution. The standard approach, widening [15], is a kind of extrapolation based on the analysis of the first few iterations of the loop under consideration. On the example above, the usual widening operator of the abstract domain of intervals would go like this.

1. Upon entry, $i \in [1, 1]$.
2. After at most one iteration, $i \in [1, 2]$.
3. After at most two iterations, $i \in [1, 3]$.
4. It seems that the upper bound increases each time, whereas the lower bound remains stable. Let's try $i \in [1, +\infty[$.
5. Interval $i \in [1, +\infty[$ is an inductive invariant.
6. An extra iteration, with candidate $i \in [1, +\infty[$, would recover $i \in [1, 100]$. This final step is called a narrowing iteration [15].

Polyhedra: a more precise numerical abstraction

Once the analyser has computed an abstraction of the reachable memory states after each instruction, it uses these abstractions to check whether the behaviour of the program under analysis is always well defined. The hoped-for case is that it is. If it isn't, the analyser warns the user: a bug in the program may have been discovered. However, the analyser may find that the program under analysis has undefined behaviours, but no actual execution of the program exhibits these undefined behaviours. The warning is a false alarm: the program is correct, but the analyser can't infer strong enough invariants to check it. This situation may originate from overapproximations performed during a join or a widening. It may also be the case that the chosen abstract domain can't represent the necessary inductive invariant. Using a more precise abstraction may alleviate the issue.

Consider function `g` of listing 4. If no information on the calling context is supplied for parameters `x` and `y`, an abstract interpretation using intervals won't give any useful information on the return value of function `g`, although you will convince yourself that $x + 2 \leq \mathtt{g}(x, y)$, irrespective of the values x and y of parameters `x` and `y`. This can't be deduced using intervals, because intervals can't capture relations between the value of variables. In function `g`, there is nothing to learn on each variable independently.

The domain of intervals is but one numerical abstraction. Its operators are very fast, but being a nonrelational abstraction often makes it imprecise. The abstract domain of convex polyhedra [17] is designed to capture relations between the variables of a program, as long as they don't involve products of variables. It can handle linear inequality or linear equality relations, such as $x + 2 \leq y$, which is enough for our needs for function `g`.

```
1  int g(int x, int y) {
2    if(x <= y) { /* x ≤ y */
3      y = y + 2; /* x + 2 ≤ y */
4    } else { /* y + 1 ≤ x */
5      y = x + 2; /* x + 2 = y */
6    }
7    /* x + 2 ≤ y */
8    return y;
9  }
```

Listing 4 – Program fragment illustrating abstract interpretation over polyhedra

The comments in listing 4 show what properties are inferred after each instruction. The assumption under which the **else** branch of the test is executed is, strictly speaking, $x > y$. Given that variables x and y are integers, it can be recast as $x \geq y + 1$. The result of the analysis $x + 2 \leq \mathsf{g}(x, y)$ follows from relation $x + 2 \leq y$ being a correct overapproximation of the disjunction $x + 2 \leq y \vee x + 2 = y$.

Verasco: proving the soundness of an abstract interpreter

Coming back to Verasco after some context has been provided, a more precise picture of the goal can be painted. We have just seen that checking safety properties in software can be delegated to static analysers, which are complex tools. In a context of safety-critical systems, the trustworthiness of the analysis results comes into question. Project Verasco aims at building an abstract interpreter and writing the proof that it performs a sound analysis using the Coq proof assistant. "Sound" means that all possible behaviours of the program being analysed have been considered. In an attempt to reduce the proof effort, only soundness is proved. Precision is sought through regular software engineering techniques. The resulting analyser handles the same subset of the C programming language as the Compcert compiler, that is all of C99 but unstructured **switch** statements, variable-length arrays and setjump and longjump. It operates on one of Compcert intermediate representations, which occurs early in the compilation process, and relies on the formalised semantics from Compcert proof of semantics preservation.

Verasco isn't the first attempt at formalising the soundness of an analyser using Coq. David Pichardie pioneered [46] the formalisation in Coq of abstract interpretation, with applications to a subset of the Java programming language. Project Verasco, however, is the first project targeting a full-blown programming language, such as C.

A high-level look at the architecture of Verasco reveals two major blocks: the analysis engine and the abstract domains.

- Each abstract domain provides a set of operators and the proof that each operator soundly overapproximates set operations on reachable states.
- The engine maps C instructions to domain operators. Its accompanying proof builds on those of the abstract domain to ensure that all possible executions of the program under analysis have been considered.

The collection of abstract domains in Verasco includes an abstract domain of intervals, an abstract domain for handling memory operations and an abstract domain of convex polyhedra. It also includes an abstract domain transformer which builds an abstract domain over machine integers from an abstract domain over mathematical integers in $\mathbb{Z}$.

Building the domain of polyhedra with the necessary proofs in Coq has been the focus of my work within project Verasco. It resulted in the Verimag Polyhedra Library, shortened VPL. VPL is integrated into the Verasco analyser [37] and is also available separately. The rest of the thesis covers the challenges I faced while building VPL.

Summary in French

Un analyseur statique est un outil permettant de démontrer l'absence de certains types d'erreurs dans les programmes. Un tel outil peut-être utilisé en remplacement de tests. Quelle confiance accorder aux résultats de l'analyseur ? Il peut avoir des bugs. Nous allons proposer une solution à ce problème basée sur la preuve formelle à l'aide de l'assistant à la preuve Coq.

L'outil Coq combine trois éléments : un langage de programmation, un langage de spécification et un langage de preuve. Une preuve démontre qu'un programme satisfait une spécification. L'intérêt d'utiliser un assistant de preuve réside dans le fait qu'il vérifie la bonne construction des preuves. En ce sens, utiliser un assistant à la preuve revient à transférer sa confiance du programme développé vers le composant de l'assistant qui valide les preuves. Coq est conçu de façon à réduire la taille de ce composant.

L'analyse statique que nous allons considérer s'appelle «interprétation abstraite». Elle consiste à calculer une approximation de l'ensemble des états possibles d'un programme après chaque instruction. L'interprétation abstraite se différencie principalement du test en ce qu'elle considère toutes les exécutions possibles d'un programme, alors que le test n'en vérifie que quelques unes.

Décrire exactement l'ensemble des états d'un programme après une instruction donnée est théoriquement impossible : il est nécessaire d'avoir recours à une approximation. Pour être correcte, une telle approximation doit inclure au moins tous les états atteignables. Elle peut aussi inclure des états en pratique inatteignables. Un type d'approximation, avec un ensemble d'opérateurs pour le manipuler, est appelé un «domaine abstrait». Le domaine abstrait le plus courant associe à chaque variable d'une programme l'intervalle de ses valeurs possibles. Le domaine abstrait des intervalles ne peut pas représenter des relations entre les valeurs des variables.

Le domaine abstrait des polyèdres sert à représenter des ensembles d'états atteignables d'un programme, en capturant les relations linéaires entre les valeurs possibles des variables.

Cette thèse présente la construction de VPL, un domaine abstrait de polyèdres, réalisé pour l'analyseur statique Verasco. Verasco est écrit et prouvé correct à l'aide de Coq. La conception de VPL permet à la fois de remplir les obligations de preuve imposées aux domaines abstraits par Verasco et d'obtenir des performances raisonnables du domaine abstrait.

Part I

An abstract domain of polyhedra with a formal soundness proof

The abstract domain of polyhedra was introduced by Patrick Cousot and Nicolas Halwachs in 1978 [17]. It is designed to capture relations between variables during the analysis of a program, as opposed to the original domain of intervals [15], which considers each variable separately. In more than three decades, many refinements [51, 1, 2] have been published, enhancing both computational efficiency and precision. In comparison, using proof assistants to build an abstract domain with a formal proof of correctness is a recent undertaking [6].

This first part reports on my contribution to bridging the performance gap between formally proved abstract domains of polyhedra and their counterparts built following traditional software engineering practices. My work on VPL keeps the result verification approach based on witnesses, which was pioneered by Frédéric Besson et al. [6]. However, it goes against the common practice of using the double description of polyhedra, in favor of a constraints-only representation.

Chapter 1

Efficient result verification

VPL guarantees that its results are correct using a small checker to verify, after they are computed, that the results meet the desired properties. Although this approach, called "result verification", adds a small checking overhead to each computation, it reduces the formal proof work to the simple checker. Before we dive further into the specifics of polyhedra, let us consider the problem we need to solve from some distance.

Verasco analyses an input C program using an abstract domain chosen by the user from a set including, among others, intervals and convex polyhedra. Choosing at run time which abstract domain to use is made possible by having all abstract domains implement the same interface, which is reproduced on listing 1.1. This interface is richer than interfaces commonly encountered in mainstream programming languages, such as Ocaml or C++: all of the fields ending with "_correct" on listing 1.1 are proof obligations. However, similarly to mainstream programs, interfaces provide structure, both to the code of the analyser and to its soundness proof.

1.1 What needs to be proved?

VPL implements interface ab_ideal_env. The name stands for "abstract domain over ideal environments", where an environment represents a memory state, in the form of a function from variable names to values. In this context, "ideal" means that variables have values from the set of integers $\mathbb{Z}$. Real program variables, however, have values from the set of machine integers, which fit in either thirty two or sixty four bits. Verasco provides a domain transformer to build an abstract domain over machine environments from a domain satisfying interface ab_ideal_env.

The interface of abstract domains is defined as a type class [49], which is similar to a module. The definition is parameterised by a type t, which is the type of abstract values, such as the type of products of intervals or the type of polyhedra.

The case of $\bot$. Actually, most operators have t+$\bot$ as return type. The abstract value $\bot$ represents the infeasible, or unreachable, state. The return value is "either an element of t or $\bot$". Distinguishing $\bot$ syntactically allows

```
1  Class ab_ideal_env (t:Type) : Type := {
2    γ :> gamma_op t (var → Z);
3    id_top: t+⊥;
4
5    id_leb: t → t → bool;
6    id_join: t → t → t+⊥;
7    id_widen: t → t → t+⊥;
8
9    assume: iexpr var → bool → t → t+⊥;
10   assign: var → iexpr var → t → t+⊥;
11   forget: var → t → t+⊥;
12
13   id_top_correct: ∀ m, m ∈ γ id_top;
14   id_leb_correct: ∀ p1 p2 m,
15     id_leb p1 p2 = true → m ∈ γ p1 → m ∈ γ p2;
16   id_join_correct: ∀ p1 p2 m,
17     m ∈ γ p1 \/ m ∈ γ p2 → m ∈ γ (id_join p1 p2);
18   assume_correct: ∀ c m p b,
19     m ∈ γ p → (if b then 1 else 0) ∈ eval_iexpr m c → m ∈ γ (assume c b p);
20   assign_correct: ∀ x e m n p,
21     m ∈ γ p → n ∈ eval_iexpr m e → (upd m x n) ∈ γ (assign x e p);
22   forget_correct: ∀ x m n p,
23     m ∈ γ p → (upd m x n) ∈ γ (forget x p);
24 }.
```

Listing 1.1 – Coq interface of Verasco abstract domains

to detect unreachable code at no cost, that is without having to test explicitly abstract values for emptiness. Consider, for example, the code fragment /*$x \geq 1$ */ **if**(x <= 0) { y = 3; }. There are two correct ways to analyse it.

- The naive way consists in accumulating bounds. Right before the assignment, the abstract value contains the information $x \geq 1$ and $x \leq 0$. There is no value for variable x satisfying these bounds, but the abstract domain hasn't detected it. After the assignment, the abstract value contains the information, $x \geq 1$, $x \leq 0$ and $y = 3$. Again there is no value for the variables x and y satisfying these conditions, but the abstract domain hasn't noticed.

- The alternative consists in *minimising*, that is drawing the most concise conclusions from the available information. Right before the assignment, the abstract value contains the information $x \geq 1$ and $x \leq 0$. The abstract domain realises that the bounds are incompatible and rewrites the conjunction as $\bot$. The analyser then knows that it can skip the rest of the block, since its entry point is unreachable.

The special value $\bot$ only makes analysis faster. We won't distinguish it in the following. Since most of what we'll cover is related to the abstract domain of polyhedra, we'll use letter p to name abstract values.

Concretization. The first component of interface ab_ideal_env relates type t of abstract values to program states. In Verasco, a program state is represented by a function from variables to integers of type Z. Such a function has type var → Z and can be thought as describing an infinite memory. Sets of infinite memories relate to abstract values through a *concretization* function γ. The set $\gamma\, p$ contains all the memories which are abstracted by abstract value p. Suppose we have a polyhedron p defined as $x \leq y$, as we had in listing 4. Then, any memory m, such that $m\, x \leq m\, y$, belongs to the set $\gamma\, p$.

An abstract domain must provide a value id_top, which abstracts all states. It is typically used as an abstraction of the initial state of the program under analysis, capturing the fact that no assumption is made. This appears in the proof obligation id_top_correct for id_top, on line 13. Next come the operators.

Inclusion test. Declared on line 5, the inclusion test is used to check that the analysis of a loop has stabilised on an inductive invariant. We note $p \sqsubseteq p'$ for id_leb p p' = true, where p and p' are abstract values. Spelt in conventional mathematical notation, the proof obligation for id_leb, on line 14, is as follows.

$$p \sqsubseteq p' \Rightarrow \forall m \in \gamma\, p,\, m \in \gamma\, p'$$

Inclusion in the abstract domain must be compatible with set inclusion of sets of concrete memory states.

Join operator. The join operator, line 6, overapproximates the disjunction which appears after an **if** statement. The result of id_join p p' is noted $p \sqcup p'$. It comes with the following proof obligation, from line 16.

$$\forall m,\, m \in \gamma\, p \,\vee\, m \in \gamma\, p' \,\Rightarrow\, m \in \gamma\, (p \sqcup p')$$

Operator $\sqcup$ is an overapproximation of disjunction.

Widening operator. When introducing abstract interpretation on an example, we mentioned the need for some kind of extrapolation in order to discover inductive invariants in loops, and we called this operation "widening". It is specific to each domain and appears on line 7. Widening is used for guessing candidate invariants. These don't need to be trusted, since the analyser checks whether they are inductive. As a result, no proof obligation applies to the widening operator.

Guard operator. The control flow of a program is based on conditions, most commonly **if** statements, which must hold for the guarded instructions to be executed. Filtering program states satisfying a guard is performed by the assume operator, on line 9 of listing 1.1. The call assume c b p abstracts the program states satisfying abstract value p where expression c evaluates to boolean b. Expression c can be any side-effect-free expression from the C programming language. Note that there is no boolean type in the C language. Boolean expressions, such as x <= y, yield an integer value: 1 if the comparison evaluates to "true" and 0 otherwise, as you can see in the proof obligation on line 18. The proof obligation for the guard, noted $\sqcap$, in the case where boolean b is true is the following.

$$\forall m,\, m \in \gamma\, p \,\wedge\, m \in \gamma\, c \,\Rightarrow\, m \in \gamma\, (p \sqcap c)$$

I abused notations in this formula: $m \in \gamma\, c$ is the set of memory states making expression c evaluate to "true". The proof obligation shows that operator $\sqcap$ is an approximation of conjunction.

Assignment operator. A memory state is altered by variable assignments. The result of assign x e p represents the effect of assignment x = e on abstract value p. The corresponding proof obligation is found on line 20. Spelt with usual mathematical notations, it reads as follows.

$$\forall m \in \gamma\, p,\ m[x := \mathrm{eval}(m, e)] \in \gamma\,(p[x := e])$$

Two program states are involved: state m before executing the assignment and state $m' \triangleq m[x := \mathrm{eval}(m, e)]$ resulting from the assignment. The latter associates to x the value of expression e evaluated in m and $m\,y$ to any other variable y. The proof obligation states that memory m' belong to the memory states abstracted by the result $p[x := e]$ of assign x e p.

Projection operator. We haven't encountered operator forget yet. Assume variable x is local to a function f of the program under analysis. Keeping information about variable x makes little sense, once the analysis of function f is over. Only the effect of function f is interesting, that is its side effects and return value. Therefore, the analyser forgets about variable x when it becomes out of scope, but keeps the rest of the information it gathered. Variable x is said to be "eliminated". This operation is noted $p\backslash x$, where p is an abstract value. Line 22 gives the proof obligation.

$$\forall m \in \gamma\, p,\ \forall n \in \mathbb{Z},\ m[x := n] \in \gamma\,(p\backslash x)$$

It states that abstract value p doesn't constrain the value of x. Operator forget is also called "projection": looking at abstract value p as a geometrical object which has one dimension per variable, eliminating variable x is tantamount to projecting abstract value p in a space of lower dimension, where variable x has been removed. From this perspective, the proof obligation can be recast equivalently as the following existential quantification.

$$\forall m, (\exists n \in \mathbb{Z},\ m[x := n] \in \gamma\, p) \Rightarrow m \in \gamma\,(p\backslash x)$$

Soundness versus precision. The formal proof of Verasco focuses on soundness. It is possible to prove precision properties, as the result of the operators of the usual domains are well-defined. However, in an attempt to lower the proof effort, it doesn't provide any guarantee on the precision of the analysis. The downside of this choice is that an abstract domain where id_leb always returns false and where all of the other operators return id_top trivially meets the specification described in listing 1.1, even if such an abstract domain wouldn't be of any use.

All Verasco abstract domains meet the specification reproduced in listing 1.1. We may now turn to how this is achieved for the abstract domain of polyhedra.

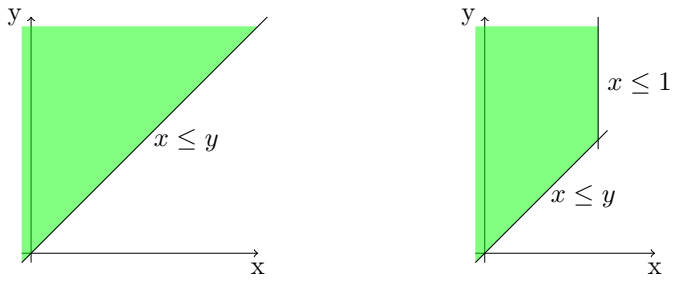


Figure 1.1 – Convex polyhedron as the intersection of half-spaces

1.2 Convex polyhedra and their representations

The code fragment on listing 4 revealed that capturing relations between program variables is sometimes necessary, in order to infer useful information. The abstract domain of convex polyhedra can represent linear equality or inequality relations between variables, such as $x \leq y$ or $x = y + 1$. These are very commonly encountered in programs.

1.2.1 Geometrical view

Convex polyhedra are used in static analysis for representing collections of reachable memory states. They are more commonly associated with the field of geometry and it is often useful to keep a geometrical view in mind.

Suppose that the program under analysis has n variables, or that the program fragment under consideration has n live variables. Let me name them x_1 to x_n. To each of these n variables, we associate one dimension of an n-dimensional space: $x_1, x_2, \ldots, x_n$. Then, each point in this space represents a memory state: the value of the i^{th} coordinate of point $m \triangleq [x_1 \mapsto \mathsf{x}_1, \ldots, x_n \mapsto \mathsf{x}_n]$ is the value associated to the i^{th} variable of the program.

Consider two program variables x_1 and x_2 mapped to dimensions x_1 and x_2 of a two-dimensional space. Both x_1 and x_2 are rational numbers. Then, a guard, such as $\mathsf{x}_1 <= \mathsf{x}_2$, defines a half-space, as illustrated on the left of figure 1.1. The aforementioned guard is satisfied by all the points above the line $x_1 = x_2$.

If another guard, such as the bound $\mathsf{x}_1 <= 1$, were encountered in the program fragment guarded by $\mathsf{x}_1 <= \mathsf{x}_2$, all the satisfying states would be in the intersection of the two half-spaces, as shown on the right of figure 1.1.

1.2.2 Constraints and generators

The example of figure 1.1 describes a convex polyhedron as a conjunction of half-spaces. There are actually two ways of defining a convex polyhedron: constraints and generators. All the numbers mentioned in the following description, both constants and variables, are rational numbers. We consider an n-dimensional space, with dimensions x_1 to x_n, and note $x \triangleq (x_1, \ldots, x_n)$ a point in this space. Let us call $c(x)$ an *affine* function if it can be written in the following way.

$$c(x) = a_0 + \sum_{j=1}^{n} a_j \cdot x_j$$

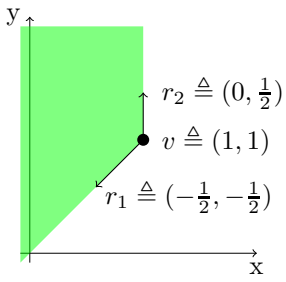


Figure 1.2 – The generators of a convex polyhedron

If $a_0 = 0$, then we call $c(x)$ a *linear* function.

Constraint representation

Suppose l affine functions $c_1(x)$ to $c_l(x)$. A convex polyhedron P is defined as the following set of points.

$$P \triangleq \left\{ x \,\middle|\, \bigwedge_{i=1}^{l} c_i(x) \geq 0 \right\}$$

Each conjunct $c_i(x) \geq 0$ is called an affine constraint. An abstract value p of the abstract domain of polyhedra is a set of such affine constraints.

$$p \triangleq \{ c_i(x) \geq 0,\, i \in \{1, \ldots, l\} \}$$

It is a representation of all the points of set P defined above. Sets P and p are two ways of looking at the same thing. They are related by the concretization function of the abstract domain of polyhedra: $P = \gamma\, p$. Therefore, we can think of set p as being the abstraction of the set P of memory states. We will use upper case P for the set of states and lower case p for the set of constraints.

It is possible for a polyhedron to contain two constraints, such as $x \geq 0$ and $x \leq 0$. Their conjunction effectively states that $x = 0$. For reasons mainly related to computational efficiency, it is desirable to set these equalities aside in the definition of a polyhedron. Suppose k affine functions $e_1(x)$ to $e_k(x)$. A polyhedron p can be defined as a pair a sets.

$$p \triangleq \Big(\{ c_i(x) \geq 0,\, i \in \{1, \ldots, l\} \}, \{ e_i(x) = 0,\, i \in \{1, \ldots, k\} \} \Big)$$

The corresponding set of states P is defined in the following way.

$$P \triangleq \left\{ x \,\middle|\, \bigwedge_{i=1}^{l} c_i(x) \geq 0 \wedge \bigwedge_{i=1}^{k} e_i(x) = 0 \right\}$$

Generator representation

Constraints appear naturally in source code and call for a constraint representation of polyhedra. However, a convex polyhedron can also be represented by

a set of generators, which is better suited for performing some operations, such a projection. As the name implies, generators describe how to build the points of the polyhedron. For example, for any point x of the polyhedron depicted on figure 1.2, there exist $\lambda_1 \geq 0$ and $\lambda_2 \geq 0$ such that $x = v + \lambda_1 \cdot r_1 + \lambda_2 \cdot r_2$. Point v is called a *vertex* and r_1 and r_2 are called *rays*. Suppose that polyhedron P has l vertices and k rays. It can be defined in the following way.

$$P \triangleq \Big\{ \sum_{j=1}^{l} \mu_j \cdot v_j + \sum_{i=1}^{k} \lambda_i \cdot r_i \;\Big|\; \sum_{j=1}^{l} \mu_j = 1 \wedge \bigwedge_{j=1}^{l} \mu_j \geq 0 \wedge \bigwedge_{i=1}^{k} \lambda_i \geq 0 \Big\}$$

Each point of set P is built from a convex combination of its vertices and a nonnegative linear combination of its rays. Similarly to the constraint representation, an equivalent polyhedron p can be defined as a pair of sets.

$$p \triangleq \big(\{v_1, \ldots, v_l\}, \{r_1, \ldots, r_k\} \big)$$

When a polyhedron has two rays r_1 and r_2 pointing to opposite directions, that is when $r_1 = -k \cdot r_2$ with $k > 0$, they can be merged into a line. The definition of polyhedron P with m lines l_1 to l_m is altered in the following way.

$$P \triangleq \Big\{ \sum_{j=1}^{l} \mu_j \cdot v_j + \sum_{i=1}^{k} \lambda_i \cdot r_i + \sum_{i=1}^{m} a_i \cdot l_i \;\Big|\; \sum_{j=1}^{l} \mu_j = 1 \wedge \bigwedge_{j=1}^{l} \mu_j \geq 0 \wedge \bigwedge_{i=1}^{k} \lambda_i \geq 0 \Big\}$$

The polyhedra which can be represented using the representations we have just described are always convex. Therefore, I'll just refer to them as "polyhedra" from now on.

1.2.3 Which representation should be used?

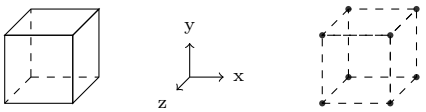
The abstract domain operators transform polyhedra so that they abstract the set of states reachable after each instruction. The complexity of the algorithms used in these operators depends on the representation of the polyhedra. For each operator, one choice of representation results in a very simple algorithm for computing the result. In the following enumeration, p and p' are two polyhedra.

inclusion test. To check whether $p \sqsubseteq p'$, it is best to have p as generators and p' as constraints. Inclusion holds if all the generators of p satisfy the constraints of p'.

join. The join $p \sqcup p'$ performs the convex hull of polyhedra p and p'. It is easiest with generators: just compute the union of the sets of generators.

widening. The standard algorithm, which is described in Nicolas Halbwachs's PhD thesis [30] operates on constraint representation. Essentially, it drops constraints whose constant term changes from one iteration to the next.

guard. With an affine guard c, which is a single-constraint polyhedron, computing $p \sqcap c$ is easiest when polyhedron p is in constraint representation. Operator $\sqcap$ computes the intersection of two polyhedra, by taking the union of their constraint set.



6 $(= 2 * 3)$ constraints 8 $(= 2^3)$ generators

Figure 1.3 – The two representations of the three-dimensional hypercube

assignment. The effect of assignment $x := e$, where e is an affine expression, is computed by applying a linear transformation in generator representation. With constraints, the assignment operator can be rephrased in terms of the guard and projection operators, as we'll see later in this chapter.

projection. Eliminating variable x from polyhedron p is easiest with generators. The result $p\backslash x$ is computed by dropping the x coordinate of all of the generators.

Representation size

Another criterion for choosing between representation is their size. One representation of a polyhedron can be exponentially bigger than the other, as illustrated on figure 1.3. The hypercube captures bounds on each variables, with no relational information. In our static analysis setting, this is a common situation. The size of the constraint representation of the hypercube is linear in the number of variables: each variable has one upper bound and one lower bound. The size of the generator representation is exponential in the number of variables. Similarly, there is a class of polyhedra with a linear number of generators and exponential number of constraints, but it is much less likely encountered.

The state of the art: using both

The previous discussion naturally leads to consider using both representations. This is what the original work by Patrick Cousot and Nicolas Halbwachs [17] actually does. Nevertheless, using both representations comes at the cost of converting between representations. Chernikova's algorithm [12] solves this problem. Unfortunately, it is an expensive algorithm: the complexity of polyhedra algorithmics is transferred from the operators to the conversion algorithm. This being said, in the best case, conversions are amortised over several operator calls. This happens when analysing nested guards, or when performing a join after the local variables in each branch have been eliminated.

State-of-the-art implementations continue to follow this design. These implementations include PolyLib [52], NewPolka, from the Apron [33] collection of abstract domains, and the Parma Polyhedra Library [3], called PPL in the remainder of this text.

1.3 Designing the proof

Having reviewed how polyhedra are represented, we may go back to implementing the specification presented in section 1.1, along with formalising the required proofs.

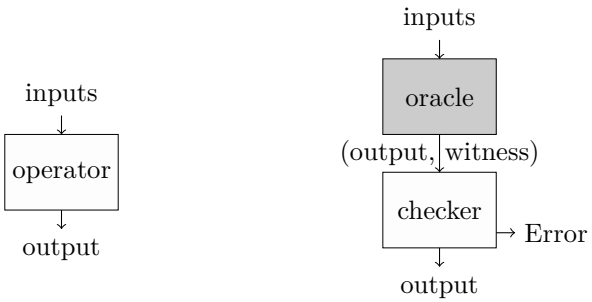


Figure 1.4 – Two approaches to proving correctness in using Coq. The trusted components are lightly shaded. The untrusted ones appear darker.

1.3.1 Proof approaches

There are two main approaches to proving the correctness of a function in Coq. Figure 1.4 illustrates them.

- Either you prove that whatever result the function returns, it fulfills the specification,
- or you write a checker which, for each result produced by the function, verifies whether the result is correct. In this case, you only need to prove that if the checker accepts a result, then the result is correct. The checker may use a *witness* of correctness, which is provided by the function.

Ultimately, what matters is that results are trustworthy. Both methods meet this goal. The first approach, proving the correctness of the implementation of a function, is the most straightforward approach: the proof is done once and afterwards, the function can be forever used and trusted. However, when the algorithms used are complex, the proof may become hard and must be maintained as the algorithm evolves. The alternative, called "result verification", may be a lot simpler in this situation. For example, the Compcert C compiler uses this technique for register allocation, which can be cast as a graph colouring problem. Proving that a checker accepts only properly coloured graphs is much easier than proving the correctness of the colouring algorithm itself. Furthermore, the algorithm may be enhanced without affecting the proofs. However, there are a few drawbacks to result verification.

- Each result needs to be checked at runtime. If checking is expensive, this approach incurs a performance penalty. However, writing correctness proofs in Coq is hard. One may choose to implement and prove correct simple algorithms, instead of elaborate, and more efficient, algorithms.
- Checking results adds the possibility of a result being incorrect, which must be dealt with in all the Coq development. This is the "Error" output on figure 1.4.

Also note that a checker which always returns an error is correct, although useless. This being said, verifying results means that no restrictions apply to the function implementation. It can be implemented as an external Ocaml

oracle, to which most of the complexity is offloaded. This is beneficial for the following reasons.

- Coq code cannot use machine arithmetic. Instead, it is limited to a formalisation of numbers as bit strings. For arithmetic intensive algorithms, this becomes problematic for performance.
- All Coq functions must be shown to terminate for all inputs. For some algorithms, such as the simplex algorithm which is widely used in the implementation of the abstract domain of polyhedra, proving termination is tricky. The burden of the termination requirement may be lifted by resorting to *fuel*, which consists in bounding statically the number of recursive calls by a very large number. When the algorithm exhausts its fuel, it returns an error, which brings us back to handling errors throughout the code.

Choosing result verification

The guiding goal for this work was to keep things as simple as possible. As we will see in the next two chapters, the algorithms used for computing over polyhedra are already complex enough. Adding the proof requirement to their implementation would have made them even more complex.

For this reason, result verification was chosen for writing the soundness proof of the abstract domain of polyhedra in Coq. This choice reduces the proof burden and separates concerns between computing and proving. The arguments required to prove the correctness of the operators of the abstract domain of polyhedra make it convenient to offload much computation to an untrusted oracle and keep only a small amount of code to be proved correct in Coq.

1.3.2 Farkas's lemma

As we'll see shortly, the proof obligation for all the operators of the abstract domain, which we saw at the beginning of this chapter, can be recast as inclusion properties, with the exception of the guard operator. Farkas's lemma provides us with a witness for inclusion, when dealing with polyhedra under constraint representation.

Farkas's lemma. *A polyhedron* $P \triangleq \left\{x \,\middle|\, \bigwedge_{i=1}^{l} c_i(x) \geq 0\right\}$ *is included in the half-space* $C \triangleq \{x \,|\, c(x) \geq 0\}$ *if and only if there exists* $\lambda_0 \geq 0, \ldots, \lambda_l \geq 0$ *such that* $c(x) = \lambda_0 + \sum_{i=1}^{l} \lambda_i \cdot c_i(x)$.

In the context of result verification, only one part of the proof needs to be written in Coq: if $\lambda_0, \ldots, \lambda_l$ are provided such that the equality $c(x) = \lambda_0 + \sum_{i=1}^{l} \lambda_i \cdot c_i(x)$ holds, then inclusion $P \sqsubseteq C$ holds. The proof relies on two helper lemmas.

scaling. $\forall a \geq 0, \forall x,\ c(x) \geq 0 \ \Rightarrow\ a \cdot c(x) \geq 0$

addition. $\forall x,\ c_1(x) \geq 0 \wedge c_2(x) \geq 0 \ \Rightarrow\ c_1(x) + c_2(x) \geq 0$

The linear combination is represented as the list of pairs (constraint identifier, coefficient). The main proof is no more than an simple induction on this list,

with each step invoking the two lemmas above. If the coefficient is negative, or if the identifier refers to a nonexisting constraint, an error is returned. With the help of some proof automation, the proof of this half of Farkas's lemma in the Coq development of VPL is two lines long.

The guarantee that the linear combination always exists whenever inclusion holds is the difficult part of the proof [18], but it isn't required for proving the checker correct. This half of Farkas's lemma merely makes us confident that building a complete inclusion test with result verification is possible. Furthermore, coefficients $\lambda_0, \ldots, \lambda_l$ make a witness for inclusion. The result of each operator can be verified by applying Farkas's lemma.

inclusion test. This is a direct application of the lemma.

join. As we saw at the beginning of this chapter, the proof obligation for the join operator is as follows.

$$\forall m,\ m \in \gamma\, p \ \vee\ m \in \gamma\, p' \ \Rightarrow\ m \in \gamma\,(p \sqcup p')$$

In order for $p_h \triangleq p \sqcup p'$ to meet this specification, case analysis requires proving that $p \sqsubseteq p_h$ and $p' \sqsubseteq p_h$. Providing a witness for each inclusion allows a checker to verify that the result is correct.

guard. If I continue abusing notation γ, the proof obligation for the guard operator as follows.

$$\forall m,\ m \in \gamma\, p \ \wedge\ m \in \gamma\, c \ \Rightarrow\ m \in \gamma\,(p \sqcap c)$$

Let $p' \triangleq\ p \sqcap c(x) \geq 0$. Polyhedron p' is correct if $p \cup \{c(x) \geq 0\} \sqsubseteq p'$. Providing, for each constraint of polyhedron p', a witness of the form $\lambda_0, \ldots, \lambda_l, \lambda_c$, where $\lambda_1, \ldots, \lambda_l$ apply to the constraints of p and λ_c applies to $c(x) \geq 0$, suffices to verify correctness. Note that $p \cup \{c(x) \geq 0\}$ is a trivially correct result of $p \sqcap c(x) \geq 0$, with an obvious inclusion witness. However, an actual implementation of the guard operator minimises the representation of the result, that is redundant constraints are removed and equality constraints may be used for rewriting.

projection. Remember from the beginning of this chapter that the specification of the projection operator is the following.

$$\forall m \in \gamma\, p,\ \forall n \in \mathbb{Z},\ m[x := n] \in \gamma\,(p \backslash x)$$

Polyhedron $p' \triangleq\ p \backslash x$ is correct if x is not bounded by p' and $p \sqsubseteq p'$. The former can be checked syntactically. The latter is an inclusion test.

A result similar to Farkas's lemma holds for the generator representation. Inclusion $p \sqsubseteq p'$ is proved if all the vertices of p are points of p' and if all the rays of p are linear combinations with positive coefficients of the rays of p'.

1.3.3 Choosing constraint representation

I chose a constraint-only representation to build an abstract domain of polyhedra along with its soundness proof. Let us review the motivations behind this choice.

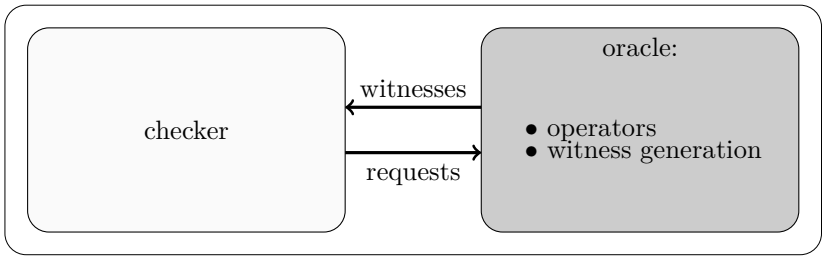


Figure 1.5 – the architecture of the abstract domain

- To the best of my knowledge, there is no algorithm computing intersection on generators. For constraint representation, the problematic operator is join, but an algorithm has been published [5] which encodes the join of two polyhedra represented by constraints as a projection.
- As the analysis of listing 4 shows, constraints appear naturally during the abstract interpretation of a program. Even if there was a simple way to convert the representation of a half-space from a single constraint to generators, this would reduce the issue to computing the intersection of two polyhedra represented by generators.
- Discovering bounds on variables during analysis is common and the polyhedron which represents the resulting product of intervals has an exponential number of generators.
- Using two representations requires proving the correctness of the results of Chernikova's algorithm. There is no known way to the result checking approach. Furthermore, proving the algorithm correct is hard: forgetting one generator results in an underapproximation, which is unsound. Therefore, proving the correctness of the algorithm requires proving a completeness result.

1.4 The core abstract domain

At this point, we have chosen to use the constraint representation of polyhedra in order to formalize the soundness proof of the abstract domain. We have also decided to build the proof using the result verification approach. Therefore, we end up with the architecture pictured on figure 1.5 for the abstract domain. It has two components: an untrusted oracle and a trusted checker.

The oracle. Most of the algorithmic complexity is delegated to an oracle. It is written in the OCaml programming language and the results it computes don't need to be trusted. However, it provides, along with its results, inclusion witnesses in the form of nonnegative coefficients of linear combinations. These coefficients are used by the checker to apply Farkas's lemma in order to prove the correctness of the result.

The checker. The result checker is part of the Coq *frontend* of the abstract domain, which we will detail in the next sections. It works by delegating computations on polyhedra to the oracle and by then checking the

```
1  Axiom t: Set.
2
3  Axiom freshId: t → positive.
4  Axiom top: t.
5  Axiom isEmpty: t → option witness.
6  Axiom isIncl: t → t → option witness.
7  Axiom guard: t → Cstr.t → option t * witness.
8  Axiom join: t → t → t * witness.
9  Axiom widen: t → t → t * Pol.t.
10 Axiom project: t → Var.t → t * witness.
11 Axiom rename: Var.t → Var.t * t → t.
```

Listing 1.2 – the axioms declaring the oracle in Coq

results. Checking relies on the witness which comes with a result in order to apply Farkas's lemma to prove the required inclusion property. The result checker is written in Coq and comes with a proof that the results it returns to the user of the abstract domain are sound. In other words, it meets the specification from section 1.1.

Result verification is a common approach to building proofs. However, it is put to practice in VPL in a slightly unusual way. In VPL, you may think of the oracle and the checker as two independent entities which are able to communicate. Each of them independently stores a current polyhedron, that is the polyhedron which captures the reachable states at the program location which the analyser is currently looking at. When the analyser calls an operator of VPL frontend, the frontend calls one or several oracle operators. The oracle operators update the oracle current polyhedron and send back witnesses to the frontend. These witnesses can been seen as instructions for the frontend to update its own current polyhedron so that it matches that of the oracle, thereby restoring synchronisation. Once the frontend has updated its own polyhedron, it returns it to the analyser as the operator result.

One interesting feature of this design is that the oracle has complete freedom for how the operators of the abstract domain are implemented. It just needs to provide the required inclusion witnesses. In particular, it would be possible for the oracle to implement a restriction of the abstract domain of polyhedra. For example, it could implement the octagon abstract domain [42], which restricts the coefficients of variables in the constraints $c(x) \geq 0$ to be either 1 or -1 and limits the number of non-zero coefficients to be at most two. Another interest of the design is that it allows the oracle to use different data structures from the frontend. This is particularly beneficial for numbers, which are represented as lists of bits in Coq: a more efficient representation is used in the oracle.

We will now describe how this high-level view of the design is actually implemented, focusing on the communication between the oracle and the checker.

1.4.1 Axiomatising

As we noted above, the frontend is written in Coq, while the oracle is written in OCaml. In order to have the frontend interact with the oracle, the frontend

```
1  Definition project (pF, pO) x :=
2    let (pO', w) := Oracle.project pO x in
3    let pF' := projectUsing w pF x in
4    (pF', pO').
```

Listing 1.3 – the Coq definition of the projection operator

needs to have a name for it. External Ocaml code can be declared to Coq, so that it can be referred to from a Coq program. These declarations are called *axioms*, as they make Coq assume the existence of an external value or function of a given type.

Listing 1.2 presents the axioms declaring the functions of the oracle to Coq. Types Cstr.t, Pol.t and Var.t are the Coq types for constraints, polyhedra and variables in the frontend. You can find an axiom for each operator of the abstract domain: the default element top, on line 4, the inclusion test isIncl on line 6, the guard, join, widening and projection operators on lines 7, 8, 9 and 10 respectively. The assignment operator is missing. As I mentioned before, it is implemented in the frontend in terms of the guard and projection operators. Simple variable renaming is also needed for assignment, which is the reason for an extra axiom to appear on line 11. Note that renaming in the frontend doesn't involve the oracle, but the frontend needs to notify the oracle that a renaming has been performed, in order to preserve synchronisation. Axiom isEmpty, on line 5, is a special case of isIncl. It checks whether there exists at least one point satisfying the constraints of a given polyhedron. Both isEmpty and isIncl have return type option witness. Type transformer option adds a special value None to type witness. When the polyhedron is empty, or when inclusion holds, the oracle returns a witness. Otherwise, it returns the special value None.

In order to avoid communicating constraints between the frontend and the oracle, unique identifiers are used to refer to them. The allocation of such identifier is handled by the oracle. As a result, the frontend calls freshId when it creates a new constraint.

Last, notice that opaque data types can be declared as axioms. Line 1 of listing 1.2 declares the data type used by the oracle to represent polyhedra. It is left opaque for Coq: a Coq program can only manipulate it through the functions declared for the oracle.

Assuming that the axioms declared in listing 1.2 are in module Oracle, listing 1.3 shows how the frontend can call the oracle. The operator taken as example is the projection operator. It takes as input the variable x to eliminate and the input polyhedron (pF, pO). The latter is a pair composed of the frontend polyhedron pF and the oracle polyhedron pO. Next section provides further details on this point. The important point for now is that pO is an element of the opaque data type Oracle.t, on which Oracle.project can be called. The returned values, the result polyhedron pO' and the witness w, can then be used like any other expression from Coq.

1.4.2 The extractor

Once a Coq function is defined, it can be evaluated from within Coq. This is the most straightforward way to execute Coq code. However, axioms, such as those

```
1  let project (pF, pO) x =
2    let (pO', w) = Oracle.project pO x in
3    let pF' = projectUsing w pF x in
4    (pF', pO')
```

Listing 1.4 – the OCaml extracted code for the projection operator

found on listing 1.2, only *declare* external functions to Coq. These declarations don't make the implementation of the axioms available. As a result, Coq code which relies on axioms can't be executed within Coq. Nevertheless, if there is an OCaml implementation of the axioms, the Coq code can be extracted to OCaml and, once linked with the implementation of the axioms, makes a complete program.

We mentioned the extractor before: it generates an OCaml program from Coq code. The extractor removes the proof-related information from a Coq development and performs some simplifications. During extraction, axioms are translated to regular function calls.

Listing 1.4 contains the OCaml code generated by the extractor for function project from listing 1.3. As you can see, keywords and some elements of syntax differ between the Coq and OCaml versions, but the structure remains the same. Translation is obvious in this case since this code fragment doesn't contain any proof-related information. Once the code on listing 1.4 is compiled and linked with module Oracle, you get an executable implementation of the projection operator.

1.4.3 The communication protocol

Three VPL operators use a witness from the oracle and produce a polyhedron: the guard, join and projection operators. Result verification leads naturally to a pattern of algorithms for the frontend, which we illustrate for the projection operator, applied on polyhedron p and variable x. The oracle returns the resulting polyhedron p', with an inclusion witness $\Lambda_i \triangleq (\lambda_{i0}, \ldots, \lambda_{il})$ for each constraint $c'_i(x) \geq 0$ of p'. We call $\Lambda \triangleq (\Lambda_1, \ldots, \Lambda_{l'})$ a witness for the whole inclusion $p \sqsubseteq p'$. For each $c'_i(x) \geq 0$ of these constraints, the checker builds affine constraint $c''_i(x) \geq 0$ using Λ_i.

$$c''_i(x) = \lambda_{i0} + \sum_{j=1}^{l} \lambda_{ij} \cdot c_j(x)$$

Then, it checks whether $c'_i(x) \geq 0$ and $c''_i(x) \geq 0$ are syntactically equal. An extra check is specific to the projection: verifying that x has a zero coefficient in $c'_i(x) \geq 0$.

Avoiding conversions. From the witness it gets from the oracle, the checker builds a polyhedron p'' and then should check whether p' and p'' are syntactically equal. We realised that, checking syntactic equality is actually unnecessary. Polyhedron p'' can be used as a result of the projection operator, as it satisfies the inclusion property, by construction. On top of sparing the equality check,

this approach removes the need for the oracle to communicate its result to the frontend: the witnesses are sufficient. The amount of data conversion between the oracle and the checker is therefore decreased. This remark applies to the projection operator, as well as to the guard and join operators. As a result, the operators follow a simpler pattern, illustrated for the projection operator on listing 1.3.

Synchronisation between frontend and oracle. The high-level view of two communicating components pictured on figure 1.5 is implemented as follows. As I mentioned at the beginning of this section, the frontend and the oracle each have their own data type for polyhedra. For the frontend, this is type Pol.t, which is transparent to the frontend: it can access the details of a value of type Pol.t. For the oracle, this is type Oracle.t, which is opaque to the frontend. A VPL polyhedron is a pair (pF, pO), where pF is a frontend polyhedron and pO is an oracle polyhedron. VPL maintains the invariant property that they represent the same geometrical polyhedron.

We have seen that the frontend and the oracle communicate through function calls. An operator of the abstract domain consists in calling the corresponding operator of the oracle, thereby obtaining the oracle version pO' of the resulting polyhedron. The oracle also produces a witness w, from which the frontend computes its version pF' of the result of the operator, along with a proof that it is correct. This restores the synchronisation between the frontend and oracle: pF' and pO' represent the same polyhedron and the pair (pF', pO') is the resulting VPL polyhedron.

The impact of bugs. The previous discussion makes the assumption that all goes well: witnesses are well-formed and yield a representation of the result computed by the oracle. However, bugs might lurk in the oracle, leading to incorrect results or erroneous witnesses. Two possible effects can be observed by the user of the abstract domain.

- If a witness is well-formed but yields a result different from that of the oracle, synchronisation is lost and the results built by the abstract domain are likely to be wildly overapproximated, yet correct.
- If an ill-formed witness—it may refer to nonexistent constraints—is produced by the oracle, the checker will report a failure. In this case, the result of the operator will be the always-correct default: top.

The frontend returns correct results in all cases: soundness bugs in the oracle can only induce precision bugs of the abstract domain. These bugs are uncovered using standard software engineering methods.

The witness language. The frontend builds correct-by-construction results using witnesses provided by the oracle. The data type witness for witnesses is given in listing 1.5. We will describe the design of the witnesses from the ground up on the example of a projection $p\backslash x$ for which the oracle has produced a witness Implies I.

In order to make the witnesses compact, the constraints of polyhedron p are identified by positive numbers. The descriptions of linear combinations—the type linComb—refer to constraints by their identifier.

```
1  Inductive witness :=
2  | Implies : list (positive * consWitness) → witness
3  | Empty : linComb → witness
4  | Bind : positive → consWitness → witness → witness.
5
6  Inductive consWitness :=
7  | Direct : linComb → consWitness
8  | SplitEq : linComb → linComb → consWitness
9  | JoinCons : linComb → linComb → consWitness.
```

Listing 1.5 – Coq definition of polyhedron build instructions

Data type consWitness describes the various ways to build one constraint of $p \backslash x$. The Direct construct is the standard application of Farkas's lemma. For efficiency reasons, an oracle may handle equality constraints specially, instead of representing them as pairs of inequalities. Two applications of Farkas's lemma are necessary to build an equality $c(x) = 0$ from polyhedron p. One builds $c(x) \geq 0$ and the other builds $-c(x) \geq 0$. The equality follows from their conjunction and SplitEq construct was introduced to handle this case.

The join operator requires a special construct, JoinCons. For each constraint $c(x) \geq 0$ of the result of $p_1 \sqcup p_2$, it must be shown that $p_1 \sqsubseteq c(x) \geq 0$ and $p_2 \sqsubseteq c(x) \geq 0$. To this end, a JoinCons witness contains one linear combination to build $c_1(x) \geq 0$ such that $p_1 \sqsubseteq c_1(x) \geq 0$ and another for $c_2(x) \geq 0$ such that $p_2 \sqsubseteq c_2(x) \geq 0$. The frontend checks whether $c_1(x) = c_2(x)$. If they are equal, they make a sound constraint of $p_1 \sqcup p_2$. Otherwise, the witness is ill-formed.

Type witness also provides a construct to build $\bot$, the polyhedron $P = \emptyset$. It can be the result of a guard, when the guard code fragment is dead code for example. An Empty l witness is used for this purpose, where the linear combination l yields a trivially contradictory constraint, such as $-1 \geq 0$.

Last, the Bind constructor of type witness was introduced to make witness generation easier. It allows to introduce a temporary constraints. It could also be used to factorise computations performed by the checker. This possibility hasn't been investigated yet, since the checker isn't a performance bottleneck.

1.5 Modular proof formalisation

What the last section describes isn't sufficient to meet the specification presented at the beginning of the chapter. The assignment operator is missing, for example. The missing functionality is provided as *functors*, entirely in the Coq frontend, building on what we have seen so far.

Functors in Coq. Coq provides modules to structure programs. These modules are similar to the modules found in Ocaml or the packages found in Java. It is also possible to write module types. Modules of a given module type are required to provide, or export, a number of functions, data types and lemmas whose name and type are fixed by the module type. In other words, a module type is an interface for modules to implement. Once the interface is fixed, code

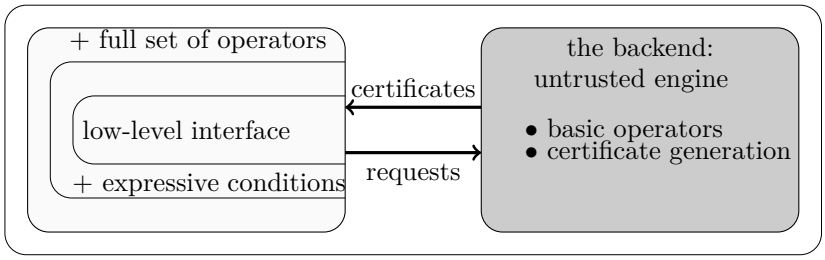


Figure 1.6 – the complete architecture of the abstract domain

using an implementation of the interface doesn't need to know the details of the implementation. A functor is similar to a function, except that it applies to modules: it takes modules as parameters and returns another module of a give module type.

The simplest functor of the abstract domain lifts an abstract domain where variables have rational values to an abstract domain where they have integer values. Dealing precisely with integers is more complex than dealing with rationals and requires the algorithms to be adapted. The functor only proves that considering integer variables as rationals is a sound overapproximation. As a result, most of the Coq development in VPL operates on rationals.

Keeping the proof manageable with functors. The complete architecture of the abstract domain is depicted on figure 1.6 and makes heavy use of functors. The shaded left-hand side is the Coq frontend. Each of the pictured layers represents a functor. The untrusted oracle stands on the right-hand side. While communication between the two is represented by arrows, remember that it reduces to function calls in the extracted frontend code.

Each functor enriches a restricted abstract domain, while lifting the operators and the proofs as necessary. This modular decomposition makes the proofs more manageable, as they only rely on well-defined interfaces, rather than on implementation details. In turn, this, results in easier proof automation.

1.5.1 The guard operator

The guard operator from the specification in section 1.1 is also provided using a functor. The fully-fledged guard operator takes side-effect-free C expressions, while previous section discussed affine constraints $c(x) \geq 0$. The functor enhances the guard operator $p \sqcap f$ so that it handles an arbitrary propositional formula as operand f, where atoms are affine constraints. The transformation from the more expressive guards to the basic guards is performed by the frontend through the following steps.

1. Negations are pushed toward the atoms using de Morgan's laws on binary operators. Double negations are eliminated. Negated comparisons are complemented: atom $\neg(c(x) \geq 0)$ is rewritten $c(x) < 0$, for example.

2. Comparison $\neq$ is rewritten as a disjunction of strict inequalities.

3. Disjunctions are overapproximated by joins.

4. The constraint $c(x) > 0$ on integers is rewritten as constraint $c(x)+1 \geq 0$. This increases precision of polyhedra computations, where all variables are rationals.

For a guard $p \sqcap f$, this algorithm performs a number of polyhedra operations that is linear in the number of operations in formula f. The functor which provides this extended guard operator to an abstract domain featuring only a basic one also contains the proof that the algorithm described above is sound.

1.5.2 Assignment

The oracle has an assignment operator, so that it is a complete OCAML abstract domain of polyhedra. However it doesn't export it to the frontend. Instead, support for assignment is built from the other operators.

Building an assignment operator. Suppose that we need to compute the effect on polyhedron p of variable x being assigned the value of expression e, that is $p\,[x := e]$. Without more information, it may well be that expression e mentions variable x. In other words, it is possible that the new value of variable x depends on its value before the assignment. In order to be able to refer to both the old value and the new value of variable x, let us introduce a fresh variable x' to represent the new value of variable x. "Fresh" means that variable x' appears neither in the constraints of polyhedron p, nor in expression e. Variable x still represents its value prior to the assignment.

The assignment is rewritten $p\,[x' := e]$. Since variable x' is fresh, this is equivalent to the guard $p \sqcap x' = e$. After the assignment, information on the old value x isn't relevant any more. Therefore, variable x is eliminated from polyhedron $p \sqcap x' = e$ through projection, yielding $(p \sqcap x' = e)\backslash x$.

All of this is internal to the assignment operator. Outside of it, the new value of variable x is called x. A renaming of x' into x, noted $p_2\,[x'/x]$, is performed. This renaming is safe since x is fresh after the projection. The assignment operator is thus defined in the following way.

$$p[x := e] \triangleq \big((p \sqcap x' = e)\backslash x\big)[x'/x]$$

Variable x' is only auxiliary: it doesn't appear in the resulting polyhedron.

The soundness proof. The soundness proof is trickier for the assignment operator than for the other operators. The major difference is that the soundness proof of the assignment operator refers to two memory states, the state before the assignment and the state after, instead of one for the other operators. The direct consequence, which appears above, is the need to generate fresh variables.

1.5.3 Framing constrained variables

Besides handling assignment, generating fresh variables has many applications for program verification, such as handling local variables or parameter passing during function inlining. However, the frontend of the abstract domain represents a polyhedron as the list of its constraints and the specifications don't provide any information about the set of variables it constrains. Loose specifications allow a modular management of fresh variables: the proofs of the core

abstract domain, which are described in section 1.4, aren't complicated with fresh variables handling. Support for *framing* the variables constrained by a polyhedron is added through a functor.

In Verasco and, consequently, in VPL, variables are represented by positive integers. Framing the variables constrained by an abstract value p consists in determining an integer β such that all the variables outside of the range $[1, \beta]$ aren't constrained by p. In other words, it is safe to allocate fresh variables above the bound β.

The framing functor, which I'll refer to as $\mathbb{P}^f$, enriches the data structure representing polyhedra with the upper bound β on the constrained variables. Invariant properties of this extended representation is expressed in the definition of the concretization function γ, which links an abstract value to the set of reachable states it represents.

A memory state is a function from variables to their value. Given a set of variables F, I'll write $m \equiv_F m'$ to mean that the two states m and m' associate the same value to each variable of set F.

$$m \equiv_F m' \triangleq \forall x \in F,\ m\ x = m'\ x$$

Then, we say that the set F *frames* a polyhedron p if the following property holds.

$$\forall m, \forall m',\ m \equiv_F m' \ \Rightarrow\ (m \in \gamma\ p \Leftrightarrow m' \in \gamma\ p)$$

The intuition behind this definition is that set F is a superset of the variables constrained by polyhedron p. As a result, if a memory state m belongs to set $\gamma\ p$, a memory state m', which associates any value to variables outside of set F, but agrees with state m on the value of variables in set F, also belongs to set $\gamma\ p$.

Variable x is fresh when the set $\{x' | x' \neq x\}$ frames p. We may now introduce an operator $\texttt{frame}(p)$ such that $\left\{x \,\middle|\, x \in [1, \texttt{frame}(p)]\right\}$ frames p. This operator returns an upper bound β on the variables constrained by polyhedron p.

Operator `frame` is provided by functor $\mathbb{P}^f$, which wraps each element p of the underlying domain into a pair (p, β), such that $\beta = \texttt{frame}(p)$. The core operators of $\mathbb{P}^f$ are defined as follows. The definition of `frame` is extended to constraints.

$$\begin{aligned}
(p, \beta) \sqsubseteq^f (p', \beta') &\triangleq p \sqsubseteq p' \wedge \beta \leq \beta' \\
(p, \beta) \sqcup^f (p', \beta') &\triangleq (p \sqcup p', \max(\beta, \beta')) \\
(p, \beta) \sqcap^f c(x) \geq 0 &\triangleq (p \sqcap c(x) \geq 0, \max(\beta, \texttt{frame}(c)))
\end{aligned}$$

This intuitive definition of $\mathbb{P}^f$ doesn't preserve the framing property. For example, consider the definition of the join operator $\sqcup^f$ above. Although the following definition of $\sqcup$ is contorted, it is correct according to the specification of listing 1.1. However, it breaks the expected framing property.

$$p \sqcup p' \triangleq \begin{cases} x \leq 0 & \text{if } p = p' = \bot \\ \texttt{top} & \text{otherwise} \end{cases}$$

If both polyhedra p and p' are empty, then $\texttt{frame}(p) = \texttt{frame}(p') = 0$. However, $\texttt{frame}(p \sqcup p')$, with the definition of $\sqcup$ above, is strictly positive and therefore greater than $\max(\texttt{frame}(p), \texttt{frame}(p')) = 0$. One solution is to keep the definitions given above, but change that of $\gamma^f(p, \beta)$. Given a memory state m, we

impose that variables above β are free in p by quantifying over states m' which may associate any value to these variables.

$$m \in \gamma^f(p, \beta) \triangleq \forall m',\ m' \equiv_{[1,\beta]} m \Rightarrow m' \in \gamma\, p$$

With framing available, introducing auxiliary fresh variables is easy and the proof of freshness comes for free. By taking x' as $\max(\texttt{frame}(e), \texttt{frame}(p)) + 1$, the assignment operator $p[x := e]$ can be defined, like before, as follows.

$$p\,[x := e] \triangleq ((p \sqcap x' = e) \setminus x)\,[x'/x]$$

1.5.4 Assignment with buffered renaming

With functor $\mathbb{P}^f$, each assignment results in one renaming, but we can do better. The following solution performs a lower amortised number of renamings. Like $\mathbb{P}^f$, this alternative is also implemented as a functor, called $\mathbb{P}^b$. Like $\mathbb{P}^f$, the soundness proof of $\mathbb{P}^b$ enforces invariant properties using an adequate definition of the concretization function γ.

Functor $\mathbb{P}^b$ makes it possible to express relations between memory states in the intermediary computations of the operators. This is achieved by duplicating the set of variable names: each variable x can be represented as $x@0$ or $x@1$, as a generalisation of the previous solution where variable x' is the updated version of variable x. One bit is appended, hence @0 and @1, to each variable x and function γ imposes that exactly one of these representatives refers to an actual variable. Its definition involves two auxiliary functions: σ and π. Function σ associates to each variable its current representative: $\sigma(x)$ is either $x@0$ or $x@1$. Function π associates the actual variable x to both $x@0$ and $x@1$, that is $\pi(x@0) = \pi(x@1) = x$. Therefore, the equality $x' = \sigma(\pi(x'))$ holds only when x' is the current representative of variable $x = \pi(x')$.

$$m \in \gamma^b(p, \sigma) \triangleq \forall m',\ m' \equiv_{\{x' \mid x' = \sigma(\pi(x'))\}} (m \circ \pi) \Rightarrow m' \in \gamma\, p$$

In functor $\mathbb{P}^b$, an assignment to variable x switches the representative of x, instead of renaming the variable in the underlying polyhedron as functor $\mathbb{P}^f$ does. Renamings from assignments are buffered until joins or inclusions, where they may be performed so that the two operands associate the same representative to each variable. Furthermore, two successive renamings on the same variable cancel out.

Functor $\mathbb{P}^b$ could be extended so as to buffer projections, which could then be reordered to get smaller intermediate results, in terms of size of representation. The decision of when to apply the delayed projections would be delegated to the oracle. This extension is not implemented yet.

Assignment in VPL**.** Our modular treatments of assignment, both $\mathbb{P}^f$ and $\mathbb{P}^b$, depart from prior work by Frédéric Besson et al. [6]. In their work, all the projections—including those who occur as part of the handling of assignments—are systematically delayed until inclusion tests. In ours, they are performed eagerly. Furthermore, our approach results in more manageable proofs, as the troubles of having to deal with several memory states are isolated in functors.

1.6 Formalising external code

We have now covered the whole frontend and its accompanying soundness proof, as formalised in Coq. The frontend builds on the declaration of the OCaml oracle as axioms. As we saw earlier, resorting to axioms prevents the frontend from being executable within Coq: it must be extracted to OCaml code. After extraction, the frontend calls to the oracle appear as function calls.

1.6.1 The pitfalls of a naive axiomatisation

Extraction roughly consists in removing all the proof-related information from a Coq development, as OCaml type system is not powerful enough to represent it. For example, it is possible in Coq to express that a function returns integers less than five directly in its type; it would be of the form $\ldots \to \{\mathsf{x} : \mathsf{Z} \mid \mathsf{x} < 5\}$. Such a function is extracted to an OCaml function returning integers, whose type is of the form $\ldots \to \underline{\mathsf{Z}}$, where type $\underline{\mathsf{Z}}$ is the extracted version of type Z. Given this difference in expressivity, the axiomatisation of the oracle in Coq makes a number of assumptions which are worth making explicit. We will review each of these assumptions, consider the danger of overlooking it and then propose a way to handle it.

In the following, underlined expressions are OCaml expressions, while the others are Coq expressions. Let $\underline{\mathsf{f}}$ be an external function of OCaml type $\underline{A} \to \underline{B}$. It is declared to Coq as a function f of type $A \to B$ and the extractor is instructed to replace calls to f with calls to $\underline{\mathsf{f}}$. Types $\underline{A}$ and $\underline{B}$ must be the extracted versions of A and B; the OCaml compiler will report an error otherwise.

Inconsistency. An axiom such as failwith below introduces inconsistency as it returns a value of any type B from a string, thereby proving that type B is not empty.

Axiom failwith: $\forall$ B, string $\to$ B.

In particular, failwith False "" gives a value of type False. However, type False is defined as **Inductive** False : **Prop** :=. It is a property, hence "**Prop**", which can't be satified: it has no constructor and, consequently, no Coq expression has type False. The pitfall of inconsistency is avoided by providing an implementation in Coq of axioms. While the resulting code is never executed, it proves that there exists Coq expressions of the type of the axiom.

Implicit axioms. If B is a precise type, such as $\{\mathsf{x}\colon \mathsf{Z} \mid \mathsf{x} < 5\}$, it may be extracted into a strictly more general OCaml type, such as $\underline{\mathsf{Z}}$ in the example above. This introduces the implicit requirement on $\underline{\mathsf{f}}$ that its results are lower than 5, which the OCaml type checker cannot ensure. This shortcoming can be avoided by manually checking that all the Coq types involved in axioms are identical to their OCaml extraction. This is the case of the axiomatisation found on listing 1.2.

Memory corruption. The oracle described in chapters 3 and 4 uses the GMP [26] C library to represent rational numbers. A bug in GMP or its OCaml frontend, ZArith [43], may corrupt arbitrary memory locations. Although

there is no shield against this problem as long as the C library is linked to the extracted Coq code, it seems unlikely that such a bug breaks soundness silently.

Implicit purity. The semantics of $\rightarrow$ are different in Coq and in OCaml. In Coq, f is implicitly a pure function: hence it is possible to prove that $\forall$ x : A, f x = f x. On the contrary, $\underline{\text{f}}$ in OCaml may use imperative features to create an implicit state such that, for a given x two distinct calls f x give different results. In other words, axiomatising f as **Axiom** f : A $\rightarrow$ B in Coq introduces an implicit functional requirement on function $\underline{\text{f}}$: it should be observationally pure. This means that $\underline{\text{f}}$ may have an implicit state, but it needs to remain hidden, as would be the case when using it for memoization for example. One of the contributions of this thesis is a method for handling the situation where there is no guarantee about the purity of the axiomatised OCaml functions. It is the topic of the following subsection.

1.6.2 A simple theory of impure computations

The soundness proofs of the abstract domain of polyhedra in Verasco don't rely on the purity of oracle functions, as is common in result verification settings. However, proofs being independent of the functional purity of the axioms implementation is usually left as a meta argument, on paper. Meanwhile, it may be difficult to ensure that an OCaml program has no observable side effects. This is especially true if it calls C code, like the oracle described in chapters 3 and 4 does. The following text introduces the *may-return monad*: a proposal to make Coq check that, indeed, the soundness proofs don't rely on the functional purity of oracle functions. This theory is inspired by simulable monads, introduced by Guillaume Claret et al. [14], without the concept of prophecy: we aren't interested in generating the oracle.

The intuition. Suppose there is an OCaml function f : A $\rightarrow$ B of the oracle, which we need to make available to Coq code. Types A and B aren't underlined as previously: we make sure that the Coq type is identical to the extracted OCaml type. To declare function f, we introduce an axiom: **Axiom** f : A $\rightarrow$?B. The return type ?B stands for function f returning *computations* yielding a value of type B. Getting a value from a computation ?B occurs through relation $\rightsquigarrow$ of Coq type ?B $\rightarrow$ B $\rightarrow$ **Prop**. This relation gives its name to our theory: k $\rightsquigarrow$ a means computation k *may return* value a. Both type transformer ? and relation $\rightsquigarrow$ are introduced as axioms: reasoning within the monad can't rely on their particular implementation.

```
Axiom impure : Type → Type.
Axiom may_return : ∀ A : Type, ? A → A → Prop.
Notation "? A" := (impure A) (at level 80).
Notation "A ⇝ B" := (may_return A B) (at level 85).
```

Using this axiomatisation, it is possible to prove that $\forall$ x, f x = f x. However, f x now has type ?B: it refers to a computation and says nothing about the value returned by the computation. Furthermore a value a returned by computation ?B relates to it through relation $\rightsquigarrow$, instead of the usual Leibniz equality =. As a result, ?B $\rightsquigarrow$ a can't be used for rewriting f x as a. The lemma capturing the functional purity of f is f_pure below.

Lemma f_pure : ∀ x a b, (f x ⇝ a) → (f x ⇝ b) → a = b.

Without further information on the implementation of type transformer ? and relation ⇝, it is impossible to prove.

The definition. Impure computations are Coq computations which may use external computations in Ocaml. As we saw, for any Coq type A, we assume a type ?A, denoting impure computations returning values of type A. Type transformer ? is equipped with a monad:

- Operator `bind` : ?A → (A → ?B) → ?B encodes the Ocaml construct **let** x = k **in** k' as `bind` k (**fun** x ⇒ k').
- Operator `unit` : A → ?A lifts a pure computation as an impure one.
- Relation ≡ : ?A → ?A → **Prop** represents equivalence of semantics between Ocaml computations.

Again, all of these are introduced as axioms: reasoning with the monad relies on the types of theses expressions. Operator `bind` is associative and admits `unit` as neutral element. Relation $\rightsquigarrow_A$, introduced above, is compatible with $\equiv_A$ and satisfies the following additional axioms.

- `unit` $a_1 \rightsquigarrow a_2 \Rightarrow a_1 = a_2$
- `bind` $k_1\, k_2 \rightsquigarrow b \Rightarrow \exists a,\ k_1 \rightsquigarrow a \wedge k_2\, a \rightsquigarrow b$

Using the monad. The theory of may-return monads is a very abstract axiomatisation of impurity: it doesn't provide any information about *side effects* of impure computations. However, this is sufficient for our needs in VPL, as the frontend only uses *results* of oracle functions. The monad is an effective solution to supporting impure oracle functions: it prevents reasoning on them as if they were pure functions. The key point for this to work is that the frontend doesn't depend on a particular implementation of the monad. For this reason, its implementation is left opaque throughout the frontend. Another way to express the same thing is: the whole frontend is parameterised by an implementation of the monad.

Since the may-return monad is used in the components of the frontend upon which everything else is built, it contaminates most of the functions and, ultimately, the interface of the abstract domain. For example, the inclusion test operator has type t → t → ?bool, instead of type t → t → bool required by Verasco abstract domain interface. The issue here is that the rest of Verasco doesn't use the monad, as is apparent in the interface of abstract domains shown at the beginning of the chapter, on listing 1.1. In order to bridge the gap, the may-return monad is instanciated as the last step of building the abstract domain. Given that the monad serves no other purpose than restricting proofs, it can be implemented by the trivial monad, as shown on figure 1.7.

Extraction of impure computations The may-return monad is useful to control Coq assumptions that would otherwise be left implicit. However, it is of no other practical interest and is removed at extraction time by providing the trivial implementation given on figure 1.7. The extractor inlines these definitions so that the monad has no runtime overhead.

$$?\mathtt{A} \triangleq \mathtt{A} \qquad \mathtt{k} \equiv \mathtt{k}' \triangleq \mathtt{k} = \mathtt{k}' \qquad \mathtt{k} \rightsquigarrow \mathtt{a} \triangleq \mathtt{k} = \mathtt{a}$$
$$\mathtt{unit\ a} \triangleq \mathtt{a} \qquad \mathtt{bind\ k\ k}' \triangleq \mathtt{k}'\mathtt{k}$$

Figure 1.7 – a trivial implementation of the may-return monad

1.6.3 Backward reasoning on impure computations

Using a monad in Coq is usually considered a source of pain in the proofs. This is probably the reason for which the implicit purity requirement on Ocaml functions declared to Coq as axioms is usually swept under the rug. It turns out that proof automation can deal with most of the proof overhead induced by the monad. The monad was introduced *after* having completed a first version of the abstract domain. The proofs didn't require much change.

The automation machinery is implemented as an Ltac tactic, which implements a *weakest liberal precondition* calculus for the may-return monad. It is based on an operator wlp and a number of accompanying lemmas, all defined in Coq.

```
Definition wlp : ?A → (A → Prop) → Prop :=
  fun k P ⇒ ∀ a, (k ⇝ a) → P a.
```

Property wlp k P states that any result returned by computation k satisfies property P. The tactic proceeds on wlp goals by repeatedly applying the following decomposition rules.

$$\textsc{Decompose unit} \quad \frac{P\ a}{\mathsf{wlp}\ (\mathtt{unit\ a})\ P}$$

$$\textsc{Decompose bind} \quad \frac{\mathsf{wlp}\ k_1\ \lambda a.\big(\mathsf{wlp}\ (k_2\ a)\ P\big)}{\mathsf{wlp}\ (\mathtt{bind}\ k_1\ k2)\ P}$$

Each of these rules deconstructs what appears under the line into what appears above it. In the Coq code, they appear as lemmas with an accompanying proof. Let me illustrate how the tactic proceeds on a simple example. Suppose we have a Coq function g, which first calls an external f returning a natural number of $\mathbb{N}$ and then increments its result.

$$g\,x \ \triangleq\ \mathtt{bind}\ (f\,x)\ \lambda n.(\mathtt{unit}\ n{+}1)$$

The property "g returns only nonzero naturals" is expressed as follows.

$$\forall x,\ \mathsf{wlp}\ (g\,x)\ \lambda n.n{\neq}0$$

Unfolding the definition of function g yields the following.

$$\forall x,\ \mathsf{wlp}\ \big(\mathtt{bind}\ (f\,x)\ \lambda n.(\mathtt{unit}\ n{+}1)\big)\ \lambda n.n{\neq}0$$

We apply rule Decompose Bind, with $k_1 = f\,x$ and $k_2 = \lambda n.(\mathtt{unit}\ n{+}1)$.

$$\forall x,\ \mathsf{wlp}\ (f\,x)\ \lambda a.\Big(\mathsf{wlp}\ \big((\lambda n.(\mathtt{unit}\ n{+}1))\,a\big)\ (\lambda n.n{\neq}0)\Big)$$

Simplifying using β-reduction, we get the goal below.

$$\forall x,\ \mathsf{wlp}\ (f\ x)\ \lambda a.\big(\mathsf{wlp}\ (\mathtt{unit}\ a{+}1)\ (\lambda n.n{\neq}0)\big)$$

Now, we can apply rule Decompose unit, β-reducing $\lambda a.(\lambda n.n{\neq}0\ (a{+}1))$.

$$\forall x,\ \mathsf{wlp}\ (f\ x)\ \lambda a.(a+1\neq 0)$$

Unfolding the definition of wlp yields the final goal, which follows from a being a natural number.

$$\forall x,\ \forall a,\ f\ x \rightsquigarrow a\ \rightarrow\ a+1\neq 0$$

There are a few more decomposition rules which pattern-match over some usual types: booleans, the option type, product types, etc.. When no decomposition applies, the tactic applies the following rule.

$$\textsc{Cut \& Unfold}\quad \frac{\mathsf{wlp}\ k\ P_1 \qquad \forall a,\ k \rightsquigarrow a\ \wedge\ P_1\ a\ \Rightarrow\ P_2\ a}{\mathsf{wlp}\ k\ P_2}$$

More precisely, the tactic tries to discharge the left premise wlp k P_1 using existing lemmas. If it fails, the definition of wlp is simply unfolded. If it succeeds, the goal is replaced using the right premise: the definition of wlp and an hypothesis $P_1\ a$ is added.

In the frontend of the abstract domain, this tactic automates most of the bureaucratic reasoning on first-order impure computations. For higher-order impure computations, such as invoking a list iterator, some manual handling is needed.

1.7 Completing the picture

We have now seen how I designed the result-verifying frontend of an abstract domain of polyhedra and its soundness proof. Parts of this work was done in collaboration with Sylvain Boulmé and was reported in a paper [23].

1.7.1 Improvement on prior work

All of this builds on prior work from Frédéric Besson et al. [6]. The work reported in this chapter improves on it in several ways.

- It deals with the assumption of functional purity made when external Ocaml functions are introduced as axioms. The resulting theory adds little to the proof effort and has zero runtime impact once the Coq code is extracted to Ocaml code.
- The soundness proof is made more modular through the use of functors. These functors enforce that individual proof fragments rely only on logical specifications, rather than implementation details. The overall proof is more modular as a result.
- Prior work buffered projections as much as possible, thereby adding many intermediate variables to the polyhedra. This exacerbates the scalability problems of the abstract domain of polyhedra. The abstract domain described here performs the projections eagerly.

- The abstract domain described here is part of a static analyser, Verasco, which handles most of the C programming language. As a result, it handles the full set of domain operators and supports complex guards.

1.7.2 The oracle

The proving part being dealt with, we still need a witness-generating oracle. There were essentially three options.

Prior art. The original work [6] relied on an off-the-shelf abstract domain of polyhedra: NewPolka, which is part of the Apron [33] collection of abstract domains. Since NewPolka doesn't generate inclusions witnesses, these were recovered using an off-the-shelf linear programming solver [27]. Besides introducing redundant computations, this approach would have set plumbing challenges for making a lot of different tools—written in different languages—cooperate together and link with the already complex Verasco infrastructure.

Instrument. Another option would have been instrumenting an existing abstract domain, such as NewPolka, to make it generate witnesses. However existing implementations use the double description of polyhedra. There is no obvious way of extracting witnesses based on constraints from computations on generators.

Start from scratch. I made the choice of starting from scratch and write an oracle based on constraint representation only. I couldn't find any convincing evidence that using double representation results in more efficient implementations, so this oracle would be a double challenge: use only constraint representation and generate witnesses.

Summary in French

Un domaine abstrait est correct s'il surapproxime correctement les opérations sur les ensembles d'états. Par exemple, l'opérateur «join» approxime l'union des ensembles d'états atteignables dans chacune des alternatives d'un test. Puisqu'il s'agit d'ensembles d'états, la surapproximation se traduit par une relation d'inclusion : l'ensemble d'états P est correctement surapproximé par l'ensemble d'états P' si l'ensemble P est inclu dans l'ensemble P'. Ce chapitre se focalise sur l'aspect preuve de VPL.

Pour démontrer la correction des opérateurs du domaine abstrait des polyèdres, il est nécessaire de s'intéresser à la façon dont les polyèdres sont représentés. Il existe deux représentations des polyèdres : la représentation par contraintes et la représentation par générateurs. Un cube, par exemple, peut être représenté par l'ensemble de ses faces, les contraintes, ou par l'ensemble de ses sommets, les générateurs. Chaque opérateur du domaine abstrait s'exprime plus simplement dans l'une ou l'autre de ces représentations. En revanche,

la conversion d'une représentation vers l'autre est coûteuse en temps d'exécution. La plupart des implémentations existantes du domaine abstrait des polyèdres utilisent les deux représentations. VPL n'utilise que la représentation par contraintes.

Le résultat de chaque opérateur est correct s'il vérifie une certaine relation d'inclusion. Par exemple, l'opérateur «join» surapproxime l'union de deux polyèdres P et P'. Son résultat P_j est correct si P est inclu dans P_j et P' est inclu dans P_j. Le lemme de Farkas permet de prouver des inclusions de polyèdres représentés par contraintes : si le polyèdre P est inclu dans le polyèdre P_j, chacune des contraintes de P_j peut s'exprimer comme un combinaison linéaire à coefficients positifs des contraintes de P. Ces coefficients permettent de mettre en œuvre efficacement une vérification de résultats a posteriori. Pour chaque opérateur, un oracle calcule le résultat et construit un témoin d'inclusion contenant les coefficients qui permettent d'appliquer le lemme de Farkas pour prouver l'inclusion. VPL est ainsi scindé en deux composants : l'oracle et un vérificateur de résultats.

La vérification de résultats à partir de coefficients de combinaisons linéaires de contraintes apparaît déjà dans un article de Frédéric Besson et al. publié en 2007. Mon travail raffine le leur de plusieurs façons.

D'après le lemme de Farkas, si un polyèdre P est inclu dans un autre polyèdre P_j, il est possible de construire les contraintes de P_j en combinant celles de P. Autrement dit, étant donnés les contraintes de P et les coefficients des combinaisons linéaires, il est possible de construire P_j. Plutôt que vérifier a posteriori la correction des résultats, nous pouvons construire directement des résultats corrects. Cela permet d'une part de réduire la quantité d'information que l'oracle doit communiquer au vérificateur et d'autre part de simplifier le travail de ce dernier.

Le vérificateur interagit avec l'oracle seulement pour un sous-ensemble des opérateurs du domaine abstrait, créant ainsi un petit noyau simple à l'interface rudimentaire. Par exemple, l'opérateur d'affectation n'apparaît pas dans cette interface, mais peut être construit à partir de l'intersection et de la projection. Un ensemble de foncteurs étend ce noyau pour construire un domaine abstrait complet. Cette décomposition, où chaque opérateur est ajouté par l'application d'un foncteur, simplifie le travail de preuve en masquant les implémentations par des interfaces opaques.

L'oracle de VPL est écrit en OCaml, alors que le vérificateur est écrit en Coq. Pour que le vérificateur puisse interagir avec l'oracle, il est nécessaire que les fonctions de ce dernier soit déclaré à Coq, par le biais d'axiomes. La sémantique des fonctions OCaml est différente de celle de Coq : en Ocaml, les fonctions peuvent avoir des effets de bord, alors que les fonctions Coq sont des fonctions mathématiques. Axiomatiser une fonction OCaml comme une fonction Coq, comme c'est le cas habituellement, permet de construire des preuves fausses lorsque celles-ci dépendent de la pureté fonctionnelle de la fonction OCaml. Ce n'est généralement pas le cas, si bien que le problème est ignoré la plupart du temps. Le développement Coq de VPL apporte des garanties supplémentaires en proposant une axiomatisation de l'oracle qui empêche les preuves Coq de dépendre de sa pureté.

Chapter 2

Proving inclusions with linear programming

Previous chapter covered the design of VPL, an abstract domain of polyhedra built for the static analyser Verasco. Meeting the interface of abstract domains in Verasco requires proving in Coq the soundness of each operator of the domain. We established that result verification was a reasonable approach to reaching this goal. Indeed, all the proof obligations boil down to inclusion properties and Farkas's lemma provides an efficient method for checking these inclusions through witnesses. On the assumption that we had a witness-producing oracle, we saw how the domain and its accompanying proofs could be built in Coq.

We will now start fulfilling the assumption of a witness-producing oracle. The primary ingredient for doing so is testing the inclusion of a polyhedron P in the half-space defined by a constraint $c(x) \geq 0$. It can be done by leveraging tools from the field of linear programming, which addresses the problem of optimising linear functions over convex sets defined by affine constraints, that is to say over convex polyhedra.

Before reviewing how, let me make the inclusion problem more precise. We consider an n-dimensional space, where a point x has coordinates $(x_1, \ldots, x_n)$. Polyhedron P and function $c(x)$ in constraint $c(x) \geq 0$ are defined as follows, all numbers being rationals.

$$P \triangleq \left\{ x \;\middle|\; \bigwedge_{i=1}^{l} c_i(x) \geq 0 \right\} \qquad \text{and} \qquad c(x) \triangleq a_0 + \sum_{i=1}^{n} a_i \cdot x_i$$

2.1 Inclusion as a maximisation problem

Constraint $c(x) \geq 0$ can equivalently be written $\sum_{i=1}^{n} -a_i \cdot x_i \leq a_0$. Written in this way, it can be thought of as a bound a_0 in the direction given by vector $(-a_1, \ldots, -a_n)$. In other words, equation $\sum_{i=1}^{n} -a_i \cdot x_i = a_0$ defines a hyperplane with normal vector $(-a_1, \ldots, -a_n)$. Its position varies with constant a_0, as figure 2.1 illustrates for normal vector $(1, 1)$. Constraints on this figure are drawn as line segments—the hyperplanes—and the described half-space lies on the shaded side.

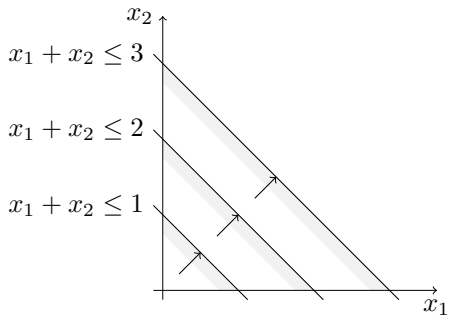


Figure 2.1 – influence of the constant term in the $\sum_{i=1}^{n} -a_i \cdot x_i \leq a_0$ writing

With this in mind, expressing the inclusion test $P \sqsubseteq \left(\sum_{i=1}^{n} -a_i \cdot x_i \leq a_0\right)$ as a maximisation problem is straightforward. Every point $x \in P$ yields a hyperplane $\sum_{i=1}^{n} -a_i \cdot x_i = a_0$, where constant a_0 varies with the chosen point x. We are going to make the hyperplane slide along vector $(-a_1, \ldots, -a_n)$, going as far as possible. That is to say, we are going to move point x within polyhedron P so as to increase the value of constant a_0 as much as possible. This is captured by the following maximisation problem.

$$\max_{x \in P} \sum_{i=1}^{n} -a_i \cdot x_i$$

Suppose now that we found the maximum to be a value $\underline{a}_0$, reached at point $\underline{x}$ of P, that is $\underline{a}_0 = \sum_{i=1}^{n} -a_i \cdot \underline{x}_i$. The following fact follows immediately from the definition of a maximum.

$$\forall x \in P,\ \sum_{i=1}^{n} -a_i \cdot x \leq \underline{a}_0$$

All the points of polyhedron P satisfy the constraint $\sum_{i=1}^{n} -a_i \cdot x \leq \underline{a}_0$. Equivalently, the constraint set p of polyhedron P *implies* constraint $\sum_{i=1}^{n} -a_i \cdot x \leq \underline{a}_0$. Deciding whether $P \sqsubseteq \left(\sum_{i=1}^{n} -a_i \cdot x_i \leq a_0\right)$ is now equivalent to testing whether $\underline{a}_0 \leq a_0$. Indeed, if $\underline{a}_0 \leq a_0$, then we have the following.

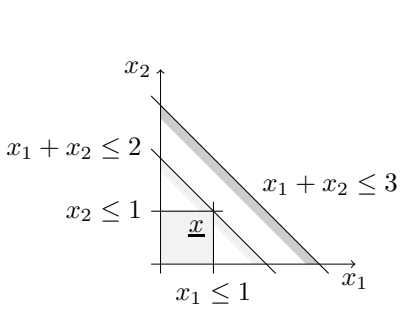
$$\forall x \in P,\ \sum_{i=1}^{n} -a_i \cdot x \leq \underline{a}_0 \leq a_0$$

We reach the conclusion.

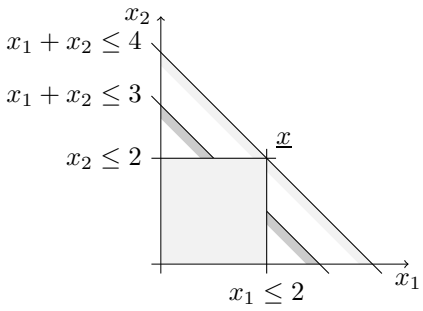
$$\forall x \in P,\ \sum_{i=1}^{n} -a_i \cdot x \leq a_0$$

Conversely, if $a_0 < \underline{a}_0$, we *know* that polyhedron P isn't included in the half-space defined by constraint $c(x) \geq 0$, since we found a point $\underline{x}$ of polyhedron P outside of the half-space. The two cases are illustrated on figures 2.2a and 2.2b.

Expressing an inclusion test as a maximisation problem allows us to decide whether inclusion holds, provided we know how to solve the maximisation problem. However, neither the solution point $\underline{x}$ nor the maximum $\underline{a}_0$ give us much information for building the inclusion witness we need in order to apply Farkas's lemma.



(a) $P \triangleq \{(x_1, x_2) \,|\, x_1 \leq 1 \wedge x_2 \leq 1\}$ is included in the half-space $x_1 + x_2 \leq 3$.



(b) $P \triangleq \{(x_1, x_2) \,|\, x_1 \leq 2 \wedge x_2 \leq 2\}$ isn't included in the half-space $x_1 + x_2 \leq 3$.

Figure 2.2 – testing inclusion with a maximisation problem

2.2 Inclusion as a minimisation problem

The maximisation approach fails to provide us with inclusion witnesses, that is to say linear combinations, with nonnegative coefficients, of the constraints of polyhedron p. The second approach to testing inclusion which we will consider now explicitely tries to build these linear combinations.

Consider figure 2.3. It pictures constraint $c(x) \geq 0$, with $c(x) \triangleq 7 - x_1 - x_2$ and polyhedron p.

$$p \triangleq \{\underbrace{4 - x_1 \geq 0}_{c_1},\ \underbrace{4 - x_2 \geq 0}_{c_2},\ \underbrace{14 - 2 \cdot x_1 - 3 \cdot x_2 \geq 0}_{c_3}\}$$

This time, our starting point is nonnegative linear combinations of constraints of polyhedron p. Let $\lambda_1 \geq 0$, $\lambda_2 \geq 0$ and $\lambda_3 \geq 0$ be the coefficients of constraints c_1, c_2 and c_3, respectively. We are looking for constraints $c'(x) \geq 0$, with $c'(x) \triangleq a'_0 + a'_1 \cdot x_1 + a'_2 \cdot x_2$, which are implied by the constraints of p, that is $p \sqsubseteq c'(x) \geq 0$. We will prove that $p \sqsubseteq c(x) \geq 0$ if we are able to build constraint $c'(x) \geq 0$ so that it implies constraint $c(x) \geq 0$. For now, we know from Farkas's lemma, $c'(x)$ must be of the following form.

$$c'(x) = \lambda_1 \cdot c_1(x) + \lambda_2 \cdot c_2(x) + \lambda_3 \cdot c_3(x)$$

Now, unfolding the definition of constraints c_i gives the following.

$$a'_0 + a'_1 \cdot x_1 + a'_2 \cdot x_2 \ = \ \lambda_1 \cdot (4 - x_1) + \lambda_2 \cdot (4 - x_2) + \lambda_3 \cdot (14 - 2 \cdot x_1 - 3 \cdot x_2)$$

Refactorising for variables x_i gives this equality.

$$a'_0 + a'_1 \cdot x_1 + a'_2 \cdot x_2 \ = \ (4 \cdot \lambda_1 + 4 \cdot \lambda_2 + 14 \cdot \lambda_3) + (-1 \cdot \lambda_1 - 2 \cdot \lambda_3) \cdot x_1 + (-1 \cdot \lambda_2 - 3 \cdot \lambda_3) \cdot x_2$$

Identifiying the terms associated to each variable x_1 and x_2 and the constant, we obtain the following three equalities.

$$\begin{aligned} a'_0 &\triangleq 4 \cdot \lambda_1 + 4 \cdot \lambda_2 + 14 \cdot \lambda_3 \\ a'_1 &\triangleq -1 \cdot \lambda_1 + -2 \cdot \lambda_3 \\ a'_2 &\triangleq -1 \cdot \lambda_2 + -3 \cdot \lambda_3 \end{aligned}$$

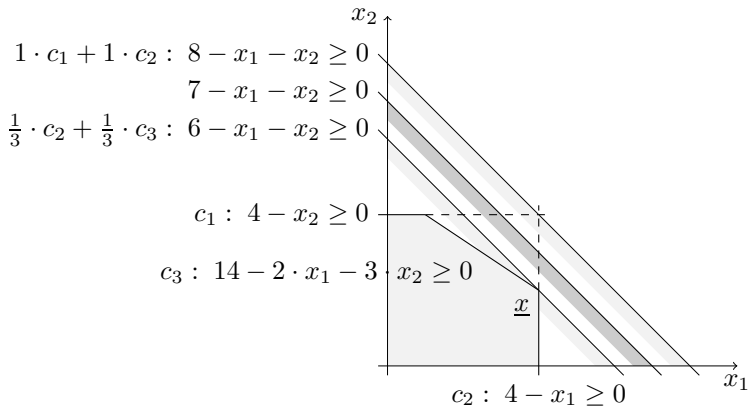


Figure 2.3 – proving inclusion using linear combinations

Going back to our goal of testing whether $p \sqsubseteq c(x) \geq 0$, we know from Farkas's lemma that there must exist $\lambda_0 \geq 0$, such that $c'(x) + \lambda_0 = c(x) = 7 - x_1 - x_2$, if inclusion holds. This constrains the coefficients a'_1 and a'_2 of variables x_1 and x_2 in constraint $c'(x) \geq 0$, yielding the following system of equations.

$$a'_1 = -1 \cdot \lambda_1 + 0 \cdot \lambda_2 + -2 \cdot \lambda_3 = -1$$
$$a'_2 = 0 \cdot \lambda_1 + -1 \cdot \lambda_2 + -3 \cdot \lambda_3 = -1$$

There are two main solutions to this system, illustrated on figure 2.3, the others being convex combinations of them. Each of these two solutions yields a constant term $a'_0 = 4 \cdot \lambda_1 + 4 \cdot \lambda_2 + 14 \cdot \lambda_3$.

- $\lambda_1 = 1$, $\lambda_2 = 1$ and $\lambda_3 = 0$ give $a'_0 = 8$
- $\lambda_1 = 0$, $\lambda_2 = \frac{1}{3}$ and $\lambda_3 = \frac{1}{3}$ give $a'_0 = 6$

The second of these two solutions yields a smaller constant term, thereby making a stronger constraint $c'(x) = 6 - x_1 - x_2 \geq 0$. Constraint $c'(x) \geq 0$ defines a half-space included in half-space $c(x) \geq 0$, since $c(x) = 1 + c'(x)$. I will call "syntactic inclusion" such an inclusion, which is proven by exhibiting a coefficient λ_0, such that $c(x) = \lambda_0 + c'(x)$. Overall, we built constraint $c'(x) \geq 0$ such that $p \sqsubseteq c'(x) \geq 0$ and we just saw that $c'(x) \geq 0 \sqsubseteq c(x) \geq 0$. These inclusions prove $p \sqsubseteq c(x) \geq 0$.

Let us now generalise from this example. Suppose the polyhedron p is made of l constraints, $c_1(x) \geq 0, \ldots, c_l(x) \geq 0$. Each of these constraints $c_i(x) \geq 0$ is of the following form.

$$a_{i0} + a_{i1} \cdot x_1 + \cdots + a_{ln} \cdot x_n \geq 0$$

Constraint $c(x) \geq 0$ is defined in a similar way.

$$a_0 + a_1 \cdot x_1 + \cdots + a_n \cdot x_n \geq 0$$

We associate coefficient $\lambda_i \geq 0$ to constraint $c_i(x) \geq 0$, for all $i \in \{1, \ldots, l\}$. Deciding whether inclusion $p \sqsubseteq c(x) \geq 0$ holds results from solving the following

minimisation problem and comparing the result with the constant term a_0 of constraint $c(x) \geq 0$.

$$\min \sum_{i=1}^{l} a_{i0} \cdot \lambda_i \text{ over the set } \left\{ (\lambda_1, \ldots, \lambda_l) \,\middle|\, \bigwedge_{j=1}^{n} \sum_{i=1}^{l} a_{ij} \cdot \lambda_i = a_i \wedge \bigwedge_{i=1}^{l} \lambda_i \geq 0 \right\}$$

This minimisation problem gathers all the elements which we have considered so far. We are verifying an inclusion by trying to apply Farkas's lemma. To this end, we are trying to express constraint $c(x) \geq 0$ as a nonnegative linear combinations of the constraints of polyhedron p, whose coefficients $\lambda_1, \ldots, \lambda_l$ we are looking for. This has three immediate consequences. The first is that coefficients $\lambda_1, \ldots, \lambda_n$ are the unknowns of our problem. Next, constraint $c'(x) \geq 0$, which results from the linear combination, should be parallel to constraint $c(x) \geq 0$. The constraints of the minimisation problem enforce this by having constraint $c'(x) \geq 0$ associate the same coefficients as constraint $c(x) \geq 0$ to all the variables $x_1, \ldots, x_n$.

Last, we should turn our attention to the constant term of constraint $c'(x) \geq 0$. By construction of $c'(x) = a'_0 + \sum_{i=1}^{n} a'_i \cdot x_n$, we know that $a'_0 = \sum_{i=1}^{l} \lambda_i \cdot a_{i0}$. In order to prove the inclusion, we need to have $c(x) = \lambda_0 + c'(x)$, with $\lambda_0 \geq 0$, so that we have the following inclusion chain.

$$p \sqsubseteq c'(x) \geq 0 \sqsubseteq c(x) \geq 0$$

To this end, the minimisation problem minimises the constant term a'_0 of constraint $c'(x) \geq 0$. Once the optimal $\underline{a}'_0$ is found, we have $\underline{\lambda}_0 = a_0 - \underline{a}'_0$. If $\underline{\lambda}_0$ is nonnegative, then inclusion holds.

From this expression of the inclusion test, it is easy to extract an inclusion witness. Indeed, optimal $\underline{a}'_0$ is reached for some point in the space of the λ_i's. The coordinates of this point describe the linear combination of the constraints of polyhedron p yielding constraint $c'(x) \geq 0$.

2.3 Linear problems and duality

It is worth noticing that the maximisation approach to deciding inclusion, on the example illustrated on figure 2.3, would have given point $\underline{x}$ of coordinates $(4, 2)$. At point $\underline{x}$, the function to be maximised, $x_1 + x_2$, has value 6. We would have concluded that polyhedron p implies the bound $x_1 + x_2 \leq 6$ in the direction $(1, 1)$. This constraint is equivalent to $6 - x_1 - x_2 \geq 0$, which we found using the minimisation approach. The two methods yielding the same solution isn't a coincidence: it is an instance of duality in linear problems, which we will now explore.

Matrix notations. Linear optimisation problems consist in a linear function, called the *objective* function, to be optimised on a polyhedron defined by a set of affine constraints. These sets of affine constraints will be easier to visualise in a more concise notation that what we used so far. Suppose that polyhedron p is made of l constraints $c_i(x) \geq 0$, with $i \in \{1, \ldots, l\}$ and $c_i(x) \triangleq a_{i0} + \sum_{j=1}^{n} a_{ij} \cdot x_j$.

We write p in matrix notation in the following way.

$$p \triangleq \begin{pmatrix} a_{10} \\ \dots \\ a_{l0} \end{pmatrix} + \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{l1} & \dots & a_{ln} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \dots \\ x_n \end{pmatrix} \geq 0$$

We then write $p \triangleq b + A \cdot x \geq 0$, with $b \triangleq (a_{10}, \dots, a_{l0})^T$, $x \triangleq (x_1, \dots, x_n)^T$ and matrix A defined as follows.

$$A \triangleq \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{l1} & \dots & a_{ln} \end{pmatrix}$$

We may also write similarly to the alternative writing $\sum_{j=1}^{n} -a_{ij} \cdot x_j \leq a_{i0}$ of constraint $a_{i0} + \sum_{j=1}^{n} a_{ij} \cdot x_j \geq 0$.

$$p \triangleq -A \cdot x \leq b$$

Dual problems. We expressed testing whether $p \sqsubseteq c(x) \geq 0$, with polyhedron $p \triangleq b + A \cdot x \geq 0$ and constraint $c(x) \triangleq a_0 + a \cdot x$, both as a maximisation problem and as a minimisation problem.

$$\textbf{max}\ -a \cdot x \ \ \textbf{under constraints}\ \ -A \cdot x \leq b$$
$$\textbf{min}\ b \cdot \lambda \ \ \textbf{under constraints}\ \ A^T \cdot \lambda = a \ \ \textbf{with}\ \lambda_i \geq 0,\ i \in \{1, \dots, l\}$$

These two linear problems are call *dual* problems. The right-hand side of one is the vector of coefficients of the objective function of the other and the rows of the constraint matrix of one are the columns of the constraint matrix of the other.

Duality theorems. Duality is captured by two theorems: the weak duality theorem and the strong duality theorem. Both of them relate *feasible* solutions of the *primal* and *dual* linear problems. A feasible solution is an assignment of the variables of a linear problem, either $x_1, \dots, x_n$ or $\lambda_1, \dots, \lambda_l$, which satisfies all the constraints. An *optimal* feasible solution is a feasible solution for which the maximum, or minimum, of the objective function is reached. The minimisation problem is called primal and the maximisation problem is called dual. However, I will avoid these terms as they can be confusing: the dual of the primal is the dual, but the dual of the dual is the primal. The statement of both of the following theorems is taken from the book "Linear Programming 2: Theories and Extensions" by George B. Dantzig and Mukund N. Thapa [18]. The proofs of these theorems appear in the second chapter of the book.

Weak duality theorem. *If $\underline{\lambda}$ is any solution to the primal problem and if $\underline{x}$ is any solution to the dual problem then $-a \cdot \underline{x} \leq b \cdot \underline{\lambda}$.*

Strong duality theorem. *If the primal problem has a feasible solution and the dual problem has a feasible solution, then there exist optimal feasible solutions $\underline{\lambda}^*$ and $\underline{x}^*$ to the primal and dual problems such that $-a \cdot \underline{x}^* = b \cdot \underline{\lambda}^*$.*

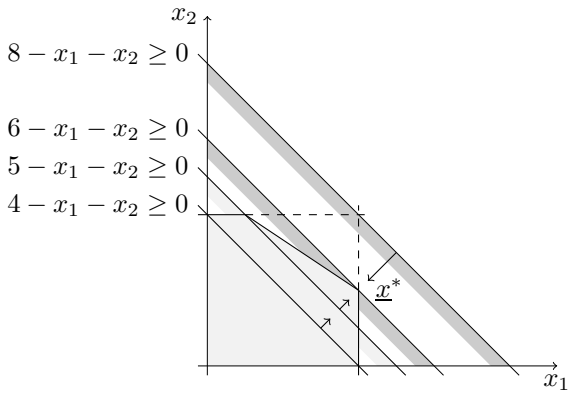


Figure 2.4 – reaching the optimum from below or from above

Figure 2.4 reproduces polyhedron $p \triangleq \{4 - x_1 \geq 0,\ 4 - x_2 \geq 0,\ 14 - 2 \cdot x_1 - 3 \cdot x_2 \geq 0\}$ illustrated on figure 2.3. Maximisation of objective function $x_1 + x_2$ could try point $(0, 4)$, yielding objective value 4, then try point $(1, 4)$, and reach objective value 5, before reaching the optimal $\underline{x}^* \triangleq (4, 2)$ and optimal objective value 6. The corresponding constraints, $4 - x_1 - x_2 \geq 0$, $5 - x_1 - x_2 \geq 0$ and $6 - x_1 - x_2 \geq 0$, are drawn on figure 2.4. On the other hand, minimisation of function $4 \cdot \lambda_1 + 4 \cdot \lambda_2 + 14 \cdot \lambda_3$ could try point $(1, 1, 0)$, yielding objective value 8, before reaching the optimal $(0, \frac{1}{3}, \frac{1}{3})$ and optimal value 6. The corresponding constraints, $8 - x_1 - x_2 \geq 0$ and $6 - x_1 - x_2 \geq 0$, are drawn on figure 2.4.

The strong duality theorem states that the two optima are equal. The weak duality theorem states that maximisation reaches the optimum from below, while minimisation reaches from above. In other words, maximisation works *inside* polyhedron p: all the bounds it finds are reached by at least one point of polyhedron p. Minimisation works *outside* of polyhedron p: the bounds it finds are satisfied by all the points of polyhedron p.

2.4 Interior point methods

We have reduced our inclusion-based proof obligations to linear problems. Now, we should have a look on how to solve them. There are two main categories of solving algorithms for linear problems.

- the simplex algorithm and its variants
- interior point methods

Interior point methods originate from the work of Narendra Karmarkar [38] who published an algorithm for solving linear problem in polynomial time, as opposed to the exponential time worst-case complexity of simplex-based methods. Although it is an interesting theoretical contribution, this new algorithm is less efficient than the simplex algorithm, whose practical complexity on actual problems is polynomial. Later work on interior point methods made them competitive with the simplex algorithm, but neither seem clearly better than the other.

All the work described afterwards in this thesis uses the simplex algorithm. Therefore, I won't cover much of interior point methods, besides my motivations for favouring simplex-based methods.

- As their name implies, interior point methods move in the interior of the exploration space defined by the constraints, excluding the boundary. Since the optimum is always reached on the boundary of the exploration space, the interior point must be moved close enough to the optimum and some post-processing has to recover the exact value.
- Interior point methods involve nonlinear and transcendental functions, such as logarithm. These functions don't behave nicely with exact computer representation of numbers as rationals. On the other hand, using floating point numbers brings in a number of extra concerns of their own, mostly related to their finite precision and to rounding.
- The simplex algorithm relies on simpler mathematics and has been used successfully on many occasions. Initially, the goal of my work wasn't advancing the state of the art of linear programming: I chose the simpler alternative.
- The simplex algorithm is simple enough to be implemented by non-experts, while implementing interior point methods is better left to specialists. Given that extracting witnesses from the solution of linear problems might have required instrumenting the linear problem solver, chosing one that I could implement seemed more appropriate.

2.5 The simplex algorithm

The simplex algorithm is the original method for solving linear problems. It was introduced in the 1950s by George B. Dantzig and is described in many books in various ways. The variant described below is one of the simplest: minimising a linear objective function under equality constraints, with nonnegative variables.

min $c \cdot \lambda$ **under constraints**

$$\sum_{j=1}^{l} a_{ij} \cdot \lambda_j = b_i,\ i \in \{1, \ldots, n\},\ \textbf{with}\ \lambda_j \geq 0 \quad (2.1)$$

This is exactly what is needed to solve the minimisation problem we have seen before for testing an inclusion. The description is intended to lay the basic ideas behind the algorithm, with just enough theory to convince you that everything works as expected. When they aren't provided, the proofs of the lemmas can be found in the book "Linear Programming 2: Theory and Extensions" [18]. Note that a very similar method can solve maximisation problems and it is based on the same concepts. More information can be found in the book referred to above.

The set of feasible solutions. The minimisation problem consists in finding which point of a set minimises the objective function. This set, in our setting,

is the intersection of positive orthant $\left\{(\lambda_1, \ldots, \lambda_l) \,\middle|\, \forall j \in \{1, \ldots, l\}, \lambda_j \geq 0\right\}$, with the set of points $(\lambda_1, \ldots, \lambda_l)$ satifying equality constraints of the problem $\sum_{j=1}^{l} a_{ij} \cdot \lambda_j = b_i,\ i \in \{1, \ldots, n\}$. We will sometimes write the i^{th} equality $a_i \cdot \lambda = b_i$. Every point $\underline{\lambda}$ in that set is a feasible solution of the linear problem. The set of feasible solutions has one main property.

Lemma 1. *The set of points corresponding to feasible solutions of the linear programming problem constitutes a convex set.*

As we'll see when discussing the actual minimisation procedure, this property guarantees that a local optimisation strategy actually leads to the global optimum we are looking for.

Basic feasible solutions. The simplex algorithm explores the set of *vertices*, also called "extreme points", of the set of feasibles solutions.

Definition. *Any point λ in a convex set C that is not a convex combination of two other distinct points in C is by definition a* vertex*, or* extreme point*, of the convex set.*

Considering only vertices reduces the set of candidate optimal points to a finite set. This is correct by the following lemma.

Lemma 2. *If the minimum $\underline{z}^*$ of objective function $c \cdot \lambda$ is finite, then there exists a vertex $\underline{\lambda}$ of the set of feasible solutions such that $c \cdot \underline{\lambda} = \underline{z}^*$.*

Each step of the optimisation performed by the simplex algorithm starts from a vertex of the set of feasible solutions and moves to a neighbouring vertex in such a way that the value of the objective function decreases. At each move, the linear problem is rewritten so that the current vertex appears syntactically in the problem. The rewritten form is called "canonical form". It is canonical to the extent that it is entirely determined by the current vertex.

Suppose that the equality constraints of our minimisation problem 2.1 are linearly independent. The canonical form is built by choosing n variables and defining them in terms of the $l - n$ others. These n variables are called "dependent variables" or "basic variables". We choose, without loss of generality, $\lambda_1, \ldots, \lambda_n$ as dependent variables. The set of dependent variables is called "the basis". Furthermore, we introduce a new variable $z = c \cdot \lambda$, whose value is that of the objective function. Leaving the nonnegativity constraints implicit, linear problem 2.1 is transformed by Gaussian elimination. The resulting system of equations is equivalent to the initial problem and is called the "dictionnary".

$$\begin{aligned} \min z &= \underline{z} + \sum_{j=n+1}^{l} c'_j \cdot \lambda_j \\ \lambda_1 &= b'_1 + \sum_{j=n+1}^{l} a'_{1j} \cdot \lambda_j \\ &\vdots \\ \lambda_n &= b'_n + \sum_{j=n+1}^{l} a'_{nj} \cdot \lambda_j \end{aligned} \tag{2.2}$$

Note that a constant term $\underline{z}$ now appears in the definition of variable z, due to the definition of variables $\lambda_1, \ldots, \lambda_n$ having constants $b'_1, \ldots, b'_n$, which are propagated by the substitutions. The *basic solution* determines the current vertex, which the algorithm starts from in order to continue minimising. The basic solution consists in setting variables $\lambda_{n+1}, \ldots, \lambda_l$ to 0 and choosing for $z, \lambda_1, \ldots, \lambda_n$ the value implied by their definitions: $\underline{z}, b'_1, \ldots, b'_n$. Furthermore, if the value of all the dependent variables is nonnegative, the basic solution is feasible.

The problem of finding an initial set of dependent variables so that the the initial basic solution is feasible can be solved by the simplex algorithm on an auxiliary problem, which has an obvious initial feasible solution, and will be discussed further in this chapter. For now, we'll assume we have a basic solution which is feasible. A basic solution is fully determined by the choice of basis, that is the choice of the dependent variables. The two following lemmas highlight the connection between bases and vertices.

Lemma 3. *A basic feasible solution to linear problem 2.2 corresponds to an extreme point in the convex set of feasible solutions to the linear problem.*

Lemma 4. *Each extreme point corresponds to one or more bases. If one of the basic feasible solutions is nondegenerate an extreme point corresponds to it uniquely.*

Degeneracy. Lemma 4 asserts the one-to-one correspondance between extreme points and bases, under the assumption of *nondegeneracy*. Degeneracy is the name given to the situation where at least one dependent variable λ_i has value 0 in the basic solution: the corresponding b'_i is 0. Under degeneracy, it is possible to swap the dependent variable λ_i with an independent variable $\lambda_{i'}$, $i' \in \{n+1, \ldots, l\}$ in the basis without affecting the basic solution. Therefore, the choice of basis describing the current vertex isn't unique. The implications of degeneracy will be clear when we will look at how the simplex algorithm iterates.

To be complete, there is another type of degeneracy, called "dual degeneracy", where several distinct vertices yield the same objective value. However, dual degeneracy doesn't impact the variant of the simplex algorithm we describe here. The problematic degeneracy, where several bases describe the same vertex, is called "primal degeneracy". Unless otherwise stated, this is the degeneracy I refer to in the following.

Decreasing the objective value. Consider the definition of the objective value in the minimisation problem in canonical form 2.2.

$$z = \underline{z} + \sum_{j=n+1}^{l} c'_j \cdot \lambda_j$$

The basic solution sets all the independent variables, or nonbasic variables, to 0, which cancels term $\sum_{j=n+1}^{l} c'_j \cdot \lambda_j$ and yields $\underline{z}$ as the current objective value. This value may be decreased by adjusting the value of one of the independent variables λ_j. Since they must stay nonnegative and their value in the basic solution is 0, our only possibility is to increase their value. To choose the value of which variable to increase, we may have look at the partial derivative

of the objective function with respect to each variable λ_j: it is equal to c'_j. Variables λ_j for which $c'_j < 0$ are good candidates: if variable $\lambda_{\underline{j}}$ has negative coefficient $c'_{\underline{j}}$ and its value is increased from 0 to $\delta_{\underline{j}} > 0$. The value of the objective function becomes $z = \underline{z} + c'_{\underline{j}} \cdot \delta_{\underline{j}} < \underline{z}$. Which of these candidate to choose is an implementation choice and is called the "pivoting rule". The following lemma states that following a direction in which the partial derivative is negative is safe.

Lemma 5. *Any solution to a linear programming problem that is a local minimum solution is also a global minimum solution.*

Once we select independent variable $\lambda_{\underline{j}}$ for increase, we must make sure that the increase doesn't violate the equality constraints of the dictionnary. Each of them sets a bound on the increase. If we call $\delta_{\underline{j}} \geq 0$ the increase of variable $\lambda_{\underline{j}}$, the definition of dependent variable λ_i becomes the following.

$$\lambda_i = b'_i + a'_{i\underline{j}} \cdot \delta_{\underline{j}}$$

The value of variable λ_i can be adjusted so as to compensate the increase. Thanks to how the problem was rewritten, such an adjustement doesn't have any effect on the other variables. However, variable λ_i should stay nonnegative. Therefore, we have the following.

$$0 \leq b'_i + a'_{i\underline{j}} \cdot \delta_{\underline{j}}$$

This yields the following bound on $\delta_{\underline{j}}$.

$$-a'_{i\underline{j}} \cdot \delta_{\underline{j}} \leq b'_i$$

Since the basic solution is feasible, $b'_i \geq 0$. This leads to three situations.

$a'_{i\underline{j}} > 0$ In this case, the definition of variable λ_i sets no bound on the growth of variable $\lambda_{\underline{j}}$. The value of variable λ_i can be increased as much as needed.

$a'_{i\underline{j}} = 0$ Changing the value of variable $\lambda_{\underline{j}}$ doesn't affect the value of variable λ_i. Constraint $\lambda_i = b'_i + a'_{i\underline{j}} \cdot \delta_{\underline{j}}$ sets no bound on the growth of variable $\lambda_{\underline{j}}$.

$a'_{i\underline{j}} < 0$ The growth of variable $\lambda_{\underline{j}}$ is bounded above: $\delta_{\underline{j}} \leq \frac{b'_i}{-a_{i\underline{j}}}$.

If no definition in the dictionary bounds the growth $\delta_{\underline{j}}$, the minimum of the problem is unbounded: the value of the objective function is arbitrarily small. Otherwise, the growth must satisfy all the bounds set on it. Suppose that the definition of dependent variable $\lambda_{\underline{i}}$ sets the lowest of these bounds: $\delta_{\underline{j}} = \frac{b'_{\underline{i}}}{-a_{\underline{i}\underline{j}}}$.

Pivoting. Once the new value $\delta_{\underline{j}}$ of independent variable $\lambda_{\underline{j}}$ has been chosen, the value of the dependent variable $\lambda_{\underline{i}}$ becomes 0 and the value of the objective function decreases to $\underline{z} + c'_{\underline{j}} \cdot \delta_{\underline{j}}$. In order to continue the minimisation in the same way, the dictionary must be rewritten so that the new current vertex becomes the basic solution. This operation is called a "pivot": variable $\lambda_{\underline{j}}$ enters the basis

and variable $\lambda_{\underline{i}}$ leaves the basis. The pivot consists in rewriting the definition of variable $\lambda_{\underline{i}} = b'_{\underline{i}} + \sum_{j=n+1}^{l} a'_{\underline{i}j} \cdot \lambda_j$ as a definition of variable $\lambda_{\underline{j}}$.

$$\lambda_{\underline{j}} = \frac{b'_{\underline{i}}}{-a_{\underline{i}\underline{j}}} + \frac{-1}{-a_{\underline{i}\underline{j}}} \cdot \lambda_{\underline{i}} + \sum_{\substack{j=n+1 \\ j \neq \underline{j}}}^{l} \frac{a'_{\underline{i}j}}{-a_{\underline{i}\underline{j}}} \cdot \lambda_j$$

This definition of variable $\lambda_{\underline{j}}$ can then be substituted in the other entries of the dictionary, thereby completing the pivot. This rewrite has made the new vertex syntactically obvious: the new constant term $\underline{z}'$ of the objective function has been updated as expected.

$$\underline{z}' = \underline{z} + c'_{\underline{j}} \cdot \delta_{\underline{j}}$$

Example minimisation. Suppose we have the following problem in canonical form, with the nonnegativity constraints left implicit.

$$\begin{aligned} z &= -\lambda_3 + \lambda_4 \\ \lambda_1 &= 2 + 3 \cdot \lambda_3 + \lambda_4 \\ \lambda_2 &= 1 + -\lambda_3 + 2 \cdot \lambda_4 \end{aligned} \tag{2.3}$$

The variables of the problem are λ_1, λ_2, λ_3 and λ_4. Variables λ_1 and λ_2 are dependent variables. Variable z was introduced to capture the value of the objective function. The basic solution is $z = 0$, $\lambda_1 = 2$, $\lambda_2 = 1$, $\lambda_3 = 0$ and $\lambda_4 = 0$, which I'll concisely write $(0, 2, 1, 0, 0)$. We want to minimise the value of variable z.

In the definition of variable z, variable λ_3 has a negative coefficient: -1. If the value of variable λ_3 increases by δ_3, the value of variable z will decrease by δ_3. The definition of variable λ_1 doesn't bound the growth of variable λ_3: increasing the value of variable λ_3 only makes variable λ_1 more positive. However, the definition of variable λ_2 bounds the definition of variable λ_3 to 1: given that $\lambda_4 = 0$, this is the maximal value that λ_3 can take which preserves $\lambda_2 \geq 0$. Setting variable λ_3 to 1 leads to a new feasible point: $(-1, 5, 0, 1, 0)$.

Now comes the pivot. The definition of variable λ_2 is rewritten so that it defines variable λ_3.

$$\lambda_3 = 1 + -\lambda_2 + 2 \cdot \lambda_4$$

This new definition is now substituted for variable λ_3 in every entry of the dictionary.

$$\begin{aligned} z &= -(1 + -\lambda_2 + 2 \cdot \lambda_4) + \lambda_4 \\ \lambda_1 &= 2 + 3 \cdot (1 + -\lambda_2 + 2 \cdot \lambda_4) + \lambda_4 \\ \lambda_3 &= 1 + -\lambda_2 + 2 \cdot \lambda_4 \end{aligned} \tag{2.4}$$

In turn, this yields the following problem.

$$\begin{aligned} z &= -\mathbf{1} + \lambda_2 + -\lambda_4 \\ \lambda_1 &= \mathbf{5} + -3 \cdot \lambda_2 + 7 \cdot \lambda_4 \\ \lambda_3 &= \mathbf{1} + -\lambda_2 + 2 \cdot \lambda_4 \end{aligned} \tag{2.5}$$

The basic solution is now apparent syntactically, as highlighted in equation 2.5. Going further in the minimisation, we would try to increase the value of variable λ_4, as it has a negative coefficient in the objective function. Its coefficient in the definition of both dependent variables is positive, implying that the growth of variable λ_4 is unbounded. Therefore, the minimum of function $-\lambda_3 + \lambda_4$ is unbounded.

Necessity of pivoting. The pivoting step rewrites the problem into an equivalent problem. The new problem makes the current basic solution appear in the syntax of the definitions. It may be tempting to decorrelate the basic solution from the system of constraints, so as to save part of the computation time spent pivoting. However, doing so breaks all of the syntactic criteria used by the simplex algorithm to take decisions. In the previous example, suppose that we keep the problem under the form given in equation 2.3, and that we keep track of having moved from point $(0, 2, 1, 0, 0)$ to point $(-1, 5, 0, 1, 0)$ separately.

In order to continue minimising, we need to choose the value of which variable we are going to increase. Increasing the value of variable λ_3 isn't possible any more. We are left with the possibility of increasing variable λ_4, although it has a positive coefficient. Doing so results in the value of variable z being decreasing, as we observed in the example. However, the decrease doesn't result directly from the increase of variable λ_4, but rather from the fact that variable λ_3 has to grow twice as fast as variable λ_4 to keep the equation $\lambda_2 = 1 + -\lambda_3 + 2 \cdot \lambda_4$ satisfied.

The canonical form of the linear problems removes dependencies between constraints. The effect of a change on a given constraint can be found by examining only this constraint.

Cycling and termination. Termination of the simplex algorithm is straightforward for a nondegenerate problem, that is when the constant term of the definitions of dependent variables is never zero. Indeed, nontermination could have two causes.

1. The optimisation takes infinitely many steps to converge to the minimum.
2. The optimisation cycles among of subset of dependent variables choices.

Without cycling, an infinity of steps is impossible, as there is a finite number of ways to choose dependent variables. Without degeneracy, cycling is impossible as well, since each iteration strictly decreases the objective value and the choice of dependent variables uniquely determines the value of the objective value.

When the problem is degenerate, the value of the objective doesn't necessarily decrease strictly at each iteration any more. Cycling is possible among the different bases describing a single vertex, thereby yielding the same objective value. There are two approaches to dealing with degeneracy.

- The simplest approach is to removing degeneracy by introducing perturbations in the system. These perturbations are usually kept symbolic [35] by introducing a new infinitesimal quantity ϵ_i in the constant of each constraint and imposing an order among the ϵ_i's, so that constants can be compared lexicographically. This has the effect of increasing the size of the problem, thereby making all pivots more computationally expensive.

- The other approach to dealing with degeneracy is to choose the variables for the pivot such that cycling is prevented. The best-known pivoting rule is Bland's rule [8]. It consists in ordering the variables and, when given a choice, pick the lowest variable.

Although it guarantees termination, Bland's rule leads to a large number of iterations during the optimisation. For this reason, a more efficient pivoting rule is usually used. Since the more efficient rule doesn't necessarily guarantee termination, it is customary to use it for the first N iterations, where N is implementation-defined, before resorting to Bland's rule to ensure termination. An example of more efficient pivoting rule is the *steepest descent* rule. When several variables may be increased to decrease the objective value, the steepest descent rule picks the one with the most negative coefficient in the objective function.

The algorithm. We have now covered all of the ingredients of the simplex algorithm. Starting from a minimisation problem in canonical form, where the basic solution is feasible, the algorithm proceeds as follows.

1. Choose a nonbasic variable λ_j which has a negative coefficient in the objective function, that is the definition of variable z. Variable λ_j will enter the basis.

2. If there is no such variable, the algorithm terminates: the optimal value of the objective function is the value of variable z in the basic solution.

3. Otherwise, find the dependent variable λ_i whose definition sets the smallest upper bound on the growth of variable λ_j. Variable λ_i will leave the basis.

4. If no definition bounds the growth of variable λ_j, the objective function isn't bounded below by the constraint. The algorithm terminates with $-\infty$ as the minimum.

5. Otherwise, pivot on λ_i and λ_j.

6. Go back to step 1.

The initial feasible basis. The algorithm which we have just outlined starts from a problem in canonical form, where the basic solution is feasible. Rewriting a minimisation problem so that it is in canonical form is easy: all that is needed is selecting a subset of the problem variables and performing a Gaussian elimination. However, there is no guarantee that the corresponding basic solution is feasible: the constant term of one or several entries in the dictionary may be negative. Since the basic solution sets the independent variables to zero and the dependent variables to the value of the constant in their definition, negative constants violate the nonnegativity constraints on the variables.

The simplex algorithm can also be used to find an initial basis such that the corresponding basic solution is feasible. The idea for doing so is to build an intermediate problem

- which has a trivial initial basis with a feasible basic solution and

- whose optimal is reached for a basis which has a feasible basic solution in the original problem.

Suppose our original problem is the following.

$$\textbf{min } c \cdot \lambda \textbf{ under constraints } \sum_{j=1}^{l} a_{ij} \cdot \lambda_j = b_i,\ i \in \{1, \ldots, n\}, \textbf{ with } \lambda_j \geq 0$$

We introduce a new variables $\alpha_i \geq 0$ for each constraint. If constant b_i is negative, the constraint is rewritten as $\alpha_i = -b_i + \sum_{j=1}^{l} a_{ij} \cdot \lambda_j$. Otherwise, it is rewritten as $\alpha_i = b_i + \sum_{j=1}^{l} -a_{ij} \cdot \lambda_j$. The intermediate problem is completed by changing the objective function to $z = \min \sum_{i=1}^{n} \alpha_i$.

This intermediate problem has an obvious initial basis $\alpha_1, \ldots, \alpha_n$. The corresponding basic solution is feasible: we arranged for the constant term of each definition to be nonnegative. All that is left to do in order to get a problem in canonical form is to substitute variables $\alpha_1, \ldots, \alpha_n$ in the objective function.

Running the simplex algorithm on the intermediate problem may yield one of two results.

- If $z = 0$, the optimal basic solution is a basic solution to the initial problem.
- If $z > 0$, then the original problem doesn't have any feasible solution, as there is no solution to the intermediate problem where only the variables $\lambda_1, \ldots, \lambda_l$ are nonzero.

The optimal value can't be negative as it is a sum of nonnegative variables.

2.6 A linear program solver

We have now covered the basics of *linear programming*—that is linear problems and associated solving algorithms—and their use for proving that a polyhedron P is included in the half-space defined by a constraint $c(x) \geq 0$. The abstract domain of polyhedra in Verasco uses a variant of the simplex algorithm. However, starting by explaining the basic algorithm let me highlight the crucial points without worrying about the subtleties of refinements.

2.6.1 From optimisation to satisfiability

If we go back and have a look at figure 2.2b on page 47, which illustrates that polyhedron $p \triangleq \{x_1 \leq 2,\ x_2 \leq 2\}$ isn't included in the half-space defined by constraint $x_1 + x_2 \leq 3$, we may realize that point $\underline{x}$ isn't the only point proving that $p \not\sqsubseteq x_1 + x_2 \leq 3$. Any point in the triangle whose vertices are points $(1, 2)$, $(2, 1)$ and $(2, 2)$, excluding those which lie on the line segment between points $(1, 2)$ and $(2, 1)$, disproves the inclusion as well. The real question which we answered using maximisation is the following. *Is there a point* $\underline{x}' \in P$ *such that* $x \notin \{(x_1, x_2) \mid x_1 + x_2 \leq 3\}$*?*

Optimisation doesn't appear to be a *required* ingredient for answering it. Indeed, this question is equivalent to that of satisfiability of the following formula [6].

$$x_1 \leq 2 \ \wedge\ x_2 \leq 2 \ \wedge\ \neg(x_1 + x_2 \leq 3)$$

This formula can then be simplified.

$$x_1 \leq 2 \ \wedge \ x_2 \leq 2 \ \wedge \ x_1 + x_2 > 3$$

More generally, testing the inclusion $p \sqsubseteq \sum_{i=1}^{n} a_i \cdot x_i \leq b$ is equivalent to checking the unsatisfiability of the following formula.

$$p \ \wedge \ \sum_{i=1}^{n} a_i \cdot x_i > b \tag{2.6}$$

A similar transformation exists for the minimisation problem. Indeed, it suffices to exhibit a linear combination which yields a constant smaller or equal to constant b: finding the smallest possible constant isn't necessary. Moving from optimisation to satisfiability opens the possibility to leverage the work done by the very active *satisfiability modulo theory* community. In particular, the solver of linear problems which I built for the abstract domain of polyhedra in Verasco is heavily based on the work which Leonardo de Moura and Bruno Dutertre [20] did to optimise the simplex algorithm for handling linear arithmetic in SMT solvers. Their variant of the simplex algorithm has a number of distinguishing features.

- There is no restriction on the sign of the variables and they can have both a lower bound and an upper bound.
- While the standard simplex algorithm encodes inequality $a \leq b$ as equality $a + s = b$ with a nonnegative slack variable s, inequalities are handled without introducing extra variables.
- Inequalities can be strict.
- The bounds on the variables can be changed dynamically in a simple way.
- A presimplication, taking advantage of the fact that only satisfiability matters, speeds the solving up.

As we'll see shortly, all of these features make it possible to reason in the space of the variables bound by the polyhedra, similarly to the maximisation problem we saw at the beginning of this chapter. One resulting benefit is the similarity between the representation of polyhedra and the problem representation in the solver.

2.6.2 Problem representation

Bruno Dutertre and Leonardo de Moura's algorithm splits the linear problem to solve into three components: bounds on variables, the current value of variables and a dictionary, which defines a subset of the variables by equations. Variables are either problem variables or auxiliary variables. An auxiliary variable is introduced for each constraint where two or more variables appear. For example, constraint $x_1 + x_2 > 3$ above would be represented by an entry in the dictionary $y_1 = x_1 + x_2$ and a bound $y_1 > 3$, where y_1 is a fresh auxiliary variable. No auxiliary variable needs to be introduced for a simple constraint such as $x_1 \leq 2$. It is directly used as a bound. As in the standard simplex algorithm, the set of

variables is split in dependent, or basic, and independent, or nonbasic, variables. The initial dependent variables are the auxiliary variables. The initial value of independent variables is either 0 or the value of one of its bounds if the bounds forbid value 0.

2.6.3 Overview of the algorithm

Since the interest is satisfiability, there is no distinguished objective function to guide the pivoting choices. Rather, the algorithm minimises the extent to which the current assignment is unsatisfiable. The algorithm maintains the invariant property that the independent variables have a current value which is within their bounds. The value of the dependent variables is fixed accordingly and may be outside of the bounds. A step of the algorithm starts with finding a dependent variable—it may be an auxiliary variable or not—whose value is outside of its bounds. Suppose it found y_1.

$$y_1 = x_1 + x_2$$

Suppose further that the current assignment is $y_1 = 0$, $x_1 = 0$ and $x_2 = 0$ and that the bounds on variables are $y_1 > 3$, $x_1 \leq 2$ and $x_2 \leq 2$.

Then, the algorithm looks for an independent variable x_j whose value could be moved within its bounds, so as to reduce the distance from the current value of variable x_i to its bound. In our example, dependent variable y_1 needs to increase. Variable x_1 has a positive coefficient: increasing its value would result in the value of variable y_1 to increase.

Then, the value of y_1 is moved so that it satisfies its closest bound and the value of variable x_1 is moved so as to compensate the move. This also illustrates the handling of strict inequalities: a symbolic positive ϵ is used.

$$y_1 = 3 + \epsilon,\ x_1 = 3 + \epsilon,\ x_2 = 0$$

A pivot on variables y_1 and x_1 is then performed.

$$x_1 = y_1 + -x_2$$

Variable y_1 now has a value within its bounds. The value of variable x_1 is now outside of its bounds. Since it is now a dependent variable, the invariant property on independent variables being within their bounds is preserved.

The next iteration of the algorithm selects dependent variable x_1, whose value $3 + \epsilon$ violates its upper bound 2. The value of variable y_1 can't decrease without violating its bound. However, variable x_2 can increase to make variable x_1 decrease.

$$x_1 = 2,\ y_1 = 3 + \epsilon,\ x_2 = 1 + -\epsilon$$

A pivot is performed.

$$x_2 = y_1 + -x_1$$

The algorithm terminates: the value of all the variables is within its bounds. Each iteration decreases the overall distance of the variables to their bounds, which guarantees termination.

Stopping criteria. The algorithm may have one of two outcomes.

- No dependent variable x_i is outside of its bounds. In this case, a satisfying assignment has been found. In our problem of interest, this means that inclusion doesn't hold.
- No independent variable x_j can be found which could compensate, at least partially, the move of variable x_i within its bounds. This situation is called a "conflict" and results from the unsatisfiability of the original formula 2.6. This is the interesting case: inclusion holds.

2.6.4 Extracting witnesses

When inclusion holds, we face the same problem as when we considered maximisation: how can a witness be extracted from a conflict? Assume that, in order to test whether inclusion $p \sqsubseteq \sum_{i=1}^{n} a_i \cdot x_i \leq b$ holds, we check whether the following formula is unsatisfiable.

$$p \wedge \sum_{i=1}^{n} a_i \cdot x_i > b$$

The solver associates a variable y_c to constraint $\sum_{i=1}^{n} a_i \cdot x_i > b$: variable y_c is defined by equality $y_c = \sum_{i=1}^{n} a_i \cdot x_i$ and has a bound $y_y > b$. Remark that, if there is a single $\underline{i} \in \{1, \ldots, n\}$ such that $a_{\underline{i}}$ is different from 0, then $y_c \triangleq x_{\underline{i}}$.

Assume further that the algorithm terminates because of a conflict in the following entry of the dictionary.

$$x_i = \sum_{j=1}^{n'} a_{ij} \cdot x_j$$

We are going to assume that polyhedron p isn't empty and, therefore, that the conflict involves variable y_c. Since the coefficient of variable y_c is nonzero, the conflict line can be rewritten.

$$y_c = \sum_{j=1}^{n'} a'_{ij} \cdot x_j$$

The constraints of polyhedron p imply a bound b_c on the right-hand side of this equality. Variable y_c has a bound $y_c > b$ from constraint $\sum_{i=1}^{n} a_i \cdot x_i > b$. There being a conflict implies that the bound $y_c > b$, is incompatible with bound b of the right-hand side. For example, we might have $b \geq b_c$ in the following.

$$b < y_c = \sum_{j=1}^{n} a'_{ij} \cdot x_j \leq b_c$$

The linear combination of the constraints of polyhedron p, which yields the bound $y_c \leq b_c$, is simply read from the right-hand side $\sum_{j=1}^{n} a'_{ij} \cdot x_j$ of the equality above. This result builds on the following remarks.

- Each constraint of the input problem has a variable associated to it, in a similar way to variable y_c is associated to input constraint $\sum_{i=1}^{n} a_i \cdot x_i > b$.

- The variables which appear in the conflict are necessarily associated to an input constraint. Indeed, only these variables have an explicit bound, by construction of the problem. If a variable with no explicit bound appeared in the equality above, its value could be adjusted as needed and there wouldn't be any conflict.

The upper bound on variable y_c implied by the constraints of polyhedron p is incompatible with the lower bound $y_c > b$. The linear combination of the constraints of p yielding the upper bound appear by substituting for each variable x_j on the conflict line, the input constraint it is associated with. If variable x_j has a positive coefficient a_{ij}, then the constraint which gives variable x_j its upper bound should be used. If coefficient a_{ij} is negative, the constraint which gives the lower bound should be used instead.

This discussion should shed light on why variable y_c is necessarily involved in the conflict if polyhedron p is nonempty. If variable y_c doesn't appear in conflict $x_i = \sum_{j=1}^{n'} a_{ij} \cdot x_j$, the bounds defined by polyhedron p are incompatible with each other, implying that polyhedron p is empty.

Instrumenting data structures. Bruno Dutertre and Leonardo de Moura's algorithm doesn't directly support the extraction of witnesses we have just seen. Some instrumentation of the working data structures of the algorithm is necessary. The basic idea is to associate a unique identifier to each input constraint and attach that identifier to its corresponding bound on a variable. Extra care is needed for bounds on problem variables such as constraint $2 \cdot x_1 \leq 3$, which is represented as bound $x_1 \leq \frac{3}{2}$ in the working data. While the two are equivalent, a witness coefficient λ_1 applying to constraint $x_1 \leq \frac{3}{2}$ should be divided by two before applying it to constraint $2 \cdot x_1 \leq 3$.

2.7 Wrapping up

This chapter bridged the gap between the necessity of proving inclusion properties and algorithms which effectively compute witnesses of the inclusion properties, when they hold. These algorithms belong to the field of linear programming, of which we have covered the basics with their theoretical justification. In practice, we use a variant of the simplex algorithm originally targeting satisfiability modulo theory solving [20]. We skimmed through the main features of this algorithm, just enough to show how witnesses are extracted from the final state of the algorithm.

I should stress that this chapter mostly covers background and state-of-the-art material, which will serve as a basis for the rest of the thesis. My contribution described in this chapter is the extraction of inclusion witnesses from a solver conflict.

Summary in French

La correction des résultats des opérateurs de VPL s'exprime comme des propriétés d'inclusion.

Le domaine de la programmation linéaire fournit des algorithmes pour tester l'inclusion de polyèdres, notamment l'algorithme du simplexe. Ce chapitre présente ces outils, en gardant à l'esprit la nécessité de générer des témoins. Il y a deux façons de déterminer si un polyèdre P est inclu dans le demi-espace décrit par la contrainte linéaire $a.x \leq b$.

La première méthode consiste à chercher le point $\bar{x}$ du polyèdre P qui maximise $a.x$. Si $a.\bar{x} \leq b$, alors tous les points de P satisfont la contrainte : le polyèdre P est inclu dans le demi-espace.

L'autre méthode est directement inspirée du lemme de Farkas : il s'agit de trouver la contrainte $a.x \leq \bar{b}$, résultant d'une combinaison linéaire à coefficients positifs des contraintes de P, qui minimise $\bar{b}$. Si $\bar{b} \leq b$, alors il y a inclusion.

Ces deux méthodes permettent d'arriver au même résultat : $a.\bar{x} = \bar{b}$. Le problème linéaire de maximisation et le problème linéaire de minimisation sont appelés «duaux».

La partie Coq de VPL dépend de la capacité de l'oracle à produire des témoins d'inclusion sous la forme de coefficients qui permettent d'appliquer le lemme de Farkas. Le problème de minimisation ci-dessus est posé en termes des coefficients du témoin : il s'agit de l'approche la plus directe. Néanmoins, il est possible grâce à une instrumentation simple d'extraire un témoin de la solution du problème de maximisation. Cela permet d'utiliser pour VPL une variante de l'algorithme du simplexe conçue pour les besoins des outils de satisfaisabilité modulo théorie, qui a l'avantage de permettre une résolution incrémentale des problèmes linéaires.

Chapter 3

Computing on polyhedra represented as constraints

We started from the goal of building an abstract domain of polyhedra for the Verasco static analyser. This abstract domain, VPL, should have its soundness proof written using the Coq proof assistant. We chose a result verification approach to the proof: an external oracle performs the computations and provides inclusion witnesses so that its results can be checked. Consequently, it is not required to trust the result provided by the oracle upfront. Then, we saw how to check whether one polyhedron is included in another, and how to extract a witness if the inclusion holds, using linear programming. This would be enough for building a state of the art witness-providing oracle [6]: compute results with a readily available abstract domain, such as NewPolka, and compute witnesses with a readily available linear problem solver, such as the GNU Linear Programming Kit [27]. However, this method adds the significant overhead of solving many linear problems after each operator to the already costly abstract domain of polyhedra.

The oracle I built for VPL integrates witness generation with result computation, in order to achieve better performance. VPL oracle, which is external to Coq, is written in the Ocaml programming language. The resulting domain compares well in performance with state-of-the-art domains NewPolka and PPL and provides result verification as a bonus. The oracle is itself a complete abstract domain which can be used independently of the result-verifying Coq frontend.

VPL being competitive results mostly from the design of the oracle. It has two distinguishing features, which this chapter elaborates on.

- It operates only on the constraint representation of polyhedra.
- It generates witness on the fly from intermediate results.

3.1 Representing polyhedra as constraints

In a performance-sensitive program, data structures are key. Therefore, the organisation and invariant properties of the constraint representation of polyhedra in VPL will be our starting point. We'll proceed from the bottom up.

Numbers representation. The basic entities that VPL deals with are numbers. VPL polyhedra bound rational variables and constraint have rational coefficients. These coefficients can grow so as to overflow native integer representation during an analysis. Working around this shortcoming requires using an arithmetic library for arbitrarily large numbers. This has a serious impact on overall performance. The oracle uses the ZArith [43] Ocaml frontend to the GNU Multiple Precision Arithmetic Library, GMP [26]. ZArith tries to lower the cost of using GMP by using native integers as long as they don't overflow. This design draws on insights from the satisfiability modulo theory solving community [11]: in most cases, extended precision is not used, hence the great importance of an arithmetic library that operates on machine words as much as possible, without allocating extended precision numbers.

Constraints. Capturing linear relations between program variables with polyhedra generally leads to sparse systems [47]: each relation involves only a few program variables. VPL oracle stores constraints as radix trees, where the path from the root to a node identifies the variable whose coefficient is stored at that node. This offers a middle ground between dense representation, as used by other widely-used implementation of the abstract domain of polyhedra and sparse representation which makes random access costly as sparsity diminishes. I borrowed the idea from previous work by Frédéric Besson et al. [6].

Polyhedra. VPL oracle represents polyhedra as sets of constraints only. The salient choice I made while designing the corresponding data structures is: the representation of polyhedra is always *minimised*. We'll see shortly what this choice encompasses; for now, you may think of it as redundancy in the constraint set being removed eagerly. The main motivation is to make the behaviour of VPL predictable: the computational cost of the operators depends only on the geometry of their inputs, without interference from a state resulting from previous computations.

What I mean by "state" is best illustrated on the performance profiles presented in the next chapter. These profiles compare VPL to NewPolka and PPL. Both NewPolka and PPL use the double representation of polyhedra. However, they differ in how they maintain these representations. NewPolka performs the conversions eagerly: both representations are always available. PPL performs the conversions lazily: it maintains one as long as possible and switches to the other when requested by an operator. The performance profiles seem to indicate that PPL has an extremely efficient convex hull operator: it is six times faster than that of NewPolka. In fact, PPL merely delays the work: all of its other operators are slower than those of NewPolka. When considering the total time spent by the execution in each domain over a whole analysis, the two perform comparably. Maintaining minimised representation of polyhedra amounts to performing work eagerly. It makes both the code and the analysis of its performance simpler.

Canonical empty polyhedron. An immediate consequence of minimising representations is that the empty polyhedron $\bot$, that is the polyhedron which has no point in it, is represented in a canonical way. If `t` is the type of collections of constraints, the data type for polyhedra can effectively be spelt as $\texttt{t} + \bot$. A

polyhedron is either the empty polyhedron $\bot$, or a collection of constraints which are satisfied by at least one point. In other words, there is no constraint representation of $\bot$ in VPL.

3.1.1 Separating equalities from inequalities

When a polyhedron p isn't empty, its set of constraints is split into two: the set p_E of its equality constraints and the set p_I of its inequality constraints. Keeping equality constraints separate has two main benefits: dimension reduction and cheaper projection.

Dimension reduction. Each equality can be regarded as a definition of one of the variables which appear in it, in a similar fashion to the construction of the dictionary presented in the previous chapter. In turn, the defined variables can be substituted by their definition in the other constraints, both equalities and inequalities. The dimension of the system of inequalities, which is the costlier to operate on, is thereby decreased.

Cheaper projection. Computing the projection of a polyhedron represented as constraints onto a subset of its dimensions involves the very costly Fourier-Motzkin elimination [18] of the corresponding variables. This elimination can be bypassed for the variables which appear in equality constraints. For those, the equality is turned into a definition and elimination is performed by substitution.

3.1.2 Invariants of the representation of equalities

Following the minimisation choice, the system of equalities of a polyhedron has no redundancy: no equality is a linear combination, with coefficients of arbitrary sign, of the others. Furthermore, it is rewritten to be in row echelon form, similarly to what the first step of Gaussian elimination does. Suppose we have the following system.

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 &= 1 \\ x_1 + 2 \cdot x_2 + 5 \cdot x_4 &= 2 \\ x_2 + x_3 &= 0 \end{aligned}$$

It doesn't make the polyhedron $\bot$, as it has at least one solution: $x_1 = \frac{3}{4}$, $x_2 = x_3 = 0$ and $x_4 = \frac{1}{4}$. In order to rewrite it in row echelon form, we start by picking one variable to be defined by the first equality, say variable x_1. The equality becomes $x_1 = 1 - x_2 - x_3 - x_4$. We substitute variable x_1 in equalities which *follow* the first one. In this case, this is the rest of the system.

$$\begin{aligned} x_1 &= 1 - x_2 - x_3 - x_4 \\ x_2 + -x_3 + 4 \cdot x_4 &= 1 \\ x_2 + x_3 &= 0 \end{aligned}$$

We now pick another variable in the second equality, say variable x_2. We rewrite the equality $x_2 = 1 + x_3 - 4 \cdot x_4$ and substitute variable x_2 in the equalities which

follow the second one, namely the last one.

$$\begin{aligned} x_1 &= 1 - x_2 - x_3 - x_4 \\ x_2 &= 1 + x_3 - 4 \cdot x_4 \\ 2 \cdot x_3 - 4 \cdot x_4 &= -1 \end{aligned}$$

We pick variable x_3 for the last equality. There is no equality remaining where substitution should be performed.

$$\begin{aligned} x_1 &= 1 - x_2 - x_3 - x_4 \\ x_2 &= 1 + x_3 - 4 \cdot x_4 \\ x_3 &= -\frac{1}{2} + 2 \cdot x_4 \end{aligned}$$

This procedure illustrates the reason I am talking about minimisation, as opposed to building a canonical form: the result depends on the choice of variable at each step and the order in which the equality constraints appear.

Maintaining minimality. When adding a new equality to a system in row echelon form, maintaining the minimised form is easy. The equality is rewritten by substituting all the defined variables, x_1, x_2 and x_3 in our example. It results in one of three cases.

Contradiction. The result of the substitutions is a trivially contradictory equality, such as $0 = 1$. In this case, the addition of the equality has made the polyhedron empty.

Redundancy. The result is a trivial equality, such as $1 = 1$. It is implied by the others and can be discarded without losing any information.

New definition. One of the variables remaining in the rewritten equality is chosen to be defined. The new definition is simply appended to system p_E, which perserves the row echelon form.

Using the row echelon form, instead of a dictionary similar to that maintained by the simplex algorithm, avoids the second phase of Gaussian elimination, which builds the reduced row echelon form. This saves some computation at the expense of enforcing an ordering on the equalities.

3.1.3 Invariants of the representation of inequalities

The set of inequalities p_I of a VPL polyhedron p also satisfies invariant properties related to minimality.

Full and decreased dimensionality. To start with, the variables defined in the set of equalities p_E are always substituted in the constraints of set p_I. What's more, there is no constraint $c(x) \geq 0$ implied by the constraints of set p_I, such that constraint $c(x) \leq 0$ is also implied. The conjunction of these two facts implies that the system of inequalities of a polyhedron

- always has full dimension: there are no implicit equalities, and

- always mentions the lowest possible number of variables.

Full dimensionality is a prerequisite for some of the linear programming algorithms discussed in the second part of this thesis. When it isn't a prerequisite, full dimensionality simplifies implementation and makes algorithms whose complexity depends on the number of variables more efficient. As mentioned before, projection is one of them. Inclusion testing is another. Suppose we need to test inclusion $p \sqsubseteq c(x) \geq 0$. Constraint $c(x) \geq 0$ is first rewritten using the equalities of p_E, yielding constraint $c'(x) \geq 0$. The linear solver described in the previous chapter then checks whether $p_I \wedge c'(x) < 0$ is satisfiable: there is no need to provide it with the equalities p_E.

Syntactic redundancy. Full dimensionality has another benefit: it gives the inequality constraints an almost-canonical representation, up to a positive scaling factor. A constraint $c(x) \geq 0$ can be arbitrarily rewritten as $\lambda \cdot c(x) \geq 0$, with $\lambda > 0$. However, how the equalities of the polyhedron are used to rewrite it is fixed by the choice of defined variables.

This property allows for a cheap syntactic criterion for testing whether inclusion $p \sqsubseteq c(x) \geq 0$ holds. Once constraint $c(x) \geq 0$ is rewritten using equalities p_E, yielding constraint $c'(x) \geq 0$, inclusion is proved if there exists a constraint $c_i(x) \geq 0$ from the set p_I such that $c'(x) - a \cdot c_i(x) \geq 0$, with $a > 0$. This is essentially saying that constraint $2 \cdot x \leq 2$ implies constraint $x \leq 3$. Redundant constraints found using the syntactic criterion just described are called *syntactic redundancies*.

Against canonicity. Enforcing a canonical representation of inequality constraints is tempting, as it would reduce syntactic redundancy elimination to a simple equality test. A possible implementation consists in selecting an order for variables and forcing the first nonzero coefficient of each constraint to be either 1 or -1. The problem with this approach is that it may scale the other coefficients of the constraints to large values, at the risk of overflowing machine representation. There is no risk for the correctness of the computations, since VPL uses arbitrary precision numbers [26]. However, arithmetic is cheaper when numbers fit in the machine representation. For this reason, canonicity isn't enforced, but factors common to all the coefficients of variables in a given constraint are removed: constraint $4 \cdot x_1 + 6 \cdot x_2 + 3 \geq 0$ is rewritten $2 \cdot x_1 + 3 \cdot x_2 + \frac{3}{2} \geq 0$.

Factorising work during minimisation. Once syntactic redundancies are removed, the minimisation procedure tests, for each constraint $c_i(x) \geq 0$ of the set p_I, whether it is implied by the other constraints of set p_I, that is whether inclusion $p_I \setminus \{c_i(x) \geq 0\} \sqsubseteq c_i(x) \geq 0$ holds. If it does, then constraint $c_i(x) \geq 0$ is redundant and can be removed. Although this is a costly operation, the specifics of the VPL linear problem solver help keeping it efficient.

As we saw in the previous chapter, the linear solver in VPL splits constraint $a_{i0} + \sum_{j=1}^{n} a_{ij} \cdot x_j \geq 0$ into the definition of an auxiliary variable $y_i = \sum_{j=1}^{n} a_{ij} \cdot x_j$ and a bound on this variable $y_i \geq -a_{i0}$. Also recall that the algorithm then preserves the invariant that the value of all the variables used in the definitions stays within their bounds, while the value of the dependent variables may not. Now, remember that the solver tests inclusion $p \sqsubseteq c_i(x) \geq 0$ by testing the satisfiability of the conjuction $p \wedge c_i(x) < 0$. Furthermore, note that transforming

constraint $c_i(x) \geq 0$ into constraint $c_i(x) < 0$ only changes the bound on the internal variable y_i associated to it. If we make sure that variable y_i is a dependent variable, we can make the change without rebuilding the linear problem from scratch. This leads naturally to the following approach.

- Build the linear satisfiability problem for constraints $p_I \triangleq \bigwedge_{i=1}^{l} c_i(x) \geq 0$. Note that this step introduces an auxiliary variable per constraint.
- Solve the problem. If it is infeasible, we know that $\bot$ is the result of the minimisation. Otherwise, we have found a point $\underline{x}$ which satisfies all the constraints.
- For each constraint $c_i(x) \geq 0$, perform the following steps.
  - Make sure that the corresponding auxiliary variable y_i is a defined variable. If it is not, pivot to make it so.
  - Change the bound on variable y_i from $y_i \geq -a_{i0}$ to $y_i < -a_{i0}$.
  - Solve the new problem. Note here that only variable y_i is out of its bounds. If the new problem is unsatisfiable, then constraint $c_i(x) \geq 0$ is redundant.

The test for each constraint starts from the same solved problem resulting from the second step. Since VPL is written in the functional subset of OCAML the copying involved is transparent.

This way of performing minimisation saves considerable duplication of initialisation work. However, the theoretical benefit is hard to evaluate since the number of pivoting steps required to solve the new problem for each constraint heavily depends on both the geometry of the input polyhedron and the solution point we start from.

3.1.4 Interaction between minimisation and verification

Now that we have spelt out how polyhedra are represented by the oracle of VPL, it may be worth taking a step back and consider how this affects the frontend and the proofs which we have discussed in chapter 1. The frontend represents a polyhedron as the set of its constraints, without making any further assumption, such as emptiness or minimality. We argued that the main interest of using a result verification approach lies in the loose coupling between the frontend and the oracle. To back this claim, we showed that hardly more than the coefficents of linear combinations of constraints need to be transferred between the two. The "hardly" in the previous sentence is due to operators which introduce new constraints, such as the intersection $p \sqcap c(x) \geq 0$. Constraint $c(x) \geq 0$ needs to be transfered to the oracle.

Whether the linear combinations specify a minimised representation of the resulting polyhedron is entirely irrelevant to the soundness of the result. Soundness follows from each resulting constraint verifying a specific inclusion property. However, minimisation has an impact on the computational efficiency of the whole analyser. Obviously, it spares useless computations on the redundant constraints. It also detects as soon as possible whether the execution path under consideration by the analyser is infeasible and can therefore be ignored.

Overall, the engineering impact of minimising representations only affects the oracle. It comes for free in the frontend when the witnesses provided by the oracle describe a minimised polyhedron.

3.2 Generating witnesses on the fly

The oracle is responsible for computing the results of the abstract domain operators and for providing the frontend with witnesses of the soundness of these results. These two aspects have been dealt with separately in preceeding attempts [6] to prove the correctness of an abstract domain of polyhedra using a proof assistant. One of my contributions with VPL resides in showing that the witnesses can be extracted with a low overhead from the working data of the operators of the domain.

3.2.1 Inclusion test

Previous chapter already gave most of the information about how to build a witness for the inclusion $p \sqsubseteq c(x) \geq 0$, once the linear solver declared that $p \wedge c(x) < 0$ is unsatisfiable. Two minor pieces are still missing, however: the handling of equalities and the syntactic test.

Constraint $c(x) \geq 0$ is rewritten as constraint $c'(x) \geq 0$ using the set of equalities $p_E \triangleq \{e_i(x) = 0 \,|\, i \in \{1, \ldots, k\}\}$ to substitute a subset of the variables.

$$c'(x) \triangleq c(x) + \sum_{i=1}^{k} \lambda_i \cdot e_i(x) = c(x) + \Lambda_E \cdot p_E$$

Note that Λ_E applies to equalities: there is no sign restriction for its coefficients. Suppose now that constraint $c'(x) \geq 0$ is then implied by the set of inequalities $p_I \triangleq \{c_i(x) \geq 0 \,|\, i \in \{1, \ldots, l\}\}$.

$$c'(x) = \lambda'_0 + \sum_{i=1}^{l} \lambda'_i \cdot c_i(x) \quad \text{with } \forall i \in \{1, \ldots, l\},\ \lambda'_i \geq 0$$

Although it leaves coefficient λ'_0 implicit, we rewrite this equality as follows.

$$c'(x) = \Lambda_I \cdot p_I$$

The inclusion witness for constraint $c(x) \geq 0$ is then rebuilt.

$$c(x) = \Lambda_I \cdot p_I + -\Lambda_E \cdot p_E = \Lambda \cdot p$$

Again, negating coefficients Λ_E isn't an issue since they apply to equality constraints. Deciding the inclusion $p_I \sqsubseteq c(x) \geq 0$ based on the syntactic criterion presented before is no more than a special case where only one coefficient λ'_i, $i \in \{1, \ldots, l\}$ is nonzero.

If the inclusion we are testing had been $p \sqsubseteq e(x) = 0$, the answer would have followed from the initial rewriting.

$$e'(x) \triangleq e(x) + \Lambda_E \cdot p_E$$

If the result $e'(x)$ is syntactically 0, then inclusion holds and $-\Lambda_E$ makes an inclusion witness.

$$e(x) = -\Lambda_E \cdot p_E$$

If $e(x)$ is different from 0, then inclusion doesn't hold. Note that testing whether inclusion $p_I \sqsubseteq e'(x) = 0$ is useless, since we maintain the invariant property that there is no implicit equality in the set p_I.

3.2.2 Intersection

Intersection of polyhedron p with a new constraint c can be trivially implemented by appending the new constraint to the constraints already constituting the involved polyhedron. However, doing so doesn't yield a minimised representation of the result. The intersection operator in VPL starts by testing whether $p \sqsubseteq c$. If the added constraint is found to be implied by the input polyhedron, polyhedron p is returned unaffected. The witness for inclusion $p \cup \{c\} \sqsubseteq p$ is obvious to build. If polyhedron p doesn't imply the new constraint, the next step depends on constraint c being an equality or an inequality.

Nonredundant equality, $c \triangleq e(x) = 0$. We wish to compute the result of intersection $p \sqcap e(x) = 0$ and we kept the rewritten constraint $e'(x) = e(x) + \Lambda_E \cdot p_E$ from the inclusion test. The new equality $e'(x) = 0$ is added to the set p_E as described when we discussed minimality, yielding the set p'_E. Next, the variable defined by equality $e'(x) = 0$ is substituted in the set p_I. The resulting set p'_I isn't minimised and, more importantly, it may now contain implicit equalities.

Looking for implicit equalities. VPL looks for implicit equalities in a set of inequalities p'_I in the simplest way possible. For each constraint $c_i(x) \geq 0$, it checks whether $p'_I \sqsubseteq c_i(x) \leq 0$. If one inclusion holds, the new equality $c_i(x) = 0$ is added to the set p'_E. Rewriting and implicit equality search is started over. Note that, when testing all the inclusions, work can be factorised in a similar fashion to what we saw when removing redundancy from a set of inequalities with no implicit equality.

Once all the implicit equalities have been discovered, added to the set of equalities, and after rewritings have been performed, the transformed set of inequalities is minimised as explained in previous section.

Nonredundant inequality, $c \triangleq c(x) \geq 0$. We wish to compute the result of intersection $p \sqcap c(x) \geq 0$ and we kept the rewritten constraint $c'(x) = c(x) + \Lambda_E \cdot p_E$ from the inclusion test $p \sqsubseteq c(x) \geq 0$. We may now look for implicit equalities in the set $p_I \cup \{c'(x) \geq 0\}$. However, since the set p_I has no implicit equality, if there is an implicit equality in set $p_I \cup \{c'(x) \geq 0\}$, it must be that $p_I \sqsubseteq c'(x) \leq 0$. If that inclusion doesn't hold, the search stops.

Otherwise, we just found implicit equality $c'(x) = 0$. Even if only one new inequality constraint is added, there may be others. Suppose that the set p_I equals to $\{x \leq y,\ y \leq z\}$ and that we add constraint $z \leq x$. Two implicit equalities are introduced: $x = z$ and $y = z$.

Again, once implicit equalities have been discovered, added to the set of equalities, and after rewritings have been performed, the transformed set of inequalities is minimized.

Generating a witness. In theory, generating a witness for intersection is very easy. Most of what we did is rewriting inequality constraints using equalities. We know what coefficients were used since we explicitely computed them. In theory, all that is needed is to remember these coefficients and the constraints they apply to. This is made manageable by the constraints being uniquely identified by an integer, for the purpose of communicating with the frontend.

However, there are two details to pay some attention to. First, some equalities $e(x) = 0$ have once been implicit and a witness for both $e(x) \leq 0$ and $e(x) \geq 0$ should be kept. This is the rationale behind the SplitEq witness constructor from chapter 1, which you may recall has the following type.

$$\mathsf{SplitEq} : \mathsf{linComb} \to \mathsf{linComb} \to \mathsf{consWitness}$$

The other detail is the sign of the coefficients applying to equalities, as illustrated in the case of the inclusion test above.

Now, consider the following worst-case scenario for intersection $p \sqcap c(x) \geq 0$. I will use p_{I1}, $p_{I2}, \ldots$ instead of p'_I, $p''_I, \ldots$.

1. The starting point is $p_{I0} \triangleq p_I \cup \{c(x) \geq 0\}$.
2. Constraint $c(x) \geq 0$ is rewritten using p_E, yielding $p_{I1} \triangleq p_I \cup \{c'(x) \geq 0\}$.
3. An implicit equality $c'(x) = 0$ is found and added to the set p_E, which becomes $p_{E1} \triangleq p_E \cup \{c'(x) = 0\}$.
4. Equality $c'(x) = 0$ is used to rewrite the set p_{I1}, yielding p_{I2}.
5. Inequality set p_{I2} is minimised, yielding the final set p_{I3}.

Introducing binding. One way to generate a witness in this situation is to compose a sequence of witnesses. If the example above, let us define a number of intermediate polyhedra: $p_0 \triangleq (p_{I0}, p_E)$, $p_1 \triangleq (p_{I1}, p_E)$, $p_2 \triangleq (p_{I1}, p_{E1})$, $p_3 \triangleq (p_{I2}, p_{E1})$ and $p_4 \triangleq (p_{I3}, p_{E1})$. Suppose now that the intersection operator provides four witnesses: Λ for $p_0 \sqsubseteq p_1$, Λ' for $p_1 \sqsubseteq p_2$, Λ'' for $p_2 \sqsubseteq p_3$ and Λ''' for $p_3 \sqsubseteq p_4$. This is equivalent to saying that $p_1 = \Lambda \cdot p_0$, $p_2 = \Lambda' \cdot p_1$, $p_3 = \Lambda'' \cdot p_2$ and $p_4 = \Lambda''' \cdot p_3$. Therefore, $p_4 = \Lambda''' \cdot \Lambda'' \cdot \Lambda' \cdot \Lambda \cdot p_0$.

Given that a number of constraints are left untouched at each step—for example, set p_{I3} is a subset of set p_{I2}, meaning that witness Λ''' is a part of witness Λ''—I introduced a finer grain mechanism than composition of full witnesses. It appears as witness constructor Bind on listing 1.5 on page 33. A witness $\mathsf{Bind\ i}\ \Lambda_\mathsf{i}\ \Lambda'$ introduces an intermediate constraint c_i, which is the result of linear combination Λ_i of input constraints or previously introduced intermediate constraints. The newly introduced constraint c_i can be used like any other constraint in the remainder Λ' of the witness. The effect of the Bind above can be understood as follows when applied on polyhedron p.

$$\textbf{let}\ \mathsf{c_i} := \Lambda_\mathsf{i} \cdot\ \mathsf{p}\ \textbf{in}\ \Lambda' \cdot (\mathsf{p} \cup \{\mathsf{c_i}\})$$

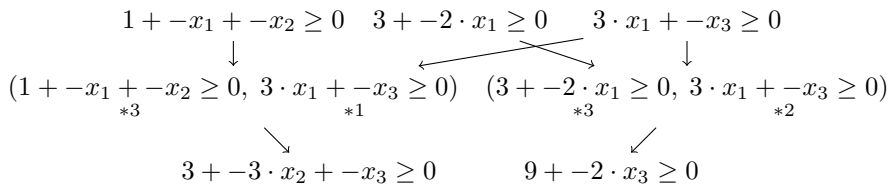


Figure 3.1 – Fourier-Motzkin elimination of variable x_1

Binding makes generating a witness for the result of an intersection easier. It is however not an ideal solution. The numbering of the bound constraints needs to be managed explicitly, which makes it error-prone and fragile since nothing prevents the witness generator from binding a given identifier to several constraints.

3.2.3 Projection

In constraint representation, the projection of an n-dimensional polyhedron p, whose dimensions are $x_1, \ldots, x_n$, on a subset of its dimensions, say $x_1, \ldots, x_m$ with $m < n$, is usually performed using repeated Fourier-Motzkin eliminations [25]. I used a refinement of this approach, originally published by Andy King and Axel Simon [47].

Eliminating one variable x_i

As a first approximation, we will consider that all the constraints of polyhedron p are inequalities. In order to eliminate variable x_i from polyhedron p, Fourier-Motzkin elimination partitions the inequalities of polyhedron p into three sets.

- $E^0_{x_i}$ contains the constraints where the coefficient of variable x_i is zero, meaning that variable x_i doesn't appear in these constraints.
- $E^+_{x_i}$ contains those having a strictly positive coefficient for variable x_i.
- $E^-_{x_i}$ contains those whose coefficient for variable x_i is strictly negative.

From these three sets, the result $p \backslash x_i$ of the projection is defined as follows.

$$p \backslash x_i \triangleq E^0_{x_i} \cup \left(\mathsf{map}\ \mathsf{elim}_{x_i}\ (E^+_{x_i} \times E^-_{x_i})\right)$$

The constraints of polyhedron p where variable x_i doesn't appear, that is the set $E^0_{x_i}$, end up directly in the result. Then, the algorithm builds all possible pairs of constraints such that one has a positive coefficient for variable x_i and the other has a negative coefficient for it. This is the set $E^+_{x_i} \times E^-_{x_i}$, which is then transformed by function map. Function map applies function elim_{x_i} on each element of the set $E^+_{x_i} \times E^-_{x_i}$. Function elim_{x_i}, when applied to two constraints $c^+(x) \geq 0$ and $c^-(x) \geq 0$, builds the linear combination $\lambda^+ \cdot c^+(x) + \lambda^- \cdot c^-(x)$, such that variable x_i doesn't appear in the result. The whole process is illustrated on figure 3.1.

Eliminating several variables

When several variables need to be eliminated, Fourier-Motzkin elimination removes them one by one, but does not enforce an elimination order. In practice, the order has a big impact on the representation size of the intermediate polyhedra. A common heuristic [47] for choosing an order of the projections consists in, at each step, choosing the variable which results in the smallest growth of the number of constraints.

The variation Δ_{x_i} in the number of constraints after the Fourier-Motzkin elimination of a single variable x_i is the following.

$$\Delta_{x_i} \triangleq |E^+_{x_i}| \cdot |E^-_{x_i}| - (|E^+_{x_i}| + |E^-_{x_i}|)$$

The left-hand side of the substraction is the number of pairs in set $E^+_{x_i} \times E^-_{x_i}$, each of which yields a constraint in the result. The right-hand side of the substraction is the number of constraints of the input polyhedron which are combined by the algorithm and therefore don't appear in the result. At each step, the heuristic chooses the variable x_i from the remaining variables which has the lowest Δ_{x_i}.

Using equality constraints

As we have seen already, each equality constraint can be used as a definition for a variable. The variable can them be substituted so as to reduce the dimension of the system of inequalities of the polyhedron. The same idea can be applied to eliminate a variable x_i. If variable x_i appears in an equality constraint, this equality is used as a definition for variable x_i and it is substituted in the rest of the polyhedron. The definition is then discarded. An elimination by substitution reduces the size of the representation of the affected polyhedron by one constraint. Therefore, when performing multiple eliminations, those which can be performed by substitution are carried out first.

Redundancy elimination

Fourier-Motzkin elimination can generate a lot of redundant constraints, which make the representation size of the result unwieldy. In the worst case, the l constraints of input polyhedron p split evenly into $E^+_{x_i}$ and $E^-_{x_i}$, and thus, after one elimination, one gets $l^2/4$ constraints. This yields an upper bound of $l^{2^n}/4^n$ constraints where n is the number of eliminated variables. Yet, the number of nonredundant constraints can only grow in single exponential [45]. As a result, most generated constraints are likely to be redundant.

To avoid the overwhelming growth of the number of constraints, a representation minimisation is performed after each variable elimination. Minimising the result of a projection is simpler than minimising that of an intersection, since no implicit equality can be introduced. Once the initial eliminations by substitution have been performed, all the work focuses on the system of inequalities. Its minimisation occurs in two steps: syntactic redundancy elimination first, and simplex-based elimination next.

Kohler's criterion

The work [47] which the projection operator of VPL is based on used another redundancy elimination technique prior to resorting to the costly simplex-based method. That technique applies a result from David Kohler [39] which states that, when you eliminate n variables from a polyhedron p, resulting in polyhedron p', each nonredundant constraint of polyhedron p' is a linear combination of at most $n + 1$ constraints of p.

It turns out that this criterion is incompatible with performing a minimisation after each elimination step, as the following example shows. Suppose we start with the following constraint system.

$$\begin{aligned} -x_1 + -x_2 + x_3 &\leq 0 \qquad (1)\\ 2 \cdot x_1 + x_2 + x_4 &\leq 0 \qquad (2)\\ 2 \cdot x_1 + x_2 + -x_4 &\leq 0 \qquad (3)\\ -x_1 + x_2 + -x_4 &\leq 0 \qquad (4)\\ -x_1 + x_2 + x_4 &\leq 0 \qquad (5) \end{aligned}$$

Now, we eliminate variable x_2, recording where each constraint comes from.

$$\begin{aligned} -2 \cdot x_1 + x_3 + x_4 &\leq 0 \qquad (6) = (1) + (5)\\ -2 \cdot x_1 + x_3 + -x_4 &\leq 0 \qquad (7) = (1) + (4)\\ x_1 + x_3 + -x_4 &\leq 0 \qquad (8) = (1) + (3)\\ x_1 + x_3 + x_4 &\leq 0 \qquad (9) = (1) + (2) \end{aligned}$$

Next, we eliminate variable x_1 in a similar way.

$$\begin{aligned} 3 \cdot x_3 + -x_4 &\leq 0 \qquad (10) = (6) + 2 \cdot (8)\\ x_3 + -x_4 &\leq 0 \qquad (11) = \frac{1}{3} \cdot (6) + \frac{2}{3} \cdot (9)\\ x_3 + x_4 &\leq 0 \qquad (12) = \frac{1}{3} \cdot (7) + \frac{2}{3} \cdot (9)\\ 3 \cdot x_3 + x_4 &\leq 0 \qquad (13) = (7) + 2 \cdot (9) \end{aligned}$$

Constraints (10) and (13) are implied by constraints (11) and (12).

- $(10) = 2 \cdot (11) + (12)$
- $(13) = (11) + 2 \cdot (12)$

Unfolding the linear combinations so that they refer to the original constraints, the system now is as follows.

$$\begin{aligned} x_3 + -x_4 &\leq 0 \qquad (11) = (1) + \frac{2}{3} \cdot (3) + \frac{1}{3} \cdot (4)\\ x_3 + x_4 &\leq 0 \qquad (12) = (1) + \frac{2}{3} \cdot (2) + \frac{1}{3} \cdot (5) \end{aligned}$$

Last, we eliminate variable x_4.

$$x_3 \leq 0 \qquad (14) = \frac{1}{2} \cdot (11) + \frac{1}{2} \cdot (12)$$

If we unfold the linear combination, we find the following.

$$(14) = (1) + \frac{1}{3} \cdot (2) + \frac{1}{3} \cdot (3) + \frac{1}{6} \cdot (4) + \frac{1}{6} \cdot (5)$$

Three variables have been eliminated and constraint $x_3 \leq 0$ is the linear combination of five input constraints. Constraint $x_3 \leq 0$ is obviously nonredundant. However, by Kohler's criterion, it should be deemed so, as it is the linear combination of five constraints and we have eliminated three variables. Had we not removed redundant constraints (10), we would have produced constraint $x_3 \leq 0$ from $\frac{1}{4} \cdot (10) + \frac{1}{4} \cdot (12)$. Unfolding, we find the following, on which Kolher's criterion doesn't apply.

$$x_3 \leq 0 \qquad \frac{1}{4} \cdot (10) + \frac{1}{4} \cdot (12) = (1) + \frac{1}{6} \cdot (2) + \frac{1}{2} \cdot (3) + \frac{1}{3} \cdot (5)$$

This example shows that Kolher's criterion is incompatible with removing redundant constraints after each variable elimination. Therefore, how to leverage Kohler's criterion in an efficient projection implementation remains unclear.

Witness generation

Generating an inclusion witness $p \sqsubseteq p \backslash x_i$ for the elimination of variable x_i from polyhedron p is illustrated on figure 3.1, on page 74. There, the coefficients of the linear combinations yielding the resulting constraints are given below each pair of input constraints. The example of the left pair is reproduced below.

$$\mathbf{3} \cdot (1 + -x_1 + -x_2) + \mathbf{1} \cdot (3 \cdot x_1 + -x_3) \;=\; 3 + -3 \cdot x_2 + -x_3$$

The coefficients, 3 and 1, are explicitely computed by function elim_{x_i} in order to compute the result of the elimination. All it takes to build a witness is to keep them along with the resulting constraint.

VPL builds witnesses for the result of multiple eliminations by annotating each constraint $c_i(x) \geq 0$ of the input polyhedron p with its witness, forming a pair $\big(c_i(x) \geq 0,\, 1 \cdot (i)\big)$. The functions operating on constraints are lifted so that they operate on pairs. Function mult, which multiplies a constraint $c_i(x) \geq 0$ by a positive constant a, becomes function mult^w below.

$$\begin{aligned} \mathsf{mult}\ \ a\ \ c_i(x) \geq 0 \quad &\triangleq \quad a \cdot c_i(x) \geq 0 \\ \mathsf{mult}^w\ a\ \ \big(c_i(x) \geq 0, \lambda_i\big) \quad &\triangleq \quad \big(a \cdot c_i(x) \geq 0,\, a \cdot \lambda_i\big) \end{aligned}$$

Function add, which adds two constraints $c_i(x) \geq 0$ and $c_j(x) \geq 0$, becomes function add^w below.

$$\begin{aligned} \mathsf{add}\ \ c_i(x) \geq 0\ \ c_j(x) \geq 0 \quad &\triangleq \quad c_i(x) + c_j(x)\ \geq 0 \\ \mathsf{add}^w\ \big(c_i(x) \geq 0,\, \lambda_i\big)\ \ \big(c_j(x) \geq 0,\, \lambda_j\big) \quad &\triangleq \quad \big(c_i(x) + c_j(x)\ \geq 0,\, \lambda_i + \lambda_j\big) \end{aligned}$$

No further adjustment than this small change is required to generate witnesses for projection results.

3.2.4 Assignment

We saw in chapter 1 that VPL handles assignment entirely in the frontend, through a functor. However, an earlier version used a witness-generating assignment operator from the oracle, which we will discuss now. Additionally, the following discussion supports the claim that the oracle itself is a complete OCaml abstract domain of polyhedra, which can be used independently of the Coq part of VPL. Although we overlooked the fact until now, assignment comes in two flavors: invertible and non-invertible.

Invertible assignment

An assignment $p' = p[x_i := e]$ is invertible if it is possible to compute back polyhedron p from polyhedron p'. This is the case when the assigned-to variable x_i appears in expression e. Suppose the assignment under consideration is $p' = p[x_i := x_i + 1]$. It can be inverted as $p = p'[x_i := x_i - 1]$. Note that we're assuming that expression e is linear here: assignment $x_i := x_i^2$ can't be inverted, although variable x_i appears on both sides.

In the invertible case, the assignment operator is defined as described before, an intersection, a projection and a renaming.

$$p[x_i := e] \triangleq \big((p \sqcap x'_i = e)\backslash x_i\big)[x'_i/x_i]$$

The corresponding witness is actually a pair of witnesses: one for the intersection and one for the projection.

Non-invertible assignment

It is safe to assume that all assignments are invertible: we aren't interested in actually inverting them. However, non-invertible assignment can be handled more efficiently. An assignment $p' = p[x_i := e]$ is non-invertible when the assigned-to variable x_i doesn't appear in expression e. For example, $x_1 := 5$ and $x_1 := 2 \cdot x_2$ are non-invertible assignments. There is no relation between the value of variable x_i before and after the assignement and, therefore, there is no need to introduce a fresh variable to be able to refer to the two at the same time.

The non-invertible assignment operator is defined as follows.

$$p[x_i := e] \triangleq p\backslash x_i \sqcap x_i = e$$

Variable x_i as it is before the assignment isn't needed any more: it is eliminated first. The result of the elimination is then intersected with $x_i = e$.

Non-invertible assignment is faster for two reasons. First, there is no need to perform a renaming. Then, computing the intersection is trivial: variable x_i doesn't appear in $p\backslash x_i$. Equality $x_i = e$ is rewritten using the set p_E of equalities of polyhedron p and then used to define variable x_i. Since variable x_i doesn't appear in the inequalities of polyhedron p, no rewriting or minimisation needs to be performed.

The witness for a non-invertible assignment is also a pair of witnesses: one for the projection and one for the intersection. Also note that the selection between invertible and non-invertible assignment is performed automatically by VPL in an encapsulating assignment operator.

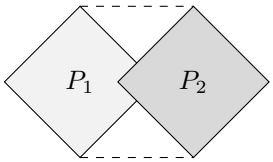


Figure 3.2 – disjunction and convex hull of polyhedra

3.2.5 Convex hull

The join operator of the abstract domain of polyhedra approximates the disjunction $P' \vee P''$ by the convex hull $P' \sqcup P''$. Figure 3.2 illustrates the need for overapproximating: the disjunction of polyhedra P_1 and P_2, that is to say their union, is the whole shaded area. However, this area isn't convex: it can't be represented by a conjunction of linear constraints. Instead, the union of polyhedra P_1 and P_2 is overapproximated by the convex hull of the two polyhedra, which adds the two areas delimited by the dashed lines. Computing a convex hull is very simple when generator representation is used. For constraint representation, this is the problematic operator. Florence Benoy, Andy King and Fred Mesnard published a method for reducing the convex hull to a projection [5], thereby providing an algorithm for computing convex hulls in constraints-only representation.

The convex hull $P \triangleq P' \sqcup P''$ is the smallest polyhedron containing all line segments joining points of polyhedron P' and points of polyhedon P''. In other words, a point $x \in P$ is a convex combination of a point $x' \in P' \triangleq b' + A' \cdot x' \geq 0$ and a point $x'' \in P'' \triangleq b'' + A'' \cdot x'' \geq 0$.

$$x = \alpha' \cdot x' + \alpha'' \cdot x'', \quad \text{with } \alpha' \geq 0,\ \alpha'' \geq 0 \text{ and } \alpha' + \alpha'' = 1$$

Adding the constraints on points x' and x'', the points of the convex hull satisfy the following set p_H of constraints.

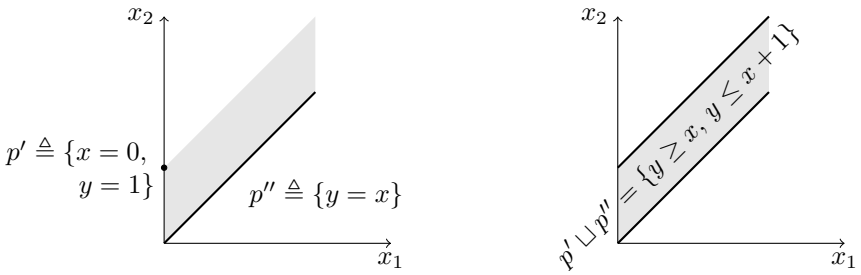
$$p_H \triangleq \left\{ \begin{array}{r} b' + A' \cdot x' \geq 0 \\ b'' + A'' \cdot x'' \geq 0 \\ x = \alpha' \cdot x' + \alpha'' \cdot x'' \\ \alpha' + \alpha'' = 1 \\ \alpha' \geq 0 \\ \alpha'' \geq 0 \end{array} \right\}$$

Then, the constraints p of convex hull result from the following projection.

$$p \triangleq p_H \backslash \{x', x'', \alpha', \alpha''\}$$

The set of variables is triplicated. The original dimensions x are those of the result. The constraints of operand p' bound variables in the x' space and the constraints of operand p'' bound variables in the x'' space. If n is the original number of variables, the problem above builds a constraint system bounding $3 \cdot n + 2$ variables and eliminates $2 \cdot n + 2$ of them: x', x'', α' and α''.

However, the projection problem we have just seen can't be solved using the projection operator for polyhedon: constraint $x = \alpha' \cdot x' + \alpha'' \cdot x''$ is nonlinear.



(a) convex hull (b) $p' \sqcup p''$

Figure 3.3 – The convex hull of p' and p'' can't be represented as constraints.

In order to solve this issue, a change of variables is performed.

$$y' \triangleq \alpha' \cdot x'$$
$$y'' \triangleq \alpha'' \cdot x''$$

The set of constraints $b' + A' \cdot x' \geq 0$ is multiplied by α' and the set of constraints $b'' + A' \cdot x'' \geq 0$ is multiplied by α''. Introducing the new variables y' and y'' leads to the following definition for p_H, which is now a polyhedron.

$$p_H \triangleq \left\{ \begin{array}{r} b' \cdot \alpha' + A' \cdot y' \geq 0 \\ b'' \cdot \alpha'' + A'' \cdot y'' \geq 0 \\ x = y' + y'' \\ \alpha' + \alpha'' = 1 \\ \alpha' \geq 0 \\ \alpha'' \geq 0 \end{array} \right\}$$

Closure of the convex hull

Polyhedron $p_H \backslash \{x', x'', \alpha', \alpha''\}$ actually isn't the *convex hull* of polyhedra p' and p''. This fine point is best illustrated on a example, which I take from the original publication on this method [5].

Consider polyhedra $p' \triangleq \{x = 0, y = 1\}$ and $p'' \triangleq \{y = x\}$ shown on figure 3.3(a). Both are closed sets and are represented using closed linear constraints, that is there is no strict inequality. Their convex hull is represented as the shaded area on figure 3.3. It includes point (0,1), but it excludes the half-line $\{y = x + 1, x > 0\}$. The convex hull of polyhedra p' and p'' can't be represented using linear constraints.

In such a case, the join operator of the abstract domain of polyhedra computes the *closure* of the convex hull. On our example, $p' \sqcup p'' = \{y \geq x, y \leq x + 1\}$, as shown on figure 3.3(b).

Extracting witnesses

Knowing how to compute a convex hull using projections, we may have a look at how witnesses for a convex hull can be built from witnesses of projections. Polyhedron $p \triangleq p_H \backslash \{x', x'', \alpha', \alpha''\}$ is a sound result of the join $p' \sqcup p''$ if both

inclusions $p' \sqsubseteq p$ and $p'' \sqsubseteq p$ hold. The projection operator provides us with a witness Λ for the inclusion $p_H \sqsubseteq p$, such that $p = \Lambda \cdot p_H$. The equality $p = \Lambda \cdot p_H$ is true for any value of variables x, x', x'', α' and α'' satisfying the constraints of polyhedron p_H. Let us choose the following values: $\alpha' = 1$, $\alpha'' = 0$, $y'' = 0$ and call σ the substitution $[\alpha'/1, \alpha''/0, y''/0]$. Polyhedron $(p_H)\sigma$ is as follows.

$$(p_H)\sigma \triangleq \left\{ \begin{array}{r} (b' \cdot \alpha' + A' \cdot y' \geq 0)\sigma \\ (b'' \cdot \alpha'' + A'' \cdot y'' \geq 0)\sigma \\ (x = y' + y'')\sigma \\ (\alpha' + \alpha'' = 1)\sigma \\ (\alpha' \geq 0)\sigma \\ (\alpha'' \geq 0)\sigma \end{array} \right\} = \left\{ \begin{array}{r} b' + A' \cdot y' \geq 0 \\ 0 \geq 0 \\ x = y' \\ 1 = 1 \\ 1 \geq 0 \\ 0 \geq 0 \end{array} \right\}$$

We may now note that, substituting variable y' by variable x, as they are equal, and removing the trivial constraints, we obtain the constraints of polyhedron p'. Let us note this equivalence $(p_H)\sigma \approx p'$. Note now that the variables for which we chose a value don't appear in polyhedron p. Therefore, we have the following.

$$p = (p)\sigma = \Lambda \cdot (p_H)\sigma \approx \Lambda \cdot p'$$

In a similar way, we may choose values $\alpha' = 0$, $\alpha'' = 1$, $y' = 0$ and obtain $p = \Lambda \cdot p''_H$, where the constraints of p''_H are the constraints of p'' with some extra trivial constraints.

Let us now consider on constraint $c_i(x) \geq 0$ of polyhedron p, under the light of the preceeding construction. We wish to obtain two witnesses Λ'_i and Λ''_i for the two inclusions $p' \sqsubseteq c_i(x) \geq 0$ and $p'' \sqsubseteq c_i(x) \geq 0$. Focusing on the first inclusion, we can decompose witness Λ_i for inclusion $(p_H)\sigma \sqsubseteq c_i(x) \geq 0$ into three subsets of coefficients.

Λ_{i1}	$(b' \cdot \alpha' + A' \cdot y' \geq 0)\sigma$
Λ_{i2}	$(b'' \cdot \alpha'' + A'' \cdot y'' \geq 0)\sigma$
Λ_{i3}	$(x = y' + y'')\sigma$
	$(\alpha' + \alpha'' = 1)\sigma$
	$(\alpha' \geq 0)\sigma$
	$(\alpha'' \geq 0)\sigma$

- Let Λ_{i1} be the subset of the coefficients of Λ_i applying to the constraints of polyhedron p' set in the x' space: $b' \cdot \alpha' + A' \cdot y' \geq 0$.
- Let Λ_{i2} be the subset of the coefficients of Λ_i applying to the constraints of polyhedron p'' set in the x'' space: $b'' \cdot \alpha'' + A'' \cdot y'' \geq 0$.
- Let Λ_{i3} be the coefficients of Λ_i applying to the constraints $x = y' + y''$, $\alpha' + \alpha'' = 1$, $\alpha' \geq 0$ and $\alpha'' \geq 0$.

Fragment Λ_{i1} applies to the constraints of p'. Fragment Λ_{i2} applies to constraints $0 \geq 0$ and can therefore be disregarded. Fragment Λ_{i3} applies to trivial constraints, only constraint $1 \geq 0$ has a nonzero contribution to $\Lambda_i \cdot (p_H)\sigma$. We

refer to its coefficient as λ_0. From these observation, equality $c_i(x) \geq 0 \;=\; \Lambda_i \cdot (p_H)\sigma$ can be rewritten as follows.

$$c_i(x) \geq 0 \;=\; \Lambda_{i1} \cdot p' + \lambda_0 \cdot (1 \geq 0)$$

This effectively shows that fragment Λ_{i1} is a witness for inclusion $p' \sqsubseteq c_i(x) \geq 0$. By a similar reasoning, fragment Λ_{i2} is a witness for inclusion $p'' \sqsubseteq c_i(x) \geq 0$. In the end, witnesses for the convex hull are simply extracted from the witness resulting from the projection $p_H \backslash \{x', x'', \alpha', \alpha''\}$.

3.3 Highlights and other features of VPL

The work covered in this chapter was carried out with Michaël Périn and David Monniaux and is described in a paper [24]. The main novelty introduced by VPL oracle is the generation of inclusion witnesses on the fly. We have shown that it takes little instrumentation and little extra computation to gather the necessary information. This makes the witness reconstruction step of prior work by Frédéric Besson et al. unnecessary.

Another contribution of my work is the realisation that Kohler's criterion for detecting some redundant constraints during projection is actually incompatible with minimising intermediate elimination results. This is rather unfortunate: VPL benchmark suite has over fifteen thousand projection problems and Kohler's criterion caused a single incorrect result.

The operators we have covered in this chapter are the standard abstract domain operators. However, VPL includes some extra operators for the user's convenience. Examples include support for parallel assignment and an operator for retrieving the lower and upper bounds of an arbitrary linear expression. Other people have contributed to VPL a linearisation method to soundly approximate nonlinear relations between program variables [9].

Summary in French

Tout comme la partie Coq de VPL, l'oracle est construit à partir de polyèdres représentés par contraintes. Ce choix est motivé par la recherche de simplicité et comme expérience à contrepied des implémentations matures du domaine abstrait des polyèdres, qui utilisent les deux représentations.

Les bonnes performances de VPL sont en partie dues à des choix de structures de données et des invariants associés. Le choix principal consiste à maintenir une représentation minimisée en permanence. La minimisation consiste à séparer les contraintes d'égalités des contraintes d'inégalités et éliminer les redondances dans chacun de ces sous-ensembles. Ainsi, aucune des contraintes d'un polyèdre ne peut être obtenue par combinaison linéaire d'autres contraintes du polyèdre. De plus, il n'y a aucune égalité implicite parmi les inégalités. Ces choix de représentation n'ont aucun impact sur la partie Coq de VPL, celui-ci ne dépendant pas de la minimalité des polyèdres qu'il construit.

Ce que nous avons décrit jusqu'à maintenant est suffisant pour construire un domaine abstrait complet, dans lequel l'oracle fait appel à l'algorithme du

simplexe après l'exécution de chaque opérateur pour construire le témoin. Les travaux inittiaux de Frédéric Besson et al. reposent sur cette approche. VPL améliore cet aspect de leur travail en montrant qu'une instrumentation légère de chaque opérateur permet d'extraire des témoins de ses données de travail. Par exemple, l'opérateur qui élimine une variable x d'un polyèdre P construit toutes les contraintes qui résultent de la combinaison linéaire à coefficients positifs de contraintes de P et qui donnent un coefficient nul pour x. Il suffit de conserver les coefficients utilisés par l'opérateur pour construire le témoin. Le surcoût en temps d'exécution résultant de la preuve formelle de VPL s'en trouve drastiquement réduit.

Chapter 4

Implementing and evaluating performance

VPL was built as part of project Verasco and is integrated to its collection of abstract domains [37]. Although this fact makes Verasco a complete analyser capable of using VPL as an abstract domain of polyhedra, it should be stressed that Verasco is above all a proof of concept for building the soundness proof of a static analyser using Coq. As a result, it suffers from scalability problems [37], even when using more efficient abstract domains, such as that of intervals. Using polyhedra makes the problem more severe, which lead me to consider another route for evaluating the performance of VPL.

4.1 Implementation size

VPL is made of six thousand lines of Ocaml and twelve thousand lines of Coq. Lines are counted using the standard Unix tool wc: the counts include both comments and blank lines. I made this choice since, if these have been inserted in a code file, it probably serves the good understanding of the code. In addition to the actual VPL code, I wrote an extensive test suite constituted of over one thousand and five hundred test cases, spanning over seven thousand and five hundred lines of Ocaml code. These tests are designed to help development and maintenance of the code by catching regressions. Finally, evaluating the performance of VPL required writing some glue C code in order to make VPL communicate with other abstract domains of polyhedra. This glue code represents two thousand lines.

VPL oracle comes with a thorough documentation. The text is found in the Ocaml interface files of the oracle and can be extracted using the Ocamldoc tool. The Coq frontend isn't documented beyond its Verasco interface.

4.2 Building programs and proofs with Coq

The Coq frontend described in chapter 1 is the result of a major rewriting of the simple checker which I wrote initially. The design of the two versions differs in the weight they assigned to the two fundamental concerns ruling a Coq

development: building a program and building a proof. The initial version—let me call it version 1—puts the emphasis on building a program. The rewrite, which I'll call version 2, puts the emphasis on building a proof.

Program first. Version 1 was designed similarly to how an equivalent OCaml program would be: each module would capture a concept, often with a main data structure. The functions in each module were decomposed so as to make the proof simple. In most cases, each function had a single accompanying correctness lemma, which followed its definition immediately.

The proofs were manual: no use was made of the automation features provided by Coq. On occasions, proofs broke the encapsulation provided by modules and depended on implementation details. As a result, changes in the algorithms and the data structures often required fixing details in the proofs.

Balancing these shortcomings, the core frontend was small, four thousand lines of Coq code, and easy to navigate.

Proof first. Version 2 was written in collaboration with Sylvain Boulmé. It is designed so as to make the proofs simpler and less brittle. It is built with functors, which make the proofs more modular. Indeed, a functor can be applied to any module which satisfies a given interface: it can't depend on implementation specifics. It builds on explicit logical interfaces. Version 2 also introduced proof automation to the frontend: proofs became shorter and more robust to change in implementation details.

While version 2 brought many benefits to the proof side of the frontend, the program is now harder to understand. Its code is scattered in functors and the abstract domain is defined by a sequence of nested functor applications.

Adding to this, the code size of the core of the frontend nearly doubled. "Core" here designates the subset of version 2 which is equivalent in functionality to version 1. The functor which provides an advanced guard operator, the framing functor and the functor which buffers renamings were added after the fact.

Lessons learnt. With the benefit of hindsight, the advanced proof architecture of version 2 is probably an overkill. The reason is mainly that VPL frontend is conceptually simple, both as a program and as a proof. Should more elaborate features be added to VPL, such as variable packing or disjunctive invariants, the weaknesses of version 1 would probably become painfully apparent. Furthermore, working on version 2 was a good way to learn about advanced proof construction methods, without the extra burden of a complex application.

4.3 The subtleties of performance evaluation

As I mentioned in the introduction to this chapter, drawing conclusions on the performance of VPL from the observation of how it performs when used by Verasco isn't satisfactory. Verasco isn't really tuned to exploit an abstract domain of polyhedra: it relies heavily on extracting value intervals from the abstract domain, which are a costly operation to perform on polyhedra. Still, I wished to compare experimentally VPL with mature implementations of the abstract domain of polyhedra, such as NewPolka and PPL. I also wished to

measure the cost of the inclusion checker: when it replays witnesses, it computes using an encoding of numbers as lists of bits, which I expected to be inefficient.

Performance comparison is a complex issue for a variety of reasons. The most obvious follows for abstract domains being only components of a bigger piece of software. Their performance varies a lot depending on their use case. Theoretical complexity doesn't really help either: all the algorithms have a very bad worst-case complexity which is rarely encountered in practice. As David Monniaux points out [44], randomly-generated polyhedra don't give a faithful evaluation: I needed a more realistic approach.

The ideal setting would have been to have an analyser designed to benefit from the abstract domain of polyhedra and capable of using VPL and other implementations, through a common interface. My wish was partially fulfilled by the existing Pagai analyser, written by Julien Henry, David Monniaux and Matthieu Moy [31]. Pagai uses the Apron [33] collection of abstract domains, meaning that it can use either NewPolka or PPL. I still had to solve the programming language problem: VPL is written in Ocaml, while NewPolka is written in the C programming language and Pagai and PPL use C++.

I first tried to extract traces of domain operator calls from the analysis of programs by Pagai, in simple abstraction interpretation mode, and then to replay them offline with each of the abstract domains I included in the comparison: for each call, the polyhedra operands were built and then, the operator was called. Preliminary experiments exhibited very bad performance for NewPolka. This uncovered the issue of state for abstract domains which use the double representation of polyhedra. Depending on which representation is available, a conversion needs to be performed prior to executing the actual operator. The cost of this conversion is then amortised over several operators. Offline replay deprived double description-based domains from the possibility to amortise conversions. Since VPL uses only always-minimised constraint representation, the performance of its operators is independent of such factors.

4.4 The experimental setting

In order to make sure that the operators of NewPolka and PPL are called on polyhedra which are in a realistic state, measurements of their performance was done *during* the analysis. My experimental setup is pictured on figure 4.1. It is composed of two programs: Pagai and a comparing program which I wrote in Ocaml. They communicate through a pipe. For a given benchmark C program, Pagai runs as usual, but each of its calls to an operator of the abstract domain, NewPolka or PPL, is intercepted. The execution time of the operator is measured and then a description of the problem, its result and the time measurement is sent to the comparator. The comparator rebuilds the operands and replays the operator using VPL functions. It measures the execution time of the VPL operator and then checks the result against the result it received. Once Pagai has finished the analysis, the comparator outputs the result of the comparison. The comparison was done for the following operations: assignment, convex hull, inclusion test and intersection. It was performed on a recent laptop, under the GNU/Linux operating system.

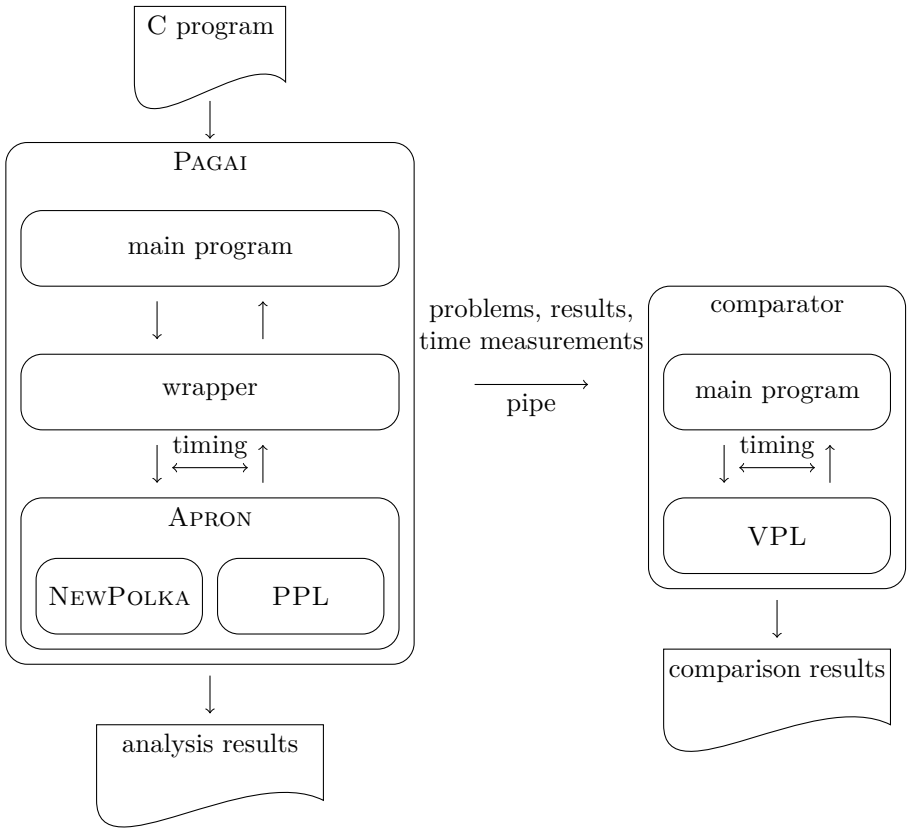


Figure 4.1 – experimental setup

Measuring time. In this setting, the overhead of the clock_gettime system call, which is used to perform the measurement, is sufficiently small for the timing of individual calls to yield meaningful results. This fact makes the measurement practically insensitive to variations due to the operating system. What's more, the measurements are made insensitive to the scheduling performed by the operating system by measuring the process execution time, rather than the real time elapsed during the execution of an operator. During the replay by VPL, measurements exclude the time needed to build the operands.

Checking results. The result of each of evaluated operators is a well-defined geometrical object. For every logged call, the results from NewPolka, PPL and VPL were checked for equality, through double inclusion. The witnesses generated by VPL were then systematically replayed by the frontend, so as to check that they yield the appropriate constraints. Furthermore, polyhedra have a minimal constraints representation, up to the variable choices in the substitutions of equalities. It was systematically checked whether VPL, NewPolka and PPL computed the same number of equalities and inequalities. In all the cases we tried, the tests of correctness and precision passed. It should be noted that PPL doesn't systematically minimise representations: its results often have redundant constraints. This is due to the lazy-by-default implementation of the operators of PPL. Since support for the eager version of the operators has been deprecated in and is being removed from PPL (see [10], § A Note on the Implementation of the Operators), we couldn't configure the library to have the same behavior as NewPolka.

Logging and measuring time online. While we elected to measure the execution time of NewPolka and PPL operators online, that is to say during the analysis by Pagai, one of primary goals was to modify neither Pagai nor Apron. A solution came from the wrapping functionality of the GNU linker ld. It makes possible to intercept calls to a shared library and execute an arbitrary function instead. In our setting, this function performs two tasks.

1. First, it calls Apron to perform the expected computation on unmodified operands. The time it takes is measured and recorded.

2. Then, it generates a text description of the operands, the result and the execution time and sends it to the comparator over a pipe. The comparator parses the description and replays the operator using VPL.

Benefits of the approach. I tried to give a feeling for how delicate a fair performance comparison is for abstract domains of polyhedra. The solution we have just outlined addresses all of the issues we raised.

First, it compares the operators of the domains on problems extracted from real analyses. The conclusion which we will draw from the evaluation will apply to realistic use cases. Then, the behaviour of abstract domains which use the double representation of polyhedra is very dependent on which representation is available when a given operator is called. Measuring the execution time of these domains in their original context, during the execution of the analyser, guarantees faithful information. On the other hand, VPL represents polyhedra as constraints only and minimises their representation eagerly. Its performance

doesn't depend on past calls. Therefore, replaying out of context doesn't make a difference.

Shortcomings. Despite its careful design, the method we have just described suffers from a number of shortcomings. Some of them may bias the outcome in favor of VPL, others may bias the outcome in favor of the domains we compare against.

- Pagai is tuned to use NewPolka. As such, some of the decisions behind the design of Pagai were guided by some trade-offs made by the authors of NewPolka. The most significant of these trade-offs is the use of a dense representation of vectors, which explicitly stores coefficients with value zero. This becomes costly when the constraint systems are very sparse. As a result, Pagai limits the number of variables it stores in the abstract values, which wouldn't be necessary for VPL, since it uses a semi-sparse representation.
- Next, the size of the vectors used to represent constraints in VPL depends logarithmically on the number of variables in the whole program under analysis. Our experimental setting only gives us access to a subset of the variables, namely the ones which are bounded by the polyhedra from NewPolka. In the event that this dependency to the global number of variables were a problem, the size of vectors in VPL could be made dependent only on the number of variables constrained by the polyhedron, through an environment similar to that of Apron.
- Then, we record the execution time of spent in the functions of the Apron shared library, which encapsulates both NewPolka and PPL. The benefit of accessing both transparently comes at the cost of a small overhead at runtime, due to the traversal of the Apron frontend.
- Last, it should be noted that the result of domain operators used by Pagai to continue the analysis aren't those of VPL but those of NewPolka or PPL. We make sure that the results of VPL are equivalent to the results of either NewPolka or PPL, but not syntactically equal. However, this shouldn't change anything to the analysis.

Benchmark programs. All but one of the programs on which I ran Pagai in order to obtain benchmark problems are taken from Pagai benchmark suite. It is itself extracted from a benchmark suite for worst-case execution time analysis. More precisely, I used the following programs.

- bf: the Blowfish cryptographic cipher
- bz2: the bzip2 compression algorithm
- dbz2: the bzip2 decompress algorithm
- jpg: an implementation of the jpeg codec
- re: the regular expression engine of GNU awk
- foo: a hand-crafted program leading to polyhedra with many constraints, large coefficients and few equalities

prog.	N	P	V	C
bf	0.2	0.4	0.1	0.1
bz2	1.6	2.8	0.7	1.2
dbz2	32.3	35.6	2.1	3.6
jpg	1.2	1.8	0.5	0.8
re	1.1	1.3	0.5	0.7
foo	0.2	0.2	0.9	0.9

(a) inclusion test

prog.	N	P	V
bf	10.7	13.4	1.2
bz2	52.3	61.1	7.9
dbz2	1687	1815	28.3
jpg	39.7	51.0	6.0
re	37.3	47.2	3.3
foo	6.7	7.1	5.5

(b) intersection

prog.	N	P	V	C
bf	3.2	1.2	2.7	2.8
bz2	23.5	11.5	66.8	68.7
dbz2	1393	231.9	532.8	535.3
jpg	28.2	7.5	24.0	24.9
re	20.2	8.4	17.9	19.2
foo	4.2	0.6	941.8	943.7

(c) convex hull

prog.	N	P	V
bf	3.7	11.4	0.5
bz2	14.6	54.1	2.9
dbz2	1618	4182	83.8
jpg	23.7	68.3	3.8
re	5.7	17.2	0.7
foo	9.2	14.8	8.5

(d) assignment

Table 4.1 – execution time in milliseconds: aggregation by program

4.5 Evaluation results and interpretation

Tables 4.1 and 4.2 show two views of the timing measurements recorded during the performance evaluation of VPL.

- Table 4.1 shows, for each of the evaluated operators, the time spent executing the operator during the analysis of each benchmark program.
- Table 4.2 shows the time spent executing each operator on problems of various sizes, summed over all the benchmark programs. This view informs about the typical distribution of problem sizes.

Reading the tables. All the time measurements reported on both tables are expressed in milliseconds. The column headers use the following single letter abbreviations to refer to the abstract domains.

N designates NewPolka.

P designates PPL.

V designates the oracle in VPL.

C designates VPL, including the frontend building the Coq polyhedron from the witness.

NewPolka and PPL were used in their default configuration. VPL has no configuration settings. Trivial problems, that are problems of size 0 and 1, weren't taken into account in the figures shown on table 4.1. The motivation for filtering is many of these problems are encountered during a run of Pagai. On these problems, PPL and NewPolka suffer from being called through the Apron frontend. I think the comparison is made fairer by this choice. Note that the second column of table 4.2, typeset in *italics*, doesn't contain time measurements, but counts of problems.

Problem size. I call "problem size" the total number of constraints in the inputs. Suppose we wish to compute the convex hull of two polyhedra p_1 and p_2. If polyhedron p_1 has n_1 constraints, mixing equality and inequality constraints, and if polyhedron p_2 has n_2 constraints, the computation of their convex hull is a problem of size $n_1 + n_2$. For an intersection $p_1 \sqcap c(x) \geq 0$, the size of the problem is $n_1 + 1$. This definition of "problem size" is convenient, as associates a single number to each problem. It is also reasonable since the behaviour of all of the operators mostly depends on the number of constraints involved.

Interpreting the results

The results presented in tables 4.1 and 4.2 show that VPL is efficient on small problems. Yet, the performance gap between VPL and the other implementations closes on bigger problems. This is especially true for the convex hull, which is the costliest operation on constraint representation. At least part of the difference in efficiency on small problems can be explained by the generality Apron provides: it provides a unified interface to several abstract domains at the expense of an extra abstraction layer. The induced overhead is more significant on small problems.

More generally, the use of ZArith in VPL is likely to lower the cost of arithmetic when compared to NewPolka and PPL, which use GMP directly. Indeed, ZArith uses native machine numbers as long as computations don't overflow, at which time it switches to GMP numbers. Program `foo` suggests this has an impact on performance: the analysis creates constraints with big coefficients, likely to overflow native number representation and the performance of the three domains is similar, except in the case of the convex hull. However, precise measurement of the effect of using ZArith would be a hard task.

Last, table 4.2 seems to show that problems are most often of rather small size. This certainly accounts for the big numbers obtained on the `dbz2` benchmark, which is the largest of the benchmark programs. As a result, the effect of better performance on small problems is amplified when there are many of them. However, this observation may be an artifact of using Pagai, or of using it configured to perform simple abstract interpretation.

Overall, these results seem promising for a constraints-only implementation of the abstract domain of polyhedra. They also argue in favor of using a higher-level language than C or C++ for implementing abstract domains: VPL achieves competitive performance in spite of being written in Ocaml. Some progress still needs to be made on the convex hull side. It is also interesting to notice the performance differences between NewPolka and PPL. Over all the operators, the performance of the two is similar, but there are big

size	*n*	N	P	V
0–1	*1482*	7.2	6.5	0.6
2–5	*1881*	9.7	12.8	1.6
6–10	*673*	9.7	10.6	1.3
11–15	*277*	3.3	4.2	0.5
16–20	*111*	5.8	7.0	1.0
21–25	*52*	4.0	3.9	0.3
26–30	*17*	4.0	3.4	0.1
31+	*4*	0	0	0

(a) inclusion test

size	*n*	N	P	V
0–1	*11458*	1389	1933	35.0
2–5	*4094*	1752	1740	30.9
6–10	*322*	52.3	158.6	18.4
11–15	*156*	27.4	91.4	8.8
16–20	*6*	1.3	4.8	0.6

(b) intersection

size	n	N	P	V
0–1	*3354*	687.9	167.5	7.0
2–5	*3373*	679.7	141.0	57.1
6–10	*1092*	434.1	68.4	133.7
11–15	*354*	119.5	22.8	131.2
16–20	*135*	68.8	16.8	1050
21–25	*65*	37.9	9.2	106.4
26–30	*14*	6.4	1.9	50.1
31+	*7*	3.5	0.9	27.8

(c) convex hull

size	*n*	N	P	V
0–1	*539*	33.8	47.5	1.1
2–5	*667*	601.8	1176	6.6
6–10	*381*	385.4	519.7	14.3
11–15	*58*	20.9	87.4	10.7
16–20	*64*	78.3	247.6	5.2
21–25	*480*	537.4	2111	39.2
26–30	*30*	59.5	81.7	15.2
31+	*16*	13.1	77.9	11.6

(d) assignment

Table 4.2 – execution time in milliseconds: aggregation by problem size

variations when considering each operator in isolation. At least part of them can be explained by the eagerness of NewPolka and the laziness of PPL.

Verification overhead. The difference between column V and column C on table 4.1 corresponds to the time spent checking witnesses by the frontend. It includes the translation of witnesses from OCaml to Coq number representations and the reconstruction of the result from the witness. The two ends of the performance spectrum are presented: the inclusion test, which is the fastest operator, and the convex hull operator, which is the slowest.

The overhead of verification is small for the inclusion test and negligible for the convex hull. It is relatively greater for the inclusion test than for the convex hull, because the former is much cheaper to compute. More precisely: computing the convex hull of two polyhedra involves testing many inclusions when minimising, as we saw in the previous chapter.

4.6 VPL: simple, verified and efficient

This chapter discussed how I designed an experiment in order to compare the runtime performance of VPL with that of existing implementations of the abstract domain of polyhedra. Making the comparison fair proved harder than expected. The design of the method I ended up with benefited from useful feedback from one of the authors of Apron and NewPolka, Bertrand Jeannet. It was published, with some tweaks, in the paper [24] which reports on the oracle.

The result of the experimental evaluation of VPL are very encouraging: it compares favorably with state-of-the-art implementations, even if these don't generate witnesses for further validation of their results. Besides result verification, this result is encouraging since VPL represents polyhedra as constraints only, which departs from established practice. The results presented above are insufficient to claim that the double representation of polyhedra is unnecessary for performance, yet they invite further inquiry.

As expected, however, computing convex hulls on constraint representation is very expensive. The second part of this thesis investigates possible remedies to this situation.

Summary in French

Nous avons maintenant décrit l'ensemble de VPL. La prochaine étape consiste à évaluer expérimentalement ses performances. L'évaluation réalisée compare le temps d'exécution des opérateurs de VPL à ceux d'autres implémentations du domaine abstrait des polyèdres : NewPolka et la Parma Polyhedra Library (PPL). La performance des opérateurs du domaine abstrait dépend fortement des polyèdres sur lesquels ils s'exécutent. Plutôt que de générer aléatoirement des polyèdres, l'évaluation expérimentale de VPL se base sur le rejeu de séquences d'appels au domaine abstrait par l'analyseur statique Pagai. Ce dernier utilise la collection de domaines abstraits Apron, dans laquelle figurent NewPolka et PPL. Les résultats de l'évaluation sont encourageants : VPL a globalement

des performances comparables à NewPolka et PPL sur un ensemble de benchmarks issus d'une collection destinée à l'analyse de temps d'exécution dans le pire cas. Ces résultats ne sont pas suffisants pour conclure qu'une conception basée sur la double réprésentation n'est pas nécessaire. En revanche, ils démontrent qu'une conception basée sur la représentation par contraintes est une alternative à considérer. Néanmoins, les performances de l'opérateur «join» de VPL se dégradent très vite lorsque le nombre de contraintes des polyèdres augmente.

Part II

Improving projection using parametric linear programming

Besides VPL being reasonably efficient, the main conclusion of the performance evaluation presented at the end of the first part is that join is the costliest operator of VPL. In the abstract domain of polyhedra, operator join performs the convex hull of two polyhedra. In turn, computing the convex hull of polyhedra can be recast as a projection problem [5]. When polyhedra are represented as sets of linear constraints, projection is implemented by a refinement of Fourier-Motzkin elimination [47]. However, it is tempting to look at Fourier-Motzkin elimination as very naive algorithm for performing projection. It builds every constraint possibly resulting from the linear combination of two input constraints and then removes those which are redundant, which is to say most of them. Refinements on Fourier-Motzkin elimination make redundancy detection cheaper, but they don't address the root problem of redundant constraints being generated in the first place.

Another weakness of Fourier-Motzkin elimination is that it can't project more than one variable at a time. Eliminating n variables is performed in n steps, each of which resulting in a intermediate result which has usually more constraints than both the input polyhedron and the final result.

An interesting line of work addresses the projection problem from a totally different angle. Indeed, it starts from observing that polyhedral projection can be encoded as a parametric linear problem [36]. This introduces a parametric variant of linear programming, in which either the coefficients of the objective functions, or the constant terms of problem constraints, are affine functions of parameters [28]. The result of a parametric linear problem is a function from values of the parameters to the optimal of the objective function for these values of the parameters. This new approach addresses the concerns mentioned above on Fourier-Motzkin elimination.

- The parametric linear problem describes the geometry of the projected polyhedron. Using a suitable exploration algorithm, the solution yields a minimised representation of the projected polyhedron.

- An arbitrary number of variables can be eliminated at once. As an extreme case, it is possible to eliminate no variable at all. In that case, the algorithm minimises the representation of the input polyhedron.

Although promising, the idea in its current state suffers from a number of issues. As we will show, the standard algorithm [22] for solving parametric linear problems performs poorly. Several new solving algorithms have been published [35, 32] and try to exploit the specifics of the parametric linear problems which encode projections. However, their experimental validation is still limited and no working implementation is publicly available. After providing some background on parametric linear programming, we'll peer into several possible encodings of projection as parametric linear problems and outline their specific characteristics in order to build a better solving algorithm.

Chapter 5

Parametric linear programming

The encoding of projection as a parametric linear problem is quite technical and relies on the specific shape of the solution to parametric problems. In an attempt at making it easier to digest, this chapter provides some background information on both parametric linear problems and the standard algorithm for solving them: the parametric simplex algorithm [22].

5.1 Parametric linear problems

Parametric linear problems have a very similar shape to linear problems, which are described in the first part of this thesis. However, they are more general, in that they capture the value of the optimal value of the objective function as a function of *parameters*. Parameters and variables don't have the same status in a parametric linear problem. Parameters are used to describe a family of related linear problems, which can be solved more efficiently all at once than one by one, especially if there are infinitely many of them. Therefore, the solution to a parametric linear problem is a function taking values for the parameters and yielding the optimal objective value of the linear problem obtained by substituting the values for the parameters. From the perspective of a user, who uses such a function as a black box, the value of the parameters is a user's choice and the function fiddles as necessary with the value of the variables so that the objective function reaches its optimal value.

Parameters may appear either in the coefficients of objective function, or in the constant right-hand side of the constraints, but not both [28].

Parametric objective function. The objective function of a usual linear problem is a linear expression of the following form.

$$\min \sum_{i=1}^{l} c_i \cdot \lambda_i$$

The coefficients c_i are constants and the λ_i are the variables of the problem. The objective function is made parametric by defining each coefficient c_i as follows.

$$c_i(x) = c_{i0} + \sum_{j=1}^{m} c_{ij} \cdot x_j$$

Each coefficient c_i becomes an affine function of parameters $x_1, \ldots, x_m$. The objective function is therefore rewritten.

$$\min \sum_{i=1}^{l} c_i(x) \cdot \lambda_i$$

When the objective function is parametric, the constraints of the parametric linear problem have the same shape as those of a regular linear problem.

Parametric right-hand side. Each constraint of a linear problem is, at least in the variant we saw, an affine equality.

$$\sum_{i=1}^{l} a_{ki} \cdot \lambda_i = b_k$$

The constant right-hand side b_k of such a constraint is made parametric by redefining it as follows.

$$b_k(x) = b_{k0} + \sum_{j=1}^{m} b_{kj} \cdot x_j$$

It is now an affine function of parameters $x_1, \ldots, x_m$. The constraint above is therefore rewritten as follows.

$$\sum_{i=1}^{l} a_{ki} \cdot \lambda_i = b_k(x)$$

When the constraints of the problem have a parametric right-hand side, the objective function has the same shape as that of a regular linear problem.

5.2 Solutions to parametric linear problems

Figure 5.1 illustrates a parametric linear problem, where the parameters x_1 and x_2 are in the coefficients of the objective function and may have arbitrary sign. The problem has two nonnegative variables λ_1 and λ_2, which are constrained to values inside triangle ABC.

You will rightfully note that triangle ABC must be built using inequality constraints, while I mentioned only equality constraints. An inequality constraint, such as $\lambda_1 + \lambda_2 \leq 3$, can be systematically turned into an equality constraint by introducing a new nonnegative variable s_1 and by rewriting the constraint $\lambda_1 + \lambda_2 + s_1 = 3$. Variables such as s_1 measure some form of distance between point (λ_1, λ_2) and the boundary of the halfspace $\lambda_1 + \lambda_2 \leq 3$. They are called *slack* variables.

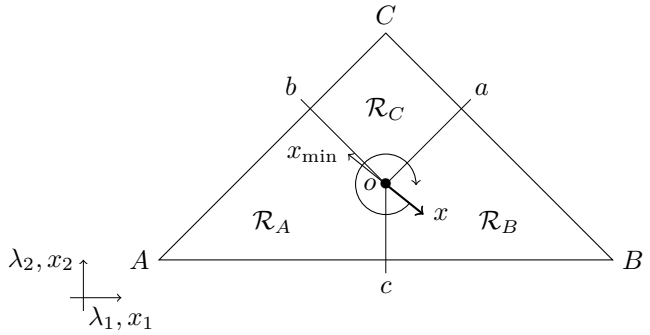


Figure 5.1 – an example of parametric objective function

With that made clear, let us go back to the example on figure 5.1. The parametric objective function is very simple: $x_1 \cdot \lambda_1 + x_2 \cdot \lambda_2$. With this objective function, the space of parameters x_1 and x_2 is superimposed on the space of variables λ_1 and λ_2. Indeed, choosing a value for parameters x_1 and x_2 specifies a direction $x \triangleq (x_1, x_2)$ in space of variables in which optimisation should be performed. It is not necessarily the case: the objective function $(x_1 + x_2) \cdot \lambda_1 + 0 \cdot \lambda_2$ can only describe directions along the λ_1 axis. Also note that the image of direction x rotating around point o on figure 5.1 should be taken with a grain of salt: it would be accurate if we imposed $||x|| = 1$. Without this nonlinear constraint on the parameters, the parameter space spans all the directions in the variable space, with many equivalent directions, such as $x = (1, 1)$ and $x = (2, 2)$.

Furthermore, note that, if we perform a minimisation, optimising in a given direction x will make us move opposite to that direction, as $\min x = -\max -x$. For example, suppose we wish to minimise function $-0.5 \cdot \lambda_1 + 0.4 \cdot \lambda_2$ in our example. This corresponds to optimising in direction $x_{\min}$ on figure 5.1. Minimising actually makes us look for a point as far in the opposite direction as possible. Point $A = (-3, 0)$ yields an objective value of 1.5, whereas point $B = (3, 0)$ yields the optimal objective value -1.5. Therefore, minimisation moves in direction labeled x on figure 5.1. For the purpose of illustration, we'll consider maximisation problems, so that we move in the direction of optimisation.

Solving a parametric linear problem amounts to considering all possible optimisation directions and finding which vertex is going to yield the optimal value for the objective function. For example, optimising in all directions between $(oa) = (1, 1)$ and $(oc) = (-1, 0)$ will end up in vertex B. More precisely, for the set of parameter values $\mathcal{R}_B = \{(x_1, x_2) \,|\, x_1 \geq 0 \wedge x_1 - x_2 \geq 0\}$, the optimal will be reached in point $B = (3, 0)$ and the objective value will be $x_1 \cdot 3 + x_2 \cdot 0$. Similarly, all the directions in the set $\mathcal{R}_C = \{(x_1, x_2) \,|\, -x_1 + x_2 \geq 0 \wedge x_1 + x_2 \geq 0\}$ will have the optimal reached in vertex $C = (0, 3)$, with objective value $x_1 \cdot 0 + x_2 \cdot 3$. Last, the directions in set $\mathcal{R}_A = \{(x_1, x_2) \,|\, -x_1 + -x_2 \geq 0 \wedge -x_1 \geq 0\}$ will have the optimal reached in vertex $A = (-3, 0)$, with objective value $x_1 \cdot (-3) + x_2 \cdot 0$.

Considering vertices

An alternative way to look at this builds on the fact that optimal values of objective functions are reached at vertices on the polyhedron described by the

constraints. When the objective function is parametric, the vertex yielding the optimal depends on the choice of values for the parameters. In our example, there are three vertices: A, B and C. Evaluating the objective function $x_1 \cdot \lambda_1 + x_2 \cdot \lambda_2$ at each of vertex gives the following three linear functions of the parameters.

$$\begin{aligned} x_1 \cdot (-3) \quad & \text{for vertex } A = (-3, 0) \\ x_1 \cdot 3 \quad & \text{for vertex } B = (3, 0) \\ x_2 \cdot 3 \quad & \text{for vertex } C = (0, 3) \end{aligned}$$

Since we are maximising the objective function and we know that the optimal is reached in one of these three points, the objective function can be recast in the following way.

$$\max(-3 \cdot x_1,\, 3 \cdot x_1,\, 3 \cdot x_2)$$

The solution of the parametric problem remains unchanged: when (x_1, x_2) are chosen in set $\mathcal{R}_A$, the first operand is the maximum. When they are chosen in set $\mathcal{R}_B$, the second is maximum, and similarly for the third. When the objective function is minimised, the minimum should be taken instead of the maximum.

Shape of the solution

Each set of directions $\mathcal{R}_i$, where $i \in \{A, B, C\}$, is a polyhedron and is called a "region". These regions partition the space of parameters: $\mathcal{R}_A \cup \mathcal{R}_B \cup \mathcal{R}_C = \mathbb{R}^2$. In each of the regions, the value of the objective function is an affine expression depending on the parameters. In the example above, the solution function $f(x_1, x_2)$ can be defined as follows.

$$f(x_1, x_2) = \begin{cases} -3 \cdot x_1 \text{ if } (x_1, x_2) \in \mathcal{R}_A \\ 3 \cdot x_1 \text{ if } (x_1, x_2) \in \mathcal{R}_B \\ 3 \cdot x_2 \text{ if } (x_1, x_2) \in \mathcal{R}_C \end{cases} \tag{5.1}$$

The solution function f has the following properties.

- Function f is piece-wise affine. This is apparent in the definition above: for all values of the parameters belonging to region $\mathcal{R}_A$, the optimum is reached at point A. The expression of the optimal for the region is therefore the following.

  $$x_1 \cdot (-3) + x_2 \cdot 0$$

  More generally, for all choices of parameters within a region $\mathcal{R}$, the optimal is reached at the same point $\underline{\lambda}$. Therefore, the expression of the optimal is the following, where $\underline{\lambda}$ is fixed and the $c_i(x)$ are affine functions of the parameters.

  $$\sum_{i=1}^{l} c_i(x) \cdot \underline{\lambda}_i$$

- Function f is continuous. When two regions share a frontier, as regions $\mathcal{R}_B$ and $\mathcal{R}_C$ share frontier $x_1 - x_2 = 0$, they yield the same objective value for directions lying on the frontier. In our example, for directions of the set $\mathcal{R}_B \cap \mathcal{R}_C = \{(x_1, x_2) \,|\, x_1 + -x_2 = 0 \wedge x_1 \geq 0\}$, we have $3 \cdot x_1 = 3 \cdot x_2$.

Constraint $x_1 \geq 0$ appears so as to restrict possible values of (x_1, x_2) to the half-line pictured on figure 5.1. Equivalently, I could have chosen $x_1 + x_2 \geq 0$.

- Function f is convex for parametric maximisation problems and concave for parametric minimisation problems. Both follow from similar remarks. Let me detail the minimisation case. Function f being concave means that, if you take two choices of values $\underline{x}$ and $\underline{x}'$ for parameters, the optimal value $f(\underline{x}'')$ associated to a convex combination $\underline{x}'' \triangleq \alpha \cdot \underline{x} + (1-\alpha) \cdot \underline{x}'$, with $\alpha \in [0,1]$, of points $\underline{x}$ and $\underline{x}'$ will be greater than the same convex combination of the optimal values associated to points $\underline{x}$ and $\underline{x}'$.

$$\alpha \cdot f(\underline{x}) + (1-\alpha) \cdot f(\underline{x}') \leq f\big(\alpha \cdot \underline{x} + (1-\alpha) \cdot \underline{x}'\big) \quad \text{with } \alpha \in [0,1]$$

Concavity follows from function f implementing the minimum of a set of functions. Suppose that, depending on the choice of parameters, two vertices $\underline{\lambda}$ and $\underline{\lambda}'$ can yield the optimal of the objective function $c(x) \cdot \lambda$. From the definition of a minimum, we have the following.

$$\min\big(\underline{\lambda} \cdot c(x),\, \underline{\lambda}' \cdot c(x)\big) \leq \underline{\lambda} \cdot c(x)$$

This still holds when both sides are multiplied by a nonnegative number.

$$\alpha \cdot \min\big(\underline{\lambda} \cdot c(x),\, \underline{\lambda}' \cdot c(x)\big) \leq \alpha \cdot \underline{\lambda} \cdot c(x)$$

Such inequalities can be summed and instantiated.

$$\begin{aligned} \alpha \cdot \min\big(\underline{\lambda} \cdot c(\underline{x}),\, \underline{\lambda} \cdot c'(\underline{x})\big) + \\ (1-\alpha) \cdot \min\big(\underline{\lambda} \cdot c(\underline{x}'),\, \underline{\lambda}' \cdot c'(\underline{x}')\big) \leq \\ \alpha \cdot \underline{\lambda} \cdot c(\underline{x}) + (1-\alpha) \cdot \underline{\lambda} \cdot c(\underline{x}') \end{aligned}$$

Objective function $c(x)$ being linear, the right-hand side can be factorised as $\underline{\lambda} \cdot c\big(\alpha \cdot \underline{x} + (1-\alpha) \cdot \underline{x}'\big)$. A similar reasoning on function $\underline{\lambda}' \cdot c(x) \geq 0$ finish the proof of concavity.

5.3 The parametric simplex algorithm

Now that we saw what a parametric linear problem is and what its solution looks like, we may have a look at algorithms which actually build the solution. The standard algorithms [28] are extensions to the simplex algorithm, which we already covered. There being two flavours of parametricity, in the coefficient of the objective function or in the constant term of the constraints, naturally leads to two solving algorithms. They build on the same idea and, therefore, I will describe only one: the algorithm for parameters in the objective function. A description of the other can be found in a paper by Paul Feautrier [22], which came with an implementation: Pip [21], which is still maintained today.

I chose the parametric objective variant since this is the one I implemented in collaboration with Alexandre Maréchal, him for extending his work [9] on linearisation and me for experimenting with projection. Some details on the experimentation are given in chapter 7

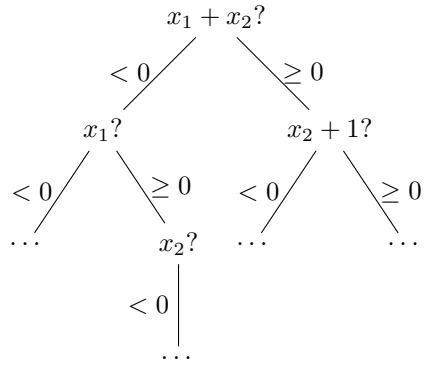


Figure 5.2 – the tree structure of the parametric simplex exploration

5.3.1 The impact of parametricity

We saw in chapter 2 that each step of the simplex algorithm consists in looking for negative coefficients in the current expression of the objective function. Among the variables which have a negative coefficients, the pivoting rule selects which enters the basis, which is the set of dependent variables. Let's call $\lambda_{\underline{i}}$ the chosen variable. Next, the coefficients of entering variable $\lambda_{\underline{i}}$ in each dependent variable definition of the dictionary is considered so as to determine which of them restricts the most the growth of variable $\lambda_{\underline{i}}$. The process leads to the choice of the dependent variable $\lambda_{\underline{j}}$ which will leave the basis.

When parameters appear in the coefficients of the objective function, it is no longer obvious to determine their sign, as they aren't constants any more, but functions. How do you know whether parametric coefficient $x_1 + x_2$ is negative or nonnegative? If you don't: both cases need to be considered. This is the main idea behind the parametric simplex algorithm. Each time, the algorithm needs to find the sign of a parametric coefficient and it doesn't know it, it creates a *branching* in the optimisation. On one branch, it considers that the coefficient is negative. On the other branch, it considers that it is nonnegative. Each branch is then explored, which may involve further branchings. This gives a tree structure to the exploration, as illustrated on figure 5.2. Note that earlier choices sometimes makes one alternative of a later choice impossible. In the exploration shown on figure 5.2, once the algorithm chose $x_1 + x_2 < 0$ and $x_1 \geq 0$, it must choose $x_2 < 0$, since there is no (x_1, x_2) such that $x_1 + x_2 < 0$, $x_1 \geq 0$ and $x_2 \geq 0$.

The context. In order to make consistent choices along a branch and therefore to realise that some sign alternatives shouldn't be considered, the choices need to be recorded and collected in a *context*. This context captures the conjunction of the sign choices: in the left-most branch of figure 5.2, the algorithm chose that $x_1 + x_2$ is negative *and* that x_1 is negative. The context is propagated and updated with further choices down each branch.

All of the sign choices accumulated in the context are affine constraints, which split the parameter space into two. More precisely, each constraint splits the space on which the current branch focuses into two, as illustrated on figure 5.3. This recursive splitting makes a partition of the parameter space into

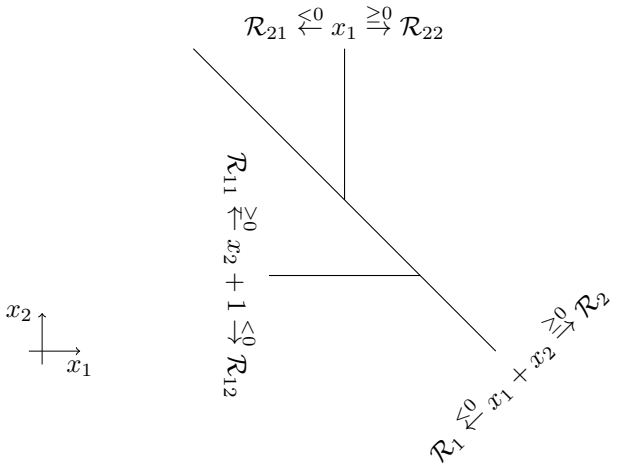


Figure 5.3 – search space splitting

regions: the search illustrated on figure 5.2 first splits the space $\mathbb{R}^2$ into two: $\mathcal{R}_1$ and $\mathcal{R}_2$, such that $\mathcal{R}_1 \cup \mathcal{R}_2 = \mathbb{R}^2$. One branch then subdivides subspace $\mathcal{R}_1$ into $\mathcal{R}_{11}$ and $\mathcal{R}_{12}$, such that $\mathcal{R}_{11} \cup \mathcal{R}_{12} = \mathcal{R}_1$, etc..

Deciding signs. Context C is the intersection of sign choices, each of which is an affine constraint: it is therefore a polyhedron. Given a context C, testing the sign of a parametric coefficient $c(x)$ consists in testing $C \sqsubseteq c(x) < 0$ and $C \sqsubseteq c(x) \geq 0$. If both are true, context C must be the empty polyhedron. If both are false, both cases need to be considered. Otherwise, the sign of the parametric coefficient is determined by context C.

Besides context. The end of a branch is reached when the value of the objective function can't be improved on any more: the context implies that all the parametric coefficients of the objective function are nonnegative, much like the stopping criterion of standard simplex algorithm.

Once it is past the choice of variable entering the basis, the algorithm operates just the same as the simplex algorithm.

5.3.2 The algorithm

With the preceeding context, the algorithm should now be straightforward. It starts from a linear problem with a parametric objective function and an initial context C, which may or may not set initial constraints on the parameters. In the latter case, all the parameter space must be explored. The problem is transformed to be in canonical form in the same way as it is performed prior to the simplex algorithm. Then, the algorithm operates as follows.

1. Choose a variable $\lambda_{\underline{i}}$ which parametric coefficient in the objective function may be negative in the current context.

2. If there is no such variable, the end of the branch is reached and the algorithm returns the parametric value of the objective function.

3. If the context implies that variable $\lambda_{\underline{i}}$ may also be nonnegative, branching happens. On one branch, extend the context C with $c_{\underline{i}}(x) \geq 0$ and perform a recursive call, which goes back to step 1. On the other branch, extend the context C with $c_{\underline{i}}(x) < 0$ and move to step 4 below.

4. Find the dependent variable $\lambda_{\underline{j}}$ which limits the most the growth of variable $\lambda_{\underline{i}}$. If the biggest limitation is given by several variables, choose one according to the pivoting rule. This step is done exactly as in the simplex algorithm: it doesn't involve the parameters.

5. If no dependent variable bounds the growth of variable $\lambda_{\underline{i}}$, then end of the branch is reached and the algorithm returns $+\infty$.

6. Perform a pivot with variables $\lambda_{\underline{i}}$ and $\lambda_{\underline{j}}$ and go back to step 1.

The result tree is built from the return values of the recursive calls. Furthermore, although the outline above sweeps it under the rug, some amount of copying happens for the recursive calls: both the context and the problem are transformed independently by each branch. A straightforward implementation would pass copies of the data at each recursive call.

Equation 5.1, on page 100, shows an alternative presentation of the result of the parametric simplex algorithm. It is essentially the collection of the leaves of the exploration tree: each region $\mathcal{R}_i$ is the conjunction of all the sign choices performed along a branch. In other words, the region is equal to the final context of the branch.

Pivoting rules

Most of the theory we discussed about the simplex algorithm carries over to its parametric variant. In particular, on a degenerate problem, the parametric simplex algorithm may cycle among several possible choices of dependent variables which all describe the same point in the space of variables. Bland's rule [8] remedies to the situation in the same way as it provides a solution for the simplex algorithm, with the same performance concerns. In the context of the parametric simplex algorithm, Bland's rule would translate as follows: given an order on the variables, pick the smallest variable which the current context doesn't imply to be nonnegative. One can think of several other pivoting rules, which proved to be more efficient than Bland's rule.

- If there is a variable which the context implies to be negative, choose it. This rule has the benefit of avoiding unnecessary branching.

- The problem with the above rule is that it requires testing for negativity all of the parametric coefficients of the objective function. Since each test involves solving a nonparametric linear problem, the benefit of not branching may be outweighed by the cost of performing many sign tests. A cheaper rule consists in checking if the parametric objective function has some negative constant coefficients, such as $(0 \cdot x_1 + 0 \cdot x_2 + -3) \cdot \lambda_5$ and choosing the corresponding variable. If there is no such constant coefficient, then use Bland's rule which picks the first variable whose coefficient may be negative.

Note that these rules don't guarantee termination. However, we can use the same trick as for the simplex algorithm: use an efficient pivoting rule for a bounded number of steps and switch to Bland's rule afterwards to ensure termination.

5.4 Wrapping up

This chapter introduced the basics of parametric linear programming. Parametric linear problems can be of two types: parametric objective function or parametric right-hand side of constraints. Although each type requires an adapted variant of the simplex algorithm, the solution to the two has the same shape: a function from parameters to objective value. The parametric simplex algorithm builds the implementation of such a function by splitting the parameter space into regions. For all choices of parameter values in one region, the optimal of the objective function is reached at the same vertex of the space of variables. It results that each region is associated an affine expression of the parameters, which yields the objective value.

With this background in mind, next chapter presents how parametric linear problem can be leveraged to compute the result of polyhedral projections.

Summary in French

L'évaluation expérimentale de VPL a montré la mauvaise performance de l'opérateur «join» du domaine abstrait. Pour la représentation par contraintes des polyèdres, cet opérateur est implémenté par une projection. La deuxième partie de la thèse consiste à étudier une approche de la projection basé sur la programmation linéaire paramétrique, comme alternative à l'algorithme habituel de projection, l'élimination de Fourier-Motzkin. La programmation linéaire paramétrique est une variante de la programmation linéaire dans laquelle l'objectif à optimiser est une fonction linéaire d'un ensemble de paramètres. L'algorithme de résolution est basé sur l'algorithme du simplexe. La solution d'un problème linéaire paramétrique est une fonction des paramètres vers la valeur optimale de l'objectif paramétrique. Cette fonction partionne l'espace des paramètres en régions. Dans chaque région, la valeur de la solution est une fonction affine des paramètres. Comme la programmation linéaire ordinaire, la variante paramétrique s'exprime comme une maximisation ou comme une minimisation, chacune étant duale de l'autre.

Chapter 6

Defining projection as a parametric linear problem

The main idea behind using parametric linear programming to solve projection problems on polyhedra is twofold.

1. The first step consists in describing the *geometry* of the result polyhedron as a parametric linear problem.
2. From that, the solving algorithm reconstructs a minimised constraint representation of the polyhedron.

The approach was introduced by Colin Jones et al. [36] and further elaborated on by Jacob Howe and Andy King [32]. It has three main advantages compared to Fourier-Motzkin elimination and its refinements.

- It focuses on the geometry of the polyhedra: the algorithm explores it independently of the details of the input constraints, which may or may not have redundancies. As a result, the output is an already minimised constraint representation of the result.
- The focus on geometry enables the elimination of several variables at a time. It is also possible to recover a minimised constraint representation of the input polyhedron by eliminating no variable at all.
- The solving algorithm explores the space looking for the boundaries set by the projected polyhedron. This exploration can be directed or stopped as needed, so that approximations can be obtained without extra cost. This contrasts with Fourier-Motzkin elimination, where all the variable eliminations need to be performed before getting output constraints.

The present chapter describes two formulations of projection as a parametric linear problem. One has parameters on the right-hand side of constraints, while the other has parameters in the objective function. I built the former in an attempt to capture the intuition of the latter, which is somewhat more abstract. As a result, I'll introduce my work prior to covering the state of the art. At the end of the day, both approaches highlight different aspects of the projection problem and its encoding as a parametric linear problem.

6.1 A polyhedron as a parametric linear problem

The first approach to encoding projection as a parametric linear problem builds on a slight change in point of view on the definition of a linear constraint. Instead of regarding $c_i(x) \geq 0$ as a constraint defining a half space, that is to say as set $\{x \,|\, c_i(x) \geq 0\}$, we are going to focus on function $c_i(x) \triangleq a_{i0} + \sum_{j=1}^{n} a_{ij} \cdot x_j$. More precisely, we would like to consider the *distance* of a point x to the hyperplane $c_i(x) = 0$. Let me define the distance in the following way.

$$d_i(x) \triangleq \frac{c_i(x)}{||(a_{i1}, \ldots, a_{in})||_2} = \frac{c_i(x)}{\sqrt{\sum_{j=1}^{n} a_{ij}^2}}$$

Actually, the distance of point x to hyperplane $c_i(x) = 0$ is $|d_i(x)|$. Removing of the absolute value gives $d_i(x)$ a nice property: when a point $\underline{x}$ satisfies constraint $c_i(x) \geq 0$, then $d_i(\underline{x}) \geq 0$, otherwise $d_i(\underline{x}) < 0$. This essentially makes $d_i(x)$ a particular choice of scaling of $c_i(x)$: it just happens that $d_i(x)$ is *normalised* with $||.||_2$. As a result, the following equality trivially holds.

$$\big\{x \,\big|\, d_i(x) \geq 0\big\} = \big\{x \,\big|\, c_i(x) \geq 0\big\}$$

From a polyhedron $p \triangleq \{c_1(x) \geq 0, \ldots, c_l(x) \geq 0\}$, we can build l functions $d_1(x), \ldots, d_l(x)$ and define a function $f(x)$ in the following way.

$$f(x) \triangleq \min d_i(x),\ i \in \{1, \ldots, l\}$$

To each point x, function f associates the distance of x to the constraint it is the closest to. It can serve as an alternative representation of polyhedron P.

$$P = \Big\{x \,\Big|\, \bigwedge_{i=1}^{l} c_i(x) \geq 0\Big\} = \big\{x \,\big|\, f(x) \geq 0\big\}$$

Before we deal with projection, we are going to elaborate on this representation of a polyhedron, by recasting it as an equivalent parametric problem as follows.

$$f(x) = \textbf{max}\ y\ \textbf{under constraints}\ y \leq d_i(x),\ i \in \{1, \ldots, l\}$$

The driving motivation is to have the solution to this parametric solver have the following shape, where constraints $d_1(x) \geq 0, \ldots, d_k(x) \geq 0$ are the nonredundant constraints of polyhedron p.

$$f(x) = \begin{cases} d_1(x) \text{ if } x \in \mathcal{R}_1 \\ \quad \vdots \\ d_k(x) \text{ if } x \in \mathcal{R}_k \end{cases}$$

Throughout this chapter, we will make the simplifying assumption that polyhedron p doesn't have any implicit equality: it is fully dimensional. Furthermore, none of the constraints of polyhedron p is strict.

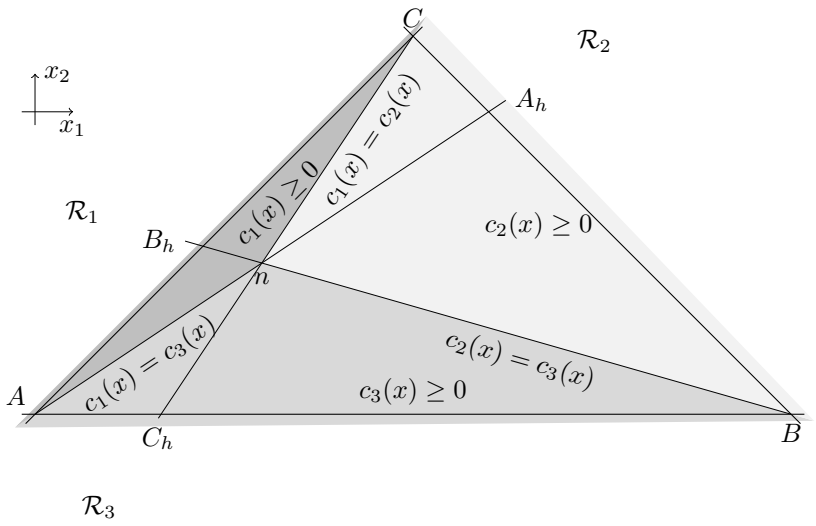


Figure 6.1 – function f on a triangle

6.1.1 The starting point

A major problem in the definition of function f is that it imposes the constraints $c_i(x) \geq 0$ to be normalised in $d_i(x) \geq 0$, using $||.||_2$ and the value of $||.||_2$ isn't necessarily a rational number. We're going to need another normalisation. For now, we won't use any and see how far we can get with the resulting variant of function f, which I'll call f_p.

$$f_p(x) = \textbf{max } y \textbf{ under constraints } y \leq c_i(x),\ i \in \{1, \ldots, l\}$$

Note that we still have $P = \{x \mid f_p(x) \geq 0\}$.

Essentially, function f_p above builds a map of the space: it divides it in regions, each associated to a constraint $c_i(x) \geq 0$. The value of function f_p for all the points of a given region is given by the same $c_i(x)$. For example, consider figure 6.1. The triangle ABC is made of three constraints, with the following definitions.

$$\begin{aligned} c_1(x) &= 10 \cdot x_1 + -10 \cdot x_2 \\ c_2(x) &= 20 + -2 \cdot x_1 + -2 \cdot x_2 \\ c_3(x) &= 5 \cdot x_2 \end{aligned}$$

All the points on the line passing through points A and A_h are as far from constraint $c_1(x) \geq 0$ as they are from constraint $c_3(x) \geq 0$. Similarly, points on the line (BB_h) are equally far from constraints $c_3(x) \geq 0$ and $c_2(x) \geq 0$. The same holds for line (CC_h) and constraints $c_2(x) \geq 0$ and $c_1(x) \geq 0$.

$$\begin{aligned} &\forall x \in (AA_h),\ c_1(x) = c_3(x) \\ &\forall x \in (BB_h),\ c_3(x) = c_3(x) \\ &\forall x \in (CC_h),\ c_2(x) = c_1(x) \end{aligned}$$

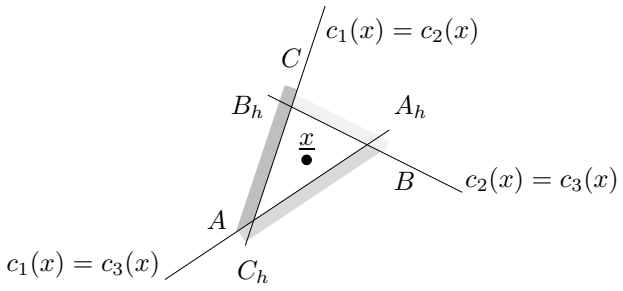


Figure 6.2 – zoom on figure 6.1 in the event point n doesn't exist

These three lines, (AA_h), (BB_h) and (CC_h), intersect at point n, at which $c_1(n) = c_2(n) = c_3(n)$. Note that, in a triangle, such a point n always exists, irrespective of the scaling factor applied to each of the three constraints. To understand why, consider point $\underline{x}$ on figure 6.2, which zooms on a variant of figure 6.1 where the three lines (AA_h), (BB_h) and (CC_h) don't intersect in one point. At point $\underline{x}$, we have the following contradiction.

$$c_1(\underline{x}) < c_3(\underline{x}) < c_2(\underline{x}) < c_1(\underline{x})$$

Let us go back to figure 6.1. From point n, the plane can be partitioned into three regions $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$, as represented on the figure. These regions are polyhedra and have the following properties with respect to function f_p.

$$\forall x \in \mathcal{R}_1,\ f_p(x) = c_1(x)$$
$$\forall x \in \mathcal{R}_2,\ f_p(x) = c_2(x)$$
$$\forall x \in \mathcal{R}_3,\ f_p(x) = c_3(x)$$

The previous discussion leads naturally to the following solution to the parametric problem.

$$f(x) = \begin{cases} c_1(x) \text{ if } x \in \mathcal{R}_1 \\ c_2(x) \text{ if } x \in \mathcal{R}_2 \\ c_3(x) \text{ if } x \in \mathcal{R}_3 \end{cases}$$

Furthermore, we can build a constraint representation of polyhedron P from the definition of function f_p above: all that is needed is discarding the information pertaining to the regions and gathering the $c_i(x)$'s. This process doesn't forget any nonredundant constraint. To see why, let us suppose that nonredundant constraint $c_{\underline{i}}(x) \geq 0$ doesn't appear in the recovered constraint representation. Now, consider any point $\underline{x} \in P$ such that $c_{\underline{i}}(\underline{x}) = 0$. Since the constraint is nonredundant, there must exist such a point. If constraint $c_{\underline{i}}(x) \geq 0$ doesn't appear in the constraints recovered from the definition of function f_p, there must be a constraint $c_{\underline{i}'}(x) \geq 0$ of polyhedron p such that $c_{\underline{i}'}(\underline{x}) < c_{\underline{i}}(\underline{x})$. Since $c_{\underline{i}}(\underline{x}) = 0$, we have $c_{\underline{i}'}(\underline{x}) < 0$, which violates the assumption that point x is in polyhedron P. Therefore, we can recover from function f_p at least the nonredundant constraints of polyhedron p. However, as we will see in a moment, recovering a constraint representation from function f_p may not filter redundant constraints.

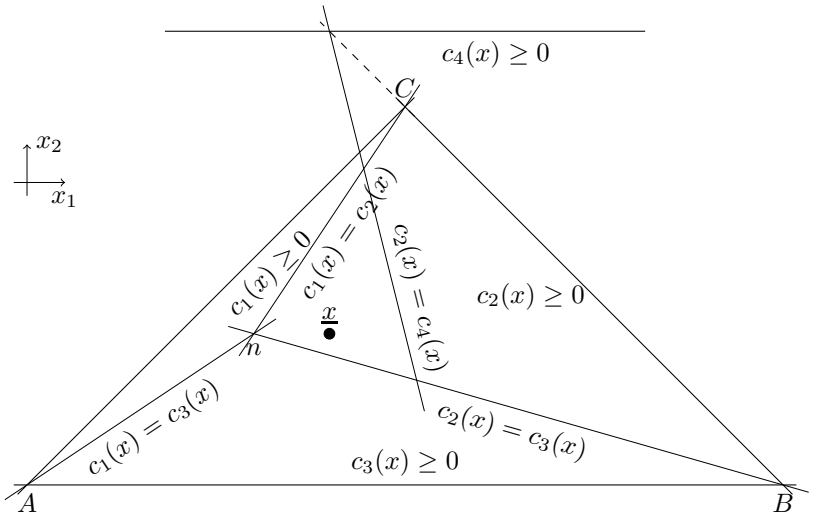


Figure 6.3 – function f with a redundant constraint

6.1.2 Redundancy

Short of additional details, the discussion of function f_p applies only to the restricted case of the triangle. Furthermore, it builds on the optimistic scenario in which there is no redundant constraint. Consider figure 6.3, which is very similar to figure 6.1, but has a new and redundant constraint $c_4(x) \geq 0$.

$$c_4(x) = 9 + -\frac{3}{2} \cdot x_2$$

This new constraint being redundant, $p \cup \{c_4(x) \geq 0\}$ is the same polyhedron as p, with a different constraint representation. Therefore, it seems reasonable to expect the solution to the parametric problem defining function f_p to be unaffected by this additional constraint. This is not the case: a new region appears in the result.

There are now too many constraints to draw all the $c_i(x) = c_j(x)$ lines, which are the frontiers of the regions. Therefore, figure 6.3 only shows those which already appear on figure 6.1 and the line $c_2(x) = c_4(x)$. Furthermore, figure 6.3 pictures a point $\underline{x}$, which would have been in region $\mathcal{R}_2$ on figure 6.1. According to the frontiers, we have the following.

$$c_4(\underline{x}) < c_2(\underline{x}) < c_3(\underline{x}) < c_1(\underline{x})$$

As a result, function f_p evaluated at point $\underline{x}$ gives value $c_4(\underline{x})$. The same holds for all the points in the small triangle made by the frontiers around point $\underline{x}$.

The issue is better illustrated on the following simpler example, with only one parameter x_1. Figure 6.4 represents three constraints $\frac{1}{2} \cdot x_1 \geq \frac{1}{2}$, $2 \cdot x_1 \geq 4$ and $x_1 \leq 4$ in an unusual manner: it plots them as functions $c_1(x_1) = -\frac{1}{2} + \frac{1}{2} \cdot x_1$, $c_2(x_1) = -4 + 2 \cdot x_1$ and $c_3(x_1) = 4 + -x_1$. If we were to build a function f_p for the polyhedron $p = \{\frac{1}{2} \cdot x_1 \geq \frac{1}{2} \wedge 2 \cdot x_1 \geq 4 \wedge x_1 \leq 4\}$, it would be as follows.

$$f_p(x_1) = \begin{cases} c_2(x_1) \text{ if } x \in \mathcal{R}_2 \\ c_1(x_1) \text{ if } x \in \mathcal{R}_1 \\ c_3(x_1) \text{ if } x \in \mathcal{R}_3 \end{cases}$$

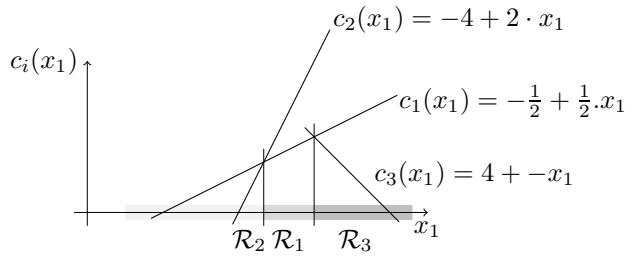


Figure 6.4 – the impact of scaling

Constraint $c_1(x_1) \geq 0$ appears in the definition of function f_p, although it is redundant in polyhedron p. The following lemma captures the issue.

Lemma 6. *For each nonempty polyhedron whose representation contains a redundant constraint, a scaling exists which makes that constraint appear in the map built using function f_p.*

Proof. Let $P = \{x \,|\, c_1(x) \geq 0 \wedge \cdots \wedge c_l(x) \geq 0\}$. Suppose that constraint $c_1(x) \geq 0$ is redundant: there are $\lambda_0 \geq 0$ and $\lambda_i \geq 0$, $i \in \{2, \ldots, l\}$ such that,

$$\forall x,\ \lambda_0 + \sum_{i=2}^{l} \lambda_i \cdot c_i(x) = c_1(x)$$

We wish to prove that there are scaling factors $\lambda'_i > 0, i \in \{1, \ldots, l\}$ such that

$$\exists \underline{x} \in P,\ \lambda'_1 \cdot c_1(\underline{x}) < \lambda'_i \cdot c_i(\underline{x}),\ \forall i \in \{2, \ldots, l\}$$

This is trivially true for any $\underline{x}$ in the interior of P, that is as long as all the $c_i(\underline{x})$ are strictly positive. Since we made the assumption at the beginning of the section that the polyhedra we are dealing with have full dimension, there exists such an interior point. □

Had we kept the normalisation, using $||.||_2$, of the input constraints, this problem wouldn't have arisen. In order to avoid redundant constraints appearing in the result of the parametric linear problem, we are going to introduce a new normalisation, which builds on a point n in the interior of polyhedron P.

For now, let us restrict ourselves to the polytopes. Point n is used to transform the space, so that point n appears to be at distance 1 from all the input constraints. For each input constraint $c_i(x) \geq 0$ of input polyhedron p, a function $d'_i(x)$ is defined as follows.

$$d'_i(x) = \frac{c_i(x)}{c_i(n)}$$

Point n being an interior point is key: if it is on the boundary of polyhedron p, meaning that there is a constraint $c_i(n) = 0$, then constraint i can't be scaled so as to reach $\lambda_i \cdot c_i(n) = 1$, with $\lambda_i > 0$. However, note that choosing distance *1* is arbitrary here: any strictly positive number would have the same effect.

The main effect of using point n to scale the constraints is to have all the regions meet in point n: since $d'_i(n) = 1$ for all i, point n is at the frontier of

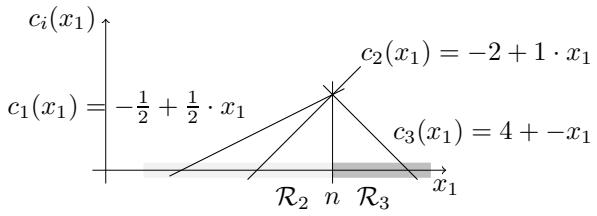


Figure 6.5 – scaling with central point n

all the regions. For this reason, let me call point n the "central point". If we go back to the example on figure 6.4 and choose point $\underline{x}$, at which $c_1(\underline{x}) = c_3(\underline{x})$, as the central point, we find ourselves in the situation depicted on figure 6.5. Note that region $\mathcal{R}_1$ has disappeared: there is always at least one nonredundant constraint below redundant constraint $c_1(x_1) \geq 0$.

The assumption that polyhedron p is a polytope, that is a bounded polyhedron, makes it possible to prove the following lemma.

Lemma 7. *If polyhedron $P \triangleq \{x \,|\, c_1(x) \geq 0, \ldots, c_l(x) \geq 0\}$ is a polytope and if there exists an interior point n of polytope P such that $\forall i \in \{1, \ldots, l\}$, $c_i(n) = 1$, then function $f_p(x) = \min_{i \in \{1,\ldots,l\}} c_i(x)$ is always smaller than or equal to 1.*

Proof. Any point $\underline{x} \in P$ is on a line segment between point n and a point $\underline{x}'$ on the boundary of P. Point $\underline{x}'$ being on the boundary of polytope P means that there is at least one $\underline{i} \in \{1, \ldots, l\}$ such that $c_{\underline{i}}(\underline{x}') = 0$. Since $c_{\underline{i}}(n) = 1$ and $\exists \alpha \in [0, 1]$, $\underline{x} = \alpha \cdot \underline{x}' + (1 - \alpha) \cdot n$, it must be that $c_{\underline{i}}(\underline{x}) \in [0, 1]$. From the definition of function f, we know that $f(\underline{x}) \leq c_{\underline{i}}(\underline{x}) \leq 1$.

For any point $\underline{x} \notin P$, we know by definition of polytope P that $f(\underline{x}) < 0$. □

This lemma leads us to the major gain of introducing point n.

Lemma 8. *Assuming that there is an interior point $n \in P$, with P a nonempty polytope, such that $c_i(n) = 1$ for all i, function f_p can be built from nonredundant constraints only.*

Proof. Assume that $c_1(x) \geq 0$ is a redundant constraint of polytope p.

$$\exists \lambda_0 \geq 0,\ \lambda_i \geq 0,\ i \in \{2, \ldots, l\},\ \forall x,\ \lambda_0 + \sum_{i=2}^{l} \lambda_i \cdot c_i(x) = c_1(x)$$

We want to prove that $\forall x,\ \exists i \in \{2, \ldots, l\},\ c_i(x) \leq c_1(x)$. Let us assume, by contradiction, that there is a point $\underline{x}$ such that, $c_1(\underline{x}) < c_i(\underline{x}), \forall i \in \{2, \ldots, l\}$. From $\lambda_i \geq 0$, we derive

$$\lambda_0 + \sum_{i=2}^{l} \lambda_i \cdot c_1(\underline{x}) < \lambda_0 + \sum_{i=2}^{l} \lambda_i \cdot c_i(\underline{x}) = c_1(\underline{x})$$

From evaluating $\forall x, \lambda_0 + \sum_{i=2}^{l} \lambda_i \cdot c_i(x) = c_1(x)$ in n, we obtain the equality $\lambda_0 + \sum_{i=2}^{l} \lambda_i = 1$. Rewriting $\sum_{i=2}^{l} \lambda_i \cdot c_1(\underline{x})$ as $c_1(\underline{x}) \cdot \sum_{i=2}^{l} \lambda_i$ and substituting $1 - \lambda_0$ for $\sum_{i=2}^{l} \lambda_i$, we get $\lambda_0 + (1 - \lambda_0) \cdot c_1(\underline{x}) < c_1(\underline{x})$. Simplifying yields $\lambda_0 < \lambda_0 \cdot c_1(\underline{x})$. Now, either $\lambda_0 = 0$ or $1 < c_1(\underline{x})$. The former yields the contradiction $0 < 0$ and the latter is in contradiction with lemma 7. □

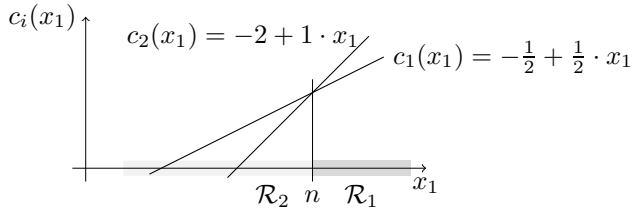


Figure 6.6 – scaling with point n in a unbounded polyhedron

6.1.3 Unbounded polyhedra

We have shown that it is possible to build function f_p from the nonredundant constraints of polyhedron p, using an interior point n to force a particular scaling. This trick, however, only works when p is a polytope. Essentially, unboundedness invalidates lemma 7 which, in turn, invalidates lemma 8. That is to say that unboundedness isn't a problem by itself. It is a problem in that it causes function f_p to have a value strictly greater than 1 for some points. Figure 6.6 illustrates this point. It pictures the function f_p associated to polyhedron $p = \{c_1(x_1) \geq 0, c_2(x_1) \geq 0\}$, with $c_1(x_1) = -1 + x_1$ and $c_2(x_2) = -2 + x_1$. The central point n is chosen at $x_1 = 3$. As a result, $c_1(x_1)$ is scaled to $-\frac{1}{2} + \frac{1}{2} \cdot x_1$, so that $c_1(n) = 1$. Note that, constraint $c_1(x_1) \geq 0$ is redundant with respect to $c_2(x_1) \geq 0$, yet the solution f_p to the parametric linear problem **max** y **under constraints** $y \leq c_i(x_1)$ is as follows.

$$f_p(x_1) = \begin{cases} c_1(x_1) \text{ if } x_1 < 3 \\ c_2(x_1) \text{ if } x_1 \geq 3 \end{cases}$$

This example shows that redundant constraints can still be found in function f_p when the corresponding polyhedron is unbounded. However, $c_1(x) < c_2(x)$ only when $f_p(x) > 1$.

One remedy consists in introducing redundancy in a controlled way, by means of the trivially redundant constraint $1 \geq 0$, which can be added to any polyhedron without changing the set of points it describes. Consider now how it affects the construction of function f_p from a polyhedron p.

$$\begin{aligned} f_p^{\leq 1}(x_1) = \ & \textbf{max } y \textbf{ under constraints} \\ & y \leq c_1(x_1) \\ & y \leq c_2(x_1) \\ & y \leq 1 \end{aligned}$$

This doesn't change anything for bounded polyhedra, when constraints are normalised with respect to a central: lemma 7 shows that $f_p \leq 1$ anyway. For unbounded polyhedra, the redundant constraint forces $f_p \leq 1$, thereby making lemma 8 applicable again. However, this also introduces redundancy in the constraints extracted from function f_p. Function f_p for the polyhedron on figure 6.6 was as follows.

$$f_p(x_1) = \begin{cases} c_2(x_1) \text{ if } x \in \mathcal{R}_2 \\ c_1(x_1) \text{ if } x \in \mathcal{R}_1 \end{cases}$$

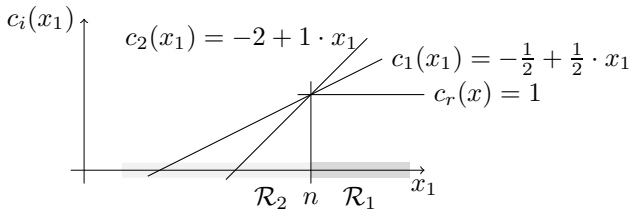


Figure 6.7 – scaling with point n in a unbounded polyhedron augmented by $1 \geq 0$

Introducing redundant constraint $1 \geq 0$ alters it as pictured on figure 6.7.

$$f_p^{\leq 1}(x_1) = \begin{cases} c_2(x_1) \text{ if } x \in \mathcal{R}_2 \\ 1 \text{ if } x \in \mathcal{R}_1 \end{cases}$$

The crucial difference between functions f_p and $f_p^{\leq 1}$ is that $c_1(x_1) \geq 0$ being redundant may be hard to find out—on more complex examples, solving a linear program may be required—whereas redundancy is limited to exactly $1 \geq 0$ in the case of $f_p^{\leq 1}$ and is therefore trivial to spot.

6.1.4 Fully dimensional regions

There is one last thing we have overlooked so far and which appears in the phrasing of lemma 8. Lemma 8 states that function $f_p^{\leq 1}$, hence the solution to the parametric problem, *can* be built from nonredundant constraints. Indeed, consider function f'_p below.

$$f'_p(x_1) = \begin{cases} c_2(x_1) \text{ if } x \in \mathcal{R}_2 \backslash \{n\} \\ c_1(x_1) \text{ if } x = n \\ 1 \text{ if } x \in \mathcal{R}_1 \backslash \{n\} \end{cases}$$

Functions $f_p^{\leq 1}$ and f'_p are extensionally equal: $\forall x_1, f_p^{\leq 1}(x_1) = f'_p(x_1)$. However, when we gather a constraint representation from function f'_p, redundant constraint $c_1(x_1) \geq 0$ appears, whereas it wouldn't appear if we were to use function $f_p^{\leq 1}$. The problem disappears when the regions are forced to have full dimension: the region corresponding to constraint $c_1(x_1)$ reduces to a single point, which has dimension 0, in 1-dimensional problem.

A simple way to enforce regions being of full dimension is to make them open, that is described by strict constraints. The missing values on the boundary of regions can be inferred by continuity of function $f_p^{\leq 1}$.

6.2 From a polyhedron to its projection

So far, we have seen how to encode a polyhedron $p = \{c_1(x) \geq 0, \ldots, c_l(x) \geq 0\}$ as a parametric linear problem, in the form of function $f_p^{\leq 1}$.

$$\begin{aligned} f_p^{\leq 1}(x) = \textbf{max } & y \textbf{ under the constraints} \\ & y \leq c_i(x),\ \forall i \in \{1, \ldots, l\} \\ & y \leq 1 \end{aligned}$$

From the solution to this parametric linear problem, a minimised constraint representation of polyhedron $\{c_1(x) \geq 0, \ldots, c_l(x) \geq 0\}$ can be extracted. Let us see how this can be used to project variables. We'll start from a polyhedron P.

$$P = \left\{(x_1, \ldots, x_m) \,\middle|\, \bigwedge_{i=1}^{l} c_i(x_1, \ldots, x_m) \geq 0\right\}$$

In order to keep the notations simple, we assume that trivial constraint $1 \geq 0$ is one of the $c_i(x) \geq 0$. Accordingly, we'll write f_p instead of $f_p^{\leq 1}$. Furthermore, we assume that the constraints have already been scaled with respect to a central point n, that is to say there exists a point n in the interior of polyhedron P such that $c_i(n) = 1$, $\forall i \in \{1, \ldots, l\}$.

We want to project polyhedron P on a subset of its dimensions $x_1, \ldots, x_k$ with $k < m$. Let us call $\tilde{P}$ the result. From the definition of the projection, we have the following expression for polyhedron $\tilde{P}$.

$$\tilde{P} = \left\{(x_1, \ldots, x_k) \,\middle|\, \exists x_{k+1}, \ldots, x_m, \bigwedge_{i=1}^{l} c_i(x_1, \ldots, x_m) \geq 0\right\}$$

Now, let us add function f_p to the mix.

$$f_p(x_1, \ldots, x_m) = \max y \in \left\{y \,\middle|\, \forall i \in \{1, \ldots, l\},\ y \leq c_i(x_1, \ldots, x_m)\right\}$$

The definition of polyhedron P becomes the following.

$$P = \left\{(x_1, \ldots, x_m) \,\middle|\, f_p(x_1, \ldots, x_m) \geq 0\right\}$$

Applying the definition of the projection to this new definition of polyhedron P, we get the following expression.

$$\tilde{P} = \left\{(x_1, \ldots, x_k) \,\middle|\, \exists x_{k+1}, \ldots, x_m,\ f_p(x_1, \ldots, x_m) \geq 0\right\}$$

Alternatively, we may push the existential quantification inside function f_p, yielding a new function $\tilde{f}_p$.

$$\begin{aligned} \tilde{f}_p(x_1, \ldots, x_k) = & \max y \\ & \text{with } y \in \left\{y \,\middle|\, \exists x_{k+1}, \ldots, x_m,\ \forall i \in \{1, \ldots, l\},\ y \leq c_i(x_1, \ldots, x_m)\right\} \end{aligned}$$

Now, we use function $\tilde{f}_p$ to define a new polyhedron.

$$P_{\tilde{f}} = \left\{(x_1, \ldots, x_k) \,\middle|\, \tilde{f}_p(x_1, \ldots, x_k) \geq 0\right\}$$

Last, we prove that $\tilde{P} = P_{\tilde{f}}$, through mutual inclusion.

$P_{\tilde{f}} \subseteq \tilde{P}$. Suppose we have a point $(\underline{x}_1, \ldots, \underline{x}_k) \in P_{\tilde{f}}$, that is $\tilde{f}_p(\underline{x}_1, \ldots, \underline{x}_k) \geq 0$. Function $\tilde{f}_p$ is defined as a maximum. We call $\underline{x}_{k+1}, \ldots, \underline{x}_m$ the value of existentially quantified variables $x_{k+1}, \ldots, x_m$ for which the maximum is reached in $\tilde{f}_p(\underline{x}_1, \ldots, \underline{x}_k)$. Instantiating $x_{k+1}, \ldots, x_m$ with $\underline{x}_{k+1}, \ldots, \underline{x}_m$ in the definition of $\tilde{P}$ yields $f_p(\underline{x}_1, \ldots, \underline{x}_m) \geq 0$, from which $(\underline{x}_1, \ldots, \underline{x}_k) \in \tilde{P}$

$\tilde{P} \subseteq P_{\tilde{f}}$. Suppose $(\underline{x}_1, \ldots, \underline{x}_k) \in \tilde{P}$, meaning that there are $\underline{x}_{k+1}, \ldots, \underline{x}_m$ such that $f_p(\underline{x}_1, \ldots, \underline{x}_m) \geq 0$. Instantiating $x_{k+1}, \ldots, x_m$ in the definition of $\tilde{P}'$, we exhibit one point where $y \geq 0$. This provides a nonnegative lower bound on the value of $\tilde{f}_p(\underline{x}_1, \ldots, \underline{x}_m)$, which is thus nonnegative. This implies that $(\underline{x}_1, \ldots, \underline{x}_k) \in P_{\tilde{f}}$

Final parametric linear problem

Gathering all the elements discussed so far, we may now build a parametric linear problem, the result of which describes the result of the projection $p \backslash \{x_{k+1}, \ldots, x_m\}$. The problem statement given below considers the following definition for each constraint $c_i(x) \geq 0$ of polyhedron p.

$$c_i(x) \triangleq a_{i0} + \sum_{j=1}^{m} a_{ij} \cdot x_j$$

A central point n is chosen for scaling the $c_i(x)$'s so that they have value 1 when evaluated at point n.

$$\frac{c_i(x)}{c_i(n)} = \frac{a_{i0}}{c_i(n)} + \sum_{j=1}^{m} \frac{a_{ij}}{c_i(n)} \cdot x_j$$

The variables $x_{k+1}, \ldots, x_m$, which are existentially quantified in the definition of polyhedron $P_{\tilde{f}}$ above, appear on the left-hand side of each constraints. The other variables, $x_1, \ldots, x_k$ appear on the right-hand side.

max y **under the constraints**

$$\begin{aligned} c_i(n) \cdot y + \sum_{j=k+1}^{m} -a_{ij} \cdot x_j \;&\leq\; a_{i0} + \sum_{j=1}^{k} a_{ij} \cdot x_j, \; \forall i \in \{1, \ldots, l\} \\ y &\leq 1 \end{aligned} \tag{6.1}$$

The problem stated in equation 6.1 is a parametric linear problem, with parameters on the right-hand side of the constraints. Its variables are $y, x_{k+1}, \ldots, x_m$ and its parameters are $x_1, \ldots, x_k$.

This presentation of how projection can be encoded as a parametric linear problem is, as far as I know, new. It helps understanding the geometry of the parametric linear problem and the reason for which it yields a minimised constraint representation of the projected polyhedron. The key element is the central point, where all the regions meet. With some amount of tweaking, Paul Feautrier's Pip solver [21] could be used to solve problems of this form. Tweaking is required since Pip handles only nonnegative parameters, while they are unconstrained in our setting, and finds a lexicographic minimum, instead of minimising a function.

6.3 Starting from positive linear combinations

The state-of-the-art approach [36, 32] to encoding projection as a parametric linear problem builds on an idea closer to Fourier-Motkzin elimination. It puts more emphasis on the resulting constraints being linear combinations of input constraints, at the expense of obscuring the geometrical view of the problem. In the end, presenting both gives two views, which complement each other. We will now present this second approach, before showing how the two are related. It may also be worth mentioning that, due to historical reasons and ongoing collaborations, the following approach is the one I implemented and experimented with.

6.3.1 The starting point

Let $p = \{c_1(x) \geq 0, \ldots, c_l(x) \geq 0\}$ be a polyhedron. As before, we assume that none of the constraints of polyhedron p is strict and that there are no implicit equality constraints. Let us now consider the set p_Λ of constraints implied by the constraints of polyhedron p.

$$p_\Lambda = \Big\{ \sum_{i=1}^{l} \lambda_i \cdot \big(c_i(x) \geq 0\big) \;\Big|\; \forall i \in \{1, \ldots, l\},\, \lambda_i \geq 0 \Big\}$$

The set p_Λ is a polyhedral *cone*.

Definition. *A cone $\mathcal{C}$ is a set such that, if $c \in \mathcal{C}$, then $\lambda \cdot c \in \mathcal{C}$, $\lambda \geq 0$. Furthermore, if $c \in \mathcal{C}$ and $c' \in \mathcal{C}$, then $c + c' \in \mathcal{C}$.*

An alternative way to define polyhedron $P = \{x \mid \bigwedge_{i=1}^{l} c_i(x) \geq 0\}$ is to say that it is the set of points which satisfy all the constraints of set p_Λ.

$$P = \big\{ x' \;\big|\; \forall\, c(x) \geq 0 \;\in p_\Lambda,\; c(x') \geq 0 \big\}$$
$$P = \Big\{ x' \;\Big|\; \forall\, \lambda_i \geq 0,\; \Big(\sum_{i=1}^{l} \lambda_i \cdot c_i(x) \Big) \geq 0 \Big\}$$

Note that this definition is useful for building proofs, but useless for algorithms computing over polyhedra. Similarly to a polyhedron, a cone has a set of *generators*, which are the elements from which all the other elements can be built by scaling and addition. The generators of cone p_Λ are the nonredundant constraints of polyhedron p. Therefore an minimised representation of polyhedron p can be obtained by computing the generators of cone p_Λ.

To be precise, the elements of cone p_Λ are vectors. A vector $(a_{i0}, a_{i1}, \ldots, a_{im})$ of cone p_Λ represents constraint $a_{i0} + \sum_{j=1}^{m} a_{ij} \cdot x_j \geq 0$. The definition of cone p_Λ is therefore more exactly stated as follows.

$$p_\Lambda = \Big\{ \sum_{i=1}^{l} \lambda_i \cdot (a_{i0}, \ldots, a_{im}) \;\Big|\; \forall i \in \{1, \ldots, l\},\, \lambda_i \geq 0 \Big\}$$

Note that there is a dualisation going on here: each element of cone p_Λ is a *constraint* over points in the x space.

Loosening constraints

The set p_Λ doesn't actually contain *all* the constraints implied by polyhedron p. Suppose that polyhedron p is defined as $\{3 + x_1 \geq 0\}$. The set p_Λ as defined above contains only constraints of the form $\lambda{\cdot}3{+}\lambda{\cdot}x_1 \geq 0$, with $\lambda \geq 0$. However, constraint $4 + x_1 \geq 0$, which is implied by polyhedron $\{3 + x_1 \geq 0\}$, doesn't appear in set p_Λ. What's missing in the definition of set p_Λ is the coefficient λ_0, which appears in the statement of Farkas's lemma in chapter 2 for loosening the constant term of a constraint. Adding it results in the following updated definition for set p_Λ.

$$p_\Lambda = \Big\{ \lambda_0 \cdot (1 \geq 0) + \sum_{i=1}^{l} \lambda_i \cdot \big(c_i(x) \geq 0\big) \;\Big|\; \forall i \in \{0, \ldots, l\},\, \lambda_i \geq 0 \Big\}$$

Or more rigorously,

$$p_\Lambda = \Big\{ \lambda_0 \cdot (1, 0, \ldots, 0) + \sum_{i=1}^{l} \lambda_i \cdot (a_{i0}, \ldots, a_{im}) \;\Big|\; \forall i \in \{0, \ldots, l\},\, \lambda_i \geq 0 \Big\}$$

Note that this is tantamount to adding the trivial constraint $1 \geq 0$ to the constraints of polyhedron p. In order to make the formula more concise, we will assume that constraint $1 \geq 0$ is one of the constraints $c_i(x) \geq 0$ in the following.

Expressing projection

Before going further down the road of finding the generators of a polyhedral cone, let us restore the focus on projection. Fourier-Motzkin elimination performs linear combinations of constraints in order to build constraints which have a zero coefficient for the eliminated variables. We can do the same to the cone p_Λ: focus on the linear combinations which cancel the coefficients of the variables $x_{k+1}, \ldots, x_m$ we wish to eliminate. This leads to building a new set $\tilde{p}_\Lambda$.

$$\tilde{p}_\Lambda = \Big\{ \sum_{i=1}^{l} \lambda_i \cdot (a_{i0}, \ldots, a_{im}) \;\Big|\; \forall j \in \{k+1, \ldots, m\}, \sum_{i=1}^{l} \lambda_i \cdot a_{ij} = 0, \quad \text{with } \lambda_i \geq 0, \forall i \in \{1, \ldots, l\} \Big\}$$

The j^{th} equality constraint in the definition of set $\tilde{p}_\Lambda$ states that the linear combination should yield a zero coefficient for variable x_j. Note that, like set p_Λ, set $\tilde{p}_\Lambda$ is a cone. Finding the generators of cone $\tilde{p}_\Lambda$ will give us a nonredundant representation of polyhedron $p \backslash \{x_{k+1}, \ldots, x_m\}$.

Finding the generators of cone $\tilde{p}_\Lambda$

Finding the generators of a polyhedral cone is a well-studied problem, whose solution dates back to Chernikova's algorithm [12]. However, our setting prevents us from applying these results: we don't start from a constraint representation of the cone, seeking its generator representation. Rather, our cone is defined

as a restriction of another cone, p_Λ. In this situation, Chernikova's algorithm behaves in exactly the same way as Fourier-Motzkin elimination.

Instead, the idea we use for finding the generators of cone $\tilde{p}_\Lambda$ is very similar to that of function f_p, which we explored before: for each point x of space, we consider the constraint of cone $\tilde{p}_\Lambda$ closest to point x. Let function g_p capture this idea.

$$g_p(x) = \min c_i(x) \in \tilde{p}_\Lambda$$

As before for function f_p, each nonredundant constraint $c'_i(x) \geq 0$ of the projected polyhedron will give the value of function g_p for all points x in a region each. Solving the parametric linear problem associated to function g_p will yield a definition of the following form.

$$g_p(x) = \begin{cases} c'_1(x) \text{ if } x \in \mathcal{R}_1 \\ \quad \vdots \\ c'_{l'}(x) \text{ if } x \in \mathcal{R}_{l'} \end{cases}$$

The actual parametric linear problem is the following.

$$\begin{aligned} &\textbf{min } \sum_{i=1}^{l} \lambda_i \cdot c_i(x) \textbf{ under the constraints} \\ &\sum_{i=1}^{l} a_{ij} \cdot \lambda_i = 0, \ \forall j \in \{k+1, \ldots, m\} \\ &\lambda_i \geq 0, \ \forall i \in \{1, \ldots, l\} \end{aligned}$$

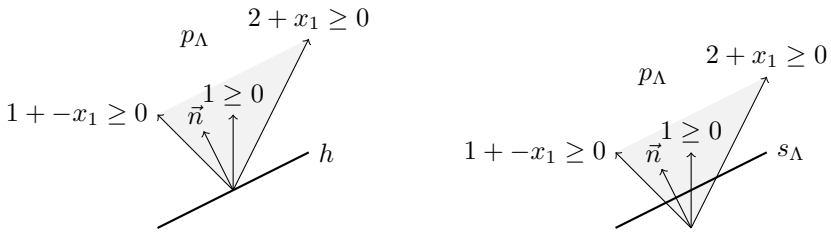
In this parametric problem, the variables are $\lambda_1, \ldots, \lambda_l$ and the parameters are $x_1, \ldots, x_m$. The parameters appear in the objective function. Note that the set of parameters can be reduced to $x_1, \ldots, x_k$ as the constraints impose that the coefficient of parameters $x_{k+1}, \ldots, x_m$ is always zero. The problem can therefore be rewritten as follows.

$$\begin{aligned} &\textbf{min} \sum_{i=1}^{l} \lambda_i \cdot \left(a_{i0} + \sum_{j=1}^{k} a_{ij} \cdot x_j\right) \textbf{ under the constraints} \\ &\sum_{i=1}^{l} a_{ij} \cdot \lambda_i = 0, \ \forall j \in \{k+1, \ldots, m\} \\ &\lambda_i \geq 0, \ \forall i \in \{1, \ldots, l\} \end{aligned} \tag{6.2}$$

6.3.2 From cone to polytope

Although the parametric problem in equation 6.2 captures the intuition well, it is not an entirely adequate formulation of projection. This becomes apparent when you try to instantiate the parameters.

Consider a point $\underline{x}$ satisfying the constraints of polyhedron p: that is to say $\forall i \in \{1, \ldots, l\}$, $c_i(\underline{x}) \geq 0$. From Farkas's lemma, point $\underline{x}$ satisfies any constraint built from nonnegative linear combinations of the constraints of polyhedron p. It follows that the minimum value of the linear problem obtained by instantiating the parameters in problem 6.2 with point $\underline{x}$ is nonnegative. Taking $\lambda_i = 0, \forall i$



(a) finding the supporting hyperplane (b) shifting to get the slice

Figure 6.8 – slicing the cone

actually yields the optimal: 0. It follows that function g_p is uniformly 0 for all points $\underline{x}$ satisfying the constraints of polyhedron p.

Again, the problem comes from the possibility of scaling the constraints. Given a constraint $c_i(x) \geq 0$ and a point $\underline{x}$ such that $c_i(\underline{x}) > 0$, the positive scaling λ_i which minimises the number $\lambda_i \cdot c_i(\underline{x})$ goes to 0. Function g_p being uniformly zero on polyhedron p also appears by looking at equation 6.2. From chapter 2, we know that the objective value is built from linear combinations of the constant term of the constraints of a linear problem. Since these constant terms are all zero in the parametric linear problem in equation 6.2: their linear combinations will only yield objective values of zero.

Going back to the cone structure of set $\tilde{p}_\Lambda$ sheds a new light on the issue. Consider constraint $c_i(x) \geq 0 \ \in \tilde{p}_\Lambda$. It follows from the definition of a cone, that $\lambda_i \cdot c_i(x) \geq 0 \in \tilde{p}_\Lambda$ for any $\lambda_i > 0$. All these constraints describe the same half-space.

$$\forall \lambda_i > 0, \{x \,|\, c_i(x) \geq 0\} = \{x \,|\, \lambda_i \cdot c_i(x) \geq 0\}$$

Selecting one instance of each of these families of constraints will make it possible to make distance comparisons. Which instance is irrelevant. Similarly to what we had before, the issue is that of normalising the constraints. The solution which is adopted in the state of the art [36, 32] is to build a *slice* of the cone $\tilde{p}_\Lambda$.

A slice $\tilde{s}_\Lambda$ of cone $\tilde{p}_\Lambda$ is a polytope such that the following property holds.

$$\forall c_i(x) \geq 0 \ \in \tilde{p}_\Lambda, \exists \lambda_i > 0, \lambda_i \cdot \big(c_i(x) \geq 0\big) \ \in \tilde{s}_\Lambda$$

In other words, slice $\tilde{s}_\Lambda$ selects a unique representative of each set of equivalent constraints $\{\lambda_i \cdot c_i(x) \geq 0 \,|\, \lambda_i > 0\}$.

Computing a slice $\tilde{s}_\Lambda$ of cone $\tilde{p}_\Lambda$

Slice $\tilde{s}_\Lambda$ is built from a hyperplane supporting cone $\tilde{p}_\Lambda$ and then shifted so as to intersect all the families of constraints described above. A hyperplane supporting the cone $\tilde{p}_\Lambda$ is a hyperplane going through the origin $(0, \ldots, 0)$ of the cone and having all of the points of cone $\tilde{p}_\Lambda$ on one side. Building such a hyperplane directly is made tricky by the fact that we don't have an explicit expression of the generators of cone $\tilde{p}_\Lambda$. However, cone $\tilde{p}_\Lambda$ is a subcone of cone p_Λ, of which we have the generators, or a superset of thereof if the representation of polyhedron p isn't minimised. Making sure that all the generators of cone p_Λ lie on one side of the supporting hyperplane guarantees that all the points generated from them will also lie on one side of it. In order to do so, the normal vector

$(n_0, \ldots, n_m)$ to the hyperplane h is chosen so that its dot product with all the generators of cone p_Λ is positive, as illustrated on figure 6.8a. We are looking for vector $(n_0, \ldots, n_m)$ which satisfies the following condition.

$$(a_{i0}, \ldots, a_{im}) \cdot (n_0, \ldots, n_m) > 0, \forall\, a_{i0} + \sum_{i=1}^{l} a_{ij} \cdot x_j \geq 0 \in p$$

Once vector $n = (n_0, \ldots, n_m)$ is determined, a constraint represented by a vector $(a_0, \ldots, a_m)$ belongs to the supporting hyperplane of normal vector n if the following condition holds.

$$\sum_{j=0}^{m} n_j \cdot a_j = 0$$

Shifting the hyperplane so that it slices the cone p_Λ, as illustrated on figure 6.8b, results in the following expression.

$$\sum_{j=0}^{m} n_j \cdot a_j = b \quad \text{with } b > 0$$

Any positive value fits for constant b. In the following, I will use $b = 1$, but that choice is arbitrary. The final step of the construction of slice $\tilde{s}_\Lambda$ consists in inserting it in equation 6.2. Doing so requires undoing temporarily a shortcut we have taken when building equation 6.2, which can be recast equivalently as follows.

$$\begin{aligned} &\textbf{min } a_0 + \sum_{j=1}^{k} a_j \cdot x_j \textbf{ under the constraints} \\ &a_j = 0, \forall j \in \{k+1, \ldots, m\} \\ &a_j = \sum_{i=1}^{l} a_{ij} \cdot \lambda_i, \forall j \in \{0, \ldots, m\} \\ &\lambda_i \geq 0, \forall i \in \{1, \ldots, l\} \end{aligned} \tag{6.3}$$

This is still a linear problem with a parametric objective function. The parameters are $x_1, \ldots, x_k$. The variables are $\lambda_1, \ldots, \lambda_l$ and a newly introduced set of variables $a_0, \ldots, a_m$. Problem 6.3 explicits that we are interested in vectors $(a_0, \ldots, a_m)$, which must be nonnegative linear combinations of the constraints of the input polyhedron p and which must not constrain the dimensions to be eliminated. Problem 6.2 is problem 6.3, where variables a_j have been eliminated using the equations which define them. We may now add the equation of the

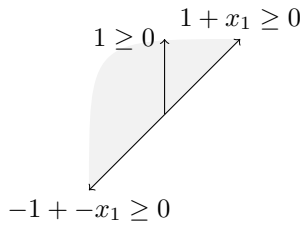


Figure 6.9 – an implicit equality means no supporting hyperplane

slicing hyperplane to problem 6.3, yielding the following updated problem.

$$\begin{aligned}
&\textbf{min } a_0 + \sum_{j=1}^{k} a_j \cdot x_j \textbf{ under the constraints} \\
&\sum_{j=0}^{m} n_j \cdot a_j = 1 \\
&a_j = 0,\ \forall j \in \{k+1, \ldots, m\} \\
&a_j = \sum_{i=1}^{l} a_{ij} \cdot \lambda_i,\ \forall j \in \{0, \ldots, m\} \\
&\lambda_i \geq 0,\ \forall i \in \{1, \ldots, l\}
\end{aligned}$$

Replacing variables a_j by their definitions, we get the following final problem.

$$\begin{aligned}
&\textbf{min } \sum_{i=1}^{l} \lambda_i \cdot \left(a_{i0} + \sum_{j=1}^{k} a_{ij} \cdot x_j\right) \textbf{ under the constraints} \\
&\sum_{i=1}^{l} \left(\sum_{j=0}^{m} n_j \cdot a_{ij}\right) \cdot \lambda_i = 1 \\
&\sum_{i=1}^{l} a_{ij} \cdot \lambda_i = 0,\ \forall j \in \{k+1, \ldots, m\} \\
&\lambda_i \geq 0,\ \forall i \in \{1, \ldots, l\}
\end{aligned} \tag{6.4}$$

The full dimensionality assumption

At the beginning of this section, I made the assumption that the polyhedra we're dealing with have no implicit equality in their constraint set. Figure 6.9 illustrates the motivation behind the assumption. It depicts the cone p_Λ built from polyhedron $\{1 + x_1 \geq 0, -1 + -x_1 \geq 0\}$, which is more intuitively written $-1 \leq x_1 \leq -1$, or even $x_1 = -1$. It would be impossible to find a vector $\vec{n}$ whose dot product with each vector $(-1, -1), (0, 1)$ and $(1, 1)$ is strictly positive.

6.4 Duality of the two encodings

So far in this chapter, we have seen two ways of encoding a projection problem as a parametric linear problem. The approach presented in section 6.2 features

a maximisation problem with a parametric right-hand side, while the approach presented in section 6.3 uses a minimisation problem with a parametric objective function. The two provide complementary views on the same object and emphasise different aspects of the problem. For example, normalisation with a central point is perhaps more intuitive than normalisation with a slicing hyperplane.

The two encodings are actually dual, which I'll try to show by first exhibiting the similarities between them. The most obvious similarity resides in the introduction of the trivial constraint $1 \geq 0$.

Slicing is finding an interior point

Another, less obvious, similarity shows between the central point and the slicing hyperplane. Both are used in order to normalise the constraints. The analogy goes deeper than that, though. Given a polyhedron $p = \{c_1(x) \geq 0, \ldots, c_l(x) \geq 0\}$, where each $c_i(x)$ is defined as $a_{i0} + \sum_{j=1}^{m} a_{ij} \cdot x_j$, the normal vector $(n_0, \ldots, n_m)$ to the slicing hyperplane satisfies the following constraints.

$$(a_{i0}, a_{i1}, \ldots, a_{im}) \cdot (n_0, \ldots, n_m) > 0, \forall i \in \{1, \ldots, l\} \tag{6.5}$$

Remember that we added the trivial constraint $1 \geq 0$ to the constraints of polyhedron p. This results on the following constraint on the normal vector.

$$(1, 0, \ldots, 0) \cdot (n_0, \ldots, n_m) > 0$$

This constraint simplifies in $n_0 > 0$. Now, if we find a vector $(n_0, \ldots, n_m)$ satisfying the constraints in equation 6.5, we can build another vector n'.

$$n' \triangleq (1, \frac{n_1}{n_0}, \ldots, \frac{n_m}{n_0})$$

Since $n_0 > 0$, this new vector n' also satisfies the constraints in equation 6.5. Furthermore, equation 6.5 can be spelt in a more concise manner, with vector $n' = (n'_0, \ldots, n'_m)$.

$$\sum_{j=0}^{m} a_{ij} \cdot n'_j \;=\; a_{i0} + \sum_{j=1}^{m} a_{ij} \cdot n'_j \;=\; c_i(n'_1, \ldots, n'_m) > 0, \forall i \in \{1, \ldots, l\}$$

It becomes apparent that vector n', without its first coefficient 1, is a point in the interior of polyhedron p. With that remark in mind, the constraint of equation 6.4 where vector n appears is changed as follows.

$$\sum_{i=1}^{l} c_i(n'_1, \ldots, n'_m) \cdot \lambda_i = 1$$

On top of being used to achieve the same end, the slicing hyperplane of section 6.3 and the central point of section 6.2 are actually the *same* object.

Matrix representation

The last point I am going to make will be more obvious with equations 6.1 and 6.4 recast in matrix format, on tables 6.1 and 6.2, respectively. Each of these tables has three groups of lines and three groups of columns, separated

	x_{k+1}	$\dots$	x_m	y		x_1	$\dots$	x_k
	0	$\dots$	0	1	a_0	a_1	$\dots$	a_k
λ_1	$-a_{1(k+1)}$	$\dots$	$-a_{1m}$	$c_1(n)$	a_{10}	a_{11}	$\dots$	a_{1k}
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
λ_l	$-a_{l(k+1)}$	$\dots$	$-a_{lm}$	$c_l(n)$	a_{l0}	a_{l1}	$\dots$	a_{lk}
λ_0	0	$\dots$	0	1	1	0	$\dots$	0

Table 6.1 – Equation 6.1 as a matrix

	λ_1	$\dots$	λ_l	λ_0	
	a_{10}	$\dots$	a_{l0}	1	a_0
x_1	a_{11}	$\dots$	a_{l1}	0	a_1
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
x_k	a_{1k}	$\dots$	a_{lk}	0	a_k
x_{k+1}	$a_{1(k+1)}$	$\dots$	$a_{l(k+1)}$	0	0
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
x_m	a_{1m}	$\dots$	a_{lm}	0	0
$c_i(n)$	$c_1(n)$	$\dots$	$c_l(n)$	1	1

Table 6.2 – Equation 6.4 as a matrix ($\forall i \in \{0, \dots, l\}$, $\lambda_i \geq 0$)

by solid lines. The first group of lines is made of only one line. This line tells which variable or parameter is associated with each column. The second group of lines is the objective function. In table 6.2, the coefficients of the objective function have parameters. Each line gives the coefficients of the parameter given in the first column. For example, variable λ_1 has parametric coefficient $a_{10} + a_{11} \cdot x_1 + \dots + a_{1k} \cdot x_k$. The third group of lines gives the coefficients of the constraints of the parametric problem. The first column provides some information about which coefficients these are. For example, the line starting with λ_1 of table 6.1 records the coefficients of constraint $y \leq c_1(x)$. The trivial constraint $1 \geq 0$ is associated to λ_0.

Of the three groups of columns, the first is merely informational: it states which variable or parameter the coefficients of the line apply to. The second group of columns gives the linear part of the constraints, which apply to the variables of the problem. The last group of columns gives the right-hand side of the constraints. On table 6.1, the second and third groups are separated by an implicit $\leq$ sign. On table 6.2, the second and third groups are separated by an implicit $=$ sign.

Now, let me recall a few key facts.

- The problem on table 6.1 is a maximisation problem. Its constraints are inequalities and its variables are unrestricted in sign.
- The problem on table 6.2 is a minimisation problem. Its constraints are equality constraints and its variables are nonnegative.
- The parametric objective function of table 6.2 is the parametric right-hand side of table 6.1.

- The constraints forcing the coefficients of variables x_{k+1} to x_m to 0 in problem 6.2 can be negated without changing the problem. With this change, the matrix of the coefficients of variables of one problem is the transposed matrix of the coefficients of variables of the other.
- The two parametric linear problems solve the same problem: they give the same solution.

These two problems are in fact the dual of one another.

6.5 Two models for projection

Through this chapter, we covered two ways of expressing as a parametric linear problem the projection of a polyhedron on a subset of its dimensions. One is a minimisation problem with parameters in the objective function. It comes from previous work by Colin Jones et al. [36] and Jacob Howe and Andy King [32]. To the best of my knowledge, the other expression of projection as a parametric linear problem is new. It was designed in an attempt to convey more of the global picture of the encoding of projection as a parametric linear problem. In the end, it highlighted the geometry of the regions in the solution—they all meet in a central point—and provided a new explanation to the resulting constraint set being minimised. This second approach results in a maximisation problem, with parameters in the constant term of the constraints. Previous section showed that the two approaches are linked by the duality theorem of linear programming.

Witness generation

The main motivation for this exploratory work is to build a better projection operator for VPL, which brings in the necessity of generating inclusion witnesses for the results produced by the operator. The original work [36] solves this concern in a straightforward manner. Indeed, the parametric problem is stated in terms of linear combination coefficients λ_i. In its solution, each region corresponds to a point in the space of λ_i's, where the optimal is reached for all parameter values in the region. This point exactly describes the linear combination of input constraints yielding the constraint associated to the region.

The situation is slightly less obvious in the encoding I proposed, where the variables of the maximisation problem are the variables constrained by the input polyhedron. However, witness extraction follows from a simple remark, similar to the one I made for extracting witnesses from the linear solver in VPL. The constraints of the parametric problem are inequality constraints. As such, a variable is going to be introduced during the initialisation of the parametric linear solver, which will uniquely identify each constraint. Depending on how the solver is implemented, this variable will either be an auxiliary variable, as in VPL solver, or a slack variable, as I described in previous chapter. When the optimal is reached in a region, the coefficients of these new variables in the definition of the objective function will give the coefficient of the input constraints yielding the constraint of the projected polyhedron associated with the region. In textbooks [18], this observation that the dual solution can be found in the coefficients of the objective function when the optimal is reached is called "complementary slackness".

From theory to practice

Now that we explored the theory for a new constraints-based projection operator, we may have a look at implementing it. As I mentioned earlier in this chapter, the maximisation problem can be solved with Pip, with some amount of extra encoding related to parameters being nonnegative in Pip and Pip computing a lexicographic minimum instead of optimising a function. However, historical reasons lead me to implement the other approach. First and foremost, the maximisation encoding wasn't available when this work started. I also collaborated with Jacob Howe and Andy King [32], who use a different approach for solving the minimisation problem. The experimental validation of the ideas presented in this chapter is still ongoing and the progress that has been made so far is reported in the next chapter.

Summary in French

Colin Jones a proposé dans sa thèse, publiée en 2005, un encodage de la projection d'un polyèdre par un problème linéaire paramétrique. J'en introduis une vision alternative, basée sur une nouvelle représentation des polyèdres. Un polyèdre est vu comme une function définie sur l'espace des variables contraintes par le polyèdre. Cette fonction est positive ou nulle pour les points du polyèdre et négative à l'extérieur. La projection d'un polyèdre est ensuite présentée comme la quantification existentielle des variables à éliminer. Cette quantification s'exprime naturellement sous la forme d'un problème linéaire paramétrique dont la solution est le polyèdre projeté représenté par une fonction de la forme décrite ci-dessus. La projection proposée par Colin Jones et la mienne sont duales l'une de l'autre. Elles offrent deux perspectives complémentaires pour comprendre la projection par programmation linéaire paramétrique.

Chapter 7

Towards a new solver

In the second part so far, we have introduced parametric linear programming and shown how it could be used to compute projections of polyhedra. This last chapter puts these ideas to practice. It turns out that the standard algorithm for solving parametric linear problems, namely the parametric simplex algorithm, performs poorly on projection benchmark problems. This observation triggered further research for a more efficient solving algorithm, drawing on insights from the automatic control community. The work reported below happens in collaboration with Alexandre Maréchal and is still ongoing. I report it nonetheless, with two main motivations.

- Some of the directions we are considering seem yet unexplored in state-of-the-art work.
- Our investigation method is pragmatic: the goal is to build a new projection operator for VPL, which is more efficient than Fourier-Motzkin elimination on projection problems encountered during the static analysis of programs.

7.1 The problem with the standard algorithm

Figure 7.1 illustrates the main issue of the parametric simplex algorithm on the example polyhedron ABC, which we used in chapter 5. Remember that

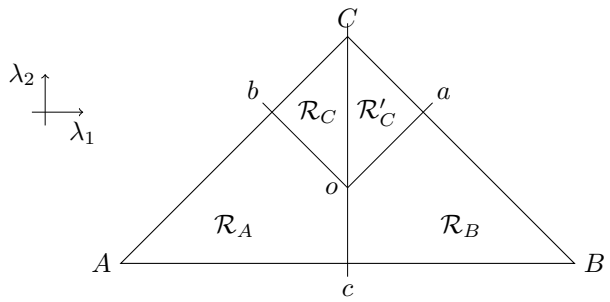


Figure 7.1 – four regions instead of three

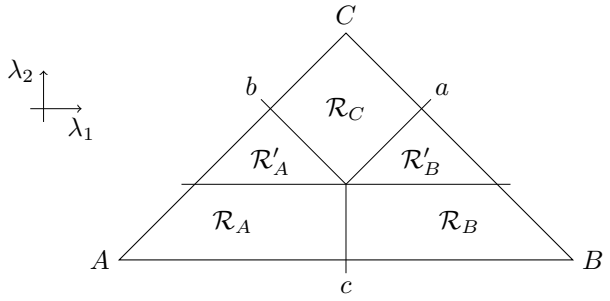


Figure 7.2 – not splitting on frontiers

each branching decision taken by the algorithm splits the current subspace to be explored into two. Therefore, the optimal splitting into three regions $\mathcal{R}_A$, $\mathcal{R}_B$ and $\mathcal{R}_C$, of polyhedron ABC, shown on figure 5.1 on page 99, can't be the result of the parametric simplex algorithm.

Figure 7.1 shows one suboptimal splitting, which the parametric simplex algorithm *can* produce. The algorithm first splits along frontier (Cc). On one branch, it then splits along $[ob)$, while it splits along $[oa)$ on the other. This yields four regions $\mathcal{R}_A$, $\mathcal{R}_B$, $\mathcal{R}_C$ and $\mathcal{R}'_C$, instead of the expected three. Of these four, $\mathcal{R}_C$ and $\mathcal{R}'_C$ have the same expression for the value of the objective function.

Oversplitting doesn't prevent the solution function to a parametric linear problem from being correct: it merely creates duplicates in the constraints built from each optimal expression. A simple postprocessing can get rid of these duplicates. However, oversplitting also duplicates the costlier computations that lead to duplicate optimal expressions at the end of branches. To put it differently, the problem with oversplitting is that it makes the exploration tree bigger, resulting in more pivots and more sign tests. These operations are costly and become a performance problem.

To make things worse, it is not true in general that all the parametric coefficients, which appear in the objective function, are frontiers of a region. Figure 7.2 illustrates the case where the initial splitting is not performed along a frontier: the result contains five regions instead of the optimal three. You should bear in mind that these are rather simple situations on a simple two-dimensional problem. With more dimensions, the phenomenon of oversplitting is amplified.

Despite the performance problems encountered during solving, the parametric linear problem approach to projection remains interesting. Indeed, it captures the geometry of polyhedra, from which a nonredundant constraint representation can be built. It also allows to eliminate several variables at once, which Fourier-Motzkin elimination can't handle. These benefits naturally lead us to look for a more efficient solving algorithm.

7.2 The method

Overall, our problem is a very concrete one: finding a more efficient projection algorithm than Fourier-Motzkin elimination. This goal was set as a result of

the performance evaluation of VPL and, more precisely, that of its convex hull operator. This evaluation is reported in chapter 4 and will serve as a base line to test new ideas against.

In order to focus on the core of the problem, we will restrict ourselves to solving a simplified problem: the elimination of any number of variables from a polyhedron which is of full dimension. There are no equalities, implicit or explicit. This simplification is still realistic for the use case of VPL: this is exactly the setting in which the projection operator on inequality constraints is called. In the context of VPL, we could also assume that the input polyhedron has a nonredundant representation, but we haven't found any use for this hypothesis yet. As a last simplifying assumption, we assume that there are no strict inequality constraints.

Our approach being driven by experimental data forces us to actually have a working implementation of the methods we are designing. While this takes some time, this also helps pruning research directions early.

Benchmark suite

The suite of benchmark problems, which we used for comparing ideas, is composed of all the convex hull problems gathered during the performance analysis of VPL. For each of these problems, the projections which would be performed by substitution using equalities were performed using VPL. The resulting benchmark problem is the projection of the remaining variables from the minimised intermediate result.

In total, the benchmark suite is composed of 15 882 projection problems, collected from abstract domain calls performed during the static analysis of programs using Pagai [31].

Comparison setup

The benchmark suite is stored in text format in a file. I wrote a small program which parses the file and runs each projection problem twice.

- First, the projection is performed using Fourier-Motzking elimination as it is implemented in VPL.
- Then, it is performed by the candidate projection method.

Each problem results in a pair of time measurements. Given the order of magnitude of the execution time difference between the two methods, a sum of the time spent by each method on the complete benchmark suite is a sufficient comparison metric. This being said, we checked the distribution of execution times over the suite so as to make sure that most of the execution time wasn't spent on a handful problems.

Besides timing, the driving program checks that the result provided by the experimental method is the same as the result provided by VPL. Results are considered the same when two criteria are met.

- The two sets of constraints represent the same set of points.
- They have the same number of constraints. Since the result provided by VPL are always minimised, this criterion ensures that the result provided by the experimental method is minimised as well.

Base line

All the time measurements reported below were carried out on a recent laptop, under the GNU/Linux operating system. By itself, each value isn't particularly relevant. Instead, they should be compared to one another. The base line which we want to improve on is the computation time of the implementation of Fourier-Motzkin elimination in VPL. It takes 3.7 seconds to compute all the projections of our benchmark suite.

Pip: a state-of-the-art solver

Paul Feautrier wrote and made publicly available [21] a solver for parametric linear problems, called Pip. It is still maintained today and we considered using it. The major obstacle is that Pip solves parametric problems with parameters in the right-hand side of the constraints. At the time we started investigating, the new encoding of projection with parameters in the right-hand side—the one we saw at the beginning of the previous chapter—didn't exist and we failed to dualise the encoding as minimisation otherwise. Then, Pip restricts both variables and parameters to be nonnegative, while we needed parameters of arbitrary sign. While it is possible to encode arbitrary sign parameters using nonnegative parameters, as explained in Pip user manual, we felt that nesting encodings would obfuscate our observations. In the end, all these elements lead us to implement our own solver.

Regions of full dimension

To be entirely precise, our solver implements a slight variation of the parametric simplex algorithm. The deviation comes from the need for regions of full dimension in order to guarantee the absence of redundant constraints in the polyhedron derived from the solution to parametric linear problem. It would be inefficient to look for implicit equalities in each context, once the whole solution tree has been built. Instead, the sign choices added to the context are systematically strictened: nonnegativity choices are turned into positivity choices. As a result, the polyhedron describing the context is open, which guarantees that it has full dimension when it is nonempty.

Initial analysis

Our initial experiment was to compare with the parametric simplex algorithm, which is described in chapter 5. We implemented in OCaml the variant of the algorithm which has parameters in the objective function and used it to solve projection encoded as a parametric minimisation problem. Chapter 5 describes three pivoting rules.

- The first is the parametric equivalent to Bland's rule. It chooses the first variable whose coefficient in the objective function may be negative. With this rule, our solver takes 101 seconds to compute all the projections in the suite.

- The second rule tries to delay branching by looking for coefficients in the objective function, which are necessarily negative according to the context.

With this rule, our solver takes 79 seconds to compute all the projections in the suite.

- The last rule is a compromise between the two. It looks from constant negative coefficients in the objective function. If there aren't any, it resorts to Bland's rule. With this rule, our solver takes 101 seconds to compute all the projections. This result seems to indicate the constant coefficients aren't common at all.

As the figures above show, our initial attempt at improving the performance of projection leaves a lot of room for improvement: it is over twenty times slower than Fourier-Motzkin elimination. Gathering some extra statistics on the execution gives us an interesting insight on possible causes of this poor performance.

Over the whole benchmark suite, Fourier-Motzkin elimination generates 243 393 constraints, only 45 683 of which are nonredundant. In the mean time, the parametric simplex algorithm with Bland's pivoting rule generates 234 516 regions which, once duplicate constraints are removed, yield 45 683 constraints as well. With the pivoting rule that looks for coefficients which are always negative in the context, 193 378 regions are generated. Two things can be concluded from these numbers.

- For each generated constraint, redundant or not, Fourier-Motzkin elimination solves at most one linear problem, when the constraint isn't syntactically redundant. On the other side, each region in the result of the parametric simplex algorithm is the result of potentially many sign tests and branches and each sign test involves solving one or two linear problems. However, there are approximately as many regions as constraints generated by Fourier-Motzkin elimination, making a good candidate explanation for the performance difference.

- When we compare the two pivoting rules, we find that the number of regions resulting from the most efficient rule is about 80% of that resulting from Bland's rule. At the same time, the execution time of the former is about 80% of that of the latter. This points to the oversplitting phenomenon described above being a major cause of the poor performance.

More evidence of oversplitting being a problem is found by examining the time spent performing sign tests for each pivoting rule: 64 seconds for the branch-delaying one, 83 seconds for Bland's. This accounts for nearly all the execution time difference between the two pivoting rules. As a result from these observations, we tried to design an algorithm which doesn't subdivide optimal regions.

7.3 Recent work

Hunting for improvement directions lead us to survey existing work in the area of solving parametric linear problems. We found two major tracks, both of which focus on problems with parametric objectives.

7.3.1 Instantiating solver

Andy King and Jacob Howe laid the basis of a solving technique for parametric linear problems based on parameter instantiation [32]. I am collaborating with them to design the actual algorithm and implement it. This is another track of work in progress, which I won't detail beyond the basic principle.

The idea is to disregard finding regions in the space of parameters. Instead of keeping the parameters symbolic, as the parametric simplex algorithm does, the solving algorithm instantiates them with concrete values. Doing so, it enumerates the vertices of the polytope $\tilde{s}_\Lambda$ described in the previous chapter.

Moving from one vertex to another requires adjusting the value of the parameters. The strategy essentially relies on the continuity of the solution to the parametric linear problem: when the value of the parameters are chosen on the frontier between two neighbouring regions, the vertices associated to the two regions yield the same objective value.

These ideas seem promising, but there are still a number of corner cases which need to be properly understood before a correct algorithm can emerge.

7.3.2 Local exploration

Another line of work emerges from Colin Jones's work [34]. Specific details are found in a subsequent paper that he wrote with Jan Maciejowski and Eric Kerrigan [35]. We will build on two main ideas:

- building regions from the coefficients of the objective function and
- local exploration by crossing frontiers.

Building regions from the objective function

The parametric simplex algorithm reaches the end of an exploration branch when the context it built implies that all the coefficients of the objective function are nonnegative. If we call $c_1(x), \ldots, c_l(x)$ these coefficients, the largest context which implies that the optimal is reached is the following.

$$c_1(x) \geq 0 \wedge \ldots \wedge c_l(x) \geq 0$$

Note that "largest" here refers to the context seen as a polyhedron: no other context implies that the optimal is reached and includes the one given above. From this observation, it has been proved [35] that the polyhedron defined by nonnegativity constraints on the coefficients of the objective function gives the whole region for which the objective value is optimal.

However this holds only on nondegenerate problems. Indeed, when a problem is degenerate, several choices of dependent variables describe the same optimal point in space. To each of these choices, corresponds a way of writing the objective function. Each of these ways yields a region fragment and their union gives the whole region. The authors avoid this problem by resorting to symbolic perturbations of the system, which introduce a small overhead at each pivot.

In any case, the region read from the objective function doesn't have a minimised constraint representation. Redundancy needs to be removed in order to find the true frontiers of the region. A "true" frontier in this context is a nonredundant constraint of a region.

Crossing frontiers

Once the frontiers of a region are known, it becomes tempting to look at what lies on the other side of each of them. From the fact that the regions partition the space of parameters, we know that there is another region on the other side of each frontier. Exploring the space using this idea reduces solving a parametric linear problem to exploring a graph whose nodes are the regions and whose edges are frontiers between them. Given a region, i.e. a node, one can cross one of its frontiers, i.e. edges, and compute a point in a neighbouring region. This point can then be used to reach the optimal vertex corresponding to the neighbouring region and, then, to recover the whole region from the objective function.

This idea lead Colin Jones et al. to a new solving algorithm. However, this algorithm relies fundamentally on the ability to find the true frontiers of the regions. As a result, the constraint representation of each region needs to be minimised. This is a costly operation which may hinder the performance of the whole algorithm. Unfortunately, its experimental validation, as reported in the paper, seems limited and no implementation is publicly available.

7.4 The idea which we start from

Our next experiment builds on two ideas:

- recovering the region from the objective function and
- finding a point in a region.

More precisely, under the assumption that we could get a point in each region for free, would we be able to build a more efficient projection operator than VPL Fourier-Motzkin elimination?

It is reasonable to expect so. Suppose we are given a point $\underline{x}$ in a region $\mathcal{R}$ and that we are trying to reconstruct region $\mathcal{R}$ and, more importantly, the corresponding parametric objective value. In the encoding of projection as a linear problem with a parametric objective function, point $\underline{x}$ can be used for testing the sign of parametric coefficients. If the parametric objective function is $\sum_{i=1}^{l} c_i(x) \cdot \lambda_i$, we may as well optimise the nonparametric function $\sum_{i=1}^{l} c_i(\underline{x}) \cdot \lambda_i$. This will lead us to the vertex $\underline{\lambda}$ of the search space which is optimal for the choice of parameters $\underline{x}$ and is therefore optimal for all region $\mathcal{R}$. What's left to do is recovering the parametric coefficients of the objective function and the parametric objective value at the optimal. This could be achieved by solving the linear problem with nonparametric objective function $\sum_{i=1}^{l} c_i(\underline{x}) \cdot \lambda_i$ and then pivoting in the original, parametric, problem so as to make $\underline{\lambda}$ the basic solution. Instead, we chose to keep the parametric objective function throughout the optimisation. Then, when the solving algorithm needs to know the sign of a parametric coefficient $c'_i(x)$, the sign of $c'_i(\underline{x})$ is used. In essence, we get the benefit of instantiating the parameters with point $\underline{x}$—the sign tests are very cheap—and the benefit of keeping the objective function parametric: the region and parametric objective value can be recovered at the end of the optimisation. This is a form of parameter instantiation similar to that performed by Andy King and Jacob Howe.

Under the assumption that we can obtain a point in each region at no computational cost, we are essentially solving one nonparametric linear problem for

each region in the result, that is to say for each *nonredundant* constraint in the projected polyhedron. In comparison, Fourier-Motzkin elimination solves a linear problem for each constraint *it generates*, provided it could not prove it redundant by syntactic inclusion. Therefore, assuming we have a point in each region should yield a more efficient projection operator. If it were the case, we would be left with the problem of finding a point in each region at little cost. Our initial attempt used an SMT solver to find a point in unexplored regions.

Finding a point in the unexplored space

As we mentioned several times already, each region in the solution of a parametric linear problem is a polyhedron. Furthermore, at any given point of the exploration, the subspace which has already been explored is given by the disjunction of all the regions which have already been discovered. Suppose for example that the solver already discovered two regions $\mathcal{R}_1$ and $\mathcal{R}_2$. Then a point $\underline{x}$ is in the subspace which has already been explored if it satisfies the following condition.

$$\underline{x} \in \mathcal{R}_1 \vee \underline{x} \in \mathcal{R}_2$$

If $\mathcal{R}_1 = \{c_1(x) \geq 0, \ldots, c_l(x) \geq 0\}$ and $\mathcal{R}_2 = \{c'_1(x) \geq 0, \ldots, c'_{l'}(x) \geq 0\}$, then this is equivalent to saying that point $\underline{x}$ satisfies the following formula.

$$\big(c_1(x) \geq 0 \wedge \ldots \wedge c_l(x) \geq 0\big) \vee \big(c'_1(x) \geq 0 \wedge \ldots \wedge c'_{l'}(x) \geq 0\big) \tag{7.1}$$

Formula 7.1 is a satisfiability modulo theory (SMT) formula. Its complement describes the points which are in the yet-unexplored subspace of the parameter space. It is also an SMT formula, which is given below with negation pushed to individual constraints using de Morgan's laws.

$$\big(c_1(x) < 0 \vee \ldots \vee c_l(x) < 0\big) \wedge \big(c'_1(x) < 0 \vee \ldots \vee c'_{l'}(x) < 0\big) \tag{7.2}$$

We may now use an SMT solver, such as Z3 [19], to find a point in the unexplored subspace, that is to say find a satisfying assignment for formula 7.2. When the SMT solver concludes that the formula above is unsatisfiable, the whole space has been covered and the algorithm terminates.

Reaching the end of a branch

After optimisation for the value of parameters $\underline{x}$ finishes, the objective value gives a nonredundant constraint of the result and the corresponding region is built from the coefficients of the objective function. The resulting region $\bigwedge_{i=1}^{l''} c''_i(x) \geq 0$ is then removed from the subspace left to explore, by conjoining formula 7.2 with the complement of the region $\bigvee_{i=1}^{l''} c''_i(x) < 0$. Note that the algorithm currently doesn't pay special attention to whether the optimal is degenerate: a region is always built from the available coefficients. As we'll see below, this results in more regions than actual constraints.

Algorithm outline

Overall, the algorithm for our experiment is as follows.

1. Use an SMT solver to find a point in the space of parameters which doesn't belong to an already-explored region.

2. Use this point to determine the sign of the coefficients of the objective function when optimising.

3. Save the optimal objective value.

4. Extract the region from the optimal objective function and remove it from the space left to explore.

5. Go back to step 1.

Evaluation

The solver uses Z3 for solving SMT formulas. When run on the benchmark suite, it generates only 66 075 regions, which make 45 683 constraints, once duplicates are removed. The gap between the two numbers is accounted for by the new solver not handling degenerate problems in a special way: when a given optimal is degenerate, the coefficients of the objective function don't give the whole region. This being said, much fewer duplicates are generated than when using the parametric simplex algorithm.

The solver takes 72 seconds to complete, which is a small improvement on the previous one, but that was not the point of this experiment. In order to draw conclusions from this experiment, the time spent in the SMT solver needs to be subtracted from the total time. A total of 38 seconds out of the total 72 seconds are spent in the SMT solver. Some more time is spent minimising the constraint representation of regions, as keeping all the constraints makes the SMT formula grow a lot. In total, 12 more seconds are spent minimising regions. We consider this time as part of the point selection step.

Finally, we can estimate to $72 - 38 - 12 = 22$ seconds the time required to recover all the regions from a point in each of them. These results are unexpected: if the cost of finding a point in each region is removed, the solver spends 22 seconds to run over the benchmark suite, as opposed to 3.7 seconds for VPL projection operator.

We still need to diagnose precisely this difference between our expectations and the actual results. Here are a few hypotheses. First, the implementation of the projection operator in VPL was profiled and reasonably optimised. In contrast, the experimental solver supports ongoing research and is patched and refactored frequently, yielding a suboptimal design. There may also be a more fundamental reason for the poor performance we observed: the experimental solver doesn't benefit from locality. Finding a new region starts with finding a point which is outside of all the discovered regions. The corresponding query to the SMT solver describes such points in general, without bias towards point closed to the already-explored subspace, which may be cheaper to reach. The issue can be illustrated on figure 7.3, which represents a polytope where each vertex is optimal for a region. Suppose vertex $\underline{\lambda}_4$ is the first vertex reached during exploration, using a first point $\underline{x}_4$. Assume now that the SMT solver gives point $\underline{x}_2$ for the next step and that the optimal vertex corresponding to point $\underline{x}_2$ is vertex $\underline{\lambda}_2$. The optimisation will reach vertex $\underline{\lambda}_2$ through a sequence of pivots, for example $\underline{\lambda}_4 \to \underline{\lambda}_5 \to \underline{\lambda}_6 \to \underline{\lambda}_1 \to \underline{\lambda}_2$. If the next point given by the SMT solver $\underline{x}_6$, for which vertex $\underline{\lambda}_6$ is optimal, the exploration process will go back to vertices it has already visited without paying attention. As a result, we suspect that picking a point in the next region which is close to the vertex

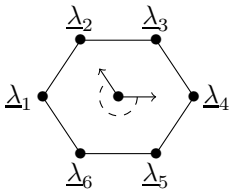


Figure 7.3 – vertex exploration order

corresponding to the region we just found will help. Of course, these are just hypotheses, which need to be checked before further progress can be made.

7.5 Further directions

So far, we experimented with ideas published by others and these experiments brought interesting remarks. However, our setting, projection for static analysis, has a number of characteristics which could be exploited in order to build an efficient solver.

Central point. The encoding of projection as a parametric linear problem is such that all the regions meet in the central point. This fact seems related to Jacob Howe and Andy King's choice of a parametric objective function for their parameter-instantiating solver [32]. Their objective function is designed so that it can describe all the possible directions in the polytope in which the variables lie. I have concerns about this working without the star-like shape of the regions. How this central point can be exploited still remains unclear.

Cheaper frontier detection. The proposal of Colin Jones et al. [35] for finding points in neighbouring regions builds on the idea of having a minimised representation of each region. Minimisation is performed just like the minimisation of a polyhedron, which is costly. However, given that the set of regions partition the parameter space, the complement of one frontier of a region is necessarily a frontier of another region. Furthermore, the regions have full dimension. Combined with the input constraints being normalised using an interior point, this forces a canonical representation for frontiers. Whether the complement of a candidate frontier has already been encountered can therefore be tested using cheaper comparisons.

Discarding regions. Further down the road of reducing the computations performed on regions, we may remark that they are no more than an intermediate result in our computation: they are of no interest in the construction of the result of projection. What is really of interest is the set of affine forms which give the value to solution function of the parametric linear problem. Jacob Howe and Andy King's approach [32] bypasses computing regions entirely by instantiating the parameters of the problem. Even without going this far, a redundant, or partially minimised, region representation may be sufficient for our needs.

Converging threads. Building on these ideas, the next step towards building an efficient solver needs to consider the following remaining questions.

- How can we get points inside unexplored regions at little cost? The idea of crossing frontiers from Colin Jones et al. seems promising, provided that the full minimisation of regions can be bypassed.
- What is the stopping criterion? A graph exploration algorithm based on the idea of Colin Jones et al. needs to consider all the vertices, that is to say that every frontier needs to be crossed. This is very costly in terms of computations.

The approach we have been following for the yet-unfinished design of a new projection operator has shed an interesting light on the concurrent work of different research groups. The ideas on all sides seem related but emerging from different research communities. We put a significant effort in bridging the terminology gap and in exhibiting similarities. We hope to contribute to making the various lines of work converge.

Summary in French

Une fois l'objectif clarifié, nous pouvons nous intéresser à sa réalisation.

Sur la collection de cas d'étude tirée de l'évaluation expérimentale de VPL, la résolution des projections par l'algorithme du simplexe paramétrique est très lente. L'analyse du profil d'exécution de l'algorithme indique que chacune des régions qui constituent la fonction solution est découverte par fragments. Le temps d'exécution de l'algorithme du simplexe paramétrique dépend du nombre de fragments, qui apparaissent en très grand nombre par rapport au nombre de régions.

J'ai essayé de résoudre ce problème en concevant un algorithme qui ne fragmente pas les régions. Cet algorithme s'inspire de nouvelles façons de résoudre les problèmes linéaires paramétriques encodant des projections, proposées récemment par plusieurs groupes de chercheurs. Mon premier essai repose sur le fait que les régions sont des polyèdres qui partitionnent l'espace et qu'il est possible de construire une région complète à partir d'un fragment découvert par l'algorithme du simplexe paramétrique. À la limite, un point constitue un fragment d'une région. L'idée est de chercher un point dans la partie encore non-explorée de l'espace des paramètres, de construire la région correspondante. La région ainsi construite est ajoutée à l'espace déjà exploré et l'algorithme cherche un nouveau point. L'espace non-exploré est le complémentaire d'une union de polyèdre : il est décrit par une formule de satisfaisabilité modulo théorie. Un nouveau point qui satisfait cette formule est trouvé à l'aide d'un outil tel que Z3.

Bien que cette nouvelle approche soit plus efficace que l'algorithme du simplexe paramétrique, l'opérateur de projection de VPL basé sur l'élimination de Fourier-Motzkin demeure un ordre de grandeur plus efficace. Une piste d'amélioration sur l'idée d'exploration par des points consiste à chercher de nouveaux points au voisinnage de la région nouvellement découverte, de façon à accéler le

reconstruction de la région suivante. Par manque de temps, ce nouvel essai est encore travail en cours.

Behind and ahead

The overarching theme is the construction of VPL, an abstract domain of polyhedra with a proof of correctness written using the Coq proof assistant.

Modular proof of correctness. My work pushed the result verification approach to guaranteeing correctness to a limit. It builds on a frontend written in Coq and an OCaml oracle. They have independent representations of polyhedra and synchronise through a limited amount of communication. The frontend itself relies on a reduced set of operators from the oracle and builds the whole domain using functors, which make the proof easier to maintain. The resulting domain exhibited only a low execution overhead related to proofs.

Witness extraction. For most part, this achievement results from inclusion witnesses being extracted on the fly by the oracle, from the working data of the domain operators. It represents a significant improvement over the state of the art [6], where witnesses are computed after the fact using a linear solver, which duplicates some of the work. In contrast, the approach I proposed relies on simple bookkeeping.

External code. When a trusted Coq program relies on external OCaml code, this code must be declared to Coq as axioms. The axioms may introduce implicit assumptions on the behaviour of the OCaml code, which are worth scrutinising. In particular, OCaml functions may not be observationally pure—they can have side effects—and their straightforward axiomatisation in Coq may allow proving falsities. In collaboration with Sylvain Boulmé, I proposed a simple formalisation of side-effecting OCaml functions, which adds little to the proof burden and guarantees that the resulting proof doesn't rely on the functional purity of the OCaml code. This work resulted in the may-return monad.

Constraint representation. VPL performs competitively with state-of-the-art implementations of the abstract domain of polyhedra, while providing extra guarantees on the results. The result is made significant by the choice of a constraints-only representation for polyhedra, thereby departing from the common practice of using both constraints and generators. From a performance point of view, going constraints-only proved to be a reasonable choice and allowed to keep the implementation of VPL simple.

Projection as a parametric linear problem. As expected, computing convex hulls is the major performance sink of VPL. The later part of my work was devoted to building a more efficient projection operator, on which the convex hull operator is based. The basis of this work is the expression of polyhedral projection as a parametric linear problem. My contribution to this line of work is proposing a new understanding by encoding projection in a dual way to existing work.

Designing a new parametric linear solver. All proposals of polyhedral projection using parametric linear programming suffer from a lack of experimental validation, especially on static analysis problems. This is a particularly salient issue, because my preliminary evaluation using the standard solving algorithm—the parametric simplex algorithm—shows poor performance. I reported on a joint effort with Alexandre Maréchal to evaluate systematically the ideas proposed towards building a new solver. Building a more efficient projection operator than that of VPL, which is based on Fourier-Motzkin elimination, is still a goal to achieve.

David Kohler's criterion. A last, minor, contribution of my work consists in remarking that David Kohler's criterion for removing some redundant constraints during the Fourier-Motzkin elimination of several variables is actually incompatible with eager minimisation of intermediate results. Since the latter is commonly performed in implementations of Fourier-Motzkin elimination, this observation raises the question of how to exploit David Kohler's criterion in a sound manner.

Publications

The work reported in the first part of this thesis resulted in three publications.

SAS' 13 "Efficient certificate generation for the abstract domain of polyhedra", with Michaël Périn and David Monniaux

TYPES' 14 "Modular and lightweight certification of polyhedral abstract domains", extended abstract, with Sylvain Boulmé

VSTTE' 14 "Certifying frontend for (sub)polyhedral abstract domains" with Sylvain Boulmé

The first paper describes mainly the overall architecture of VPL and the design of the oracle. The two other publications cover the design of the Coq frontend.

Tools

Besides writing papers, I implemented tools to support my work. The most mature of them, VPL, is integrated to the collection of abstract domains in the Verasco analyser.

- VPL is an abstract domain of polyhedra which comes with a soundness proof and is built using the Coq proof assistant. It has two components: a witness-checking frontend and an OCaml oracle. The frontend was written in collaboration with Sylvain Boulmé. The oracle itself is a complete

abstract domain written in OCaml, which can be used independently in analysers.

- I also wrote, in collaboration with Alexandre Maréchal, a solver for parametric linear problems, with parameters in the objective function. It was initially based on the parametric simplex algorithm and evolves as we evaluate new ideas both from the published prior art and our own.

- Last, I implemented a parameter-instantiating parametric linear solver from the ideas of a paper by Jacob Howe and Andy King [32]. This was done in collaboration with the authors and is still ongoing work.

Further directions

There is still progress to be made in the area of projection using parametric linear programming. As the conclusion of chapter 7 points out, many ideas, from several groups of researchers, have been published recently. They all seem to converge: time may be ripe to give the final stroke.

On a more pessimistic note, one could note that Fourier-Motzkin elimination is the simplest algorithm for eliminating variables in polyhedra. Yet, our elaborate attempts at designing a more efficient algorithm aren't competitive, by a large margin. It would be interesting to find out theoretical justifications of projection as parametric linear problem being an inferior approach.

Taking one step back, integrating VPL as an abstract domain in the Verasco analyser has revealed a number of performance issues in the combination of the two. Part of the problem seems related to Verasco repeatedly asking VPL for the value range of expressions, which is costly for the abstract domain of polyhedra, since it requires solving two linear problems each time. These repeated requests come from the way in which machine arithmetic is handled. VPL itself has no support for it: it relies on a functor in Verasco which repeatedly checks whether the value interval of each variable lies within the range of machine integers. Several approaches to fixing this performance issue may be considered. The most straightforward is adding machine arithmetic support to VPL, following the work of Axel Simon and Andy King [48]. Another approach may be considered, in the spirit of the work of Arie Gurfinkel et al. [29]: invariants found with abstractions over integers could be used as a help for inferring sound invariants over machine integers.

Longer-term ideas

Two longer-term directions emerge from the work I reported in this thesis. They should be regarded more as potential research programs, rather than immediate projects.

Selective enabling. Polyhedra are a computationally expensive abstraction which hardly scales. As a result, many restrictions of polyhedra, such as octagons [42] or subpolyhedra [41], have been introduced. An alternative approach may be worth investigating: limiting their use in scope. Instead of having the domain active throughout the analysis, it may be interesting to turn it on for portions of the code where the extra precision is necessary, when the analyser fails to prove an assertion with a less precise domain for example. Limiting the

use of polyhedra may also take the form of using the anytime approximation characteristic of projection as a parametric problem: solving it for only an interesting subset of the parameter space may save significant computation effort. Buffering calls to operators so that they may be reordered follows a similar spirit. In all cases, the abstract domain would require a closer cooperation from the analyser, the details of which make an interesting topic.

Linear programming Coq tool box. Implementing and designing algorithms from the field of linear programming proved to be surprisingly tricky. Many, rarely encountered, corner cases compromise the correctness of either algorithm or code. The delicate use of David Kohler's criterion is but one example. One way to address this issue may be to build a tool box in Coq to help reason on the topic. It may come as a formalisation of the core concepts with an accompanying collection of lemmas. Although, it wouldn't make *designing* easier, it would help uncovering hidden assumptions and easily overlooked corner cases in promising ideas.

Summary in French

La construction de VPL est le thème principal de mon travail. VPL est un domaine abstrait de polyèdres dont une preuve de correction est écrite à l'aide de l'assistant à la preuve Coq. La preuve est construite suivant le principe de la vérification de résultats. Cette architecture permet de décorréler l'implémentation des opérateurs de la preuve de leur correction. La preuve elle-même est découpée en un ensemble de foncteurs, chacun ajoutant un opérateur à un squelette simple. Vérifier en Coq des résultats calculés par un programme Ocaml nécessite une communication entre les deux environnements. Ce lien est réalisé à l'aide d'axiomes. Une de mes contributions, réalisée en collaboration avec Sylvain Boulmé, est une théorie Coq simple qui permet de se protéger des contradictions que peuvent introduire de l'ajout de nouveaux axiomes dans Coq. Une autre contribution est de montrer qu'il est réaliste de n'utiliser que la représentation par contraintes des polyèdres. En effet, les performances de VPL sont comparables aux implémentations existantes, bien que VPL soit le seul domaine dont les résultats sont formellement prouvés corrects. Ce faible coût de la preuve découle d'une nouvelle façon de construire les témoins d'inclusion, en les extrayant des données de travail des opérateurs du domaine. L'opérateur «join» est le plus coûteux de VPL. Il est construit à partir de l'opérateur de projection, pour laquelle de nouveaux algorithmes ont récemment été proposés. Mon travail propose une nouvelle explication des fondements théoriques de ces algorithmes. Il se conclut sur une évaluation expérimentale en cours de ces nouvelles idées.

Mon travail a fait l'objet de deux articles publiés dans des conférences internationales et d'un exposé présenté dans un workshop. J'ai développé deux outils : le domaine abstrait VPL et un outil de résolution de problèmes linéaires paramétriques.

Bibliography

[1] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58, 2005.

[2] R. Bagnara, P. M. Hill, and E. Zaffanella. Not necessarily closed convex polyhedra and the double description method. *Formal Aspects of Computing*, 17, 2005.

[3] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2), 2008.

[4] G. Balakrishnan, T. W. Reps, D. Melski, and T. Teitelbaum. Wysinwyx: What you see is not what you execute. In *Verified Software: Theories, Tools and Experiments*, 2005.

[5] F. Benoy, A. King, and F. Mesnard. Computing convex hulls with a linear solver. *Theory and Practice of Logic Programming*, 5(1–2), 2005.

[6] F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Technical Report RR-6333, INRIA, 2007.

[7] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation*, 2003.

[8] R. G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2, 1977.

[9] S. Boulmé and A. Maréchal. Refinement to certify abstract interpretations, illustrated on linearization for polyhedra. In *Interactive Theorem Proving*, 2015.

[10] Bugseng. *The Parma Polyhedra Library*, 1.0 edition, 2012.

[11] D. Caminha Barbosa de Oliveira and D. Monniaux. Experiments on the feasibility of using a floating-point simplex in an SMT solver. In *Workshop on Practical Aspects of Automated Reasoning*, 2012.

[12] N. V. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 1968.

[13] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2013.

[14] G. Claret, L. D. C. Gonzalez Huesca, Y. Régis-Gianas, and B. Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *Interactive Theorem Proving*, 2013.

[15] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *International Symposium on Programming*, 1976.

[16] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, 1977.

[17] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages*, 1978.

[18] G. B. Dantzig and M. N. Thapa. *Linear Programming 2: Theory and Extensions*. Springer, 1997.

[19] L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[20] B. Dutertre and L. de Moura. Integrating simplex with dpll(t). Technical report, SRI International, 2006.

[21] P. Feautrier. PIP/PipLib. http://piplib.org.

[22] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22, 1988.

[23] A. Fouilhé and S. Boulmé. A certifying frontend for (sub)polyhedral abstract domains. In *Verified Software: Theories, Tools and Experiments*, 2014.

[24] A. Fouilhé, D. Monniaux, and M. Périn. Efficient certificate generation for the abstract domain of polyhedra. In *Static Analysis Symposium*, 2013.

[25] J. Fourier. Histoire de l'académie, partie mathématique. In *Mémoire de l'Académie des sciences de l'Institut de France*. Imprimerie Royale (Paris), 1827.

[26] Free Software Foundation. *The GNU Multiple Precision Arithmetic Library*, 5.0 edition, 2012.

[27] Free Software Foundation. *The GNU Linear Programming Kit*, 4.55 edition, 2014.

[28] T. Gal and J. Nedoma. Multiparametric linear programming. *Management Science*, 18, 1972.

[29] A. Gurfinkel, A. Belov, and J. Marques-Silva. Synthetizing safe bit-precise invariants. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2014.

[30] N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme.* PhD thesis, Université scientifique et médicale de Grenoble, 1979. In French.

[31] J. Henry, D. Monniaux, and M. Moy. PAGAI: a path sensitive static analyser. In *Tools for Automatic Program Analysis*, 2012.

[32] J. Howe and A. King. Polyhedral analysis using parametric objectives. In *Static Analysis Symposium*, 2012.

[33] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer-aided Verification*, 2009.

[34] C. N. Jones. *Polyhedral tools for control.* PhD thesis, University of Cambridge, 2005.

[35] C. N. Jones, E. C. Kerrigan, and J. M. Maciejowski. Lexicographic perturbation for multiparametric linear programming with applications to control. *Automatica*, 2007.

[36] C. N. Jones, E. C. Kerrigan, and J. M. Maciejowski. On polyhedral projection and parametric programming. *Journal of Optimization Theory and Applications*, 138, 2008.

[37] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *Principles of Programming Languages*, 2015.

[38] N. Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4, 1984.

[39] D. A. Kohler. *Projections of convex polyhedral sets.* PhD thesis, University of California, Berkeley, 1967.

[40] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009.

[41] F. Logozzo and V. Laviron. SubPolyhedra: a (more) scalable approach to infer linear inequalities. In *Verification, Model Checking and Abstract Interpretation*, 2009.

[42] A. Miné. The octagon abstract domain. In *Workshop on Analysis, Slicing, and Transformation*, 2001.

[43] A. Miné and X. Leroy. ZArith. http://forge.ocamlcore.org/projects/zarith.

[44] D. Monniaux. On using floating-point computations to help an exact linear arithmetic decision procedure. In *Computer-aided Verification*, 2009.

[45] D. Monniaux. Quantifier elimination by lazy model enumeration. In *Computer-aided verification*, 2010.

[46] D. Pichardie. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés.* PhD thesis, Université Rennes 1, 2005. In French.

[47] A. Simon and A. King. Exploiting sparsity in polyhedral analysis. In *Static Analysis Symposium*, 2005.

[48] A. Simon and A. King. Taming the wrapping of integer arithmetic. In *Static Analysis Symposium*, 2007.

[49] M. Sozeau and N. Oury. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, 2008.

[50] The Coq Development Team. *The Coq proof assistant reference manual*. INRIA, 8.4 edition, 2012.

[51] H. L. Verge. A note on Chernikova's algorithm. Technical report, INRIA, 1992.

[52] D. Wilde. PolyLib. http://icps.u-strasbg.fr/polylib.

Index

Abstract

The work reported in this thesis revisits in two ways the abstract domain of polyhedra used for static analysis of programs. First, strong guarantees are provided on the soundness of the operations on polyhedra, by using of the Coq proof assistant to check the soundness proofs. The means used to ensure correctness don't hinder the performance of the resulting Verimag Polyhedra Library (VPL). It is built on the principle of result verification: computations are performed by an untrusted oracle and their results are verified by a checker whose correctness is proved in Coq. In order to make verification cheap, the oracle computes soundness witnesses along with the results. The other distinguishing feature of VPL is that it relies only on the constraint representation of polyhedra, as opposed to the common practice of using both constraints and generators. Despite this unusual choice, VPL turns out to be a competitive abstract domain of polyhedra, performance-wise.

As expected, the join operator of VPL, which performs the convex hull of two polyhedra, is the costliest operator. Since it builds on the projection operator, this thesis also investigates a new approach to performing projections, based on parametric linear programming. A new understanding of projection encoded as a parametric linear problem is presented. The thesis closes on a progress report in the design of a new solving algorithm, tailored to the specifics of the encoding so as to achieve good performance.

Résumé

Cette thèse revisite de deux manières le domaine abstrait des polyèdres utilisé pour l'analyse statique de programmes. D'abord, elle montre comment utiliser l'assistant à la preuve Coq pour apporter des garanties sur la correction des opérations sur les polyèdres sans compromettre l'efficacité de l'outil VPL issu de ces travaux. L'outil est fondé sur le principe de la vérification de résultats : un oracle, auquel on ne fait pas confiance, fait les calculs, puis les résultats sont vérifiés par un validateur dont la correction est prouvée avec Coq. De plus, l'oracle fournit des témoins de la correction des résultats afin d'accélérer la vérification. L'autre caractéristique de VPL est l'utilsation de la seule représentation par contraintes des polyèdres, par opposition à l'approche habituelle qui consiste à utiliser à la fois des contraintes et des générateurs. Malgré ce choix inhabituel, les performances de VPL s'avèrent compétitives.

Comme on pouvait le prévoir, l'opérateur *join*, qui calcule l'enveloppe convexe de deux polyèdres, est le plus coûteux. Puisqu'il nécessite un grand nombre de projections, cette thèse explore plusieurs nouvelles approches de l'opérateur de projection, basées sur la programmation linéaire paramétrique. Elle propose une sythèse des variantes et des combinaisons possibles. La thèse se termine sur les éléments clés d'un nouvel algorithme de résolution tirant parti des spécificités de l'encodage afin d'obtenir de bonnes performances.