Static analysis of memory manipulations by abstract interpretation – Algorithmics of tropical polyhedra, and application to abstract interpretation

Xavier Allamigeon

Thèse présentée pour obtenir le grade de

DOCTEUR DE L'ÉCOLE POLYTECHNIQUE

Spécialité : Informatique

par

Xavier Allamigeon

Static analysis of memory manipulations by abstract interpretation

Algorithmics of tropical polyhedra, and application to abstract interpretation

soutenue le 30 novembre 2009 devant le jury composé de :

Jean-Eric Pin	LIAFA – CNRS	président du jury
Nicolas Halbwachs	Verimag – CNRS	rapporteur
Michael Joswig	Technische Universität Darmstadt	rapporteur
Peter Butkovič	University of Birmingham	examinateur
Stéphane Gaubert	INRIA Saclay	examinateur
David Monniaux	Verimag – CNRS	examinateur
Eric Goubault	CEA List, MeASI	directeur de thèse
Charles Hymans	EADS Innovation Works	encadrant de thèse

Merci à / Thanks to ...

Je tiens de tout coeur à remercier :
I'd really like to thank:

- mes trois directeurs de thèse (officiels ou officieux), Eric, Charles et Stéphane pour leur encadrement de si grande qualité,

  my three advisors (officially or unofficially), Eric, Charles, and Stéphane, for their great supervision,

- Nicolas et Michael pour avoir rapporté ce long manuscrit, pour leur expertise et leurs remarques constructives,

  Nicolas and Michael for having reported such a long manuscript, for their expertise and their helpful remarks,

- Peter, David pour avoir accepté spontanément de participer au jury de ma thèse, et Jean-Eric pour l'avoir présidé,

  Peter, David, for having spontaneously accepted to take part in the jury, and Jean-Eric for having chaired it,

- mes collègues d'EADS et du CEA, avec qui j'ai passé de très bons moments tout au long de cette thèse,

  my officemates at EADS and CEA, for the great moments we shared together,

- mes amis pour leurs encouragements,

  my friends for their encouragements,

- ma famille pour m'avoir toujours soutenu,

  my family for their everlasting support,

- et Audrey, pour tout.

  and Audrey, for all.

Abstract

In this thesis, we define a static analysis by abstract interpretation of memory manipulations. It is based on a new numerical abstract domain, which is able to infer program invariants involving the operators min and max. This domain relies on tropical polyhedra, which are the analogues of convex polyhedra in tropical algebra. Tropical algebra refers to the set $\mathbb{R} \cup \{-\infty\}$ endowed with max as addition and + as multiplication.

This abstract domain is provided with sound abstract primitives, which allow to automatically compute over-approximations of semantics of programs by means of tropical polyhedra. Thanks to them, we develop and implement a sound static analysis inferring min- and max-invariants over the program variables, the length of the strings, and the size of the arrays in memory.

In order to improve the scalability of the abstract domain, we also study the algorithmics of tropical polyhedra. In particular, a tropical polyhedron can be represented in two different ways, either internally, in terms of extreme points and rays, or externally, in terms of tropically affine inequalities. Passing from the external description of a polyhedron to its internal description, or inversely, is a fundamental computational issue, comparable to the well-known vertex/facet enumeration or convex hull problems in the classical algebra. It is also a crucial operation in our numerical abstract domain.

For this reason, we develop two original algorithms allowing to pass from an external description of tropical polyhedra to an internal description, and vice versa. They are based on a tropical analogue of the double description method introduced by Motzkin *et al.*. We show that they outperform the other existing methods, both in theory and in practice. The cornerstone of these algorithms is a new combinatorial characterization of extreme elements in tropical polyhedra defined by means of inequalities: we have proved that the extremality of an element amounts to the existence of a strongly connected component reachable from any node in a directed hypergraph. We also show that the latter property can be checked in almost linear time in the size of the hypergraph.

Moreover, in order to have a better understanding of the intrinsic complexity of tropical polyhedra, we study the problem of determining the maximal number of extreme points in a tropical polyhedron. In the classical case, this problem is addressed by McMullen upper bound theorem. We prove that the maximal number of extreme points in the tropical case is

bounded by a similar result. We introduce a class of tropical polyhedra appearing as natural candidates to be maximizing instances. We establish lower and upper bounds on their number of extreme points, and show that the McMullen type bound is asymptotically tight when the dimension tends to infinity and the number of inequalities defining the polyhedra is fixed.

Finally, we experiment our tropical polyhedra based static analyzer on programs manipulating strings and arrays. These experimentations show that the analyzer successfully determines precise properties on memory manipulations, and that it scales up to highly disjunctive invariants which could not be computed by the existing methods.

The implementation of all the algorithms and abstract domains on tropical polyhedra developed in this work is available in the Tropical Polyhedra Library TPLib [All09].

Contents

CHAPTER 1

Introduction

1.1 Context of this work

The context of this thesis is software analysis. This area has known an important development over the last decades. Indeed, software is nowadays omnipresent in highly critical systems, such as nuclear installation supervision, flight control in airplanes (*fly-by-wire*), medical imaging, driving assistance in cars, *etc*. If a program does not behave as expected, which we usually call a *bug*, the consequences can be disastrous. For instance, a bug in software control of the radiation therapy machine Therac-25 caused in the late eighties at least six massive radiation overdoses, three of which were lethal. Other well-known examples are the explosion of the first flight of the rocket Ariane 5 [LLF+96], or the accident due to the failure of the Patriot missile at Dhahran in Saudi Arabia [Off92]. Bugs can also introduce software vulnerabilities which can be exploited by outside attacks. An important class of such bugs is formed by *buffer overflows*. They are caused by programmation errors in the manipulation of computer memory. A typical example of a buffer overflow is when a program accesses to the $(n+1)$-th element of an array of size n. Buffer overflows may cause software crashes in the best case (usually known as *segmentation faults*), or in the worst case, allow attackers to remotely take the control of the machine, and executing arbitrary codes. One of the first computer viruses, the *Morris worm*, infected in 1988 around 10% of the machines connected to Internet, by exploiting a buffer overflow [ER89]. Nowadays, buffer overflows still belong to the top 25 most dangerous programming errors referenced by the SANS Institute [SAN07]. For a recent account on vulnerabilities due to buffer overflows, see [CWE].

Year	Operating system	Lines of code (in million)
1993	Windows NT 3.1	6
1994	Windows NT 3.5	10
1996	Windows NT 4.0	16
2000	Windows 2000	29
2001	Windows XP	40
2005	Windows Vista Beta 2	50

Figure 1.1: Evolution of the size of the code of Microsoft Windows [Mar05]

On top of being unavoidable, software is today incredibly more complex. Whatever the application field, the size of program codes tends to grow significantly. The evolution of the size of Microsoft operating systems, given in Figure 1.1, is representative of this trend. As a more illustrative example, a Boeing 777 aircraft and a recent car embed respectively 4 and 35 million lines of code [RJP96, JRGJF06]. Now, the bigger a code is, the more bugs it may contain. McConnell estimates in [McC04] that industrial programs usually contain from 15 to 50 bugs for every 1000 lines of code.

Software analysis, or *program analysis*, aims at automatically determining properties on the behavior of programs. Examples of such properties are "the implementation of the cosinus function returns values between -1 and $+1$", or "the program does not perform any access out of the bounds of arrays". Two main classes of program analyses can be distinguished:

- *dynamic analysis*, which consists in observing how a program behaves by executing it on sample inputs,
- and *static analysis*, which, in contrast, is performed on a static representation of the program, such as its source code, without executing it.

Most of program analysis tools are not *sound*, in sense that they do not take into account all the possible behaviors of a given program. For instance, Ganssle estimates in [Gan05] that software tests, which belong to the class of dynamic analyses, only cover 50% of the behaviors.[1] Static analysis techniques may also be not sound. For instance, some tools, such as [Fla], only verify syntactically the absence of some particular functions known as "dangerous" (like the function `strcpy` in C).

Naturally, program analysis has to be sound to really ensure the absence of bugs in programs. Sound static analyses mainly rely on one of the three following approaches:

- *model checking* [CES86, EMCGP99], which consists, given a program and a property to be verified, in examining every state of the program which can arise during the execution, in order to determine whether the property is satisfied. However, this requires the number of possible states to be finite. In practice, this condition holds, since the amount of memory in a machine is finite. However, the number of states may be very large: for instance, without further assumptions, a program which manipulates d integer variables can take 2^{32d} different states on a 32-bit machine. As a consequence, model checking is rather adapted to the analysis of small programs, or of high-level models,

[1] Nevertheless, code covering tools may be used to increase this rate. They allow for instance every part of the code to be executed at least once during a set of tests. But it still does not guaranteed the exhaustivity of the analysis.

such as communication protocols, hardware designs, *etc.*[2] Examples of model checking based tools are BLAST [Bla], NuSMV [NuS], PVS [PVS], and Spin [Spi].

- *theorem proving*, which consists in translating a program and the property to be verified into formula in a given logic (see for instance [Fil03, Why]). Then, proof assistants, such as Coq [Coq] or Isabelle [Isa], can be used in order to prove the formula. The major drawback of this method is that it is not entirely automatic.

- *abstract interpretation* [CC77], which allows to automatically compute an over-approximation of the whole set of behaviors of a program. This over-approximation represents itself properties which are valid for any execution of the program. It can be used to verify for instance that the analyzed program does not contain any bug. However, if this over-approximation is not precise enough, a bug can be detected by the analysis while the program does not contain any. This is what we call a *false alarm*, or equivalently a *false positive*. Examples of static analysis tools based on abstract interpretation are Astrée [CCF+05, Ast], CGS [VB04], Fluctuat [GP06], Penjili [AGH06, Pen], Polyspace [Pol], and TVLA [SRW98, TVL].[3]

Each of the three methods has particular inconveniences. However, the "perfect" static analysis tool, which automatically and exactly determines whether a program satisfies a given property, *does not exist*. This is a consequence of Rice's theorem [Ric56], which states that any non-trivial property on a Turing-complete language is not decidable.

1.2 Analyzing memory manipulations by abstract interpretation

The initial motivation of this work is to define a sound static analysis of a certain class of memory manipulations, using abstract interpretation. In this section, we give a further insight into the principles of abstract interpretation and memory manipulations, in order to understand why the existing techniques of analysis are not fully satisfactory.

1.2.1 Main principles of abstract interpretation

The method of abstract interpretation determines properties on the set of all possible behaviors of programs. The set of the behaviors of a program is formalized as a mathematical object, which is called its *semantics*. For instance, we can use a collecting semantics, which captures, for each control point ℓ of the program, the whole set of machine states which can arise at the point ℓ during any of the executions. Ideally, examining this semantics would allow to prove properties on the program. However, because of Rice's theorem, the semantics is not computable in the general case. That is why abstract interpretation proposes to determine an over-approximation, which, in contrast, is computable. For each control point of the program, this over-approximation provides program invariants, *i.e.* properties which hold for any possible execution of the program. They can be used to check that the program has indeed the expected behavior, and that it does not cause any error.

[2] Note however that some methods such as [CGL94] have been introduced to reduce the number of states by means of abstractions, up to losing the exactness of the analysis. Moreover, some compact representations such as binary decision diagrams [Ake78] can be used to improve the performance of the approach.

[3] Note that in a more general setting, abstract intepretation also allows to compute under-approximations.

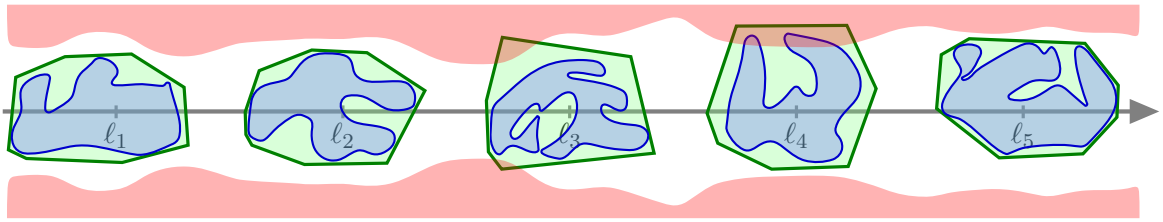


Figure 1.2: Principle of abstract interpretation

Figure 1.2 gives an illustration. For each control point of the program, the set of the collected machine states is represented in blue. The complexity of the blue shapes is intented to underline that the collecting semantics is not computable. In comparison, its over-approximation, represented by simpler shapes depicted in green, is computable. The set of erroneous machine states is represented in red. When it does not intersect the over-approximation, then it is guaranted that there is no error in the program. This happens for instance at control points ℓ_1, ℓ_2, and ℓ_5. Otherwise, the analysis raises an alarm, and two cases can be distinguished:

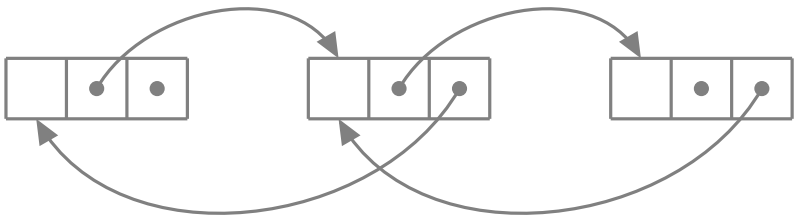
- either the concrete semantics also meets the set of erroneous states, like at control point ℓ_4, so that the alarm is indeed *true*.
- or the alarm is raised because the abstraction is not precise enough (like at control point ℓ_3), in which case it is *false*.

The over-approximation of the semantics, also called *abstract semantics*, is determined thanks to an *abstract domain*. The role of the abstract domain is twofold: (i) it characterizes the nature of the over-approximation which is performed, *i.e.* intuitively, its level of precision, (ii) and it is equipped with a set of functions, the *abstract primitives*, which allow to compute the abstract semantics. Abstract domains are consequently a key ingredient in the methodology of abstract interpretation. The following section discusses the existing abstract domains which have been designed to analyze memory manipulations.

1.2.2 Abstractions for memory manipulations

In the existing literature, we can distinguish two kinds of abstractions for memory manipulations:

- some abstractions consider the memory as a graph defined by the relations induced by pointers. For instance, a doubly-linked list of three elements will be represented as follows:



Such analyses are called *shape analyses* or *points-to analyses*. They are adapted to the analysis of manipulations of symbolic data structures, such as lists, trees, graphs, see among other the work of Steensgaard [Ste96], Reps *et al.* [SRW98], or Berdine *et al.* [BCC+07a]. They allow for instance to check the absence of NULL-pointer dereferencing.

- some others reduce the analysis of memory manipulations to the abstraction of a numerical problem. For instance, in the context of the verification of the absence of buffer overflows, the memory of the machine is seen in a low-level way, as a large sequence of disjoint blocks of consecutive bytes, see Figure 1.3 for an illustration. Each block forms a contiguous memory area, which stores the variables of the program, the data contained in arrays or character strings, *etc.* Note that blocks may be separated by parts of the memory which are not allocated to the program (represented by dots in Figure 1.3). As mentioned in Section 1.1, a *buffer overflow* is an error occurring when the program tries to access to a part of the memory which is not allocated (compare the good and bad memory accesses in Figure 1.3, respectively represented by green and red arrows). Now, an access in a block of memory of size sz is valid if and only if it is performed at a position i within the bounds of this block, which can be expressed as the numerical inequality $0 \leq i < sz$. Thus, checking the absence of buffer overflows can be seen as a numerical problem. This approach has been used in the work of Simon *et al.* [SK02], Dor *et al.* [DRS03], Venet *et al.* [VB04], Jung *et al.* [JKSY05], Miné [Min06], Allamigeon *et al.* [AGH06, AH08], *etc.*

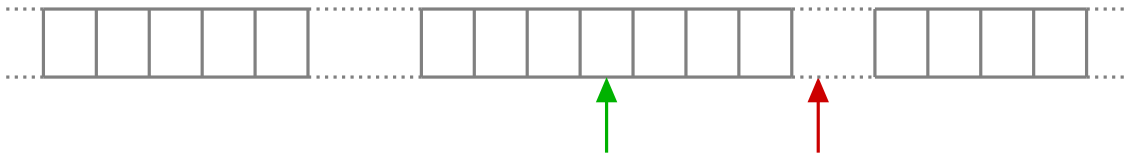


Figure 1.3: Representation of memory as a sequence of blocks

Similarly, the analysis of the manipulations of symbolic data structures, such as lists, trees, can also be reduced to a problem of numerical nature. This approach, introduced by Deutsch [Deu92, Deu94] and studied later by Venet [Ven02, Ven04], is referred to as *non-uniform pointer analysis*. In that case, the accesses in these structures are modelled as paths decorated by numerical information. Intuitively, an access to the i-th element of a list of length n is represented by a path in the list decorated by the integer i, and it is safe if and only if the condition $i \leq n$ holds.

The analyses of the second class are parameterized by a numerical abstract domain. The latter is used to compute sound invariants which hold between the numerical data, such as the size of the memory blocks, the indexes at which the memory is accessed, the integer variables, *etc.* Consequently, the precision of the whole analysis directly depends on the precision of the underlying numerical abstract domain.

Many different numerical abstract domains have been defined in the literature. Each of them is able to infer a particular class of properties over a given set $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d$ of variables. Among the most famous ones, (i) the interval abstract domain [CC77] infers invariants of the form $a \leq \boldsymbol{v}_i \leq b$, $a, b \in \mathbb{R} \cup \{-\infty, +\infty\}$ for every variable $\boldsymbol{v}_i$, (ii) the abstract domain of zones [Min01a] expresses properties of the form $\boldsymbol{v}_i - \boldsymbol{v}_j \leq c$, (iii) the

domain of octagons [Min01b] provides invariants of the form $\pm\boldsymbol{v}_i \pm \boldsymbol{v}_j \leq c$, (iv) the Karr domain [Kar76, MOS04] infers affine equality invariants $\sum_i \alpha_i \boldsymbol{v}_i = \beta$, (v) and the domain of closed convex polyhedra [CH78] provides affine inequality properties $\sum_i \alpha_i \boldsymbol{v}_i \geq \beta$. The vast majority of numerical abstract domains express conjunctions of (sub)affine relationships over the variables. Geometrically, if each variable $\boldsymbol{v}_i$ is associated to the i-th axis of $\mathbb{R}^d$, these properties all represent convex sets.

However, precisely analyzing memory manipulations often requires to express *disjunctive invariants*, *i.e.* disjunctions of constraints over the $\boldsymbol{v}_i$. As an illustration, an important feature in programming languages like C is the ability to manipulate character strings. A string corresponds to a sequence of characters stored in an array. The *null character*, which is encoded by the value 0 in ASCII code, plays the role of the delimiter of the end of strings. The array elements stored after this terminal character are meaningless w.r.t. the string. The length of a string is defined as the position of the first null character in the array.

Strings can be manipulated with usual memory manipulation functions, such as the function `memcpy`, which copies exactly the `n` first characters of the string `src` into the string `dst`. This can be written as the following pseudo-code:

```
1: int i := 0;
2: while i <= n-1 do
3:   dst[i] := src[i];
4:   i := i+1;
5: done;
```

In memory manipulation analysis, precise invariants over the length of the strings are needed to show the absence of buffer overflows.[4] Let us denote by $len_{\texttt{src}}$ and $len_{\texttt{dst}}$ the length of the strings stored in `src` and `dst` respectively. No analysis equipped with a conjunctive numerical abstract domain is able to determine a precise invariant about the resulting length of the string $len_{\texttt{dst}}$. For instance, using the domain of convex polyhedra, which infers affine inequalities, we only get $len_{\texttt{dst}} \geq 0$.[5] Indeed, two cases have to be distinguished:

(i) either `n` is strictly greater than $len_{\texttt{src}}$, in which case the null terminal character of `src` is copied into `dst` at the same index, thus $len_{\texttt{dst}} = len_{\texttt{src}}$.

For instance, if `src` and `dst` are two arrays of size 10, the former containing the string "example" and the latter being initialized by arbitrary values (represented by the symbol `?`), a call to `memcpy` with `n` = 9 yields the following memory state:

The characters which have been copied into `dst` are highlighted in blue. After this operation, the length of `dst` is equal to $len_{\texttt{src}} = 7$, as expected.

(ii) or `n` is smaller than the source length $len_{\texttt{src}}$, so that only non-null characters are copied into `dst`, hence $len_{\texttt{dst}} \geq$ `n`. Using the previous example, but with `n` = 5, we get:

[4] In particular, [SK02, DRS03, AGH06] are devoted to string analysis.

[5] Note the domain of convex polyhedra is known as one of the most precise conjunctive abstract domains.

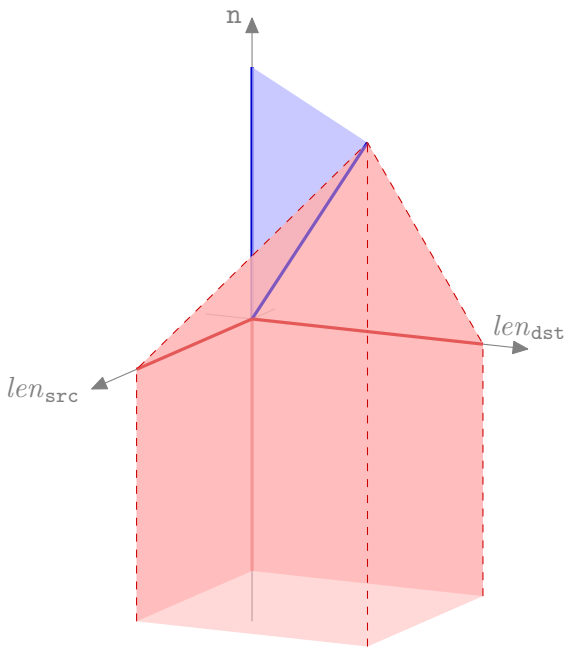
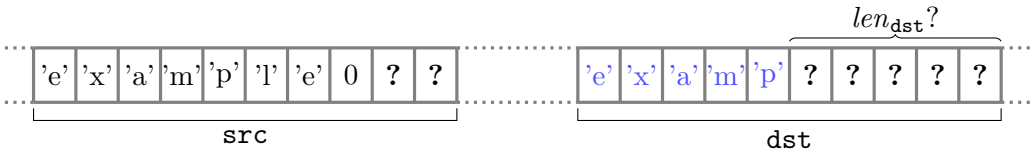


Figure 1.4: Representation of the invariant of the memory manipulating function `memcpy`



Since the five first characters of `dst` are non-null, and the last ones are arbitrary, the strongest property which holds is indeed $len_{\texttt{dst}} \geq 5$.

The representation of the corresponding invariant in $\mathbb{R}^3$ is given in Figure 1.4. The case (i) is represented in blue, and the case (ii) in red. The whole set is clearly not convex, as opposed to invariants inferred by existing numerical abstract domains.[6]

A few techniques already exist to compute disjunctive numerical invariants. For instance, *disjunction completion* [CC79] (see also the related work [GR98, BHZ06]) consists in directly expressing such invariants using a (logical) disjunction of classical abstractions. However, the number of disjunctions may grow arbitrarily when computing the invariants, so that it is not practicable as such. Thus, heuristics have to be used to reduce the number of disjunctions, but this irremediably leads to a loss of precision.[7] Similarly, *trace partitioning* [MR05b, RM07] is a form of disjunctive analysis, in which machine states arising from distinct paths in the execution of the program are abstracted separately. Such a technique does not help in our context, since the nature of the disjunction of cases (i) and (ii) is purely semantic: it depends on the value of the parameter `n`, and not on the execution of particular paths in the program.

[6] Further motivating examples of programs involving such non-convex invariants will be provided in Chapter 7.

[7] On top of that, it is particularly difficult to find heuristics appropriate to a large class of programs and invariants. Thus, in practice, the loss of precision is often uncontrolled.

As a consequence, we cannot precisely analyze such memory manipulations unless we define a new numerical abstract domain which is intrinsically able to express disjunctions. Observe that the disjunctive invariant of the `memcpy` function can be expressed as the following simple identity:

$$\min(len_{\mathtt{src}}, \mathtt{n}) = \min(len_{\mathtt{dst}}, \mathtt{n}),$$

or, passing to the opposite,

$$\max(-len_{\mathtt{src}}, -\mathtt{n}) = \max(-len_{\mathtt{dst}}, -\mathtt{n}). \tag{1.1}$$

Indeed, relationships involving the min and/or max operators allow to express a certain amount of disjunctions.

Invariant (1.1) can also be seen as a linear relationship between $-len_{\mathtt{src}}$, $-len_{\mathtt{dst}}$, and $-\mathtt{n}$, but with the laws of the *max-plus semiring*. The latter is defined as the set $\mathbb{R} \cup \{-\infty\}$ equipped with the operations max and + as addition and multiplication. Following the footsteps of Cousot and Halbwachs who defined thirty years ago the domain of convex polyhedra, this led us to introduce a new domain based on tropical polyhedra.

1.3 An overview of tropical polyhedra

Tropical polyhedra are the analogues of convex polyhedra in tropical algebra. The max-plus semiring (also called max-plus algebra) is one of the possible instantiations of tropical algebra. Tropical polyhedra are defined as the intersection of finitely many tropical halfspaces, which are sets of elements $\boldsymbol{x} = (\boldsymbol{x}_i)$ satisfying an inequality of the form

$$\max(a_0, \max_i(a_i + \boldsymbol{x}_i)) \leq \max(b_0, \max_i(b_i + \boldsymbol{x}_i)).$$

Similarly, the notion of convex sets and cones have analogues.

Tropical or max-plus convex sets have been studied for various motivations, often independently by several people, and under different names. It first appeared in a work of Zimmermann [Zim77], establishing a separation result, motivated by discrete optimization problems. Cuninghame-Green studied max-plus convex cones as max-plus analogues of linear spaces [CG79]. Later, under the name of idempotent spaces, they were considered by Litvinov, Maslov, and Shpiz for an algebraic approach of idempotent functional analysis. Cohen, Gaubert, and Quadrat [CGQ01, CGQ04] also studied them under the name of semimodules, for a geometric approach of discrete event systems [CGQ99], further developed in [Kat07, DLGKL09]. In [NS07], it was considered by Singer in the context of abstract convex analysis [Sin97]. Briec, Horvath, and Rubinov studied max-plus convexity under the name of $\mathbb{B}$-convexity [BH04, BHR05]. Develin and Sturmfels gave in [DS04] another approach of tropical convexity, pointing out important connections with tropical geometry [RGST05]. They developed a different combinatorial point of view, thinking of tropical polyhedra as polyhedral complexes in the usual sense. Following this line, Joswig proposed a study of tropical halfspaces [Jos05], and Develin and Yu gave some conjectures related to the notion of faces of tropical polytopes [DY07].

Several important mathematical results on tropical convex sets have already been established, such as tropical analogues of classical theorems, including those of Minkowski [GK07, BSS07], Helly, Radon, or Carathéodory [GM08], or Hahn-Banach [Zim77, CGQS05, DS04].

In constrast, algorithmic aspects of tropical polyhedra have not yet been thoroughly explored. In particular, a tropical polyhedron can be represented in two different ways, either internally, in terms of extreme points and rays, or externally, in terms of affine inequalities. Passing from the external description of a polyhedron to its internal description, or inversely, is a *fundamental* computational issue, comparable to the well-known vertex/facet enumeration or convex hull problems in the classical case.

Butkovič and Hegedus [BH84] gave an algorithm to compute a generating set of a tropical polyhedral cone described by linear inequalities. Gaubert gave a similar one and derived the equivalence between the internal and external representations [Gau92, Ch. III] (see also [GP97]). Both algorithms rely on a successive elimination of inequalities, but have the inconvenience of squaring at each step the number of candidate generators. Then, an elimination technique must be incorporated to eliminate the redundant candidates. A first implementation of these ideas was included in the maxplus toolbox of Scilab [CGMQ].

In [Jos08], Joswig defined a method which is able to compute pseudo-vertices (which correspond to the vertices of the associated polyhedral complex in the sense of [DS04]) of a tropical polytope generated by a given set of points. It then allows to compute a representation by tropical halfspaces. While such algorithms are of interest from a combinatorial point of view, there are in general too many pseudo-vertices in tropical polyhedra, so that they cannot form an efficient representation in our context.[8]

1.4 Contributions

In this section, we present the contributions of this work, following the chronological order in which they have been developed.

We have first defined a new numerical abstract domain allowing to infer min- and max-invariants. It relies on tropical polyhedra, and is able to express disjunctive program invariants, such as those encountered in static analysis of memory manipulations. We have defined sound abstract primitives to automatically compute over-approximations of semantics of programs by means of tropical polyhedra. In particular, this abstract domain is more precise than the numerical abstract domain of zones, and is able to precisely interact with the abstract domain of octagons. Thanks to this abstract domain, we have developed and implemented a sound static analyzer inferring numerical invariants over the program variables, the length of the strings, and the size of the arrays in memory.

Developing a new abstraction is not only a challenge in semantics (in the sense that the abstractions have to be sound and as precise as possible), but also in algorithmics. Indeed, the scalability of the whole abstract domain directly depends on the complexity of the abstract primitives of the numerical domain. This has led us to develop more efficient algorithms on tropical polyhedra, focusing on the conversion of external representations to internal ones, and inversely, which is a crucial operation in our numerical abstract domain.

First, we have developed an original algorithm allowing to compute a minimal representation by vertices and rays of tropical polyhedra defined by tropical affine inequalities. It is based on a tropical analogue of the double description method introduced by Motzkin *et al.* in [MRTT53]. The cornerstone of this algorithm is a new combinatorial characterization of extreme elements in tropical polyhedra defined by means of inequalities: we have proved that

[8] Any extreme point is a pseudo-vertex, while the converse is not true. In practice, many pseudo-vertices are redundant in the tropical sense.

the extremality of an element amounts to the existence of a strongly connected component reachable from any node in a directed hypergraph. This algorithm has a much better complexity than the other methods discussed in the previous section. In practice, we have shown on several benchmarks that it outperforms the existing implementations, allowing to solve instances which were previously by far inaccessible.

We have also defined a dual algorithm, able to determine a set of inequalities from a description by generating set of a tropical polyhedra. Given a tropical polyhedron $\mathcal{P}$, this algorithm returns the extreme elements of its polar, which is intuitively the set formed by the inequalities satisfied by $\mathcal{P}$. From the extremality criterion of the dual case, we have established that the extreme elements of the polar can be characterized very simply by means of the generating set of $\mathcal{P}$ given as input.

Moreover, in order to have a better understanding of the intrinsic complexity of tropical polyhedra, we have studied the problem of determining the maximal number of extreme points in a tropical polyhedron. In the classical case, this problem is addressed by McMullen upper bound theorem [McM70]. We have found that the maximal number of extreme points in the tropical case is bounded by a similar result. We have introduced a class of tropical polyhedra appearing as natural candidates to be maximizing instances. We have established lower and upper bounds on their number of extreme points, and shown that the McMullen type bound is asymptotically tight when the dimension tends to infinity and the number of inequalities defining the polyhedra is fixed.

Having an efficient algorithm to evaluate the extremality criterion based on directed hypergraphs (see §4) has also appeared to be essential. This has raised the following problem of independent interest: determining in a directed hypergraph all the maximal strongly connected components. Directed hypergraphs are generalizations of directed graphs, in which the tail and the head of edges are sets of nodes. Here, maximality has to be understood as maximality for the partial order induced by the reachability relation. In directed graphs, linear algorithms like Tarjan's [Tar72] are able to compute all strongly connected components, including maximal ones. However, they do not generalize to directed hypergraphs. The only existing method for the latter was suboptimal, and consisted in determining for each node the set of reachable nodes. For this reason, we have introduced a novel algorithm on directed hypergraphs. It directly computes the maximal strongly connected components. We have proved that its complexity is quasi-linear.

Following these algorithmic breakthroughs, we have experimented our tropical polyhedra based static analyzer on programs manipulating strings and arrays. It successfully determines precise properties on memory manipulations, and scales up to highly disjunctive invariants which could not be computed by the existing methods.

Note that the implementation of all the algorithms and abstract domains on tropical polyhedra developed in this work, is available in the Tropical Polyhedra Library TPLib [All09]. This library is written in approximatively 5 000 lines of code in Objective Caml, and is distributed under the GNU Lesser General Public License.

1.5 Organization of the manuscript

The manuscript is divided into two parts.

Part I is devoted to the combinatorial and algorithmic study of tropical polyhedra:

- In Chapter 2, we introduce the basic notions related to convexity in tropical algebra.

- In Chapter 3, we establish the combinatorial characterization of extremality in tropical polyhedra defined by means of inequalities, previously discussed.
- Chapter 4 studies the evaluation of this criterion, with the development of the quasi-linear algorithm on directed hypergraphs.
- Finally, in Chapter 5, we define our two main algorithms converting one kind of representation of tropical polyhedra to the other. It is completed by the study of the maximal number of extreme elements in tropical polyhedra.

Note that Chapter 4 can be read independently of the others.

In part II, we discuss the application to static analysis by abstract interpretation:

- Chapter 6 introduces the theory of abstract interpretation, and the static analysis on memory manipulations.
- In Chapter 7, we define the numerical abstract domain based on tropical polyhedra, and present the experiments with the resulting static analyzer.

Chapter 6 follows the lines of the articles [AH07, AH08]. Most of the content of Chapter 7 has been published in [AGG08]. Many results of Part I can also be found in the preprint [AGG09b], and in the articles [AGG10, AGK10].

Appendix A includes an independent work, albeit related to the analysis of memory manipulations, developed during the first year of this thesis, and published in the article [All08]. It presents an abstract domain able to infer numerical invariants over consecutive array elements. Note that this work has not received further developments since two years.[9] However, in a future work, this contribution could be generalized in order to use the abstract domain of tropical polyhedra, which would allow to increase its precision.

1.6 A few words on notations

A list of symbols is provided in Appendix C. Note that the order relations are denoted by symbols composed with the equality symbol $=$, such as $\leq$, $\preceq$, or $\sqsubseteq$. The only exception to this convention is the set inclusion relation, which is denoted by $\subset$, and not by $\subseteq$.

[9] In particular, the article is included here as such, and the section devoted to relative work has not been updated.

Part I

Combinatorial and algorithmic aspects of tropical polyhedra

CHAPTER 2

Introduction to tropical convexity

In this chapter, we study the analogue of convexity in tropical algebra. In Section 2.1, we discuss general notions relative to tropical algebra. Section 2.2 introduces the main concepts of tropical convexity, such as tropical convex sets, tropical cones, extremality in convex sets, *etc*. Finally, Section 2.3 is focused on the study of the particular class of convex sets formed by tropical polyhedra and polyhedral cones. Section 2.4 is a summary of the results which will be essential in the rest of the manuscript.

2.1 Preliminaries on tropical algebra

Tropical algebra usually refers to the *min-plus semiring*, which is the set $\mathbb{R} \cup \{ +\infty \}$ where the addition of two elements x and y is defined as $\min(x, y)$, and the multiplication as the classical addition $x + y$. However, in this manuscript, we will use the instantiation by the *max-plus semiring*, *i.e.* the set $\mathbb{R}_{\max} \stackrel{def}{=} \mathbb{R} \cup \{ -\infty \}$ endowed with the additive law $\oplus$ defined as $x \oplus y \stackrel{def}{=} \max(x, y)$, and the multiplicative law $\otimes$ defined as $x \otimes y \stackrel{def}{=} x + y$. Observe that this formulation is totally equivalent since the max-plus and min-plus semirings are in one-to-one correspondence by the function $x \mapsto -x$ (with the convention that $-\infty$ is mapped to $+\infty$).

Other instantiations of tropical algebra could also be used (see [Pin98] for a survey on tropical semirings). For instance, we could consider the semiring $\mathbb{R}_+$ of positive reals, equipped with max and $\times$ as additive and multiplicative laws. Besides, the max-plus semiring of integers

$(\mathbb{Z} \cup \{-\infty\}, \oplus, \otimes)$ is of particular interest for applications to computer science. Indeed, it can be implemented with multi-precision integers, and does not raise the critical question of the representation of reals by floating-point numbers (see [Mon08]). Similarly, booleans $\{0, 1\}$ form a semiring, equipped with the disjunction as addition, and the conjunction as multiplication.

Semirings are defined similarly to rings, but their elements do not necessarily have an inverse with respect to the additive law:

Definition 2.1. A set S equipped with the binary operations $\oplus$ and $\otimes$ is a *semiring* if the following requirements are satisfied:

- $(S, \oplus)$ is a commutative monoid equipped with a zero element $\mathbb{0}$,
- $(S, \otimes)$ is a monoid provided with a unit element $\mathbb{1}$,
- the multiplication is distributive over the addition, *i.e.* for all $x, y, z \in S$,

$$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z) \quad \text{and} \quad (y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x),$$

- and $\mathbb{0}$ is an absorbing element for the multiplication, *i.e.* for all $x \in S$, $x \otimes \mathbb{0} = \mathbb{0} \otimes x = \mathbb{0}$.

The semiring S is said to be *idempotent* when it is idempotent for the addition, *i.e.* for every $x \in S$, $x \oplus x = x$. It is said to be *commutative* when the multiplication is commutative.

The max-plus semiring is an example of commutative and idempotent semiring. Its zero and unit elements are respectively defined by $\mathbb{0} \stackrel{def}{=} -\infty$ and $\mathbb{1} \stackrel{def}{=} 0$.

An order $\preceq$ can be associated to every idempotent and commutative semiring, defined by $x \preceq y$ if and only if $y = x \oplus y$. For the max-plus semiring, it coincides with the usual order $\leq$ (extended by $-\infty \leq x$ for all $x \in \mathbb{R}_{\max}$).

The max-plus semiring is usually called *max-plus algebra* because every non-zero element has an inverse w.r.t. the multiplication. The inverse of $x \neq \mathbb{0}$ is denoted by x^{-1}, and is defined as the classical opposite $-x$ of x.

Note that for the sake of simplicity, the multiplication $x \otimes y$ of two elements x and y will be often denoted by xy. To avoid confusion, the use of classical addition and multiplication instead of the tropical laws will be explicitly mentioned.

2.2 Preliminaries on tropical convexity

In this section, we introduce the basic notions relative to the analogues of convex sets in tropical algebra. In Sections 2.2.2 and 2.2.3, we define convex sets and convex cones in the tropical sense. In Section 2.2.4, we study the notion of extremality in convex sets, and see that convex sets are generated by their extreme elements. Section 2.2.5 provides a characterization of such generating representations which are minimal. Finally, Section 2.2.6 is devoted to the tropical homogenization technique, which allows to represent closed convex sets by tropical cones, up to adding a dimension to the space.

2.2.1 Notations

The set $\mathbb{R}_{\max}^d$ denotes the d-th fold of $\mathbb{R}_{\max}$. Its elements will be denoted by bold symbols, for instance $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d)$. The ith entry of the element $\boldsymbol{x}$ will be denoted by $\boldsymbol{x}_i$. The elements $\mathbb{0}$ and $\mathbb{1}$ will refer to the vectors whose coordinates are all equal to $\mathbb{0}$ and $\mathbb{1}$ respectively.

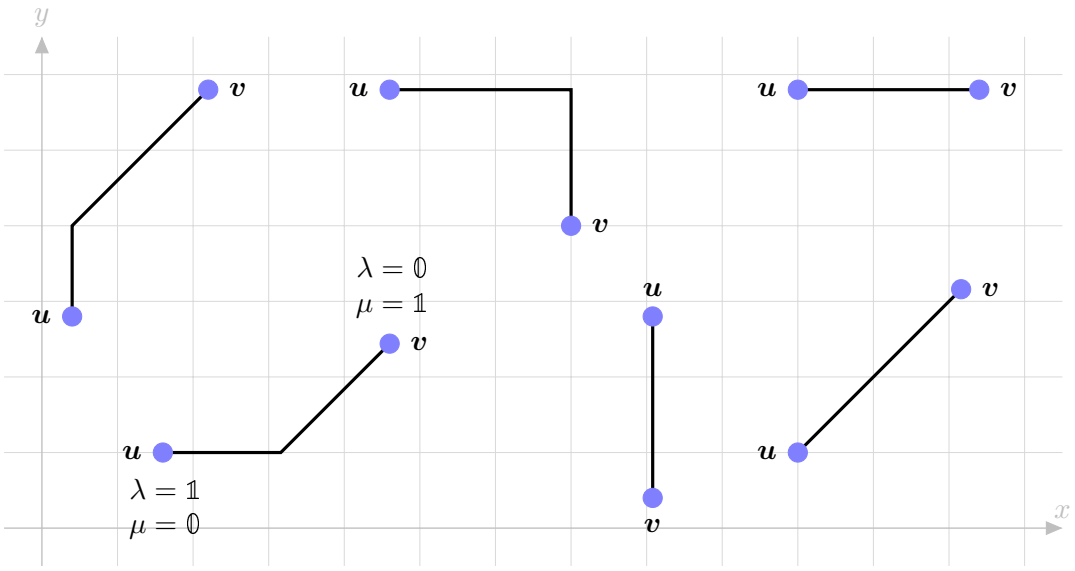


Figure 2.1: The six kinds of tropical segments in $\mathbb{R}^2_{\max}$

The elements of $\mathbb{R}^d_{\max}$ can be thought as points of an affine space, or as vectors. The addition and multiplication can be naturally extended to $\mathbb{R}^d_{\max}$. Given two elements $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d_{\max}$, $\boldsymbol{x} \oplus \boldsymbol{y}$ is the element with entries $\boldsymbol{x}_i \oplus \boldsymbol{y}_i$. Similarly, the multiplication of a vector $\boldsymbol{x} \in \mathbb{R}^d_{\max}$ by a scalar $\lambda \in \mathbb{R}_{\max}$ is denoted by $\lambda\boldsymbol{x}$, and is the element with entries $\lambda\boldsymbol{x}_i$.

Finally, the tropical addition is extended to the *Minkowski sum* of two sets $S, S' \subset \mathbb{R}^d_{\max}$, denoted by $S \oplus S'$, and defined as $\{\, \boldsymbol{x} \oplus \boldsymbol{x}' \mid (\boldsymbol{x}, \boldsymbol{x}') \in S \times S' \,\}$.

2.2.2 Tropical convex sets

Definition 2.2. A set $\mathcal{C} \subset \mathbb{R}^d_{\max}$ is said to be *a tropical convex set* if for all $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{C}$ and $\lambda, \mu \in \mathbb{R}_{\max}$ such that $\lambda \oplus \mu = \mathbb{1}$,

$$\lambda\boldsymbol{u} \oplus \mu\boldsymbol{v} \in \mathcal{C}.$$

Tropically convex sets are defined as the tropical analogues of convex sets. However, observe that the condition that λ, μ are nonnegative is omitted. Indeed, in the tropical setting, all the scalars are "nonnegative", because $\mathbb{0} = -\infty \leq \lambda$ holds for all $\lambda \in \mathbb{R}_{\max}$.

The set of the combinations $\lambda\boldsymbol{u} \oplus \mu\boldsymbol{v}$ for $\lambda \oplus \mu = \mathbb{1}$ represents the *tropical segment* joining the two elements $\boldsymbol{u}$ and $\boldsymbol{v}$. The ends $\boldsymbol{u}$ and $\boldsymbol{v}$ of the segment are obtained respectively with the parameters $\lambda = \mathbb{1}$ and $\mu = \mathbb{0}$, and $\lambda = \mathbb{0}$ and $\mu = \mathbb{1}$.

Example 2.1. Figure 2.1 depicts the six kinds of tropical segments in dimension $d = 2$. These are concatenations of at most two ordinary segments of slope 0, 1 or ∞.

Note that in the general case $d \geq 2$, tropical segments can also be expressed as concatenations of at most d ordinary segments, see [DS04, Proposition 3].

An example of tropical convex set in $\mathbb{R}^2_{\max}$ is depicted in Figure 2.2 (the border is included). It can be easily verified that it contains all tropical segments of any of its two points. From these examples, it can be observed that tropical convex sets are not convex in the classical sense.

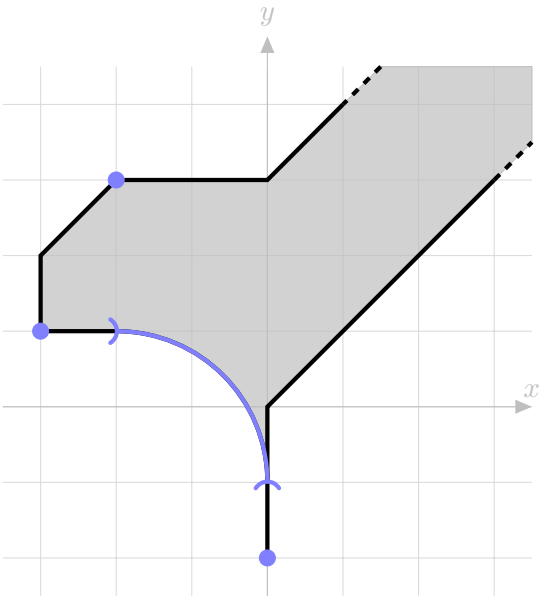


Figure 2.2: A tropical convex set in $\mathbb{R}^2_{\max}$

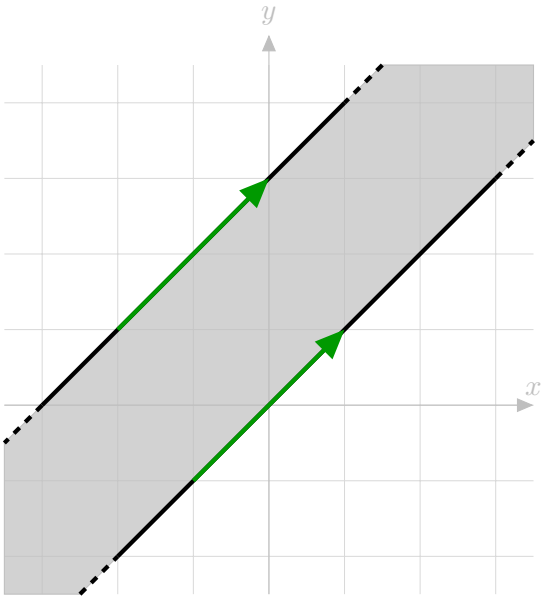


Figure 2.3: A tropical cone in $\mathbb{R}^2_{\max}$

The notion of tropical convex hull is defined similarly:

Definition 2.3. Given a subset $S \subset \mathbb{R}^d_{\max}$, the *tropical convex hull* $\mathsf{co}(S)$ of S is defined as the set of the *tropically convex combinations* $\alpha_1 \boldsymbol{x}_1 \oplus \ldots \oplus \alpha_p \boldsymbol{x}_p$ where $p \geq 1$, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p \in S$, $\alpha_1, \ldots, \alpha_p \in \mathbb{R}_{\max}$, and $\alpha_1 \oplus \ldots \oplus \alpha_p = \mathbb{1}$.

2.2.3 Tropical cones

Definition 2.4. A non-empty subset $\mathcal{C} \subset \mathbb{R}^d_{\max}$ is said to be a *tropical convex cone* if $\lambda \boldsymbol{u} \oplus \mu \boldsymbol{v} \in \mathcal{C}$ for all $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{C}$ and $\lambda, \mu \in \mathbb{R}_{\max}$.

For a subset $S \subset \mathbb{R}^d_{\max}$, the *tropical cone generated by* S, denoted by $\mathsf{cone}(S)$, is the set of the *tropical linear combinations* $\alpha_1 \boldsymbol{x}_1 \oplus \ldots \oplus \alpha_p \boldsymbol{x}_p$ where $p \geq 1$, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p \in S$ and $\alpha_1, \ldots, \alpha_p \in \mathbb{R}_{\max}$.

In the sequel, tropical convex cones will be simply referred to as tropical cones. An example of tropical cone in $\mathbb{R}^2_{\max}$ is given in Figure 2.3. Observe that the two parallel lines bounding the cone intersect at the (tropical) null point $(\mathbb{0}, \mathbb{0})$. Another example of tropical cones are *rays*, defined as follows:

Definition 2.5. A *ray* of $\mathbb{R}^d_{\max}$ is a set of the form $\{ \lambda \boldsymbol{x} \mid \lambda \in \mathbb{R}_{\max} \}$, for a given non-null element $\boldsymbol{x} \in \mathbb{R}^d_{\max}$, and is then denoted by $\langle \boldsymbol{x} \rangle$.

A non-null element $\boldsymbol{y} \in \mathbb{R}^d_{\max}$ is said to be a *representative* of the ray $\langle \boldsymbol{x} \rangle$ if it satisfies $\langle \boldsymbol{y} \rangle = \langle \boldsymbol{x} \rangle$.

We will assume that $\mathbb{R}^d_{\max}$ is equipped with the usual topology, defined by the metric $(\boldsymbol{x}, \boldsymbol{y}) \mapsto \max_{1 \leq i \leq d} |e^{\boldsymbol{x}_i} - e^{\boldsymbol{y}_i}|$. The closure operation is denoted by $cl(\cdot)$. Note that if $\mathcal{C}$ is a tropical convex set (resp. a tropical cone), $cl(\mathcal{C})$ is also a tropical convex set (resp. a tropical cone) by continuity of the tropical addition and the multiplication by a scalar.

Definition 2.6. Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a closed convex set. The *recession cone* $\mathsf{rec}(\mathcal{C})$ is defined as the set

$$\{\, \boldsymbol{v} \mid \boldsymbol{x} \oplus \lambda \boldsymbol{v} \in \mathcal{C} \text{ for all } \lambda \in \mathbb{R}_{\max} \,\},$$

where $\boldsymbol{x}$ is an arbitrary element of $\mathcal{C}$.

We refer to the work of Gaubert and Katz [GK07] in which it is shown that the recession cone does not depend on the choice of the element $\boldsymbol{x}$ when $\mathcal{C}$ is closed. Naturally, as soon as $\mathcal{C}$ is a cone, it coincides with its recession cone.

Example 2.2. The recession cone of the convex set given in Figure 2.2 is depicted in Figure 2.3.

Remark 2.3. In the tropical setting, all lines are reduced to rays. Recall that in the classical setting, rays and lines are respectively sets of the form $\{\, \lambda \boldsymbol{u} \mid \lambda \in \mathbb{R}_+ \,\}$ and $\{\, \lambda \boldsymbol{u} \mid \lambda \in \mathbb{R} \,\}$ (for a given element $\boldsymbol{u} \in \mathbb{R}$). It is immediate that the tropical analogues of the two previous definitions are identical, since any scalar of $\mathbb{R}_{\max}$ is positive.

It follows that the lineality space of a convex cone $\mathcal{C}$, defined in the classical setting as $\{\, \boldsymbol{u} \in \mathbb{R}^d \mid \boldsymbol{x} + \lambda \boldsymbol{u} \in \mathcal{C} \text{ for all } \lambda \in \mathbb{R} \,\}$ (where $\boldsymbol{x}$ is an arbitrary element of $\mathcal{C}$), does not have any analogue in the tropical setting. In other words, all tropical convex cones are pointed, intuitively because $\mathbb{R}^d_{\max}$ is not symmetric with respect to its origin $\mathbb{0}$.

2.2.4 Extreme elements

Like classical convex sets, tropical convex sets admit extreme elements. They are defined as follows:

Definition 2.7. Given a convex set $\mathcal{C} \subset \mathbb{R}^d_{\max}$, an element $\boldsymbol{x} \in \mathcal{C}$ is said to be an *extreme point* of $\mathcal{C}$ if for all $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{C}$ and $\lambda, \mu \in \mathbb{R}_{\max}$ such that $\lambda \oplus \mu = \mathbb{1}$,

$$\boldsymbol{x} = \lambda \boldsymbol{u} \oplus \mu \boldsymbol{v} \implies \boldsymbol{x} = \boldsymbol{u} \text{ or } \boldsymbol{x} = \boldsymbol{v}. \tag{2.1}$$

Definition 2.8. Given a cone $\mathcal{C} \subset \mathbb{R}^d_{\max}$, a non-null element $\boldsymbol{x} \in \mathcal{C}$ is said to be an *extreme generator* of $\mathcal{C}$ if for all $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{C}$,

$$\boldsymbol{x} = \boldsymbol{u} \oplus \boldsymbol{v} \implies \boldsymbol{x} = \boldsymbol{u} \text{ or } \boldsymbol{x} = \boldsymbol{v}.$$

A ray $\langle \boldsymbol{x} \rangle$ is said to be *extreme* if $\boldsymbol{x}$ is an extreme generator of $\mathcal{C}$.

Note that each representative of an extreme ray is an extreme generator, so that the definition of an extreme ray does not depend on the choice of the representative.

Remark 2.4. In tropical cones, extreme points are not of interest. Indeed, it can be shown that their unique extreme point is the element $\mathbb{0}$. Consequently, for a tropical cone, the term *extreme element* will always refer to an extreme generator, and not to an extreme point.

Example 2.5. The extreme points of the convex set $\mathcal{C}$ given in Figure 2.2 are depicted in blue, and consists of the open arc $\left\{ \left(x, -1+\sqrt{4-(x+2)^2}\right) \mid -2 < x < 0 \right\}$, and the points $(-2, 3)$, $(-3, 1)$, and $(0, -2)$.

The extreme rays of its recession cone, which are represented by green arrows in Figure 2.3, are the rays $\langle (0, 0) \rangle$ and $\langle (0, 3) \rangle$.

The sets of extreme points (respectively generators) of a convex set (resp. cone) $\mathcal{C}$ will be denoted by $\mathsf{extp}(\mathcal{C})$ (resp. $\mathsf{extg}(\mathcal{C})$).

In [GK07], a "Minkowski-Carathéodory"-type theorem was established for closed convex sets in the tropical setting. It states that a closed convex set of $\mathbb{R}^d_{\max}$ is entirely generated by the combinations of at most $(d+1)$ extreme points and extreme generators of the recession cone:

Theorem 2.1 ([GK07, Theorem 3.3])**.** *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a closed convex set. Then any element $\boldsymbol{x}$ of $\mathcal{C}$ can be expressed as the sum of the convex combination of n extreme points $\boldsymbol{p}^1, \dots, \boldsymbol{p}^n$ of $\mathcal{C}$, and p extreme generators $\boldsymbol{g}^1, \dots, \boldsymbol{g}^p$ of $\mathsf{rec}(\mathcal{C})$, with $n + p \leq d + 1$:*

$$\boldsymbol{x} = \bigoplus_{i=1}^{n} \lambda_i \boldsymbol{p}^i \oplus \bigoplus_{j=1}^{p} \boldsymbol{g}^j, \quad \text{where } \bigoplus_{i=1}^{n} \lambda_i = \mathbb{1}.$$

In particular,

$$\mathcal{C} = \mathsf{co}\left(\mathsf{extp}(\mathcal{C})\right) \oplus \mathsf{cone}\left(\mathsf{extg}\left(\mathsf{rec}(\mathcal{C})\right)\right).$$

A similar statement holds for closed cones:

Theorem 2.2 ([GK07, Theorem 3.1])**.** *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a closed cone. Then any element of $\mathcal{C}$ can be expressed as the sum of d extreme generators of $\mathcal{C}$. Consequently,*

$$\mathcal{C} = \mathsf{cone}\left(\mathsf{extg}(\mathcal{C})\right).$$

In Chapter 3, we will provide a direct proof of both results.

2.2.5 Minimal generating representations

As previously established, extreme elements form generating representations of closed convex sets and cones. This section deals with such representations which are minimal.

Definition 2.9. Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a closed cone. A *generating set* of $\mathcal{C}$ is a set $G \subset \mathbb{R}^d_{\max}$ such that $\mathcal{C} = \mathsf{cone}(G)$.

A generating set of $\mathcal{C}$ is said to be *minimal* if it is a minimal element for the inclusion among the generating sets of $\mathcal{C}$.

We introduce the norm $\|\cdot\|$ over $\mathbb{R}^d_{\max}$ defined by $\|\boldsymbol{x}\| \overset{def}{=} \max_{1 \leq i \leq d} e^{\boldsymbol{x}_i}$. An element $\boldsymbol{x} \in \mathbb{R}^d_{\max}$ is said to be *scaled* if it satisfies $\|\boldsymbol{x}\| = 1$. By extension, a set $G \subset \mathbb{R}^d_{\max}$ is said to be *scaled* if each of its elements is scaled. Let σ be the function from $\mathbb{R}^d_{\max} \setminus \{\, \mathbb{0} \,\}$ to itself, which maps each $\boldsymbol{x}$ to the scaled element $\|\boldsymbol{x}\|^{-1}\boldsymbol{x}$.

The following theorem gives a characterization of the minimal generating sets of tropical cones, stating that they are unique up to scaling their elements.

Theorem 2.3. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a closed cone. The minimal generating sets G of $\mathcal{C}$ are precisely the sets consisting of exactly one representative of each extreme ray of $\mathcal{C}$.*

In particular, $\sigma(\mathsf{extg}(\mathcal{C}))$ is the unique scaled minimal generating set of $\mathcal{C}$.

Proof. Let us first prove the second part of the statement. Let $G = \sigma(\mathsf{extg}(\mathcal{C}))$ be the set formed by the scaled extreme generators of $\mathcal{C}$, and consider H a scaled generating set. For each $\boldsymbol{g} \in G$, there exist $\lambda_1, \dots, \lambda_p \in \mathbb{R}_{\max}$ and $\boldsymbol{h}_1, \dots, \boldsymbol{h}_p \in H$ such that $\boldsymbol{g} = \bigoplus_{i=1}^{p} \lambda_i \boldsymbol{h}_i$.

Using the extremality of $\boldsymbol{g}$, there exists i such that $\boldsymbol{g} = \lambda_i \boldsymbol{h}_i$, and since $\boldsymbol{g}$ and $\boldsymbol{h}_i$ are both scaled, they are necessarily equal. This shows that $G \subset H$, and it follows that G is indeed the unique minimal generating set of $\mathcal{C}$.

For the first part of the statement, first consider a minimal generating set H' of $\mathcal{C}$. Then $\sigma(H)$ is a scaled minimal generating set of $\mathcal{C}$, so that $\sigma(H) = G$. This proves that H contains exactly one representative of each extreme ray.

Conversely, let us show that if G' is formed by exactly one representative of each extreme ray, it is minimal. Let $H' \subset G'$ such that $\mathcal{C} = \mathsf{cone}(H')$. Then $\sigma(H') \subset \sigma(G') = G$, hence $\sigma(H') = G$. It follows that $H' = G'$. □

Similarly, minimal generating representations of closed convex sets can be defined:

Definition 2.10. Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a closed convex set. A *generating representation* of $\mathcal{C}$ is a pair (P, R), where $P, R \subset \mathbb{R}^d_{\max}$ verifying:

$$\mathcal{C} = \mathsf{co}(P) \oplus \mathsf{cone}(R).$$

It is said to be *minimal* if for all generating representations (Q, S) of $\mathcal{C}$:

$$Q \subset P \text{ and } S \subset R \implies (Q, S) = (P, R).$$

A generating representation (P, R) is said to be *scaled* if R is a scaled set. Like cones, convex sets admits a unique scaled minimal generating representation:

Theorem 2.4. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a closed convex set. The minimal generating representations of $\mathcal{C}$ are precisely the couple $(\mathsf{extp}(\mathcal{C}), R)$ where R consists of exactly one representative of each extreme ray of $\mathsf{rec}(\mathcal{C})$.*

In particular, $(\mathsf{extp}(\mathcal{C}), \sigma(\mathsf{extg}(\mathsf{rec}(\mathcal{C}))))$ is the unique scaled minimal generating representation of $\mathcal{C}$.

Proof. Like in the proof Theorem 2.3, let us begin with the second part of the statement. Let $(P, R) = (\mathsf{extp}(\mathcal{C}), \sigma(\mathsf{extg}(\mathsf{rec}(\mathcal{C}))))$. Using Theorem 2.3, any scaled generating representation (Q, S) of $\mathcal{C}$ satisfies $R \subset S$. Besides, let $\boldsymbol{p} \in P$. As $\boldsymbol{p}$ belongs to $\mathcal{C}$, it can be written under the form:

$$\begin{aligned} \boldsymbol{p} &= \bigoplus_{i=1}^{n} \alpha_i \boldsymbol{q}_i \oplus \bigoplus_{j=1}^{m} \beta_j \boldsymbol{s}_j \qquad (\boldsymbol{q}_i \in Q,\ \boldsymbol{s}_j \in S,\ \alpha_i, \beta_j \in \mathbb{R}_{\max}, \text{ and } \bigoplus_{i=1}^{n} \alpha_i = \mathbb{1}) \\ &= \bigoplus_{\substack{1 \le i \le n \\ 1 \le j \le m}} \alpha_i (\boldsymbol{q}_i \oplus \beta_j \boldsymbol{s}_j) \end{aligned}$$

so that $\boldsymbol{p} = \boldsymbol{q}_i \oplus \beta_j \boldsymbol{s}_j$ for a given (i, j), as $\boldsymbol{p}$ is an extreme point. Suppose that there exists $\boldsymbol{p} \neq \boldsymbol{q}_i$. Then writting $\boldsymbol{p} = \boldsymbol{q}_i \oplus (-1)(\boldsymbol{q}_i \oplus (\beta_j + 1)\boldsymbol{s}_j)$ contradicts the extremality of $\boldsymbol{q}_i$ as $\boldsymbol{q}_i \oplus (\beta_j + 1)\boldsymbol{s}_j \in \mathcal{P}$ and $0 \oplus (-1) = \mathbb{1}$. Therefore $\boldsymbol{p} \in Q$, which proves $P \subset Q$.

After that, the first part of the theorem is straightforward. □

2.2.6 Tropical homogenization

In this section, we present a technique to represent closed tropical convex sets by tropical cones, which consists in adding a dimension to the latter to represent the affine component of the former. In the classical setting, this method is known as *homogenization* (see for instance [Zie98]). In the tropical setting, it has been first introduced in [GK07].

The tropical homogenization relies on the following property:

Proposition 2.1. *Let $\mathcal{C} \subset \mathbb{R}_{\max}^d$ be a closed tropical convex set. Then the set $\widehat{\mathcal{C}} \subset \mathbb{R}_{\max}^{d+1}$ defined by*

$$\widehat{\mathcal{C}} \stackrel{def}{=} cl\left(\{\, (\alpha \boldsymbol{x}, \alpha) \mid \boldsymbol{x} \in \mathcal{C},\, \alpha \in \mathbb{R}_{\max} \,\}\right) \tag{2.2}$$

is a closed tropical cone. Besides, the following relations hold:

$$\mathcal{C} = \left\{\, \boldsymbol{x} \mid (\boldsymbol{x}, \mathbb{1}) \in \widehat{\mathcal{C}} \,\right\}, \tag{2.3}$$

$$\mathsf{rec}(\mathcal{C}) = \left\{\, \boldsymbol{x} \mid (\boldsymbol{x}, \mathbb{0}) \in \widehat{\mathcal{C}} \,\right\}. \tag{2.4}$$

Proof. First, let us show that $\mathcal{D} \stackrel{def}{=} \{\, (\alpha \boldsymbol{x}, \alpha) \mid \boldsymbol{x} \in \mathcal{C},\, \alpha \in \mathbb{R}_{\max} \,\}$ is a tropical cone. Consider $\boldsymbol{u} = (\alpha \boldsymbol{x}, \alpha), \boldsymbol{v} = (\beta \boldsymbol{y}, \beta) \in \mathcal{D}$ and $\lambda, \mu \in \mathbb{R}_{\max}$. Clearly, when α, β, λ or μ is equal to $\mathbb{0}$, it is obvious that $\lambda \boldsymbol{u} \oplus \mu \boldsymbol{v}$ belongs to $\mathcal{D}$. Otherwise, $\kappa \stackrel{def}{=} (\alpha\lambda) \oplus (\beta\mu)$ is strictly greater than $\mathbb{0}$. If $\kappa^{-1} \stackrel{def}{=} (-\kappa)$ is its tropical inverse w.r.t the max-plus multiplicative law, then $(\alpha\lambda\kappa^{-1}) \oplus (\beta\mu\kappa^{-1}) = \mathbb{1}$, so that $(\alpha\lambda\kappa^{-1})\boldsymbol{x} \oplus (\beta\mu\kappa^{-1})\boldsymbol{y}$ is an element of $\mathcal{C}$. It follows that $\lambda \boldsymbol{u} \oplus \mu \boldsymbol{v} = (\kappa((\alpha\lambda\kappa^{-1})\boldsymbol{x} \oplus (\beta\mu\kappa^{-1})\boldsymbol{y}), \kappa)$ belongs to $\mathcal{D}$.

Hence, $\widehat{\mathcal{C}}$ is also a tropical cone as the closure of $\mathcal{D}$.

The relation $\mathcal{C} \subset \{\, \boldsymbol{x} \mid (\boldsymbol{x}, \mathbb{1}) \in \widehat{\mathcal{C}} \,\}$ is trivial. Conversely, consider $(\boldsymbol{x}, \mathbb{1}) \in \widehat{\mathcal{C}}$, and let $(\alpha_n \boldsymbol{x}^n, \alpha_n)$ be a sequence converging to $(\boldsymbol{x}, \mathbb{1})$, with $\boldsymbol{x}^n \in \mathcal{C}$ and $\alpha_n \in \mathbb{R}_{\max}$ for each n. Then $\lim_{n \to +\infty} \alpha_n = \mathbb{1}$, and so that $\boldsymbol{x} = \lim_{n \to +\infty} \boldsymbol{x}^n$. As $\mathcal{C}$ is closed, the element $\boldsymbol{x}$ belongs to $\mathcal{C}$. This shows (2.3).

Now, consider $\boldsymbol{v} \in \mathsf{rec}(\mathcal{C})$. If $\boldsymbol{x} \in \mathcal{C}$, then $\boldsymbol{x} \oplus (n\boldsymbol{v}) \in \mathcal{C}$ for all $n \in \mathbb{N}$. The sequence $(n^{-1}(x \oplus (n\boldsymbol{v})), n^{-1})$ of elements of $\widehat{\mathcal{C}}$ obviously converges to the element $(\boldsymbol{v}, \mathbb{0})$ when $n \to +\infty$, which shows that the latter belongs to $\widehat{\mathcal{C}}$. Conversely, consider $(\boldsymbol{v}, \mathbb{0}) \in \widehat{\mathcal{C}}$, and $\boldsymbol{x} \in \mathcal{C}$ and $\lambda \in \mathbb{R}_{\max}$. We want to show that $\boldsymbol{x} \oplus \lambda \boldsymbol{v}$ belongs to $\mathcal{C}$. If $\lambda = \mathbb{0}$, this is obvious. Otherwise, consider a sequence $(\alpha_n \boldsymbol{x}^n, \alpha_n)$ such that $\boldsymbol{x}^n \in \mathcal{C}$, $\alpha_n \in \mathbb{R}_{\max}$, and which converges to $(\boldsymbol{u}, \mathbb{0})$. Naturally, $\lim_{n \to +\infty} \alpha_n = \mathbb{0}$, so that without loss of generality, it can be supposed that $\alpha_n \leq \lambda^{-1}$ for each n. It follows that $\mathbb{1} \oplus \alpha_n \lambda = \mathbb{1}$, hence $\boldsymbol{x} \oplus \lambda(\alpha_n \boldsymbol{x}^n)$ belongs to $\mathcal{C}$ for each n. As $\mathcal{C}$ is closed, taking the limit when $n \to +\infty$ yields $\boldsymbol{x} \oplus \lambda \boldsymbol{u} \in \mathcal{C}$. This proves (2.4). □

Now, a one-to-one correspondence between extreme elements of closed convex sets and extreme elements of their homogenized cones can be established:

Proposition 2.2. *Let $\mathcal{C} \subset \mathbb{R}_{\max}^d$ be a closed tropical convex set. Then the following relations hold:*

$$\mathsf{extp}(\mathcal{C}) = \left\{\, \boldsymbol{x} \mid (\boldsymbol{x}, \mathbb{1}) \in \mathsf{extg}(\widehat{\mathcal{C}}) \,\right\}, \tag{2.5}$$

$$\mathsf{extg}(\mathsf{rec}(\mathcal{C})) = \left\{\, \boldsymbol{x} \mid (\boldsymbol{x}, \mathbb{0}) \in \mathsf{extg}(\widehat{\mathcal{C}}) \,\right\}. \tag{2.6}$$

Proof. The proof of (2.6) is almost immediate. Let us show (2.5).

Suppose that $\boldsymbol{x}$ is an extreme point of $\mathcal{C}$. By (2.3), $(\boldsymbol{x}, \mathbb{1})$ belongs to $\widehat{\mathcal{C}}$. Let us consider $\boldsymbol{u} = (\boldsymbol{y}, \lambda), \boldsymbol{v} = (\boldsymbol{z}, \mu) \in \widehat{\mathcal{C}}$ such that $(\boldsymbol{x}, \mathbb{1}) = \boldsymbol{u} \oplus \boldsymbol{v}$. As $\boldsymbol{x} = \boldsymbol{y} \oplus \boldsymbol{z}$ and $\boldsymbol{x}$ is extreme, suppose for instance that $\boldsymbol{x} = \boldsymbol{y}$. If $\lambda = \mathbb{1}$, then $(\boldsymbol{x}, \mathbb{1}) = \boldsymbol{u}$. Otherwise, $\lambda < \mathbb{1}$ and $\mu = \mathbb{1}$ (since $\lambda \oplus \mu = \mathbb{1}$). Two cases can be distinguished:

(i) either $\lambda > \mathbb{0}$, in which case $(\boldsymbol{y}, \lambda) \in \widehat{\mathcal{C}}$ implies $(\lambda^{-1}\boldsymbol{y}, \mathbb{1}) \in \widehat{\mathcal{C}}$ (as $\widehat{\mathcal{C}}$ is a cone), and using (2.3), $\lambda^{-1}\boldsymbol{x} = \lambda^{-1}\boldsymbol{y} \in \mathcal{C}$,

(ii) or $\lambda = \mathbb{0}$, so that $\boldsymbol{x} = \boldsymbol{y} \in \mathsf{rec}(\mathcal{C})$ by (2.4). It follows that $\boldsymbol{x} \oplus \kappa\boldsymbol{x} \in \mathcal{C}$ for all κ. As soon as $\kappa > \mathbb{1}$, $\kappa\boldsymbol{x}_i \geq \boldsymbol{x}_i$ for all i, so that $\kappa\boldsymbol{x} \in \mathcal{C}$.

In both cases, we have found $\alpha > \mathbb{1}$ such that $\alpha\boldsymbol{x} \in \mathcal{C}$. Besides $\boldsymbol{x} = \alpha^{-1}(\alpha\boldsymbol{x}) \oplus \boldsymbol{z}$ since for all i, $\boldsymbol{x}_i \geq \boldsymbol{z}_i$. Using the fact that $\boldsymbol{x}$ is extreme and $\alpha^{-1} \oplus \mathbb{1} = \mathbb{1}$, it implies $\boldsymbol{x} = \alpha\boldsymbol{x}$ or $\boldsymbol{x} = \boldsymbol{z}$. In the former case, $\boldsymbol{x}$ has to be equal to $\mathbb{0}$, which leads to $\boldsymbol{x} = \boldsymbol{z}$, just as in the latter case. In both situations, we have $(\boldsymbol{x}, \mathbb{1}) = \boldsymbol{v}$, which shows the extremality of $(\boldsymbol{x}, \mathbb{1})$.

Conversely, suppose that $(\boldsymbol{x}, \mathbb{1})$ is extreme in $\widehat{\mathcal{C}}$. Then $\boldsymbol{x} \in \mathcal{C}$ using (2.3). Consider $\boldsymbol{y}, \boldsymbol{z} \in \mathcal{C}$ and $\lambda, \mu \in \mathbb{R}_{\max}$ such that $\boldsymbol{x} = \lambda\boldsymbol{y} \oplus \mu\boldsymbol{z}$ and $\lambda \oplus \mu = \mathbb{1}$. Then $(\lambda\boldsymbol{y}, \lambda), (\mu\boldsymbol{y}, \mu)$ are both elements of $\widehat{\mathcal{C}}$, and their sum is equal to $(\boldsymbol{x}, \mathbb{1})$. Hence, for instance, $(\boldsymbol{x}, \mathbb{1}) = (\lambda\boldsymbol{y}, \lambda)$, which shows that $\boldsymbol{x} = \boldsymbol{y}$. Hence, $\boldsymbol{x}$ is an extreme point of $\mathcal{C}$. □

Using Theorems 2.3 and 2.4, Proposition 2.2 can be refined considering minimal generating representations of convex sets and their homogenized cones:

Corollary 2.3. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a closed convex set. Then (P, R) is a minimal generating representation of $\mathcal{C}$ if and only if $(P \times \{\mathbb{1}\}) \cup (R \times \{\mathbb{0}\})$ is a minimal generating set of $\widehat{\mathcal{C}}$.*

Proof. If (P, R) is a minimal generating representation of $\mathcal{C}$, then by Theorem 2.4, $P = \mathsf{extp}(\mathcal{C})$, and R contains exactly one representative of each extreme ray of $\mathsf{rec}(\mathcal{C})$. Hence, using Proposition 2.2, the elements of the form $(\boldsymbol{p}, \mathbb{1})$ and $(\boldsymbol{r}, \mathbb{0})$, where $\boldsymbol{p} \in P$ and $\boldsymbol{r} \in R$, are all extreme generators of $\widehat{\mathcal{C}}$. Besides, they are clearly representatives of pairwise distinct extreme rays of $\mathsf{extg}(\widehat{\mathcal{C}})$. Reciprocally, considering an extreme ray $\langle(\boldsymbol{g}, \alpha)\rangle$ of $\mathsf{extg}(\widehat{\mathcal{C}})$,

(i) either $\alpha = \mathbb{0}$, in which case $\boldsymbol{g}$ is extreme in $\mathsf{rec}(\mathcal{C})$, so that $\langle\boldsymbol{g}\rangle$ is represented by one of the elements of $\boldsymbol{r} \in R$. Hence $\langle(\boldsymbol{g}, \alpha)\rangle$ is represented by the extreme element $(\boldsymbol{r}, \mathbb{0}) \in R \times \{\mathbb{0}\}$,

(ii) or $\alpha > \mathbb{0}$, so that $\alpha^{-1}\boldsymbol{g}$ belongs to P. It follows that $\langle(\boldsymbol{g}, \alpha)\rangle$ is represented by the extreme generator $(\alpha^{-1}\boldsymbol{g}, \mathbb{1}) \in P \times \{\mathbb{1}\}$.

According to Theorem 2.3, this proves that $(P \times \{\mathbb{1}\}) \cup (R \times \{\mathbb{0}\})$ is a minimal generating set of $\widehat{\mathcal{C}}$.

Conversely, using Proposition 2.2, each element of P (resp. R) is an extreme point of $\mathcal{C}$ (resp. generator of $\mathsf{rec}(\mathcal{C})$). Besides, the elements of R represent pairwise distinct extreme rays of $\mathsf{rec}(\mathcal{C})$. On top of that, for any point $\boldsymbol{p}' \in \mathsf{extp}(\mathcal{C})$, $(\boldsymbol{p}', \mathbb{1})$ is an extreme generator of $\widehat{\mathcal{C}}$, so that there exists $(\lambda, \boldsymbol{p}) \in \mathbb{R}_{\max} \times P$ such that $(\boldsymbol{p}', \mathbb{1}) = \lambda(\boldsymbol{p}, \mathbb{1})$, thus $\lambda = \mathbb{1}$ and $\boldsymbol{p}' = \boldsymbol{p}$. This shows that $P = \mathsf{extp}(\mathcal{C})$. Similarly, considering an extreme ray $\langle\boldsymbol{r}'\rangle$ of $\mathsf{rec}(\mathcal{C})$, $\langle(\boldsymbol{r}', \mathbb{0})\rangle$ is an extreme ray of $\mathsf{rec}(\mathcal{C})$, and is necessarily represented by an element of $(P \times \{\mathbb{1}\}) \cup (R \times \{\mathbb{0}\})$ which is of the form $(\boldsymbol{r}, \mathbb{0})$, so that $\boldsymbol{r} \in R$. It follows that $\boldsymbol{r}$ belongs to the ray $\langle\boldsymbol{r}'\rangle$. As a result, (P, R) forms a minimal generating representation of $\mathcal{C}$. □

2.3 Tropical polyhedra and polyhedral cones

This section deals with tropical polyhedra and polyhedral cones, which are the analogues of the classic convex polyhedra and polyhedral convex cones. They are defined as the intersection of tropical halfspaces, as discussed in Section 2.3.1. Section 2.3.2 then establishes a tropical analogue of the Minkowski-Weyl theorem, stating that tropical polyhedra and polyhedral cones can be equivalently represented by means of a finite generating representation. Finally, Section 2.3.3 restates the homogenization technique introduced in Section 2.2.6 in the context of tropical polyhedra.

2.3.1 Definition

We first introduce the notion of tropical halfspaces:

Definition 2.11. A *tropical halfspace* is a set consisting of the elements $\boldsymbol{x} = (\boldsymbol{x}_i) \in \mathbb{R}_{\max}^d$ verifying an inequality constraint of the form:

$$\bigoplus_{1\leq i\leq d} \boldsymbol{a}_i \boldsymbol{x}_i \;\leq\; \bigoplus_{1\leq i\leq d} \boldsymbol{b}_i \boldsymbol{x}_i, \tag{2.7}$$

where $\boldsymbol{a} = (\boldsymbol{a}_i), \boldsymbol{b} = (\boldsymbol{b}_i) \in \mathbb{R}_{\max}^{1\times d}$.

Definition 2.12. A *tropical affine halfspace* is a set consisting of the elements $\boldsymbol{x} = (\boldsymbol{x}_i) \in \mathbb{R}_{\max}^d$ verifying an inequality constraint of the form:

$$\Big(\bigoplus_{1\leq i\leq d} \boldsymbol{a}_i \boldsymbol{x}_i\Big) \oplus c \;\leq\; \Big(\bigoplus_{1\leq i\leq d} \boldsymbol{b}_i \boldsymbol{x}_i\Big) \oplus d, \tag{2.8}$$

where $\boldsymbol{a} = (\boldsymbol{a}_i), \boldsymbol{b} = (\boldsymbol{b}_i) \in \mathbb{R}_{\max}^{1\times d}$, and $c, d \in \mathbb{R}_{\max}$.

The constraints given in (2.7) and (2.8) are respectively two-sided linear and affine inequalities with the laws of the semiring $\mathbb{R}_{\max}$. Unlike their classical analogues, they cannot be reduced to one-sided inequalities (since the additive law has no inverse).

However, some simplications can be performed. In (2.7), it can be supposed that for all i, either $\boldsymbol{a}_i = \mathbb{0}$ or $\boldsymbol{b}_i = \mathbb{0}$, or equivalently $\boldsymbol{a}_i \otimes \boldsymbol{b}_i = \mathbb{0}$. Indeed, if $\mathcal{H} \subset \mathbb{R}_{\max}^d$ is a tropical halfspace defined by the inequality given in (2.7), then it also coincides with the set of $\boldsymbol{x} = (\boldsymbol{x}_i)$ satisfying the inequality $\bigoplus_i \boldsymbol{a}'_i \boldsymbol{x}_i \leq \bigoplus_i \boldsymbol{b}'_i \boldsymbol{x}_i$, where:

$$\boldsymbol{a}'_i = \begin{cases} \boldsymbol{a}_i & \text{if } \boldsymbol{a}_i > \boldsymbol{b}_i, \\ \mathbb{0} & \text{if } \boldsymbol{a}_i \leq \boldsymbol{b}_i, \end{cases} \qquad \boldsymbol{b}'_i = \begin{cases} \boldsymbol{b}_i & \text{if } \boldsymbol{a}_i \leq \boldsymbol{b}_i, \\ \mathbb{0} & \text{if } \boldsymbol{a}_i > \boldsymbol{b}_i. \end{cases}$$

In that case, $\boldsymbol{a}'$ and $\boldsymbol{b}'$ will be said to be *orthogonal*.[1]

Similarly, in (2.8), the vectors $\begin{pmatrix}\boldsymbol{a} & c\end{pmatrix}$ and $\begin{pmatrix}\boldsymbol{b} & d\end{pmatrix}$ of $\mathbb{R}_{\max}^{d+1}$ can be supposed to be orthogonal, and $c \oplus d = \mathbb{1}$ as soon as c and d are not both equal to $\mathbb{1}$ (up to multiplying the whole inequality by $(c \oplus d)^{-1}$).

Under these assumptions, the *apex* of the halfspace is defined as the element $\boldsymbol{c} \in \mathbb{R} \cup \{+\infty\}$ such that $\boldsymbol{c}_i \overset{def}{=} (\boldsymbol{a}_i \oplus \boldsymbol{b}_i)^{-1}$ (with the convention $(-\infty)^{-1} = +\infty$).

[1] This term is due to the fact that the tropical "scalar product" $\bigoplus_i \boldsymbol{a}'_i \boldsymbol{b}'_i$ of $\boldsymbol{a}'$ and $\boldsymbol{b}'$ is equal to $\mathbb{0}$.

Example 2.6. For $d = 2$, consider the affine halfspace defined by the inequality:

$$(3\boldsymbol{x}_1) \oplus (-2\boldsymbol{x}_2) \oplus 5 \leq (-1\boldsymbol{x}_1) \oplus (3\boldsymbol{x}_2) \oplus 8$$

It can be first reduced to an inequality in which the two members are orthogonal:

$$3\boldsymbol{x}_1 \leq (3\boldsymbol{x}_2) \oplus 8.$$

Then, multiplying by 8^{-1} yields:

$$-5\boldsymbol{x}_1 \leq (-5\boldsymbol{x}_2) \oplus 0.$$

Its apex is therefore the element $(5, 5)$.

Figure 2.4 gives an illustration the twelve families of affine halfspaces in $\mathbb{R}^2_{\max}$, excluding $\emptyset$ and the whole set $\mathbb{R}^2_{\max}$. They are represented in green or red (the border depicted in black is included). For each, the apex is also depicted when its coordinates are finite. The six last halfspaces can be considered as degenerate instances of the six first ones.

Definition 2.13. A *tropical polyhedral cone* of $\mathbb{R}^d_{\max}$ is defined as the intersection of finitely many tropical halfspaces of $\mathbb{R}^d_{\max}$.

A *tropical polyhedron* of $\mathbb{R}^d_{\max}$ is defined as the intersection of finitely many tropical affine halfspaces of $\mathbb{R}^d_{\max}$.

Equivalently, tropical polyhedra and polyhedral cones can be expressed as the set of the solutions of systems of inequality constraints. First, let us extend the additive and multiplicative laws of $\mathbb{R}_{\max}$ to matrices:

$$\begin{aligned} A \oplus B &\stackrel{def}{=} (a_{ij} \oplus b_{ij}) && \text{for } A = (a_{ij}), B = (b_{ij}) \in \mathbb{R}^{p\times d}_{\max}, \\ C\,D &\stackrel{def}{=} \Big(\bigoplus_{k=1}^{p} c_{ik} d_{kj}\Big) && \text{for } C = (c_{ij}) \in \mathbb{R}^{n\times p}_{\max}, D = (d_{ij}) \in \mathbb{R}^{p\times d}_{\max}. \end{aligned}$$

In particular, if $A = (a_{ij}) \in \mathbb{R}^{p\times d}_{\max}$ and $\boldsymbol{x} \in \mathbb{R}^d_{\max}$, $A\boldsymbol{x}$ is the vector of $\mathbb{R}^p_{\max}$ whose i-th entry is given $\bigoplus_{j=1}^{d} a_{ij}\boldsymbol{x}_j$. Besides, we can partially order the elements of $\mathbb{R}^d_{\max}$ using the entrywise extension of $\leq$, *i.e.* $\boldsymbol{x} \leq \boldsymbol{y}$ if and only if $\boldsymbol{x}_i \leq \boldsymbol{y}_i$ for all i. With these notations, a subset of $\mathbb{R}^d_{\max}$ is a tropical polyhedron if and only if it is of the form

$$\{\, \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d} \,\},$$

where $A, B \in \mathbb{R}^{p\times d}_{\max}$, $\boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}^p_{\max}$, and $p \geq 0$. Each row of the system is an affine inequality, and corresponds to an affine halfspace. Similarly, a polyhedral cone can be defined as the set of the solutions of a system $A\boldsymbol{x} \leq B\boldsymbol{x}$, with $A, B \in \mathbb{R}^{p\times d}_{\max}$ ($p \geq 0$).

The following proposition provides a description of the recession cone of tropical polyhedra from their halfspaces:

Proposition 2.4. *Let $\mathcal{P} = \{\, \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d} \,\}$ be a non-empty tropical polyhedron ($A, B \in \mathbb{R}^{p\times d}_{\max}$, $\boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}^p_{\max}$). Then the following relation holds:*

$$\mathsf{rec}(\mathcal{P}) = \{\, \boldsymbol{y} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{y} \leq B\boldsymbol{y} \,\}.$$

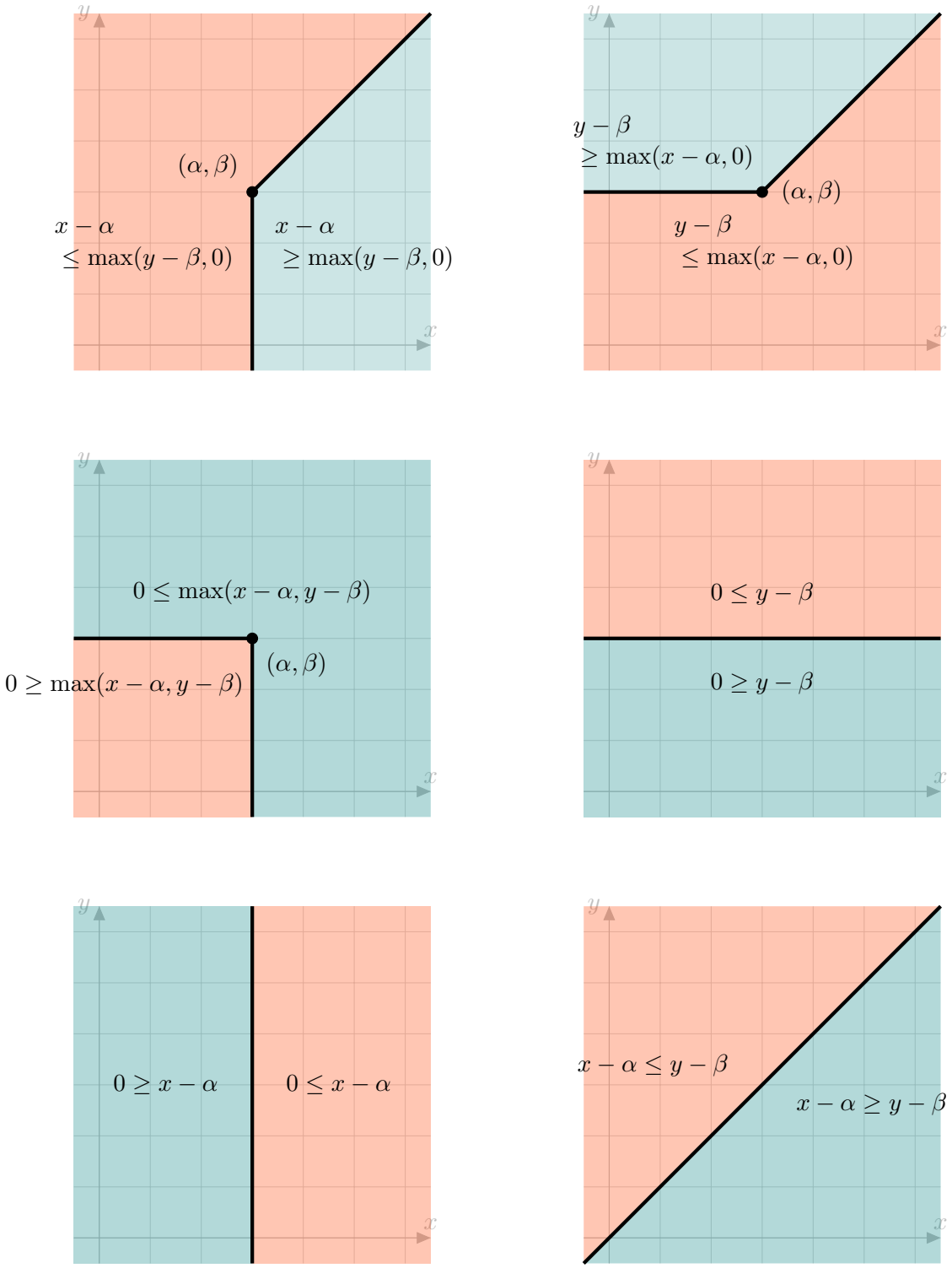


Figure 2.4: The twelve families of non-trivial affine halfspaces in $\mathbb{R}^2_{\max}$

Proof. Consider $\boldsymbol{x} \in \mathcal{P}$, and let $\boldsymbol{y} \in \mathsf{rec}(\mathcal{P})$. Consider $i \in \{1, \dots, p\}$. By definition, for all $\lambda \in \mathbb{R}_{\max}$,

$$A_i\boldsymbol{x} \oplus \boldsymbol{c}_i \oplus \lambda(A_i\boldsymbol{y}) \;\leq\; B_i\boldsymbol{x} \oplus \boldsymbol{d}_i \oplus \lambda(B_i\boldsymbol{y})$$

where A_i and B_i are respectively the i-th row of A and B. If $A_i\boldsymbol{y} = \mathbb{0}$, then obviously $A_i\boldsymbol{y} \leq B_i\boldsymbol{y}$. Otherwise, for large values of λ, we have $\lambda(A_i\boldsymbol{y}) > B_i\boldsymbol{x} \oplus \boldsymbol{d}_i$, which implies that $A_i\boldsymbol{y} \leq B_i\boldsymbol{y}$ (as in particular, $\lambda > \mathbb{0}$).

Conversely, suppose that $A_i\boldsymbol{y} \leq B_i\boldsymbol{y}$ for each i. Then for all $\lambda \in \mathbb{R}_{\max}$, $\lambda(A_i\boldsymbol{y}) \leq B_i(\boldsymbol{x} \oplus \boldsymbol{d}_i) \oplus \lambda(B_i\boldsymbol{y})$, and clearly $A_i\boldsymbol{x} \oplus \boldsymbol{c}_i \leq (\boldsymbol{x} \oplus \boldsymbol{d}_i) \oplus \lambda(B_i\boldsymbol{y})$ (using $\boldsymbol{x} \in \mathcal{P}$). It follows that $\boldsymbol{x} \oplus \lambda\boldsymbol{y} \in \mathcal{P}$. □

Remark 2.7. By analogy with the classical case, *tropical polytopes* can be defined as tropical polyhedra bounded in $\mathbb{R}^d_{\max}$. It can be easily shown that they are precisely characterized by a recession cone reduced to the singleton $\{\mathbb{0}\}$.

Observe that our definition of tropical polytopes is more general that Develin and Sturmfels' one [DS04], in which tropical polytopes are required to be bounded sets of $\mathbb{R}^d$ (equivalently, none of their points have a coordinate equal to $\mathbb{0}$).

Also note that Joswig provides in [Jos05] a slightly different definition of tropical (affine) halfspaces. It relies on tropical affine hyperplanes. A *tropical affine hyperplane* is associated to an element $\boldsymbol{c} \in \mathbb{R}^{d+1}$, and is defined as the set of elements $\boldsymbol{x} \in \mathbb{R}^d$ such that the maximum

$$\boldsymbol{c}_1\boldsymbol{x}_1 \oplus \ldots \oplus \boldsymbol{c}_d\boldsymbol{x}_d \oplus \boldsymbol{c}_{d+1} = \max(\boldsymbol{c}_1 + \boldsymbol{x}_1, \dots, \boldsymbol{c}_d + \boldsymbol{x}_d, \boldsymbol{c}_{d+1})$$

is attained at least twice. Up to multiplying $\boldsymbol{c}$ by a scalar, it can be supposed that $\boldsymbol{c}_{d+1} = \mathbb{1}$. Then $\boldsymbol{c}$ can be assimilated to the element $(\boldsymbol{c}_1, \dots, \boldsymbol{c}_d)$ of $\mathbb{R}^d$. A tropical hyperplane and its associated element in $\mathbb{R}^2_{\max}$ are depicted in Figure 2.5 in black.

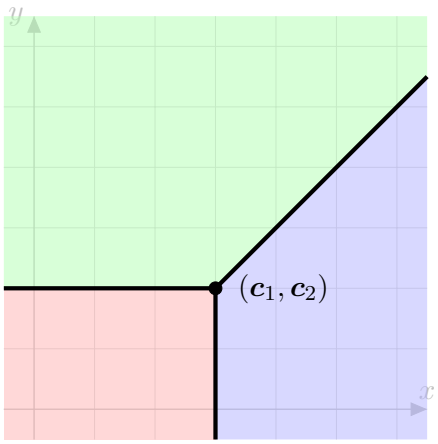


Figure 2.5: A tropical hyperplane

A tropical hyperplane splits the space $\mathbb{R}^d$ into $(d+1)$ connected components, named *open sectors* (see blue, red, and green areas in Figure 2.5). Their closure are called *closed sectors*. Tropical affine halfspaces are then defined as the union of at least one and at most d closed sectors of a given tropical hyperplane. It can be shown that they are in fact tropical affine halfspaces in the sense of Definition 2.12. Their apex is precisely the point $(\boldsymbol{c}_1, \dots, \boldsymbol{c}_d)$ (supposing that $\boldsymbol{c}_{d+1} = \mathbb{1}$).

2.3.2 Minkowski-Weyl theorem

A tropical analogue of the Minkowski-Weyl theorem has been established in several works (see for instance [GK06, DS04]), stating that tropical polyhedra and polyhedral cones are exactly finitely generated convex sets and cones respectively:

Theorem 2.5. *The tropical polyhedra of $\mathbb{R}^d_{\max}$ are precisely the sets of the form $\mathsf{co}(P) \oplus \mathsf{cone}(R)$ where P and R are finite subsets of $\mathbb{R}^d_{\max}$.*

Theorem 2.6. *The tropical polyhedral cones of $\mathbb{R}^d_{\max}$ are precisely the sets of the form $\mathsf{cone}(G)$ where G is a finite subset of $\mathbb{R}^d_{\max}$.*

A constructive proof will be provided in Chapter 5. As finitely generated sets, polyhedra and tropical cones can be shown to be closed:

Lemma 2.5. *Tropical polyhedra and polyhedral cones are closed sets.*

Proof. We only give the details of the proof for cones, as the case of polyhedra is very similar.

Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical polyhedral cone, and let $G = (\boldsymbol{g}^i)_{1\leq i\leq p}$ be a generating set. Without loss of generality, it can be assumed that $\boldsymbol{g}^i \neq \mathbb{0}$ for each i. Consider a sequence $(\boldsymbol{x}^n)_n \in \mathcal{C}^{\mathbb{N}}$ which converges to an element $\boldsymbol{x} \in \mathbb{R}^d_{\max}$. Then for each n, there exists $\lambda^n_1, \ldots, \lambda^n_p \in \mathbb{R}_{\max}$ such that $\boldsymbol{x}^n = \bigoplus_{i=1}^p \lambda^n_i \boldsymbol{g}^i$. For all $i \in \{1, \ldots, p\}$, let j such that $\boldsymbol{g}^i_j \neq \mathbb{0}$. The sequence λ^n_i is bounded, since $\lambda^n_i \leq (\boldsymbol{g}^i_j)^{-1}\boldsymbol{x}^n_j$ and $(\boldsymbol{x}^n_j)_n$ is also bounded (as it is converging to $\boldsymbol{x}_j$). It follows from Bolzano-Weierstrass theorem that, up to extracting a subsequence, the sequence $(\lambda^n_i)_n$ is also converging towards a limit $\lambda_i \in \mathbb{R}_{\max}$. Hence, $\boldsymbol{x} = \bigoplus_{i=1}^p \lambda_i \boldsymbol{g}^i$, which proves that $\boldsymbol{x}$ belongs to $\mathcal{C}$. □

A consequence of Theorems 2.1 and 2.2 is that the minimal generating representations of tropical polyhedra and polyhedral cones are all finite, and have the same size:

Proposition 2.6. *Any minimal generating set $G \subset \mathbb{R}^d_{\max}$ of a polyhedral cone $\mathcal{C}$ is finite, and its cardinality is equal to the number of extreme rays of $\mathcal{C}$.*

Similarly, for any minimal representation (P, R) of a tropical polyhedron $\mathcal{P}$, the sets P and R are both finite. Their cardinality is equal to the number of extreme points in $\mathcal{P}$ and of extreme rays in $\mathsf{rec}(\mathcal{P})$ respectively.

Proof. Trivial from Lemma 2.5, and Theorems 2.3 and 2.4. □

Remark 2.8. Note however that the size of the minimal generating representations of polyhedra (resp. cones) can be arbitrarily large as soon as $d \geq 2$ (resp. $d \geq 3$). For instance, in $\mathbb{R}^d_{\max}$, consider a tropical analogue of the cyclic polytope (see for instance [Zie98] for a definition of cyclic polytopes in the classical case), given a choice of n distinct elements $\mathbb{0} < t_1 < \ldots < t_n$ of $\mathbb{R}^d_{\max}$, and defined by:

$$\Gamma_d(t_1, \ldots, t_n) \stackrel{def}{=} \mathsf{co}(\boldsymbol{p}(t_1), \ldots, \boldsymbol{p}(t_n)),$$

where $\boldsymbol{p}(t) \stackrel{def}{=} (t, t^2, \ldots, t^d)$.[2] It can be proved that the n points defining $\Gamma_d(t_1, \ldots, t_n)$ are all extreme, so that the size of any minimal representation of the polytope is exactly n.

Figure 2.6 depicts the cyclic polytope of $\mathbb{R}^2_{\max}$ given by $n = 6$ and $t_i = i - 1$ for all i.

[2]The notation x^i corresponds to the tropical exponentiation, *i.e.* $x^i = i \times x$.

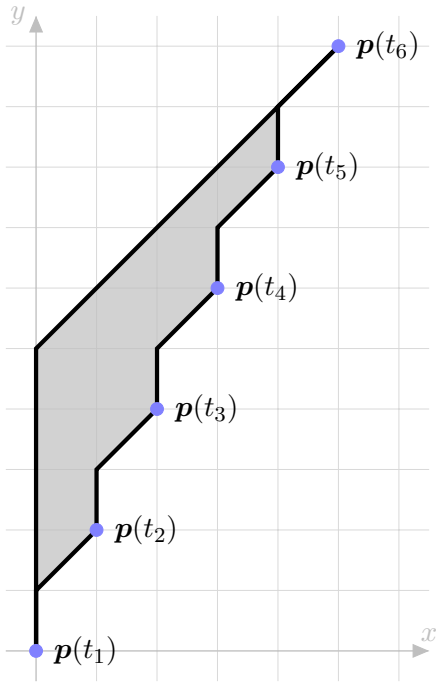


Figure 2.6: A tropical cyclic polytope in $\mathbb{R}^2_{\max}$

2.3.3 Homogenization of tropical polyhedra

We can now express homogenization for tropical polyhedra. In what follows, the concatenation of two matrices $M \in \mathbb{R}_{\max}^{n\times p}$ and $N \in \mathbb{R}_{\max}^{n\times q}$ is denoted by $\begin{pmatrix} M & N \end{pmatrix}$.

Proposition 2.7. *Let $\mathcal{H} = \{ \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid \boldsymbol{a}\boldsymbol{x} \oplus c \leq \boldsymbol{b}\boldsymbol{x} \oplus d \}$ be a non-empty affine halfspace ($\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d_{\max}$, $c, d \in \mathbb{R}_{\max}$). Then $\widehat{\mathcal{H}}$ is a tropical halfspace, and is given by:*

$$\widehat{\mathcal{H}} = \{ \boldsymbol{z} \in \mathbb{R}^{d+1}_{\max} \mid \begin{pmatrix} \boldsymbol{a} & c \end{pmatrix} \boldsymbol{z} \leq \begin{pmatrix} \boldsymbol{b} & d \end{pmatrix} \boldsymbol{z} \}.$$

Proposition 2.8. *Let $\mathcal{P} = \{ \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d} \}$ be a non-empty tropical polyhedron ($A, B \in \mathbb{R}^{p\times d}_{\max}$, $\boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}^d_{\max}$). Then $\widehat{\mathcal{P}}$ is a polyhedral cone, and the following equality holds:*

$$\widehat{\mathcal{P}} = \{ \boldsymbol{z} \in \mathbb{R}^{d+1}_{\max} \mid \begin{pmatrix} A & \boldsymbol{c} \end{pmatrix} \boldsymbol{z} \leq \begin{pmatrix} B & \boldsymbol{d} \end{pmatrix} \boldsymbol{z} \}.$$

Proof. By Lemma 2.5, $\mathcal{P}$ is closed, so that by applying Proposition 2.1, we get:

$$\widehat{\mathcal{P}} = \{ (\alpha\boldsymbol{x}, \alpha) \mid \boldsymbol{x} \in \mathcal{C},\ \alpha \in \mathbb{R}_{\max} \} \cup \{ (\boldsymbol{y}, \mathbb{0}) \mid \boldsymbol{y} \in \mathsf{rec}(\mathcal{P}) \}.$$

Then using Proposition 2.4, we clearly have $\begin{pmatrix} A & \boldsymbol{c} \end{pmatrix} \boldsymbol{z} \leq \begin{pmatrix} B & \boldsymbol{d} \end{pmatrix} \boldsymbol{z}$ for all $\boldsymbol{z} \in \widehat{\mathcal{P}}$.

Conversely, suppose that $\begin{pmatrix} A & \boldsymbol{c} \end{pmatrix} \boldsymbol{z} \leq \begin{pmatrix} B & \boldsymbol{d} \end{pmatrix} \boldsymbol{z}$. If $\boldsymbol{z} = (\boldsymbol{y}, \mathbb{0})$, then $A\boldsymbol{y} \leq B\boldsymbol{y}$, so that $\boldsymbol{y} \in \mathsf{rec}(\mathcal{P})$ (applying Proposition 2.4 to $\mathcal{P}$ which is not empty) and $\boldsymbol{z} \in \widehat{\mathcal{P}}$ (Proposition 2.1). Otherwise, $\boldsymbol{z} = (\boldsymbol{x}, \alpha)$ with $\alpha > \mathbb{0}$, so that $\boldsymbol{x}' \stackrel{def}{=} \alpha^{-1}\boldsymbol{x}$ satisfies $A\boldsymbol{x}' \oplus \boldsymbol{c} \leq B\boldsymbol{x}' \oplus \boldsymbol{d}$. It follows by (2.2) that $\boldsymbol{z} = (\alpha\boldsymbol{x}', \alpha)$ belongs to $\widehat{\mathcal{P}}$. □

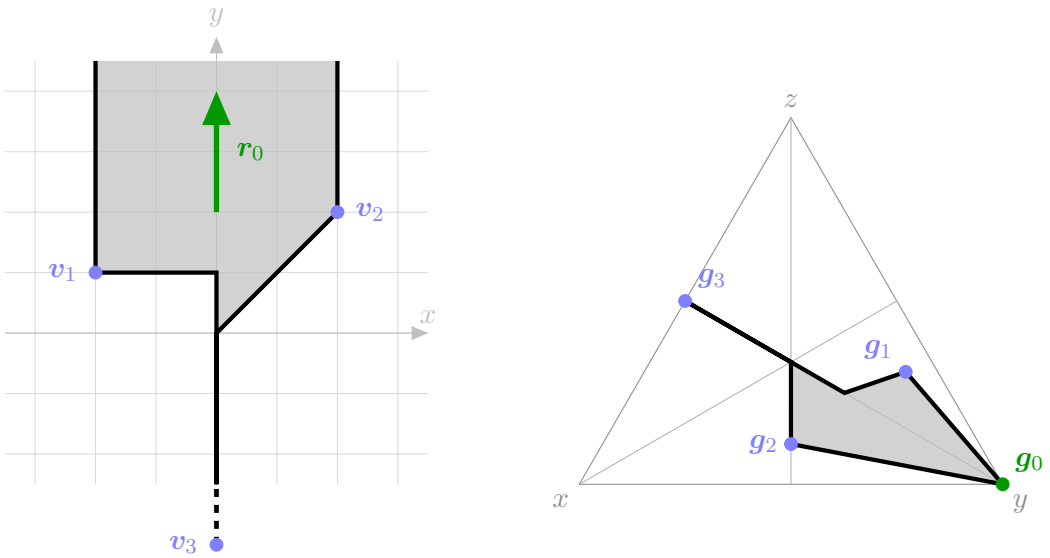


Figure 2.7: A tropical polyhedron in $\mathbb{R}^2_{\max}$ (left), and an equivalent representation by a cone in $\mathbb{R}^3_{\max}$ (right)

The following proposition formalizes the inverse correspondence between a certain class of polyhedral cones and tropical polyhedra:

Proposition 2.9. *Let $\mathcal{C} \subset \mathbb{R}^{d+1}_{\max}$ such that $\{ \boldsymbol{z} \in \mathcal{C} \mid \boldsymbol{z}_{d+1} = \mathbb{1} \}$ is not empty. Then there exists a unique tropical polyhedron $\mathcal{P}$ such that $\widehat{\mathcal{P}} = \mathcal{C}$.*

Proof. Let $\mathcal{P}$ be defined as $\{ \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid (\boldsymbol{x}, \mathbb{1}) \in \mathcal{C} \}$.

Suppose that $\mathcal{C} = \{ \boldsymbol{z} \in \mathbb{R}^{d+1}_{\max} \mid C\boldsymbol{z} \leq D\boldsymbol{z} \}$, with $C, D \in \mathbb{R}^{p \times d+1}_{\max}$. Let $A, B \in \mathbb{R}^{p \times d}_{\max}$ and $\boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}^p_{\max}$ be defined as $\begin{pmatrix} A & \boldsymbol{c} \end{pmatrix} = C$ and $\begin{pmatrix} B & \boldsymbol{d} \end{pmatrix} = D$. Then we have:

$$\mathcal{P} = \{ \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d} \}.$$

Besides, since $\mathcal{P}$ is not empty, Proposition 2.8 implies that $\widehat{\mathcal{P}} = \mathcal{C}$.

The uniqueness of $\mathcal{P}$ comes from Proposition 2.1, which ensures that any tropical polyhedron $\mathcal{Q}$ which satisfies $\widehat{\mathcal{Q}} = \mathcal{C}$ also satisfies $\mathcal{Q} = \{ \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid (\boldsymbol{x}, \mathbb{1}) \in \mathcal{C} \} = \mathcal{P}$. □

Remark 2.9. Observe that for any tropical cone $\mathcal{C} \neq \{ \mathbb{0} \}$ such that for all $\boldsymbol{z} \in \mathcal{C}$, $\boldsymbol{z}_{d+1} = \mathbb{0}$, there is no tropical polyhedron $\mathcal{P}$ verifying $\widehat{\mathcal{P}} = \mathcal{C}$.

In particular, when $\mathcal{C} = \{ \boldsymbol{z} \in \mathbb{R}^{d+1}_{\max} \mid \begin{pmatrix} A & \boldsymbol{c} \end{pmatrix} \boldsymbol{z} \leq \begin{pmatrix} B & \boldsymbol{d} \end{pmatrix} \boldsymbol{z} \}$, and for any $\boldsymbol{z} \in \mathcal{C}$, $\boldsymbol{z}_{d+1} = \mathbb{0}$, then the tropical polyhedron $\mathcal{P}$ given by $\mathcal{P} = \{ \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d} \}$ is necessarily empty. In that case, it homogenized cone is reduced to $\{ \mathbb{0} \}$.

The results relative to homogenization in the context of tropical polyhedra is recapitulated in the following corollary:

Corollary 2.10. *There is a one-to-one correspondence between non-empty tropical polyhedra of $\mathbb{R}^d_{\max}$ and tropical polyhedral cones of $\mathbb{R}^{d+1}_{\max}$ whose intersection with the set $\{ \boldsymbol{z} \in \mathbb{R}^{d+1}_{\max} \mid$*

$\boldsymbol{z}_{d+1} = \mathbb{1}\,\}$ *is not empty:*

$$\mathcal{P} \longmapsto \widehat{\mathcal{P}},$$
$$\{\,\boldsymbol{x} \in \mathbb{R}^d_{\max} \mid (\boldsymbol{x}, \mathbb{1}) \in \mathcal{C}\,\} \longleftarrow\!\shortmid \mathcal{C}.$$

Their unique scaled minimal generating representations and their associated systems of inequalities are related in the following way:

$$(P, R) \longmapsto \iota(P, R) \stackrel{def}{=} (\sigma(P \times \{\,\mathbb{1}\,\}) \cup (R \times \{\,\mathbb{0}\,\}),$$
$$\iota^{-1}(G) \stackrel{def}{=} \begin{pmatrix} \{\,\alpha^{-1}\boldsymbol{p} \mid (\boldsymbol{p}, \alpha) \in G, \alpha \neq \mathbb{0}\,\}, \\ \{\,\boldsymbol{r} \mid (\boldsymbol{r}, \mathbb{0}) \in G\,\} \end{pmatrix} \longleftarrow\!\shortmid G,$$
$$\{\,\boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d}\,\} \longmapsto\!\shortmid\!\longleftarrow \{\,\boldsymbol{z} \in \mathbb{R}^{d+1}_{\max} \mid \begin{pmatrix} A & \boldsymbol{c} \end{pmatrix} \boldsymbol{z} \leq \begin{pmatrix} B & \boldsymbol{d} \end{pmatrix} \boldsymbol{z}\,\}.$$

Proof. Let us begin by showing that the first map is indeed a one-to-one correspondence. Clearly, any cone $\mathcal{C} = \widehat{\mathcal{P}}$ where $\mathcal{P}$ is a non-empty polyhedron is such that $\mathcal{C} \cap \{\,\boldsymbol{z} \mid \boldsymbol{z}_{d+1} = \mathbb{0}\,\}$. Conversely, thanks to Proposition 2.9, any cone which satisfies this property admits a polyhedron $\mathcal{P}$ such that $\widehat{\mathcal{P}} = \mathcal{C}$. It is necessarily unique and non-empty, since according to Proposition 2.1, it satisfies $\mathcal{P} = \{\,\boldsymbol{x} \in \mathbb{R}^d_{\max} \mid (\boldsymbol{x}, \mathbb{1}) \in \mathcal{C}\,\}$.

Now, consider the unique minimal and scaled generating representation (P, R) of $\mathcal{P}$. Using Corollary 2.3, we know that $(P \times \{\,\mathbb{1}\,\}) \cup (R \times \{\,\mathbb{0}\,\})$ is a minimal generating set of $\widehat{\mathcal{P}}$. Every element of the form $(\boldsymbol{r}, \mathbb{0})$ is scaled as soon as $\boldsymbol{r}$ is itself scaled, so that $\iota(P, R)$ is indeed the unique minimal and scaled generating set of $\widehat{\mathcal{P}}$.

Reciprocally, let G be the unique scaled and minimal generating set of $\widehat{\mathcal{P}}$. Let H be the set formed by the elements $\boldsymbol{g} \in G$ such that $\boldsymbol{g}_{d+1} = \mathbb{0}$, and by the $\boldsymbol{g}_{d+1}^{-1}\boldsymbol{g}$ for every $\boldsymbol{g} \in G$ such that $\boldsymbol{g}_{d+1} \neq \mathbb{0}$. The set H contains precisely one representative of each extreme ray of $\widehat{\mathcal{P}}$, so that it is also a minimal generating set of $\widehat{\mathcal{P}}$ (Theorem 2.3). Then using Corollary 2.3, the couple (P, R) which satisfies $H = (P \times \{\,\mathbb{1}\,\}) \cup (R \times \{\,\mathbb{0}\,\})$ is also a minimal generating representation of $\mathcal{P}$. Since every element $(\boldsymbol{r}, \mathbb{0}) \in G$ are scaled, each $\boldsymbol{r} \in R$ is also scaled, so that (P, R) is a scaled representation.

The correspondence between systems of inequalities is a direct consequence of Proposition 2.8. □

Example 2.10. In Chapters 3 and 5, we will illustrate our results on the tropical polyhedron $\mathcal{P}$ defined by the system (2.9), or equivalently on its homogenized cone $\mathcal{C}$ defined by the system (2.10):[3]

$$\begin{cases} 0 \leq x + 2 \\ x \leq \max(y, 0) \\ x \leq 2 \\ 0 \leq \max(x, y - 1) \end{cases} \qquad (2.9) \qquad\qquad \begin{cases} z \leq x + 2 \\ x \leq \max(y, z) \\ x \leq z + 2 \\ z \leq \max(x, y - 1) \end{cases} \qquad (2.10)$$

The polyhedron $\mathcal{P}$ is depicted in solid gray (the black border is included) in the left hand side of Figure 2.7. It is generated by the extreme points $\boldsymbol{v}_1 = (-2, 1)$, $\boldsymbol{v}_2 = (2, 2)$, and $\boldsymbol{v}_3 = (0, \mathbb{0})$, and by the extreme ray $\langle \boldsymbol{r}_0 \rangle$ where $\boldsymbol{r}_0 = (\mathbb{0}, 0)$.

[3] When dealing with examples in $\mathbb{R}^2_{\max}$ or in $\mathbb{R}^3_{\max}$, we shall consider vectors the entries of which are denoted by x, y, z rather than $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$.

The right side of Figure 2.7 is a representation of the polyhedral cone $\mathcal{C}$ in barycentric coordinates: each element (x, y, z) is represented as a barycenter with weights (e^x, e^y, e^z) of the three vertices of the outermost triangle. Then two representatives of a same ray are represented by the same point. The tropical cone $\mathcal{C}$ is generated by the extreme elements $\boldsymbol{g}_0 = (\mathbb{0}, 0, \mathbb{0})$, $\boldsymbol{g}_1 = (-2, 1, 0)$, $\boldsymbol{g}_2 = (2, 2, 0)$, and $\boldsymbol{g}_3 = (0, \mathbb{0}, 0)$.

2.4 Conclusion of the chapter

This chapter introduces the basics of tropical convexity. We highlight the results of the chapter which will be fundamental in the rest of the manuscript, and introduce some additional terminology.

Firstly, tropical polyhedra and polyhedral cones admit two possible representations (Theorems 2.5 and 2.6):

- by means of a system of affine/linear inequalities (or equivalently as the intersection of finitely many (affine) halfspaces). We will also use the term *external representation*.
- by means of a finite generating representation, which we will also call *internal representation*.

Secondly, minimal generating representations of polyhedra and polyhedral cones are precisely provided by their extreme elements (Theorems 2.4 and 2.3). For each, there exists a unique scaled minimal generating representation, which can be seen as a canonical representation.

Finally, using homogenization, tropical polyhedra of $\mathbb{R}^d_{\max}$ can be equivalently represented by means of polyhedral cones of $\mathbb{R}^{d+1}_{\max}$. In particular, their minimal generating representations are related by a one-to-one correspondence denoted by ι (Corollary 2.10). For this reason, the vast majority of the results of Chapters 3 and 5 will be established for tropical cones.

CHAPTER 3

Combinatorial characterization of extremality from the description of polyhedra by halfspaces

In this chapter, we develop a combinatorial criterion to characterize the extremality in tropical polyhedral cones. The novelty is that this criterion relies on the description of polyhedra by means of halfspaces.

As far as we know, this is the first time that such a criterion is established in the tropical setting. This idea came from the existence of similar ways to characterize extremality in classical polyhedral cones. Indeed, Motzkin's double description method [MRTT53] and Chernikova's algorithm [Che68], which both compute generating sets of classical cones defined by inequalities, rely on an elimination technique of non-extreme rays based on some considerations on the descriptions by halfspaces given as input. Their elimination criterion uses the notion of saturated inequalities: in a classical cone $\mathcal{C} \subset \mathbb{R}^d$ defined by a system $A\boldsymbol{x} \geq 0$, an inequality $\boldsymbol{a}\boldsymbol{x} \geq 0$ of the system is said to be saturated by an element $\boldsymbol{g}$ of the cone if $\boldsymbol{a}\boldsymbol{g} = 0$. Then, the extremality of $\boldsymbol{g} \in \mathcal{C}$ can be characterized by one of the two equivalent criteria:[1]

- the rank of the matrix formed by the inequalities saturated by $\boldsymbol{g}$ is equal to $d-1$,
- the set of the inequalities saturated by $\boldsymbol{g}$ is maximal (among the other elements of the cone, or equivalently, among the other elements of a generating set).

[1] Note that we assume here that the cone is pointed.

We refer to the survey of Fukuda and Prodon [FP96] and the note of Le Verge [LV92] for further details.

Unfortunately, these criteria do not have direct tropical analogues. The characterization of extremality established in this chapter relies on the saturated inequalities, but further properties on these inequalities are required.

In Section 3.1, we first introduce equivalent ways to express extremality in tropical cones. In particular, it will be shown that the extremality of an element can be characterized from its neighborhood in the cone. Section 3.2 will develop the notion of tangent cone, and how it can be used to establish a first extremality criterion. In Section 3.3, we will rewrite this criterion in terms of directed hypergraphs, finally leading to our combinatorial criterion.

Note that we do not detail the corresponding criterion in tropical polyhedra, which can be easily derived thanks to the homogenization technique discussed in Chapter 2.

3.1 Preliminaries on extremality

In this section, we establish some general properties related to extremality. They not only hold for polyhedral cones, but also for tropical cones.

The first one is crucial for the rest of chapter. It states that extremality can be expressed as a minimality property. It is in fact a variation on the proof of Theorem 3.1 of [GK07] and on Theorem 14 of [BSS07]. In particular, it allows to associate a combinatorial type to extreme elements of tropical cones.

The minimality property is defined as follows:

Definition 3.1. Let $S \subset \mathbb{R}^d_{\max}$ and $\boldsymbol{g} \in \mathbb{R}^d_{\max}$.

Then $\boldsymbol{g}$ is said to be *minimal of type t in S* ($1 \leq t \leq d$) if it is a minimal element of the set $\{\, \boldsymbol{x} \in S \mid \boldsymbol{x}_t = \boldsymbol{g}_t \,\}$, *i.e.* $\boldsymbol{g} \in S$ and for each $\boldsymbol{x} \in S$,

$$(\, \boldsymbol{x} \leq \boldsymbol{g} \text{ and } \boldsymbol{x}_t = \boldsymbol{g}_t \,) \implies \boldsymbol{x} = \boldsymbol{g}.$$

Proposition-Definition 3.1. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical cone, and $\boldsymbol{g} \in \mathbb{R}^d_{\max}$. The element $\boldsymbol{g}$ is extreme in $\mathcal{C}$ if and only if there exists $1 \leq t \leq d$ such that $\boldsymbol{g}$ is minimal of type t in $\mathcal{C}$.*

In that case, $\boldsymbol{g}$ is said to be extreme of type t.

Proof. If there exists $1 \leq t \leq d$ such that $\boldsymbol{g}$ is minimal in $\{\, \boldsymbol{x} \in \mathcal{C} \mid \boldsymbol{x}_t = \boldsymbol{g}_t \,\}$, then let $\boldsymbol{x}^1, \boldsymbol{x}^2 \in \mathcal{C}$ such that $\boldsymbol{g} = \boldsymbol{x}^1 \oplus \boldsymbol{x}^2$. In that case, for each $i \in \{\, 1, 2 \,\}$, $\boldsymbol{x}^i \leq \boldsymbol{g}$, and there is an i such that $\boldsymbol{x}^i_t = \boldsymbol{g}_t$, so that $\boldsymbol{x}^i = \boldsymbol{g}$.

Conversely, assume that for every index t, $\boldsymbol{g}$ is not minimal in the set $\{\, \boldsymbol{x} \in \mathcal{C} \mid \boldsymbol{x}_t = \boldsymbol{g}_t \,\}$, so that we can find a vector $\boldsymbol{x}^t$ such that $\boldsymbol{x}^t \leq \boldsymbol{g}$, $\boldsymbol{x}^t_t = \boldsymbol{g}_t$, and $\boldsymbol{x}^t \neq \boldsymbol{g}$. As a result, $\boldsymbol{g} = \boldsymbol{x}^1 \oplus \ldots \oplus \boldsymbol{x}^d$. Since no $\boldsymbol{x}^t$ is equal to $\boldsymbol{g}$, this shows that $\boldsymbol{g}$ cannot be extreme. □

Example 3.1. In Figure 3.1, the light blue area represents the set of the elements (x, y, z) of $\mathbb{R}^3_{\max}$ such that $(x, y, z) \leq \boldsymbol{g}^2$ implies $x < \boldsymbol{g}^2_x$. It clearly contains the whole cone except $\boldsymbol{g}^2$, which shows that the latter element is extreme of type x.[2]

Proposition 3.1 implies the following technical lemma, which will be quite useful in some proofs:

[2]For the sake of simplicity, we identify the type to the corresponding coordinate of the vector, saying for instance that the vector is extreme "of type y", instead of "type 2".

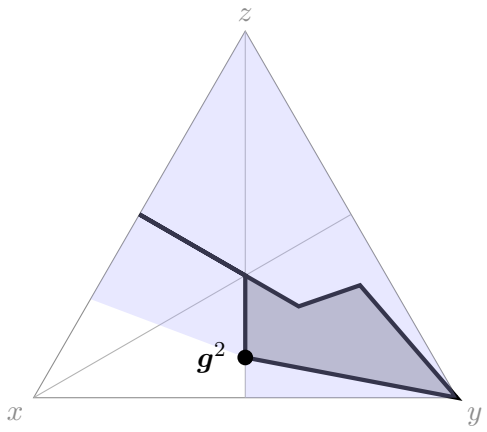


Figure 3.1: Extremality of the element $\boldsymbol{g}^2$

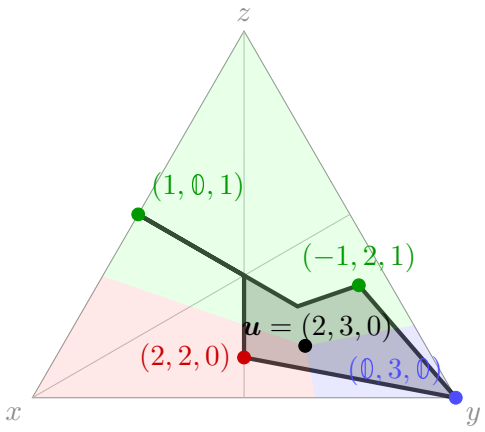


Figure 3.2: Illustration of the proof of Theorem 2.2

Lemma 3.2. *If $\boldsymbol{g} \neq \mathbb{0}$ is extreme of type t in a cone $\mathcal{C} \subset \mathbb{R}^d_{\max}$, then $\boldsymbol{g}_t \neq \mathbb{0}$.*

Proof. Suppose that $\boldsymbol{g}_t = \mathbb{0}$, and let $\boldsymbol{g}' = (-1)\boldsymbol{g}$. Then $\boldsymbol{g}' \in \mathcal{C}$, $\boldsymbol{g}' \leq \boldsymbol{g}$, and $\boldsymbol{g}' \neq \boldsymbol{g}$ since $\boldsymbol{g} \neq \mathbb{0}$. This is a contradiction with the extremality of type t of $\boldsymbol{g}$. □

For $d \geq 1$, the set of the integers $\{1, \ldots, d\}$ is denoted by $[d]$. For every $\boldsymbol{x} = (\boldsymbol{x}_i) \in \mathbb{R}^d_{\max}$, the *support* of the vector $\boldsymbol{x}$ is defined as the set of the indexes of its non-null coordinates:

$$\mathsf{supp}(\boldsymbol{x}) \stackrel{def}{=} \{\, i \in [d] \mid \boldsymbol{x}_i \neq \mathbb{0} \,\}.$$

The following proposition states that the extremality of an element in a tropical cone can be established only by considering the vectors of the cone which have a smaller support:

Proposition 3.3. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical cone, and $\boldsymbol{g}$ an element of $\mathcal{C}$. Then the two following statements are equivalent:*

(i) $\boldsymbol{g}$ is extreme of type t in $\mathcal{C}$,

(ii) $\boldsymbol{g}$ is extreme of type t in $\{\, \boldsymbol{x} \in \mathcal{C} \mid \mathsf{supp}(\boldsymbol{x}) \subset \mathsf{supp}(\boldsymbol{g}) \,\}$.

Proof. Let $\mathcal{D} \stackrel{def}{=} \{\, \boldsymbol{x} \in \mathcal{C} \mid \mathsf{supp}(\boldsymbol{x}) \subset \mathsf{supp}(\boldsymbol{g}) \,\}$. It is straightforward that $\mathcal{D}$ is a tropical cone.

Supposing $\boldsymbol{g}$ extreme of type t in $\mathcal{C}$, $\boldsymbol{g}$ is obviously extreme of type t in $\mathcal{D}$, since $\boldsymbol{g} \in \mathcal{D}$ and $\mathcal{D} \subset \mathcal{C}$.

Conversely, suppose $\boldsymbol{g}$ extreme of type t in the cone $\mathcal{D}$. By Lemma 3.2, t belongs to $\mathsf{supp}(\boldsymbol{g})$. Consider $\boldsymbol{x} \in \mathcal{C}$ such that $\boldsymbol{x} \leq \boldsymbol{g}$ and $\boldsymbol{x}_t = \boldsymbol{g}_t$. Then necessarily $\mathsf{supp}(\boldsymbol{x}) \subset \mathsf{supp}(\boldsymbol{g})$, which proves that $\boldsymbol{x} = \boldsymbol{g}$. Thus $\boldsymbol{g}$ is extreme ot type t in $\mathcal{C}$. □

In a tropical cone, we claim that the minimality of type t of an element holds if only if it holds in a neighborhood of this element. This is a consequence of the tropical convexity of the cone. This allows to reduce the extremality of an element to a local property:

Proposition 3.4. *Given a tropical cone $\mathcal{C} \subset \mathbb{R}^d_{\max}$, $\boldsymbol{g}$ is extreme of type t if and only if there exists a neighborhood N of $\boldsymbol{g}$ such that $\boldsymbol{g}$ is minimal of type t in the set $\mathcal{C} \cap N$.*

Proof. If such a neighborhood N exists, let us consider $\boldsymbol{x} \in \mathcal{C}$ such that $\boldsymbol{x} \leq \boldsymbol{g}$ and $\boldsymbol{x}_t = \boldsymbol{g}_t$. Suppose that $\boldsymbol{x}$ is distinct from $\boldsymbol{g}$. Then any element of the form $\boldsymbol{y} = \boldsymbol{x} \oplus \alpha\boldsymbol{g}$ with $\alpha < 0$ also satisfies $\boldsymbol{y} \leq \boldsymbol{g}$, $\boldsymbol{y}_t = \boldsymbol{g}_t$, and $\boldsymbol{y} \neq \boldsymbol{g}$. Now, for α enough close to $\boldsymbol{g}$, $\boldsymbol{y}$ belongs to N, which contradicts the extremality of $\boldsymbol{g}$.

The converse is straightforward. □

Back to Carathéodory theorem. In this paragraph, we make a digression in order to show how to prove the tropical analogue of Carathéodory theorem for closed cones and convex sets using the characterization of extreme elements provided by Proposition 3.1.

Proof of Theorem 2.2. Consider $\boldsymbol{u} \in \mathcal{C}$, and for all $k \in [d]$, let $\mathcal{C}_k \overset{def}{=} \{\, \boldsymbol{x} \in \mathcal{C} \mid \boldsymbol{x} \leq \boldsymbol{u} \text{ and } \boldsymbol{x}_k = \boldsymbol{u}_k \,\}$. As a closed and bounded set, $\mathcal{C}_k$ is compact, and since it is non-empty, it admits a minimal element $\boldsymbol{g}^k \in \mathcal{C}$. Each $\boldsymbol{g}^k$ is extreme of type k, as any $\boldsymbol{y} \in \mathcal{C}$ satisfying $\boldsymbol{y} \leq \boldsymbol{g}^k$ and $\boldsymbol{y}_k = \boldsymbol{g}^k_k$ belongs to $\mathcal{C}_k$, so that $\boldsymbol{y} = \boldsymbol{g}^k$ by minimality of $\boldsymbol{g}^k$. Besides, $\boldsymbol{u} = \bigoplus_{i=1}^{d} \boldsymbol{g}^k$, which proves that $\boldsymbol{u}$ can be expressed as the sum of d extreme generators. □

Example 3.2. Figure 3.2 provides an illustration of the proof of Theorem 2.2 when $\mathcal{C}$ is the cone defined in Example 2.10 and $\boldsymbol{u} = (2, 3, 1)$. The sets $\mathcal{C}_x$, $\mathcal{C}_y$, and $\mathcal{C}_z$ are the parts of $\mathcal{C}$ overlapping the areas in light red, blue, and green respectively. The minimal elements of these sets (depicted in the same color) are extreme elements of the cone $\mathcal{C}$.

The generalization to closed convex sets follows by homogenization.

Proof of Theorem 2.1. If $\boldsymbol{x} \in \mathcal{C}$, then $(\boldsymbol{x}, \mathbb{1})$ is an element of $\widehat{\mathcal{C}}$, hence, according to Theorem 2.2, there exists $(d+1)$ extreme elements $(\boldsymbol{g}_1, \alpha_1), \ldots, (\boldsymbol{g}_{d+1}, \alpha_{d+1})$ of $\widehat{\mathcal{C}}$ such that $\boldsymbol{x} = \bigoplus_{i=1}^{d+1} \boldsymbol{g}_i$. Up to writting $\boldsymbol{x}$ under the form $\bigoplus_{i=1}^{d+1} \lambda_i \boldsymbol{g}_i$ for some scalar $\lambda_i \in \mathbb{R}_{\max}$, it can be supposed that either $\alpha_i = \mathbb{0}$ or $\alpha_i = \mathbb{1}$.

Assume, without loss of generality, that $\alpha_1 = \ldots = \alpha_p = \mathbb{1}$ and $\alpha_{p+1} = \ldots = \alpha_{d+1} = \mathbb{0}$. Then $\bigoplus_{i=1}^{p} \lambda_i = \mathbb{1}$, and using Proposition 2.2, $\boldsymbol{g}_i \in \mathsf{extp}(\mathcal{C})$ if $1 \leq i \leq p$, while $\boldsymbol{g}_i \in \mathsf{extg}(\mathsf{rec}(\mathcal{C}))$ if $p+1 \leq i \leq d+1$. Defining $q = d+1-p$, it follows that $\boldsymbol{x}$ can be written as the sum of the convex combinations of the p extreme points $\boldsymbol{g}_1, \ldots, \boldsymbol{g}_p$ of $\mathcal{C}$, and the q extreme elements $\boldsymbol{g}_{d-q+2}, \ldots, \boldsymbol{g}_{d+1}$ of $\mathsf{rec}(\mathcal{C})$. □

3.2 Characterizing extremality using the tangent cone

In the previous section, we have seen that extremality can be determined locally (Proposition 3.4). In Section 3.2.1, we use this property to establish a reduction from the extremality of a vector $\boldsymbol{g}$ in a polyhedral cone $\mathcal{C}$, to the extremality of an associated element in the tangent cone to $\mathcal{C}$ at the element $\boldsymbol{g}$ (Theorem 3.1). This tangent cone is indeed a polyhedral cone which provides a local description of $\mathcal{C}$ in the neighborhood of $\boldsymbol{g}$ (Proposition 3.6).

On top of that, the tangent cone is very particular, because it is defined by inequalities whose coefficients are all equal to $\mathbb{0}$ and $\mathbb{1}$. In Section 3.2.2, we will see that such cones are in fact the tropical analogues of the 0/1-polytopes (Proposition 3.8), in the sense that they are generated by elements of $\{\, \mathbb{0}, \mathbb{1} \,\}^d$. We establish a simple combinatorial criterion of extremality for this class of cones (Proposition 3.10), and derive the corresponding characterization in the general case (Theorem 3.2).

3.2.1 Tangent cone

When an inequality $\boldsymbol{ax} \leq \boldsymbol{bx}$ satisfies $\boldsymbol{ag} = \boldsymbol{bg}$, it is said to be *saturated on* $\boldsymbol{g}$. The tangent cone of a cone $\mathcal{C}$ at an element $\boldsymbol{g}$ is essentially defined by the inequalities of the system defining $\mathcal{C}$ which are saturated on $\boldsymbol{g}$, on top of the implicit saturated inequalities $\boldsymbol{x}_l = \mathbb{0}$ induced by the support of $\boldsymbol{g}$:

Definition 3.2. Let $\mathcal{C} = \{ \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \leq B\boldsymbol{x} \}$, and $\boldsymbol{g} \in \mathcal{C}$. The *tangent cone* to $\mathcal{C}$ at the element $\boldsymbol{g}$ is the tropical polyhedral cone $\mathcal{T}(\boldsymbol{g}, \mathcal{C})$ defined by the following intersection of halfspaces:

$$\mathcal{T}(\boldsymbol{g}, \mathcal{C}) = \bigcap_{\substack{1 \leq k \leq p \\ A_k\boldsymbol{g} = B_k\boldsymbol{g} > \mathbb{0}}} \Big\{ \boldsymbol{u} \in \mathbb{R}^d_{\max} \,\Big|\, \bigoplus_{i \in \arg\max(A_k\boldsymbol{g})} \boldsymbol{u}_i \;\leq\; \bigoplus_{j \in \arg\max(B_k\boldsymbol{g})} \boldsymbol{u}_j \Big\} \cap \bigcap_{l \notin \mathsf{supp}(\boldsymbol{g})} \{ \boldsymbol{u} \in \mathbb{R}^d_{\max} \mid \boldsymbol{u}_l = \mathbb{0} \},$$

where for any $\boldsymbol{c} = (\boldsymbol{c}_i) \in \mathbb{R}^{1\times d}_{\max}$, $\arg\max(\boldsymbol{cg})$ is defined as the argument of the maximum $\max_{1 \leq i \leq d}(\boldsymbol{c}_i + \boldsymbol{g}_i)$.

The fact that the tangent cone is a local representation of the initial cone comes from a differentiation property of *tropical linear forms*, which are functions of the form $\boldsymbol{x} \mapsto \boldsymbol{cx}$ for some row vector $\boldsymbol{c} \in \mathbb{R}^{1\times d}_{\max}$:

Lemma 3.5 (Differenciation of tropical linear forms)**.** *Let $\boldsymbol{c} = (\boldsymbol{c}_i) \in \mathbb{R}^{1\times d}_{\max}$ be a tropical linear form, and $\boldsymbol{x} = (\boldsymbol{x}_i) \in \mathbb{R}^d_{\max}$ such that $\boldsymbol{cx} > \mathbb{0}$. Then there exists a neighborhood $D(\boldsymbol{c})$ of $\boldsymbol{x}$ such that for all $\boldsymbol{y} = (\boldsymbol{y}_i) \in D(\boldsymbol{c})$,*

$$\boldsymbol{cy} = \boldsymbol{cx} + \bigoplus_{i \in \arg\max(\boldsymbol{cx})} (\boldsymbol{y}_i - \boldsymbol{x}_i),$$

with the convention $-\infty + \infty = -\infty$.

Proof. For all $i \in \arg\max(\boldsymbol{cx})$, $\boldsymbol{c}_i + \boldsymbol{x}_i > \boldsymbol{c}_j + \boldsymbol{x}_j$ as soon as $j \notin \arg\max(\boldsymbol{cx})$. Then let N be a neighborhood such that $\boldsymbol{c}_i + \boldsymbol{y}_i > \boldsymbol{c}_j + \boldsymbol{y}_j$ for any element $\boldsymbol{y} \in N$ and any pair $(i, j) \in \arg\max(\boldsymbol{cx}) \times ([d] \setminus \arg\max(\boldsymbol{cx}))$. It follows that for all $\boldsymbol{y} \in N$,

$$\begin{aligned}
\boldsymbol{cy} &= \max_{1 \leq i \leq d} (\boldsymbol{c}_i + \boldsymbol{y}_i) \\
&= \max_{i \in \arg\max(\boldsymbol{cx})} (\boldsymbol{c}_i + \boldsymbol{y}_i) \\
&= \max_{i \in \arg\max(\boldsymbol{cx})} (\boldsymbol{c}_i + \boldsymbol{x}_i + \boldsymbol{y}_i - \boldsymbol{x}_i) \quad (\text{as } \boldsymbol{cx} > \mathbb{0} \text{ ensures } \boldsymbol{x}_i > \mathbb{0} \text{ for all } i \in \arg\max(\boldsymbol{cx})) \\
&= \max_{i \in \arg\max(\boldsymbol{cx})} (\boldsymbol{cx} + \boldsymbol{y}_i - \boldsymbol{x}_i) \\
&= \boldsymbol{cx} + \max_{i \in \arg\max(\boldsymbol{cx})} (\boldsymbol{y}_i - \boldsymbol{x}_i).
\end{aligned}$$

□

Proposition 3.6. *Let $\mathcal{C} = \{ \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \leq B\boldsymbol{x} \}$. Then there exists a neighborhood N of $\boldsymbol{g}$ such that for all $\boldsymbol{x} \in N$, $\boldsymbol{x}$ belongs to $\{ \boldsymbol{y} \in \mathcal{C} \mid \mathsf{supp}(\boldsymbol{y}) \subset \mathsf{supp}(\boldsymbol{g}) \}$ if and only if $\boldsymbol{x}$ is an element of $\boldsymbol{g} + \mathcal{T}(\boldsymbol{g}, \mathcal{C})$.*

Proof. Consider a neighborhood N of $\boldsymbol{g}$ defined by the elements $\boldsymbol{x}$ such that:

(i) $A_k\boldsymbol{x} < B_k\boldsymbol{x}$ for all k such that $A_k\boldsymbol{g} < B_k\boldsymbol{g}$,

(ii) $\boldsymbol{x} \in D(A_k) \cap D(B_k)$ for every k satisfying $A_k\boldsymbol{g} = B_k\boldsymbol{g} > \mathbb{0}$.

Let $\boldsymbol{x} \in N$.

First suppose that $\boldsymbol{x} \in \mathcal{C}$ and $\mathsf{supp}(\boldsymbol{x}) \subset \mathsf{supp}(\boldsymbol{g})$. Let us consider $\boldsymbol{u} = (\boldsymbol{u}_i)$ defined by $\boldsymbol{u}_i = \boldsymbol{x}_i - \boldsymbol{g}_i$ for all i. Then $\mathsf{supp}(\boldsymbol{u}) \subset \mathsf{supp}(\boldsymbol{x})$, hence for all $l \notin \mathsf{supp}(\boldsymbol{g})$, $\boldsymbol{u}_l = \mathbb{0}$. Besides, for every k such that $A_k\boldsymbol{g} = B_k\boldsymbol{g} > \mathbb{0}$, the inequality $A_k\boldsymbol{x} \le B_k\boldsymbol{x}$ implies:

$$A_k\boldsymbol{g} + \bigoplus_{i \in \arg\max(A_k\boldsymbol{g})} \boldsymbol{u}_i \le B_k\boldsymbol{g} + \bigoplus_{j \in \arg\max(B_k\boldsymbol{g})} \boldsymbol{u}_j \quad \text{by definition of } N \text{ and Lemma 3.5}$$

so that by eliminating $A_k\boldsymbol{g}$ and $B_k\boldsymbol{g}$,

$$\bigoplus_{i \in \arg\max(A_k\boldsymbol{g})} \boldsymbol{u}_i \le \bigoplus_{j \in \arg\max(B_k\boldsymbol{g})} \boldsymbol{u}_j.$$

Finally, $\boldsymbol{x}_i = \boldsymbol{g}_i + \boldsymbol{u}_i$ clearly holds when $i \in \mathsf{supp}(\boldsymbol{g})$. If $i \notin \mathsf{supp}(\boldsymbol{g})$, then $\boldsymbol{g}_i + \boldsymbol{u}_i = \mathbb{0} = \boldsymbol{x}_i$ since $\mathsf{supp}(\boldsymbol{x}) \subset \mathsf{supp}(\boldsymbol{g})$. This proves that $\boldsymbol{x} = \boldsymbol{g} + \boldsymbol{u}$.

Conversely, given $\boldsymbol{u} \in \mathcal{T}(\boldsymbol{g}, \mathcal{C})$ such that $\boldsymbol{x} = \boldsymbol{g} + \boldsymbol{u}$, we have $\mathsf{supp}(\boldsymbol{u}) \subset \mathsf{supp}(\boldsymbol{g})$ by definition of $\mathcal{T}(\boldsymbol{g}, \mathcal{C})$ so that $\mathsf{supp}(\boldsymbol{x})$ is also included into $\mathsf{supp}(\boldsymbol{g})$. Besides, for all $i \in \mathsf{supp}(\boldsymbol{g}_i)$, $\boldsymbol{u}_i = \boldsymbol{x}_i - \boldsymbol{g}_i$, while if $i \notin \mathsf{supp}(\boldsymbol{g}_i)$, $\boldsymbol{u}_i = \mathbb{0} = \boldsymbol{x}_i - \boldsymbol{g}_i$ with the convention $-\infty + \infty = -\infty$. Clearly, for all k such that $A_k\boldsymbol{g} < B_k\boldsymbol{g}$, we also have $A_k\boldsymbol{x} < B_k\boldsymbol{x}$ by definition of N. Now if $A_k\boldsymbol{g} = B_k\boldsymbol{g} > \mathbb{0}$, then

$$\bigoplus_{i \in \arg\max(A_k\boldsymbol{g})} \boldsymbol{u}_i \le \bigoplus_{j \in \arg\max(B_k\boldsymbol{g})} \boldsymbol{u}_i$$

hence

$$A_k\boldsymbol{g} + \bigoplus_{i \in \arg\max(A_k\boldsymbol{g})} (\boldsymbol{x}_i - \boldsymbol{g}_i) \le B_k\boldsymbol{g} + \bigoplus_{j \in \arg\max(B_k\boldsymbol{g})} (\boldsymbol{x}_j - \boldsymbol{g}_j)$$

and by Lemma 3.5, we obtain $A_k\boldsymbol{x} \le B_k\boldsymbol{x}$. Finally, if $A_k\boldsymbol{g} = B_k\boldsymbol{g} = \mathbb{0}$, let $I = \{\, i \mid a_{ki} > \mathbb{0} \,\}$ and $J = \{\, i \mid b_{ki} > \mathbb{0} \,\}$. Necessarily, $I \cup J$ and $\mathsf{supp}(\boldsymbol{g})$ have an empty intersection. Since $\boldsymbol{x}$ has a smaller support than $\boldsymbol{g}$, we conclude that $(I \cup J) \cap \mathsf{supp}(\boldsymbol{x}) = \emptyset$, so that $A_k\boldsymbol{x} = B_k\boldsymbol{x} = \mathbb{0}$. □

Given $I \subset [d]$, we denote by $\boldsymbol{\varepsilon}_I$ the element of $\mathbb{R}^d_{\max}$ whose i-th coordinate is equal to $\mathbb{1}$ if $i \in I$, and $\mathbb{0}$ otherwise. We are going to show that the tangent cone $\mathcal{T}(\boldsymbol{g}, \mathcal{C})$ contains the element $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$:

Lemma 3.7. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a polyhedral cone, and $\boldsymbol{g} \in \mathcal{C}$. Then $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$ belongs to $\mathcal{T}(\boldsymbol{g}, \mathcal{C})$.*

Proof. Let us first show that $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$ indeed belongs to $\mathcal{T}(\boldsymbol{g}, \mathcal{C})$. By Proposition 3.6, $\boldsymbol{g}$ belongs to $\boldsymbol{g} + \mathcal{T}(\boldsymbol{g}, \mathcal{C})$, so that there exists $\boldsymbol{u} \in \mathcal{T}(\boldsymbol{g}, \mathcal{C})$ such that $\boldsymbol{g} = \boldsymbol{g} + \boldsymbol{u}$. Clearly, for any $i \in \mathsf{supp}(\boldsymbol{g})$, $\boldsymbol{u}_i = \mathbb{1}$. Besides, $\mathsf{supp}(\boldsymbol{u}) \subset \mathsf{supp}(\boldsymbol{g})$ by definition of $\mathcal{T}(\boldsymbol{g}, \mathcal{C})$, so that $\boldsymbol{u} = \boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$. □

We can now prove that the extremality of an element $\boldsymbol{g}$ in a cone $\mathcal{C}$ can be reduced to the extremality of $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$ of the tangent cone at $\boldsymbol{g}$:

Theorem 3.1. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical polyhedral cone. Then $\boldsymbol{g}$ is extreme in $\mathcal{C}$ if and only if $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$ is extreme in $\mathcal{T}(\boldsymbol{g}, \mathcal{C})$.*

In that case, both elements have the same type(s) of extremality in the sense of Proposition 3.1.

Proof. According to Proposition 3.3, $\boldsymbol{g}$ is extreme of type t in $\mathcal{C}$ if and only if it is extreme of type t in $\{\, \boldsymbol{x} \in \mathcal{C} \mid \mathsf{supp}(\boldsymbol{x}) \subset \mathsf{supp}(\boldsymbol{g}) \,\}$.

Let N be a neighborhood such that Proposition 3.6 holds. Using Proposition 3.4, if follows that $\boldsymbol{g}$ is extreme of type t in $\mathcal{C}$ if and only if it is extreme of type t in $\boldsymbol{g} + \mathcal{T}(\boldsymbol{g}, \mathcal{C})$.

First observe that $\boldsymbol{g} \in \boldsymbol{g} + \mathcal{T}(\boldsymbol{g}, \mathcal{C})$ is equivalent to $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})} \in \mathcal{T}(\boldsymbol{g}, \mathcal{C})$ by Lemma 3.7.

Suppose that $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$ is extreme of type t in $\mathcal{T}(\boldsymbol{g}, \mathcal{C})$. Let $\boldsymbol{x} \in \boldsymbol{g} + \mathcal{T}(\boldsymbol{g}, \mathcal{C})$ such that $\boldsymbol{x} \leq \boldsymbol{g}$ and $\boldsymbol{x}_t = \boldsymbol{g}_t$. Let $\boldsymbol{u} \in \mathcal{T}(\boldsymbol{g}, \mathcal{C})$ verifying $\boldsymbol{x} = \boldsymbol{g} + \boldsymbol{u}$. Clearly, $\mathsf{supp}(\boldsymbol{u}) \subset \mathsf{supp}(\boldsymbol{g})$ by definition of $\mathcal{T}(\boldsymbol{g}, \mathcal{C})$, and for all $i \in \mathsf{supp}(\boldsymbol{g})$, $\boldsymbol{u}_i \leq \mathbb{1}$, which implies $\boldsymbol{u} \leq \boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$. Besides, $t \in \mathsf{supp}(\boldsymbol{g})$ by Lemma 3.2, so that $\boldsymbol{u}_t = \mathbb{1}$. It follows that $\boldsymbol{u} = \boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$, hence $\boldsymbol{x} = \boldsymbol{g}$. This shows that $\boldsymbol{g}$ is extreme of type t in $\boldsymbol{g} + \mathcal{T}(\boldsymbol{g}, \mathcal{C})$.

Conversely, suppose that $\boldsymbol{g}$ is extreme of type t in $\boldsymbol{g} + \mathcal{T}(\boldsymbol{g}, \mathcal{C})$. Let $\boldsymbol{u} \in \mathcal{T}(\boldsymbol{g}, \mathcal{C})$ verifying $\boldsymbol{u} \leq \boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$ and $\boldsymbol{u}_t = (\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})})_t = \mathbb{1}$ (by Lemma 3.2). Let $\boldsymbol{x} = \boldsymbol{g} + \boldsymbol{u}$. Clearly $\boldsymbol{x} \leq \boldsymbol{g}$ and $\boldsymbol{x}_t = \boldsymbol{g}_t$, so that $\boldsymbol{x} = \boldsymbol{g}$. Since $\mathsf{supp}(\boldsymbol{u}) \subset \mathsf{supp}(\boldsymbol{g})$, we have $\boldsymbol{u} = \boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$. It follows that $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$ is extreme of type t in $\mathcal{T}(\boldsymbol{g}, \mathcal{C})$. □

Example 3.3. Continuing Example 2.10, let us illustrate Theorem 3.1 on the element $\boldsymbol{g}^2 = (2, 2, 0)$. In (3.1), the inequalities of the system given in (2.10) which are saturated by $\boldsymbol{g}^2$ are colored in red, and the terms which belong to the arguments of the members of saturated inequalities are underlined. It directly provides a system of inequalities defining the cone $\mathcal{T}(\boldsymbol{g}^2, \mathcal{C})$, in (3.2).

$$\begin{cases} z \leq x + 2 \\ \underline{x} \leq \max(\underline{y}, z) \\ \underline{x} \leq \underline{z + 2} \\ z \leq \max(x, y - 1) \end{cases} \quad (3.1) \qquad \begin{cases} x \leq y \\ x \leq z \end{cases} \quad (3.2)$$

Figure 3.3 illustrates that the cones $\mathcal{C}$ and $\boldsymbol{g}^2 + \mathcal{T}(\boldsymbol{g}^2, \mathcal{C})$ locally coincide in a neighborhood of $\boldsymbol{g}^2$. Figure 3.4 provides an illustration of the extremality of type x of $\mathbb{1} = \boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g}^2)}$ in the cone $\mathcal{T}(\boldsymbol{g}^2, \mathcal{C})$: the light green area, which depicts the set of elements (x, y, z) such that $(x, y, z) \leq \mathbb{1}$ implies $x < \mathbb{1}$, contains the whole cone except $\mathbb{1}$.

3.2.2 The $\{\, \mathbb{0}, \mathbb{1} \,\}$-cones and their extreme elements

The tangent cone $\mathcal{T}(\boldsymbol{g}, \mathcal{C})$ belongs to a particular class of polyhedral cones, which we call $\{\, \mathbb{0}, \mathbb{1} \,\}$-*cones*, because they are defined by system of inequalities with coefficients in $\{\, \mathbb{0}, \mathbb{1} \,\}$:

Definition 3.3. Let $\mathcal{C} \subset \mathbb{R}_{\max}^d$. The set $\mathcal{C}$ is said to be a $\{\, \mathbb{0}, \mathbb{1} \,\}$-*cone* if it can be expressed as the set of the solutions of $A\boldsymbol{x} \leq B\boldsymbol{x}$ with $A, B \in \{\, \mathbb{0}, \mathbb{1} \,\}^{d \times p}$ $(p \geq 0)$.

An interesting property of such cones is that they are the tropical analogues of 0/1-*polytopes*. A 0/1-polytope is defined as the convex hull of points of the regular cube $\{\, 0, 1 \,\}^d$ (see [Zie00] for a survey). Here, we show that $\{\, \mathbb{0}, \mathbb{1} \,\}$-cones are indeed generated by elements of $\{\, \mathbb{0}, \mathbb{1} \,\}^d$. Observe that this property is specific to the tropical case, since in the classical case, there is no reason that a 0/1-polytope be defined by a system of 0/1-inequalities.

Proposition 3.8. *Let $\mathcal{C} \subset \mathbb{R}_{\max}^d$. Then $\mathcal{C}$ is a $\{\, \mathbb{0}, \mathbb{1} \,\}$-cone if and only if it is of the form* $\mathsf{cone}(G)$, *with $G \subset \{\, \mathbb{0}, \mathbb{1} \,\}^d$.*

In particular, every scaled extreme elements of a $\{\, \mathbb{0}, \mathbb{1} \,\}$-cone belongs to $\{\, \mathbb{0}, \mathbb{1} \,\}^d$.

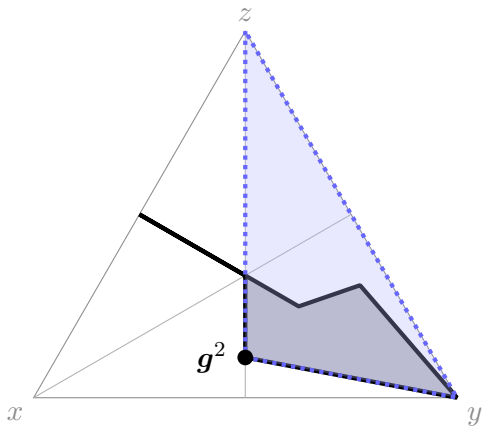


Figure 3.3: The set $\boldsymbol{g^2} + \mathcal{T}(\boldsymbol{g}^2, \mathcal{C})$ (in light blue)

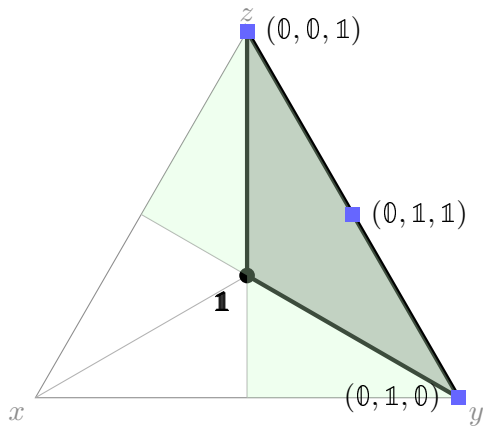


Figure 3.4: Extremality of $\mathbf{1}$ in the cone $\mathcal{T}(\boldsymbol{g}^2, \mathcal{C})$

Proof. Let us first prove the "only if" part. Let $A, B \in \{\mathbb{0}, \mathbb{1}\}^{d \times p}$ such that $\mathcal{C} = \{\boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \leq B\boldsymbol{x}\}$. We are going to show that any scaled extreme element of $\mathcal{C}$ belongs to $\{\mathbb{0}, \mathbb{1}\}^d$.

Consider a scaled extreme element $\boldsymbol{g} = (\boldsymbol{g}_i)$ of type t in $\mathcal{C}$. Let $I = \{i \mid \boldsymbol{g}_i \leq \boldsymbol{g}_t\}$ and $J = \{j \mid \boldsymbol{g}_i > \boldsymbol{g}_t\}$. Let $\alpha > 0$ such that $\boldsymbol{g}_j \geq \boldsymbol{g}_t + \alpha$ for all $j \in J$. Observe that given $\boldsymbol{c} \in \{\mathbb{0}, \mathbb{1}\}^{1 \times d}$, if $\boldsymbol{cg} > \mathbb{0}$, then it is equal to one of the $\boldsymbol{g}_k$, so that $\arg(\boldsymbol{cg})$ is contained either into I or into J. In the former case, $\boldsymbol{cg} \leq \boldsymbol{g}_t$, while in the latter case, $\boldsymbol{cg} > \boldsymbol{g}_t + \alpha$. Let $\boldsymbol{x} = (\boldsymbol{x}_i)$ be defined by $\boldsymbol{x}_i = \boldsymbol{g}_i$ for all $i \in I$, and $\boldsymbol{x}_j = \boldsymbol{g}_j - \alpha$. Since $\mathsf{supp}(\boldsymbol{x}) = \mathsf{supp}(\boldsymbol{g})$, $\boldsymbol{cx} > \mathbb{0}$ if and only if $\boldsymbol{cg} > \mathbb{0}$, and in that case, the following relation holds:

$$\boldsymbol{cx} = \begin{cases} \boldsymbol{cg} & \text{if } \arg(\boldsymbol{cg}) \subset I, \text{ or} \\ \boldsymbol{cg} - \alpha & \text{otherwise, } \textit{i.e.} \ \arg(\boldsymbol{cg}) \subset J. \end{cases} \tag{3.3}$$

We claim that for all k, $A_k\boldsymbol{x} \leq B_k\boldsymbol{x}$. First suppose that $A_k\boldsymbol{g} > \mathbb{0}$. Since $A_k\boldsymbol{g} \leq B_k\boldsymbol{g}$, exactly one of three conditions is satisfied: (i) $\arg(A_k\boldsymbol{g}), \arg(B_k\boldsymbol{g}) \subset J$, (ii) $\arg(A_k\boldsymbol{g}) \subset I$ and $\arg(B_k\boldsymbol{g}) \subset J$, in which case we even have $A_k\boldsymbol{g} + \alpha \leq \alpha + \boldsymbol{g}_t \leq B_k\boldsymbol{g}$, (iii) or $\arg(A_k\boldsymbol{g}), \arg(B_k\boldsymbol{g}) \subset I$. Applying the formula given in 3.3 to the tropical forms A_k and B_k in that three cases allows to show that $A_k\boldsymbol{x} \leq B_k\boldsymbol{x}$ holds. Now, if $A_k\boldsymbol{g} = \mathbb{0}$, then $A_k\boldsymbol{x} = \mathbb{0}$ as mentionned above, so that $A_k\boldsymbol{x} \leq B_k\boldsymbol{x}$ is trivially satisfied.

Consequently, the set J has to be empty, otherwise the element $\boldsymbol{x} \in \mathcal{C}$ contradicts the minimality of type t of $\boldsymbol{g}$. This means that for all $i \in [d]$, $\boldsymbol{g}_i \leq \boldsymbol{g}_t$, and since $\boldsymbol{g}$ is scaled, then $\boldsymbol{g}_t = \mathbb{1}$. Now suppose that there exists $i \in [d]$ such that $\mathbb{0} < \boldsymbol{g}_i < \mathbb{1}$. The element $\boldsymbol{y} = 2 \times \boldsymbol{g}$ obviously belong to $\mathcal{C}$, and satisfies $\boldsymbol{y}_t = \mathbb{1}$, $\boldsymbol{y} \leq \boldsymbol{g}$, and $\boldsymbol{y}_i < \boldsymbol{g}_i$, which is impossible under our assumptions. As a result, for all $i \in [d]$, either $\boldsymbol{g}_i = \mathbb{0}$ or $\boldsymbol{g}_i = \mathbb{1}$, which completes the proof.

Proving the "if" part requires some considerations on the polar of a tropical cone, which will be developed in Chapter 5. Indeed, Corollary 5.11 proves that a cone of the form $\mathsf{cone}(G)$ with $G \subset \{\mathbb{0}, \mathbb{1}\}^d$ is given by inequalities $\boldsymbol{ax} \leq \boldsymbol{bx}$, such that, according to Proposition 5.10, each ${}^t\begin{pmatrix}\boldsymbol{a} & \boldsymbol{b}\end{pmatrix}$ is a scaled extreme element of a $\{\mathbb{0}, \mathbb{1}\}$-cone of $(\mathbb{R}^d_{\max})^2$. The coefficients of the system given in (5.6) are all in $\{\mathbb{0}, \mathbb{1}\}$, since $G \subset \{\mathbb{0}, \mathbb{1}\}^d$. Therefore, using the "only if" part, we have $\boldsymbol{a}, \boldsymbol{b} \in \{\mathbb{0}, \mathbb{1}\}^d$, which shows that $\mathsf{cone}(G)$ is a $\{\mathbb{0}, \mathbb{1}\}$-cone. □

Remark 3.4. Proposition 3.8 implies that the number of extreme rays in a $\{\mathbb{0}, \mathbb{1}\}$-cone of $\mathbb{R}^d_{\max}$ is bounded by $2^d - 1$. The general case will be discussed in Section 5.3.

The following technical lemma is a rewritting of Lemma 3.2 in the context of $\{\mathbb{0}, \mathbb{1}\}$-cones, in the light of Proposition 3.8:

Lemma 3.9. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a $\{\mathbb{0}, \mathbb{1}\}$-cone. If $\boldsymbol{g} \in \mathbb{R}^d_{\max}$ is a scaled extreme element of type t in $\mathcal{C}$, then $\boldsymbol{g}_t = \mathbb{1}$.*

Proposition 3.8 leads to a first combinatorial characterization of extreme elements in $\{\mathbb{0}, \mathbb{1}\}$-cones. It states that the extremality in such cones is *entirely* determined by elements in $\{\mathbb{0}, \mathbb{1}\}^d$:

Proposition 3.10. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a $\{\mathbb{0}, \mathbb{1}\}$-cone, and $\boldsymbol{g} \in \mathcal{C}$. Then the three following statements are equivalent:*

(i) $\boldsymbol{g}$ is a scaled extreme element of $\mathcal{C}$ of type t.

(ii) for all scaled extreme element $\boldsymbol{h}$ of $\mathcal{C}$, $\boldsymbol{h} \leq \boldsymbol{g} \implies \boldsymbol{h}_t = \mathbb{0}$ or $\boldsymbol{h} = \boldsymbol{g}$.

(iii) for all element $\boldsymbol{x} \in \mathcal{C} \cap \{\mathbb{0}, \mathbb{1}\}^d$, $\boldsymbol{x} \leq \boldsymbol{g} \implies \boldsymbol{x}_t = \mathbb{0}$ or $\boldsymbol{x} = \boldsymbol{g}$.

Proof. (i) $\implies$ (iii) Lemma 3.9 implies that $\boldsymbol{g}_t = \mathbb{1}$. Therefore, considering an element $\boldsymbol{x} \in \mathcal{C} \cap \{\mathbb{0}, \mathbb{1}\}^d$ such that $\boldsymbol{x} \leq \boldsymbol{g}$ and $\boldsymbol{x} \neq \boldsymbol{g}$, $\boldsymbol{x}_t$ is necessarily equal to $\mathbb{0}$, otherwise it would contradict the minimality of type t of $\boldsymbol{g}$.

(iii) $\implies$ (ii) It is straighforward from Proposition 3.8.

(ii) $\implies$ (i) Consider $\boldsymbol{x} \in \mathcal{C}$ such that $\boldsymbol{x} \leq \boldsymbol{g}$ and $\boldsymbol{x}_t = \boldsymbol{g}_t$. If $G = (\boldsymbol{g}^i)_{1 \leq i \leq n}$ is the set formed by scaled extreme elements of $\mathcal{C}$, then by Theorem 2.3, there exists $\lambda_1, \dots, \lambda_n \in \mathbb{R}_{\max}$ such that $\boldsymbol{x} = \bigoplus_{i=1}^n \lambda_i \boldsymbol{g}^i$. In particular, there exists $1 \leq i \leq n$ such that $\boldsymbol{x}_t = \lambda_i \boldsymbol{g}^i_t$. Since $\boldsymbol{x}_t = \boldsymbol{g}_t = \mathbb{1}$ (using Lemma 3.9), $\boldsymbol{g}^i_t \neq \mathbb{0}$, thus $\boldsymbol{g}^i_t = \mathbb{1}$ by Proposition 3.8. Therefore, $\lambda_i = \mathbb{1}$, so that $\boldsymbol{g}^i \leq \boldsymbol{x} \leq \boldsymbol{g}$. It follows that $\boldsymbol{g}^i = \boldsymbol{g}$, which proves $\boldsymbol{x} = \boldsymbol{g}$. □

Instantiating Proposition 3.10 with tangent cones provides a combinatorial characterization of the extremality in polyhedral cones in the general case:

Theorem 3.2. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical polyhedral cone. Then $\boldsymbol{g}$ is extreme of type t in $\mathcal{C}$ if and only if $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$ is the unique element $\mathcal{T}(\boldsymbol{g}, \mathcal{C}) \cap \{\mathbb{0}, \mathbb{1}\}^d$ whose t-th coordinate is equal to $\mathbb{1}$.*

Proof. It is a straightforward consequence of Theorem 3.1, Proposition 3.10, and the fact that all elements $\boldsymbol{x} \in \mathcal{T}(\boldsymbol{g}, \mathcal{C}) \cap \{\mathbb{0}, \mathbb{1}\}^d$ satisfies $\boldsymbol{x} \leq \boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$. □

Example 3.5. Continuing Example 3.3, the three elements of $\mathcal{T}(\boldsymbol{g}^2, \mathcal{C}) \cap \{\mathbb{0}, \mathbb{1}\}^d$ distinct from $\mathbb{1}$ are represented by blue squares in Figure 3.4. They all have $\mathbb{0}$ as first coordinate, which confirms that $\boldsymbol{g}^2$ is extreme of type x.

Corollary 3.11. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical polyhedral cone, and $\boldsymbol{g} \in \mathcal{C}$. Then $\boldsymbol{g}$ is extreme of type t in $\mathcal{C}$ if and only if for every $l \in \mathsf{supp}(\boldsymbol{g})$, the following property holds:*

$$\forall \boldsymbol{x} \in \mathcal{T}(\boldsymbol{g}, \mathcal{C}) \cap \{\mathbb{0}, \mathbb{1}\}^d,\ \boldsymbol{x}_l = \mathbb{0} \implies \boldsymbol{x}_t = \mathbb{0}. \tag{3.4}$$

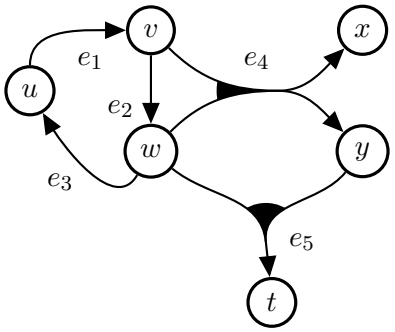


Figure 3.5: A directed hypergraph

Proof. Suppose that $\boldsymbol{g}$ is extreme of type t. By Theorem 3.2, $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$ is the unique element of $\mathcal{T}(\boldsymbol{g},\mathcal{C}) \cap \{\,\mathbb{0},\mathbb{1}\,\}^d$ whose t-th coordinate is $\mathbb{1}$. Let $l \in \mathsf{supp}(\boldsymbol{g})$, and $\boldsymbol{x} \in \mathcal{T}(\boldsymbol{g},\mathcal{C}) \cap \{\,\mathbb{0},\mathbb{1}\,\}^d$ such that $\boldsymbol{x}_l = \mathbb{0}$. Then $\boldsymbol{x} \neq \boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$, so that $\boldsymbol{x}_t = \mathbb{0}$.

Conversely, since $\boldsymbol{g} \in \mathcal{C}$, then $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})} \in \mathcal{T}(\boldsymbol{g},\mathcal{C})$ by Lemma 3.7. Besides, consider $\boldsymbol{x} \in \mathcal{T}(\boldsymbol{g},\mathcal{C}) \cap \{\,\mathbb{0},\mathbb{1}\,\}^d$ such that $\boldsymbol{x}_t = \mathbb{1}$. Then $\mathsf{supp}(\boldsymbol{x}) \subset \mathsf{supp}(\boldsymbol{g})$, and for all $l \in \mathsf{supp}(\boldsymbol{g})$, $\boldsymbol{x}_l = \mathbb{1}$ by (3.4). Thus $\boldsymbol{x} = \boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$. This proves that $\boldsymbol{\varepsilon}_{\mathsf{supp}(\boldsymbol{g})}$ is the unique element of $\mathcal{T}(\boldsymbol{g},\mathcal{C}) \cap \{\,\mathbb{0},\mathbb{1}\,\}^d$ whose t-th coordinate is $\mathbb{1}$, which implies that $\boldsymbol{g}$ is extreme of type t by Theorem 3.2. □

3.3 Characterizing extremality using directed hypergraphs

We are now going to show how the combinatorial extremality criterion based on the $\{\,\mathbb{0},\mathbb{1}\,\}$-elements of the tangent cone can be expressed using directed hypergraphs.

Directed hypergraphs and related notions are introduced in Section 3.3.1. Directed hypergraphs are a generalization of directed graphs, just as hypergraphs for graphs.

In Section 3.3.2, we introduce the notion of *tangent directed hypergraph*, which is an equivalent encoding of the tangent cone as a hypergraph. Corollary 3.11 is then rewritten in terms of the reachability relation of the tangent hypergraph. Theorem 3.3 proves that the extremality reduces to the existence of a particular strongly connected component playing the role of a sink.

3.3.1 Preliminaries on directed hypergraphs

A directed hypergraph is given by a set of nodes and of hyperedges. Hyperedges are extensions of digraph edges: a hyperedge is defined as an arc from a set of nodes to another one.

Definition 3.4. A *directed hypergraph* is a pair (N, E) such that all element $e \in E$ is of the form (T, H) with $T, H \subset N$. The elements of N and E are respectively its *vertices* and *hyperedges*.

Given a hyperedge $e = (T, H) \in E$, the sets T and H represent the *tail* and the *head* of e respectively, and are also denoted by $T(e)$ and $H(e)$.

Example 3.6. Figure 3.5 depicts an example of hypergraph whose nodes are u, v, w, x, y, z, and of hyperedges $e_1 = (\{u\}, \{v\})$, $e_2 = (\{v\}, \{w\})$, $e_3 = (\{w\}, \{u\})$, $e_4 = (\{v, w\}, \{x, y\})$, and $e_5 = (\{w, y\}, \{t\})$.[3]

The notion of reachability can be extended from directed graphs to directed hypergraphs. It is defined recursively: when all the nodes of the tail of a hyperedge e are reachable from a node u, then every node of the head of e is also reachable from u:

Definition 3.5. Let $\mathcal{H} = (N, E)$ be a directed hypergraph, and $u, v \in N$. Then v is said to be *reachable from u in $\mathcal{H}$*, which will be denoted by $u \rightsquigarrow_{\mathcal{H}} v$, if

- $u = v$,
- or there exists a hyperedge e such that $v \in H(e)$ and all the elements of $T(e)$ are reachable from u.

This definition naturally induces a notion of hyperpaths:

Definition 3.6. Let $\mathcal{H} = (N, E)$ be a directed hypergraph, and $u, v \in N$. A *hyperpath from u to v in $\mathcal{H}$* is a sequence of p hyperedges $e_1, \dots, e_p \in E$ satisfying one of the two following conditions:

- $p = 0$ and $u = v$,
- or $p \geq 1$ and

$$\begin{aligned} &T(e_i) \subset \{ u \} \cup H(e_1) \cup \cdots \cup H(e_{i-1}) \quad \text{for all } 1 \leq i \leq p \\ &\{ v \} \subset H(e_p). \end{aligned}$$

The hyperpath is said to be *minimal* if none of its subsequences is a hyperpath from u to v.

Of course, v is reachable from u if and only if there is a hyperpath from u to v.

Example 3.7. Consider the hypergraph depicted in Figure 3.5.

Applying the recursive definition of reachability from u discovers the node v, then w, which leads to the two nodes x and y through the hyperedge e_4, and finally t through e_5.

It can be checked that t is reachable from u through the hyperpath e_1, e_2, e_4, e_5.

3.3.2 Tangent directed hypergraph

The definition of the tangent directed hypergraph derives from the system of inequalities defining the tangent cone (see (3.2)). Its nodes correspond to the coordinates l such that $\boldsymbol{g}_l > \mathbb{0}$, and its hyperedges are induced by the saturated inequalities and their associated arg max:

Definition 3.7. Let $\mathcal{C} = \{ \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \leq B\boldsymbol{x} \}$ be a tropical polyhedral cone ($A = (a_{ij}), B = (b_{ij}) \in \mathbb{R}^{p \times d}_{\max}$), and let $\boldsymbol{g} \in \mathcal{C}$. The *tangent directed hypergraph* at $\boldsymbol{g}$, denoted by $\mathcal{H}(\boldsymbol{g}, \mathcal{C})$, is defined by:

$$\begin{aligned} &N = \mathsf{supp}(\boldsymbol{g}), \\ &E = \{ (\arg\max(B_k \boldsymbol{g}), \arg\max(A_k \boldsymbol{g})) \mid k \in [p] \text{ and } A_k \boldsymbol{g} = B_k \boldsymbol{g} > \mathbb{0} \}. \end{aligned}$$

[3] When a hyperedge leaves several nodes, it is decorated with a black solid disk portion.

Note that for all k such that $A_k\boldsymbol{g} = B_k\boldsymbol{g} > \mathbb{0}$, $\arg\max(A_k\boldsymbol{g})$ and $\arg\max(B_k\boldsymbol{g})$ are necessarily included into $\mathsf{supp}(\boldsymbol{g})$, so that the hypergraph $\mathcal{H}(\boldsymbol{g},\mathcal{C})$ is well defined.

The extremality criterion provided by Corollary 3.11 suggests to evaluate, given an element of $\mathcal{T}(\boldsymbol{g},\mathcal{C})\cap\{\,\mathbb{0},\mathbb{1}\,\}^d$, the effect of setting its l-th coordinate to the other coordinates. Suppose that it has been discovered that $\boldsymbol{u}_l = \mathbb{0}$ implies $\boldsymbol{u}_{j_1} = \ldots = \boldsymbol{u}_{j_n} = \mathbb{0}$. Then for any hyperedge e of $\mathcal{H}(\boldsymbol{g},\mathcal{C})$ such that $T(e) \subset \{\, l, j_1, \ldots, j_n \,\}$, $\boldsymbol{u}$ satisfies:

$$\max_{i\in H(e)} \boldsymbol{u}_i \leq \max_{j\in T(e)} \boldsymbol{u}_j = \mathbb{0},$$

so that $\boldsymbol{u}_i = \mathbb{0}$ for all $i \in H(e)$. Thus, the propagation of the value $\mathbb{0}$ from the l-th coordinate to other coordinates mimicks the inductive definition of the reachability relation from the node l in $\mathcal{H}(\boldsymbol{g},\mathcal{C})$:

Proposition 3.12. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical polyhedral cone, and $\boldsymbol{g} \in \mathcal{C}$. Then for all $l \in \mathsf{supp}(\boldsymbol{g})$, the property given by* (3.4) *holds if and only if t is reachable from l in the tangent hypergraph $\mathcal{H}(\boldsymbol{g},\mathcal{C})$.*

Proof. Let $l \in \mathsf{supp}(\boldsymbol{g})$, and suppose that t is reachable from l in $\mathcal{H}(\boldsymbol{g},\mathcal{C})$. Suppose that $\boldsymbol{u} \in \mathcal{T}(\boldsymbol{g},\mathcal{C}) \cap \{\,\mathbb{0},\mathbb{1}\,\}^d$ such that $\boldsymbol{u}_l = \mathbb{0}$. Let us show by induction on the definition of the reachability in $\mathcal{H}(\boldsymbol{g},\mathcal{C})$ that $\boldsymbol{u}_t = \mathbb{0}$:

- if $t = l$, then obviously $\boldsymbol{u}_t = \mathbb{0}$.
- otherwise, there exists a hyperedge e such that $t \in H(e)$ and for all $j \in T(e)$, j is reachable from l. By induction hypothesis, $\boldsymbol{u}_j = \mathbb{0}$. Let $A_k\boldsymbol{u} \leq B_k\boldsymbol{u}$ be the inequality associated to e. Then $\arg\max(A_k\boldsymbol{g}) = H(e)$ and $\arg\max(B_k\boldsymbol{g}) = T(e)$. Since $\boldsymbol{u}$ satisfies $\bigoplus_{i\in\arg\max(A_k\boldsymbol{g})} \boldsymbol{u}_i \leq \bigoplus_{j\in\arg\max(B_k\boldsymbol{g})} \boldsymbol{u}_j$, we have $\max_{i\in H(e)} \boldsymbol{u}_i \leq \max_{j\in T(e)} \boldsymbol{u}_j = \mathbb{0}$. Hence $\boldsymbol{u}_t = \mathbb{0}$, so that (3.4) holds.

Conversely, suppose t is not reachable from l. Let us define $\boldsymbol{u} = (\boldsymbol{u}_i)$ by $\boldsymbol{u}_i = \mathbb{0}$ if i is reachable from l or $i \notin \mathsf{supp}(\boldsymbol{g})$, and $\mathbb{1}$ otherwise. Let $1 \leq k \leq p$ such that $A_k\boldsymbol{g} = B_k\boldsymbol{g} > \mathbb{0}$, and e the associated hyperedge in $\mathcal{H}(\boldsymbol{g},\mathcal{C})$. Remember that $\arg\max(A_k\boldsymbol{g}) = H(e)$ and $\arg\max(B_k\boldsymbol{g}) = T(e)$. If for all $j \in T(e)$, j is reachable from l, then all elements $i \in H(e)$ are also reachable from l, so that $\bigoplus_{i\in\arg\max(A_k\boldsymbol{g})} \boldsymbol{u}_i = \bigoplus_{j\in\arg\max(B_k\boldsymbol{g})} \boldsymbol{u}_j = \mathbb{0}$. Otherwise, there exists $j \in T(e)$ which is not reachable from l, so that $\bigoplus_{j\in\arg\max(B_k\boldsymbol{g})} \boldsymbol{u}_j = \mathbb{1}$. Since $\bigoplus_{i\in\arg\max(A_k\boldsymbol{g})} \boldsymbol{u}_i$ is always less than or equal to $\mathbb{1}$, the inequality $\bigoplus_{i\in\arg\max(A_k\boldsymbol{g})} \boldsymbol{u}_i \leq \bigoplus_{j\in\arg\max(B_k\boldsymbol{g})} \boldsymbol{u}_i$ holds. It follows that $\mathcal{T}(\boldsymbol{g},\mathcal{C}) \cap \{\,\mathbb{0},\mathbb{1}\,\}^d$ admits an element $\boldsymbol{u}$ such that $\boldsymbol{u}_l = \mathbb{0}$ and $\boldsymbol{u}_t = \mathbb{1}$. □

Thanks to Proposition 3.12, we can now prove that the extremality criterion of Corollary 3.11 holds if and only if the node t is reachable from any other node l in the tangent hypergraph. Such a node is called a *sink*. This sink property also holds for any other node t' belonging to the same strongly connected component of $\mathcal{H}(\boldsymbol{g},\mathcal{C})$. In directed hypergraphs, strongly connected components are defined as follows:

Definition 3.8. Let $\mathcal{H}$ be a directed hypergraph. A *strongly connected component* (Scc for short) of a hypergraph $\mathcal{H}$ is an equivalence class of the relation $\equiv_{\mathcal{H}}$, defined by $u \equiv_{\mathcal{H}} v$ if $u \rightsquigarrow_{\mathcal{H}} v$ and $v \rightsquigarrow_{\mathcal{H}} u$.

Like the SCCs of directed graphs, the SCCs of hypergraphs form a partition of the set of nodes. The reachability relation in a hypergraph induces a partial order over the SCCs, defined as follows:

Proposition-Definition 3.13. *Let $\mathcal{H}$ be a directed hypergraph. Let $\preceq_{\mathcal{H}}$ be the relation over the* SCC*s of $\mathcal{H}$ by $C_1 \preceq_{\mathcal{H}} C_2$ if C_1 and C_2 admit a representative u and v respectively such that $u \rightsquigarrow_{\mathcal{H}} v$.*

Then $\preceq_{\mathcal{H}}$ is a partial order on the set of the SCC*s of $\mathcal{H}$.*

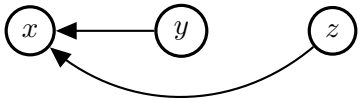
Then we claim that a node of a hypergraph is a sink if and only if the SCC which contains it is the greatest SCC for the order $\preceq_{\mathcal{H}}$. This implies:

Theorem 3.3. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a polyhedral cone, and $\boldsymbol{g} \in \mathcal{C}$. Then $\boldsymbol{g}$ is extreme if and only if the set of the* SCC*s of the hypergraph $\mathcal{H}(\boldsymbol{g}, \mathcal{C})$, partially ordered by $\preceq_{\mathcal{H}(\boldsymbol{g},\mathcal{C})}$, admits a greatest element.*

In that case, the greatest SCC *precisely contains all the elements t such that $\boldsymbol{g}$ is extreme of type t.*

Proof. From Proposition 3.12 and Corollary 3.11, $\boldsymbol{g}$ is extreme of type t if and only if t is reachable from any node l of the hypergraph $\mathcal{H}(\boldsymbol{g}, \mathcal{C})$. This holds if and only if t belongs to a SCC C such that $D \preceq_{\mathcal{H}(\boldsymbol{g},\mathcal{C}} C$ for any SCC D. □

Example 3.8. Consider the tangent directed hypergraph $\mathcal{H}(\boldsymbol{g}^2, \mathcal{C})$ associated to the element $\boldsymbol{g}^2$ of the cone introduced in Example 2.10. Remember that this element is extreme of type x. Following the system of equations defining the tangent cone (see 3.2), the hypergraph is:



Obviously, the node x forms the greatest SCC of the hypergraph.

3.4 Conclusion of the chapter

We have obtained a combinatorial characterization of extremality in polyhedral cones. It is based on the existence of a greatest SCC in a directed hypergraph derived from the description by halfspaces.

After all, our combinatorial criterion seems more complex than its classical analogues presented in the introduction of this chapter. Nevertheless, as discussed in Chapter 4, it can be evaluated by a very efficient algorithm. Therefore, we will see in Chapter 5 that, thanks to our criterion, the characterization of extremality is algorithmically easier in the tropical world than in the classical world.

CHAPTER 4

Determining the maximal strongly connected components in directed hypergraphs

In this chapter, we develop an efficient algorithm which, given a directed hypergraph $\mathcal{H}$, computes the SCCs which are maximal for the partial order $\preceq_{\mathcal{H}}$ introduced in Chapter 3. Its complexity is quasi-linear in the size of the hypergraph. It will allow to evaluate the combinatorial criterion provided by Theorem 3.3, since the set of the SCCs of $\mathcal{H}$ admits a greatest element if and only if there is a unique maximal element.

Directed hypergraphs have a very large number of applications. Among others, they are used to solve problems related to satisfiability in propositional logic (see for instance [AI91, AFFG97, GGPR98, Pre03]), network routing [Pre00], functional dependencies in database theory [Ull82], Petri nets [Yen92], AND-OR graphs in artificial intelligence [Nil71], fixpoint computations in model checking [LS98], chemical reaction networks [Özt08], transportation networks (see for instance [NP89, NPG98]), *etc.* Consequently, many algorithmic aspects of directed hypergraphs have been studied, including reachability, but above all, optimization related problems, such as maximum flows, minimum cardinality cuts, minimum weighted hyperpaths, *etc.* We refer to the surveys of Ausiello *et al.* [AFF01] and of Gallo *et al.* [GLPN93] for further details.

Surpringly, the problem of determining the SCCs has never been addressed before (as far as we know), while it is an elementary question. Besides, it is well understood in the case of

directed graphs: (i) the first known solution is due to Tarjan [Tar72], (ii) later, Aho, Hopcroft, and Ullman attributed to Kosaraju a second method [AHU83], (iii) and finally, Cheriyan and Mehlhorn [CM96], and independently Gabow [Gab00], discovered a third algorithm. Although directed hypergraphs are generalization of directed graphs, none of these methods allow to compute the SCCs in directed hypergraphs. The main reason is that there is an important gap between the notion of reachability in directed hypergraphs and in directed graphs.

The chapter is organized as follows. In Section 4.1, we present an algorithm determining the set of nodes reachable from a given node in a directed hypergraph, firstly introduced in [GLPN93]. We show how it can be used to compute (maximal) SCCs, and explain why this approach is not optimal. That is why, in Section 4.2, we develop an original method whose aim is to compute maximal SCCs. Its complexity is almost linear in the size of the hypergraph. As a conclusion, Section 4.3 informally explains why computing all the SCCs appears to be a harder problem. Finally, Section 4.4 provides the complexity and correctness proof of our algorithm.

4.1 Reachability in directed hypergraphs

The algorithm ReachableFrom that we describe here has been proposed by Gallo *et al.* in [GLPN93]. Its complexity in time and space is proved to be linear in the size of $\mathcal{H}$, which is denoted by $\mathsf{size}(\mathcal{H})$, and defined as:

$$\mathsf{size}(\mathcal{H}) = |N| + \sum_{(T,H)\in E} (|T| + |H|), \quad \text{where } \mathcal{H} = (N, E).$$

The algorithm is presented in Figure 4.1. Let us briefly discuss its principle.

A call to ReachableFrom(u, $\mathcal{H}$) determines the sets of the nodes which are reachable from u in the hypergraph $\mathcal{H}$. The result is stored in the variable ρ. Starting from $\rho = \emptyset$, it is computed by successive iterations of the main loop (Lines 5 to 15). Each time a new node v is added to ρ (Line 6), the hyperedges $e = (T, H)$ newly satisfying $T \subset \rho$ are examined in order to collect all the nodes w of their head H, which are clearly now reachable from the node u in $\mathcal{H}$.

However, a particular data structure is used in order that the time complexity be linear. First, each node v is assumed to be linked to the list E_v of the hyperedges $e = (T, H)$ such that $v \in T$.[1] Thus, when the node v is added to ρ, the hyperedges newly satisfying $T \subset \rho$ are necessarily elements of the list E_v. Besides, instead of performing the expensive test of the inclusion of T into ρ, a counter c_e representing the number of remaining nodes in the set $T \setminus \rho$ is decremented. This counter is initially set to the cardinality of T, and when it reaches the value 0 (Line 9), it means that the hyperedge satisfies the inclusion $T \subset \rho$. In that case, each node w appearing in H is reachable from u, so that it is pushed on a stack S in order to be treater later (Lines 10 to 12). In this way, the stack S temporarily stores some nodes which are known to be reachable from u, but which are not yet in the set ρ. Note that each node v is tagged with a boolean $visited_v$ which is true when v is stored in the stack S or in the set ρ. This avoids to push a node on the stack more than once. The algorithm stops when there is no new node to visit anymore, *i.e.* when S is empty.

Proposition 4.1. *Let $\mathcal{H}$ be a directed hypergraph, and u a node of $\mathcal{H}$. Then the time complexity of the function* ReachableFrom(u, $\mathcal{H}$) *is* $O(\mathsf{size}(\mathcal{H}))$.

[1] The construction of these lists can be clearly performed in linear time by traversing all the hyperedges.

```
1: function ReachableFrom(u, H = (N, E))
2:     for all w ∈ N do visited_w := false
3:     for all e = (T, H) ∈ E do c_e := |T|
4:     visited_u := true, S := [u], ρ := ∅
5:     while S is not empty do
6:         pop v from S, append v to ρ
7:         for all e = (T, H) ∈ E_v do
8:             decrement c_e
9:             if c_e = 0 then
10:                for all w ∈ H such that visited_w = false do
11:                    push w on S, visited_w := true
12:                done
13:            end
14:        done
15:    done
16:    return ρ
17: end
```

Figure 4.1: Linear algorithm computing the sets of nodes reachable from u

Proof. The initialization steps (Lines 2 to 4) clearly have a linear time complexity.

Each node v is treated at most once by the main loop, and the time complexity of the corresponding iteration is $O(1) + O(|E_v|)$ if the instructions between Lines 9 and 13 are not considered. These latter are executed at most once per hyperedge e in the whole execution of the algorithm and their time complexity is $O(|H(e)|)$ for each. The whole time complexity of the main loop is therefore:

$$\sum_{v \in N} (O(1) + O(|E_v|)) + \sum_{e \in E} O(|H(e)|) = O(|N|) + \sum_{e \in E} O(|T(e)|) + O(|H(e)|),$$

so that it is linear in the size of the hypergraph $\mathcal{H}$. □

Gallo's algorithm can be used to determine the reachability graph of $\mathcal{H} = (N, E)$, which is the directed graph G formed by the nodes of N and the set $E' = \{ (u, v) \mid u \rightsquigarrow_{\mathcal{H}} v \}$ of directed edges.[2] It can be indeed computed by evaluating ReachableFrom(u, $\mathcal{H}$) on each node $u \in N$. Following Proposition 4.1, the time complexity of this operation is $O(|N| \times \mathsf{size}(\mathcal{H}))$.

The (maximal) SCCs of $\mathcal{H}$ coincide with the (maximal) SCCs of its reachability graph. The SCCs of the latest can be computed in linear time in the size of G (*i.e.* $O(|N| + |E'|)$), using for instance Tarjan's algorithm [Tar72]. Maximal ones can also be obtained with the same complexity, using a variant of Tarjan's method, which will be described in Section 4.2.2. Therefore, the total time complexity to determine (maximal) SCCs is $O(|N| \times \mathsf{size}(\mathcal{H}) + |N| + |E'|) = O(|N| \times \mathsf{size}(\mathcal{H}))$ (since $|E'|$ is of order of $|N|^2$).

Nevertheless, this approach is not optimal. Firstly, the algorithm ReachableFrom(u, $\mathcal{H}$) is not recursive. In particular, if ReachableFrom(v, $\mathcal{H}$) has been already computed, and that $u \rightsquigarrow_{\mathcal{H}} v$, there is no way to exploit the result of ReachableFrom(v, $\mathcal{H}$) in the execution of ReachableFrom(u, $\mathcal{H}$). This implies that many redundant operations are performed when the whole reachability graph of $\mathcal{H}$ is determined. Secondly, discovering maximal SCCs in directed graphs is known to be solved in linear time. In constrast, if the current approach is executed on a directed graph $G = (N, E)$ (or more exactly, on the corresponding directed

[2] Recall that a *directed graph* is a couple (N, E'), where $E' \subset N \times N$. Its size is defined as $|N| + |E|$.

hypergraph $(N, \{\, (\{u\}, \{v\}) \mid (u, v) \in E \,\}))$, the time complexity is $O(|N| \cdot \mathsf{size}(G))$, which is suboptimal. These two reasons have motivated us to make further investigations in order to find a more efficient solution.

4.2 Computing maximal strongly connected components

In this section, we describe an algorithm which determines the maximal SCCs for the order $\preceq$ in directed hypergraphs. In particular, it returns the number of such SCCs.[3]

Its principle is discussed in Section 4.2.1. We provide a first and suboptimal sketch of the algorithm. The key idea is that maximal SCCs in a directed hypergraph can be computed by an alternative sequence of operations of two kinds: (i) merging some nodes in the directed hypergraph, (ii) discovering the maximal SCCs in an underlying directed graph. For this reason, we define in Section 4.2.2 a variation of Tarjan's algorithm which determines the maximal SCCs of a directed graph in linear time.

Finally, in Section 4.2.3, we build an optimized version of the sketch of the algorithm of Section 4.2.1, which executes in quasi-linear time. It relies on a particular instrumentation of hyperedges, which is quite technical. That is why we recommend that the reader make use of the execution trace given in Section 4.2.4.

4.2.1 Principle of the algorithm for directed hypergraphs

4.2.1.a Underlying directed graph. First observe that a directed graph $G(\mathcal{H}) = (N, E')$ can be associated to any directed hypergraph $\mathcal{H} = (N, E)$, by defining $E' = \{\, (t, h) \mid (\{t\}, H) \in E \text{ and } h \in H \,\}$. The directed graph $G(\mathcal{H})$ is generated by the *simple* hyperedges of $\mathcal{H}$, *i.e.* the elements $e \in E$ such that $|T(e)| = 1$. In Proposition 4.3, we first point out a remarkable special case in which the maximal SCCs of $\mathcal{H}$ and $G(\mathcal{H})$ coincide.

Lemma 4.2. *Let $\mathcal{H}$ be a directed hypergraph. Each* SCC *C of $\mathcal{H}$ is of the form $\cup_i C'_i$ where the C'_i are the* SCC*s of $G(\mathcal{H})$ such that $C \cap C'_i \neq \emptyset$.*

Proof. Consider $u \in C$. Then there exists a SCC C' of $G(\mathcal{H})$ such that $u \in C$ (since the SCCs of $G(\mathcal{H})$ form a partition of the set N), and obviously $C \cap C' \neq \emptyset$.

Conversely, suppose that C' is a SCC of $G(\mathcal{H})$ such that $C \cap C' \neq \emptyset$. Let $u \in C \cap C'$. Then for any $v \in C'$, $u \rightsquigarrow_{G(\mathcal{H})} v \rightsquigarrow_{G(\mathcal{H})} u$, so that $u \rightsquigarrow_{\mathcal{H}} v \rightsquigarrow_{\mathcal{H}} u$, hence $v \in C$. □

Proposition 4.3. *Let $\mathcal{H}$ be a directed hypergraph such that each maximal* SCC *of $G(\mathcal{H})$ is reduced to a singleton. Then $\mathcal{H}$ and $G(\mathcal{H})$ have the same maximal* SCC*s.*

Proof. First suppose that $\{\, u \,\}$ is a maximal SCC of $G(\mathcal{H})$. Suppose that there exists $v \neq u$ such that $u \rightsquigarrow_{\mathcal{H}} v$. Consider a hyperpath $e_1, \ldots, e_p$ from u to v in $\mathcal{H}$. Then there must be a hyperedge e_i such that $T(e_i) = \{\, u \,\}$ and $H(e_i) \neq \{\, u \,\}$ (otherwise, the hyperpath is a cycle and $v = u$). Let $w \in H(e_i) \setminus \{\, u \,\}$. Then (u, w) is an edge of $G(\mathcal{H})$. Since $\{\, u \,\}$ is a maximal SCC of $G(\mathcal{H})$, this enforces $w = u$, which is a contradiction. Hence $\{\, u \,\}$ is a maximal SCC of $\mathcal{H}$.

Conversely, consider a maximal SCC C of $\mathcal{H}$. Let $u \in C$, and let D be the SCC of $G(\mathcal{H})$ containing u. Consider D' a maximal SCC of $G(\mathcal{H})$ such that $D \preceq_{G(\mathcal{H})} D'$, and let C' be a

[3] This will allow to easily evaluate whether there exists a greatest component, as motivated by the extremality criterion of Chapter 3.

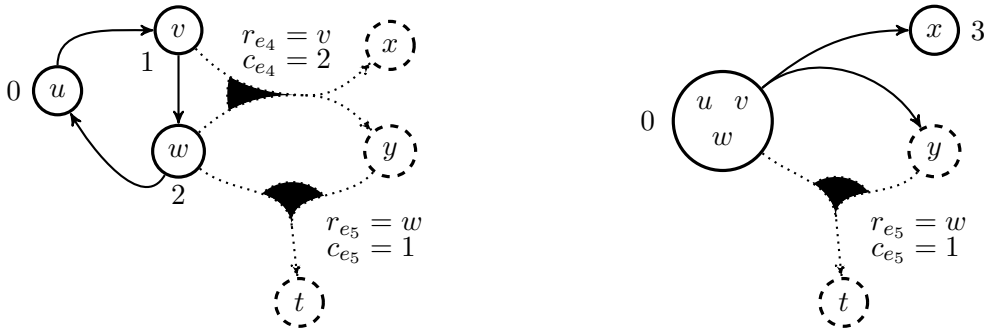


Figure 4.2: A node merging step (the index of the visited nodes is given beside)

Scc of $\mathcal{H}$ such that $D' \cap C' \neq \emptyset$. By Lemma 4.2, we have $D' \subset C'$. It follows that $C \preceq_{\mathcal{H}} C'$, hence $C = C'$ by maximality of C. Thus, $D' \subset C$, and since D' is a singleton, it also forms a Scc of $\mathcal{H}$ using the first part of the proof. This shows $D' = C$ (since the Sccs of $\mathcal{H}$ form a partition of the set of nodes), so that C is a maximal Scc of $G(\mathcal{H})$. □

Naturally, this statement does not hold in the general case.

4.2.1.b Merging nodes in a directed hypergraph. If f is a function from N to an arbitrary set, we denote by $f(\mathcal{H})$ the directed hypergraph of nodes $f(N)$ and of hyperedges $\{\, (f(T(e)), f(H(e))) \mid e \in E \,\}$. The following proposition ensures that, in a directed hypergraph, merging two nodes of a same Scc does not alter the reachability relation:

Proposition 4.4. *Let $\mathcal{H} = (N, E)$ be a directed hypergraph, and let $x, y \in N$ such that $x \equiv_{\mathcal{H}} y$. Consider the function f mapping any node distinct from x and y to itself, and both x and y to a same node z (with $z \notin N \setminus \{x, y\}$). Then $u \rightsquigarrow_{\mathcal{H}} v$ if and only if $f(u) \rightsquigarrow_{f(\mathcal{H})} f(v)$.*

The proof is given in Appendix B. Thus, the image of the reachability graph of $\mathcal{H}$ by f is equal to the reachability graph of $f(\mathcal{H})$. In particular, the maximal Sccs of $\mathcal{H}$ and $f(\mathcal{H})$ are in one-to-one correspondence. These properties can be straightforwardly extended to the operation of merging several nodes of a same Scc simultaneously.

4.2.1.c Sketch of a first algorithm. Using Proposition 4.3 and 4.4, we now sketch a method which computes the maximal Sccs in a directed hypergraph $\mathcal{H}$:

Starting from the directed hypergraph $\mathcal{H}_{cur}$ image of $\mathcal{H}$ by the map $u \mapsto \{\, u \,\}$,

(i) compute the maximal Sccs of the directed graph $G(\mathcal{H}_{cur})$.

(ii) if one of them, say C, is not reduced to a singleton, replace $\mathcal{H}_{cur}$ by $f(\mathcal{H}_{cur})$, where f merges all the elements U of C into the node $\bigcup_{U \in C} U$. Then go back to Step (i).

(iii) otherwise, return the number of maximal Sccs of the directed graph $G(\mathcal{H}_{cur})$.

Each time the *node merging step* (Step (ii)) is executed, new edges may appear in the directed graph $G(\mathcal{H}_{cur})$. This case is illustrated in Figure 4.2. In both sides, the edges of $G(\mathcal{H}_{cur})$ are depicted in solid, and the non-simple hyperedges of $\mathcal{H}_{cur}$ in dotted line. The

```
1: function GMaxSccCount(G = (N, E))
2:     n := 0, nb := 0, S := [ ], Finished := ∅
3:     for all e ∈ E do r_e := undef, c_e := 0
4:     for all u ∈ N do
5:         index[u] := undef, low[u] := undef
6:     done
7:     for all u ∈ N do
8:         if index[u] = undef then GVisit(u)
9:     done
10:    return nb
11: end

12: function GVisit(u)
13:    index[u] := n, low[u] := n, n := n + 1
14:    ismax[u] := true
15:    push u on the stack S
16:    for all (u, w) ∈ E do
17:        if index[w] = undef then GVisit(w)
18:        if w ∈ Finished then
19:            ismax[u] := false
20:        else
21:            low[u] := min(low[u], low[w])
22:            ismax[u] := ismax[u] && ismax[w]
23:        end
24:    done
25:    if low[u] = index[u] then
26:        if ismax[u] = true then nb := nb + 1
27:        repeat
28:            pop v from S, add v to Finished
29:        until index[v] = index[u]
30:    end
31: end
```

Figure 4.3: Computing the maximal SCCs in directed graphs

nodes of $\mathcal{H}_{cur}$ contain subsets of N, but enclosing braces are omitted. Applying Step (i) from node u (left side) discovers a maximal SCC formed by u, v, and w in the directed graph $G(\mathcal{H}_{cur})$. At Step (ii) (right side), the nodes are merged, and the hyperedge e_4 is transformed into two graph edges leaving the new node.

The termination of this method is ensured by the fact that the number of nodes in $\mathcal{H}_{cur}$ is strictly decreased each time Step (ii) is applied. When the method is terminated, maximal SCCs of $\mathcal{H}_{cur}$ are all reduced to single nodes, which contain subsets of N. Propositions 4.3 and 4.4 prove that these subsets are precisely the maximal SCCs of $\mathcal{H}$. Besides, the method returns the exact number of maximal SCCs in $\mathcal{H}$.

However, in order for this approach to be optimal, the algorithm should avoid computing a same maximal SCC several times. For this reason, we are going to inline the node merging step in an algorithm which computes the maximal SCCs in directed graphs.

4.2.2 Computing maximal strongly connected components in directed graphs

The maximal SCCs in a directed graph (or *digraph* for short) can be determined in linear time by an instrumentation of Tarjan's algorithm. This approach is developed in the algorithm GMaxSccCount given in Figure 4.3.

Here is the main principle of the algorithm. As in the classical algorithm, the array *index* tracks the order in which the nodes are visited: $index[u] = i$ if the node u is the i-th one to be visited. The value $low[u]$ is used to determine the minimal index of the nodes which are reachable from u in the digraph (see Line 21). A SCC C is discovered as soon as a node u satisfies $low[u] = index[u]$ (Line 25). Then C consists of all the nodes stored in the stack S above u. The node u is the node of the SCC which has been visited first, and is called its *root*.

The main difference between our algorithm and Tarjan's original one is that the nodes v are provided with a boolean $ismax[v]$ allowing to track the maximality of the SCC. A SCC is maximal if and only if its root u satisfies $ismax[u] = true$. In particular, the boolean $ismax[v]$ is set to *false* as soon as it is connected to a node w located in a distinct SCC (Line 19) or satisfying $ismax[w] = false$ (Line 22). The counter nb determines the number of maximal SCCs which have been discovered (see Line 26).

For the sake of brevity, we have removed the operations allowing to return the SCCs.

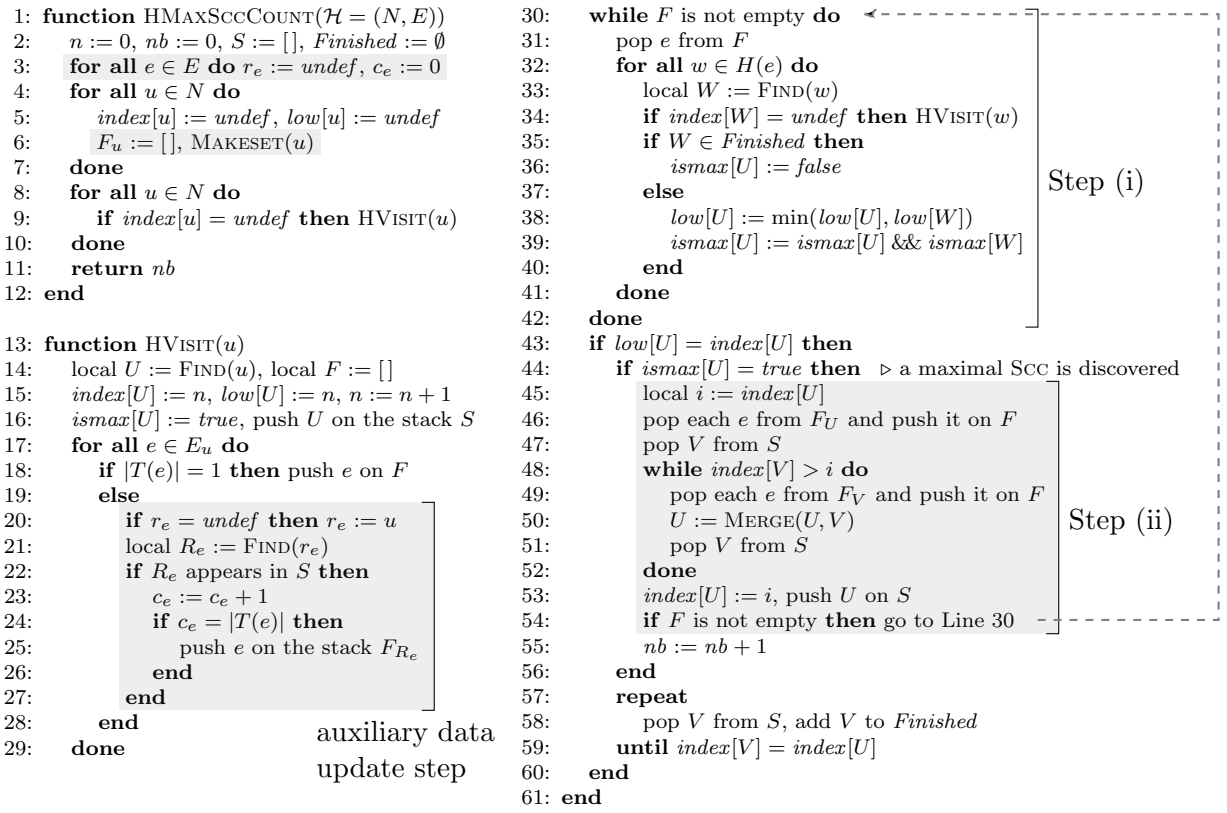


Figure 4.4: Computing the maximal SCCs in directed hypergraphs

Instead, when a node is discovered to be in a SCC, it is placed in the set *Finished* (Line 28). Nevertheless, SCCs can still be recovered after the execution of GMaxSccCount using the arrays *low* and *ismax*: the maximal SCCs are precisely the sets of the form $\{ u \in N \mid low[u] = i \text{ and } ismax[u] = true \}$ for a given i.

4.2.3 Optimized algorithm

We now present the optimized algorithm for directed hypergraphs, following the sketch given in Section 4.2.1. As mentioned before, it consists in incorporating the node merging step directly into the instrumented version of Tarjan's method defined in Section 4.2.2.

First observe that the nodes of the hypergraph $\mathcal{H}_{cur}$ always form a partition of the initial set N of nodes. Instead of referring to them as subsets of N, we use a union-find structure, which consists in three functions FIND, MERGE, and MAKESET (see *e.g.* [CSRL01, Chap. 21]):

- a call to FIND(u) returns, for each original node $u \in N$, the unique node of $\mathcal{H}_{cur}$ containing u.
- two nodes U and V of $\mathcal{H}_{cur}$ can be merged by a call to MERGE(U, V), which returns the new node.
- the "singleton" nodes $\{ u \}$ of the initial $\mathcal{H}_{cur}$ are created by the function MAKESET.

With this structure, each node of $\mathcal{H}_{cur}$ is represented by an element $u \in N$, and then corresponds to the subset $\{ v \in N \mid \text{FIND}(v) = u \}$. Nevertheless, we avoid confusion by denoting

the nodes of the hypergraph $\mathcal{H}$ by lower case letters, and the nodes of $\mathcal{H}_{cur}$ by capital ones. By convention, if $u \in N$, FIND(u) will correspond to the associated capital letter U. Note that when an element $u \in N$ has never been merged with another one, it satisfies FIND$(u) = u$.

The resulting algorithm on directed hypergraphs is given in Figure 4.4. We present the main ideas used in the correctness proof, highlighting the differences with the algorithm on digraphs.

Albeit the hypergraph $\mathcal{H}_{cur}$ is not explicitly manipulated, it can always be inferred as the image of $\mathcal{H}$ by the function FIND. The visiting function HVISIT(u) computes the maximal SCCs reachable from the node FIND(u) in the digraph $G(\mathcal{H}_{cur})$, using the same method as in GVISIT (see the part corresponding to Step (i), from Lines 30 to 42). However, as soon as a maximal SCC is discovered, the node merging step (Step (ii)) is executed.

4.2.3.a Node merging step. This step is performed from Lines 45 to 54, when it is discovered that the node $U =$ FIND(u) is the root of a maximal SCC in the digraph $G(\mathcal{H}_{cur})$. All nodes V which have been collected in that SCC are merged to U (Line 50). Let $\mathcal{H}_{new}$ be the resulting hypergraph.

At Line 54, the stack F is supposed to contain the new edges of $G(\mathcal{H}_{new})$ leaving the newly "big" node U (this point will be explained in the next paragraph). If it is empty, $\{U\}$ is a maximal SCC of $G(\mathcal{H}_{new})$, hence also of $\mathcal{H}_{new}$ (Proposition 4.3). Thus nb is incremented. Otherwise, we go back to the beginning of Step (i) to discover maximal SCCs from the new node U in the digraph $G(\mathcal{H}_{new})$.

4.2.3.b Discovering the new graph edges. In this paragraph, we explain informally how the new graph edges arising after a node merging step (like in Figure 4.2) are efficiently discovered, *i.e.* without examining all the non-simple hyperedges. The formal proof of this technique is provided in Section 4.4.

During the execution of HVISIT(u), the local stack F is used to collect the hyperedges which represent edges leaving the node FIND(u) in the digraph $G(\mathcal{H}_{cur})$. Initially, when HVISIT(u) is called, the node FIND(u) is still equal to u. Then, the loop from Lines 17 to 29 iterates over the set E_u of the hyperedges $e \in E$ such that $u \in T(e)$. At the end of the loop, it can be verified that F is indeed filled with all the simple hyperedges leaving $u =$ FIND(u) in $\mathcal{H}_{cur}$, as expected.

Now the main difficulty is to collect in F the edges which are added to the digraph $G(\mathcal{H}_{cur})$ after a node merging step. To overcome this problem, each non-simple hyperedge $e \in E$ is provided with two auxiliary data:

- a node r_e, called the *root* of the hyperedge e, and which is the first node x of the tail $T(e)$ to be visited by a call to HVISIT,
- and a counter $c_e \geq 0$, which determines the number of nodes $x \in T(e)$ which have been visited, and such that FIND(x) is reachable from FIND(r_e) in the current digraph $G(\mathcal{H}_{cur})$.

These auxiliary data are maintained in the *auxiliary data update step*, from Lines 20 to 27. Initially, the root r_e of any hyperedge e is set to the special value *undef*. The first time a node u such that $e \in E_u$ is visited, it is assigned to u (see Line 20). Besides, at the call to HVISIT(u), the counter c_e of each non-simple hyperedge $e \in E_u$ is incremented, but only when $R_e =$ FIND(r_e) belongs to the stack S (see Line 23). This is indeed a necessary and sufficient

condition to the fact that FIND(u) is reachable from FIND(r_e) in the digraph $G(\mathcal{H}_{cur})$ (see Invariant 4.6 in Section 4.4).

It follows from these invariants that, when the counter c_e reaches the threshold value $|T(e)|$, all the nodes $X = \text{FIND}(x)$ (for $x \in T(e)$) are reachable from R_e in the digraph $G(\mathcal{H}_{cur})$. Now suppose that, later, it is discovered that R_e belongs to a maximal SCC of $G(\mathcal{H}_{cur})$. Then all the nodes X must stand in the same SCC. (Indeed, if C is a maximal SCC and $t \in C$, then z is reachable from t if and only if $z \in C$.) Therefore, when the node merging step is applied on this SCC, the nodes X are merged into a single node U. Hence, the hyperedge e necessarily generates new simple edges leaving U in the new version of the digraph $G(\mathcal{H}_{cur})$.

Now let us verify that in this situation, e is correctly placed into F by our algorithm: as soon as c_e reaches the threshold $|T(e)|$, e is placed into a temporary stack F_{R_e} associated to the node R_e (Line 25). It is then emptied into F at Lines 46 or 49 during the node merging step.

Example 4.1. For example, in the left side of Figure 4.2, the execution of the loop from Lines 17 to 29 during the call to HVISIT(v) sets the root of the hyperedge e_4 to the node v, and c_{e_4} to 1. Then, during HVISIT(w), c_{e_4} is incremented to $2 = |T(e_4)|$. The hyperedge e_4 is therefore pushed on the stack F_v (because $R_{e_4} = \text{FIND}(r_{e_4}) = \text{FIND}(v) = v$). Once it is discovered that u, v, and w form a maximal SCC of $G(\mathcal{H}_{cur})$, e_4 is collected into F. It then allows to visit the nodes x and y from the new node (rightmost hypergraph). A fully detailed execution trace is provided in Section 4.2.4 below.

4.2.3.c Correctness and complexity.

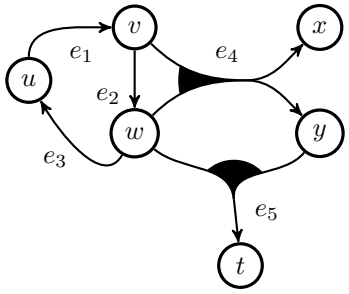
Using disjoint-set forests with union by rank and path compression as union-find structure (see [CSRL01, Chapter 21]), a sequence of p operations MAKESET, FIND, or MERGE can be performed in time $O(p \times \alpha(|N|))$, where α is the very slowly growing inverse of the map $x \mapsto A(x, x)$, and where A is the Ackermann function. Then the following statement holds:

Theorem 4.1. *Let $\mathcal{H} = (N, E)$ be a directed hypergraph. Then* HMAXSCCCOUNT($\mathcal{H}$) *returns the number of maximal* SCC*s in $\mathcal{H}$ in time $O(\mathsf{size}(\mathcal{H}) \times \alpha(|N|))$. Besides, the maximal* SCC*s are formed by the sets $\{ v \in N \mid \text{FIND}(v) = U$ and $ismax[U] = true \}$.*

For any practical value of x, $\alpha(x) \leq 4$. That is why the complexity of HMAXSCCCOUNT is said to be *almost linear* in $\mathsf{size}(\mathcal{H})$.

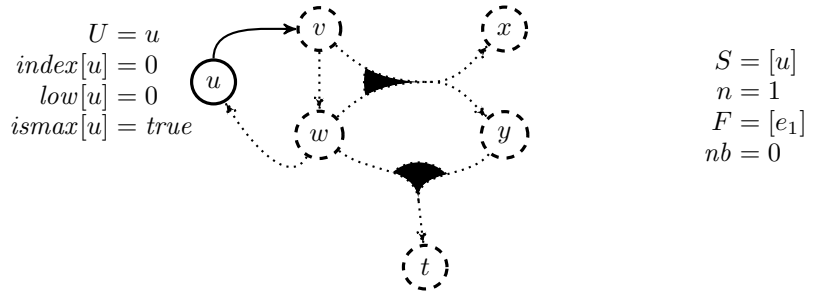
4.2.4 Example of a complete execution trace

We give the main steps of the execution of the algorithm MAXSCCCOUNT on the directed hypergraph depicted in Figure 3.5:

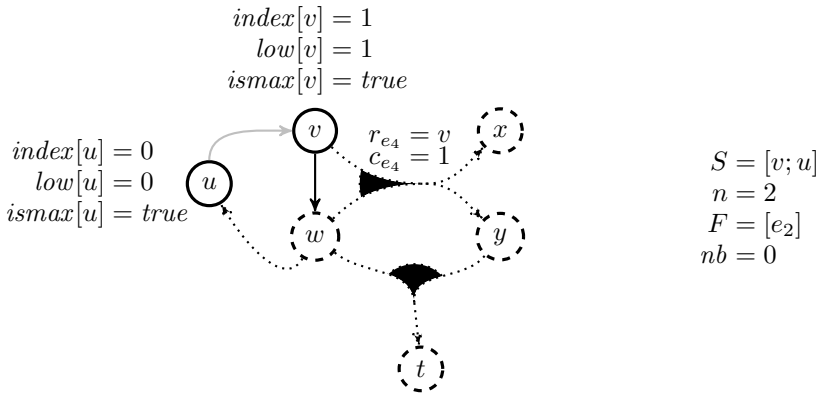


Nodes are depicted by solid circles if their index is defined, and by dashed circles otherwise. Once a node is placed into *Finished*, it is depicted in gray. Similarly, a hyperedge which has never been placed into a local stack F is represented by dotted lines. Once it is pushed into F, it becomes solid, and when it is popped from F, it is colored in gray (note that for the sake of readability, gray hyperedges mapped to cycles after a node merging step will be removed). The stack F which is mentioned always corresponds to the stack local to the last non-terminated call of the function Visit.

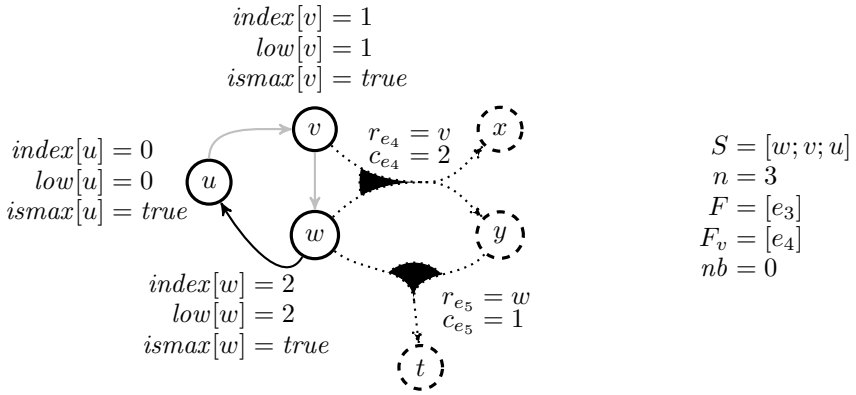
Initially, Find$(z) = z$ for all $z \in \{ u, v, w, x, y, t \}$. We suppose that HVisit$(u)$ is called first. After the execution of the block from Lines 14 to 29, the current state is:



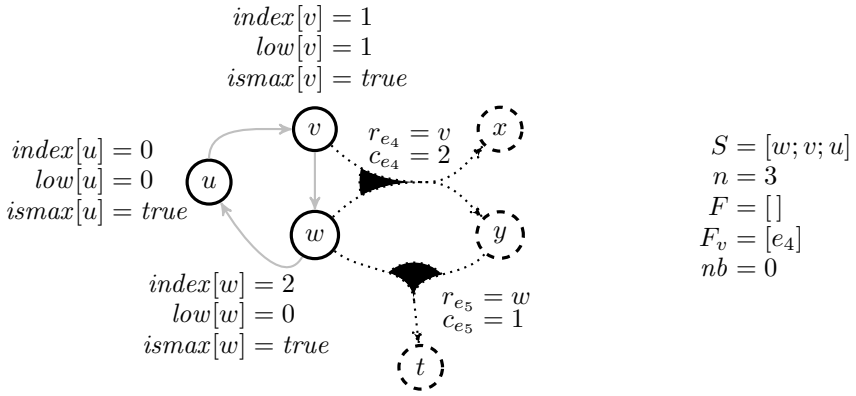
Following the hyperedge e_1, HVisit(v) is called during the execution of the block from Lines 30 to 42 of HVisit(u). After Line 29 in HVisit(v), the root of the hyperedge e_4 is set to v, and the counter c_{e_4} is incremented to 1 since $v \in S$. The state is:



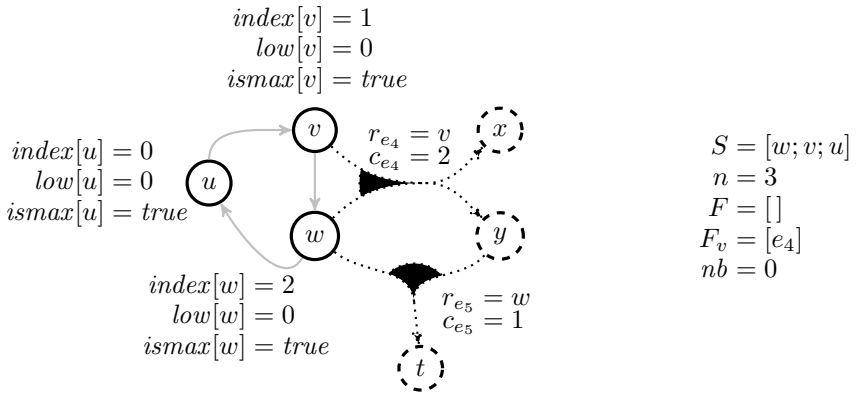
Similarly, the function HVisit(w) is called during the execution of the loop from Lines 30 to 42 in HVisit(v). After Line 29 in HVisit(w), the root of the hyperedge e_5 is set to w, and the counter c_{e_5} is incremented to 1 since $w \in S$. Besides, c_{e_4} is incremented to $2 = |T(e_4)|$ since Find(r_{e_4}) = Find$(v) = v \in S$, so that e_4 is pushed on the stack F_v. The state is:



The execution of the loop from Lines 30 to 42 of HVISIT(w) discovers that $index[u]$ is defined but $u \notin Finished$, so that $low[w]$ is set to $\min(low[w], low[u]) = 0$ and $ismax[w]$ to $ismax[w]$&& $ismax[u] = true$. At the end of the loop, the state is therefore:

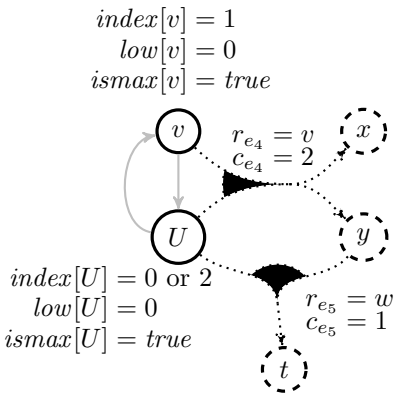


Since $low[w] \neq index[w]$, the block from Lines 43 to 61 is not executed, and HVISIT(w) terminates. Back to the loop from Lines 30 to 42 in HVISIT(v), $low[v]$ is assigned to the value $\min(low[v], low[w]) = 0$, and $ismax[v]$ to $ismax[v]$ && $ismax[w] = true$:



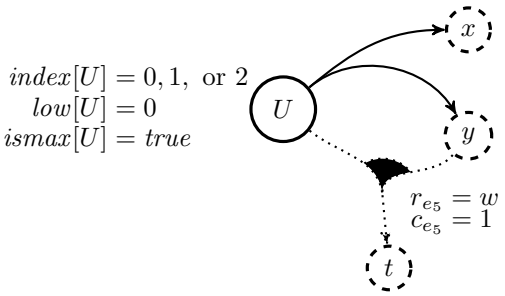
Since $low[v] \neq index[v]$, the block from Lines 43 to 61 is not executed, and HVISIT(v) terminates. Back to the loop from Lines 30 to 42 in HVISIT(u), $low[u]$ is assigned to the value $\min(low[u], low[v]) = 0$, and $ismax[u]$ to $ismax[u]$ && $ismax[v] = true$. Therefore, at Line 43, the conditions $low[u] = index[u]$ and $ismax[u] = true$ hold, so that a node merging step is

executed. At that point, the stack F is empty. After that, i is set to $index[u] = 0$ (Line 45), and $F_u = [\,]$ is emptied to F (Line 46), so that F is still empty. Then w is popped from S, and since $index[w] = 2 > i = 0$, the loop from Lines 48 to 52 is iterated. Then the stack $F_w = [\,]$ is emptied in F. At Line 50, $\textsc{Merge}(u, w)$ is called. The result is denoted by U (in practice, either $U = u$ or $U = w$). The state is:



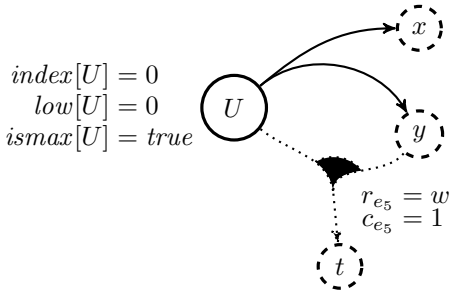
$$\begin{aligned} S &= [v; u] \\ n &= 3 \\ F_v &= [e_4] \\ i &= 0 \\ F &= [\,] \\ U &= \textsc{Find}(u) = \textsc{Find}(w) \\ nb &= 0 \end{aligned}$$

Then v is popped from S, and since $index[v] = 1 > i = 0$, the loop Lines 48 to 52 is iterated again. Then the stack $F_u = [e_4]$ is emptied in F. At Line 50, $\textsc{Merge}(U, v)$ is called. The result is set to U (in practice, U is one of the nodes u, v, w). The state is:



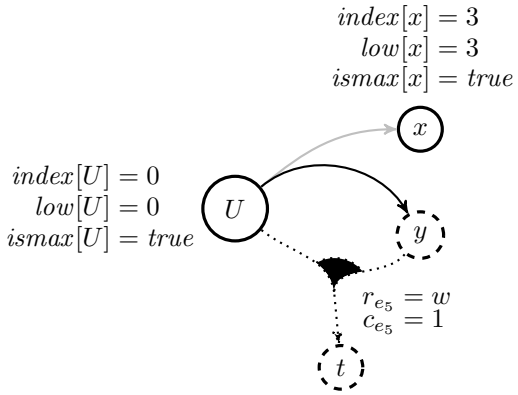
$$\begin{aligned} S &= [u] \\ n &= 3 \\ F_v &= [\,] \\ i &= 0 \\ F &= [e_4] \\ U &= \textsc{Find}(u) = \textsc{Find}(v) \\ &= \textsc{Find}(w) \\ nb &= 0 \end{aligned}$$

After that, u is popped from S, and as $index[u] = 0 = i$, the loop is terminated. At Line 53, $index[U]$ is set to i, and U is pushed on S. Since $F \neq \emptyset$, we go back to Line 30, in the state:

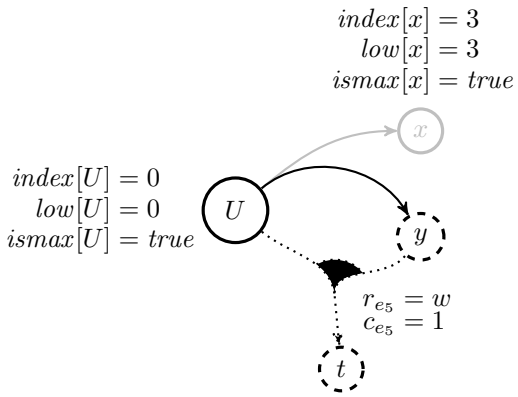


$$\begin{aligned} S &= [U] \\ n &= 3 \\ F &= [e_4] \\ U &= \textsc{Find}(u) = \textsc{Find}(v) \\ &= \textsc{Find}(w) \\ nb &= 0 \end{aligned}$$

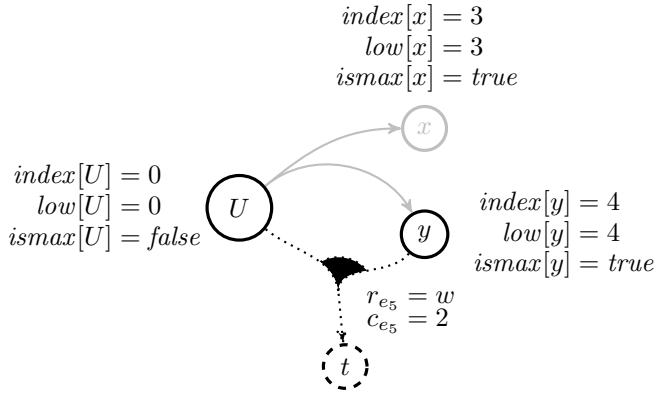
Then e_4 is popped from F, and the loop from 32 to 41 iterates over $H(e_4) = \{\, x, y \,\}$. Suppose that x is treated first. Then $\textsc{Visit}(x)$ is called. During its execution, at Line 29, the state is:



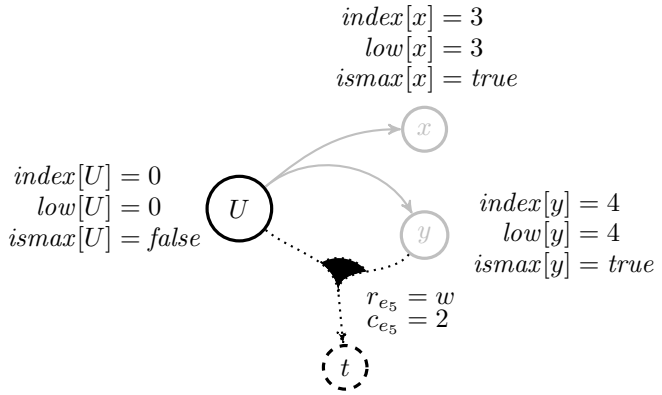
Since F is empty, the loop from Lines 30 to 42 is not executed. At Line 43, $low[x] = index[x]$ and $ismax[x] = true$, so that a trivial node merging step is performed, only on x, since it is the top element of S. At Line 53, it can be verified that $S = [x; U]$, $index[x] = 3$ and $F = [\,]$. Therefore, the goto statement at Line 54 is not executed, and nb is incremented at Line 55. It follows that the loop from Lines 57 to 59 is executed, and after that, the state is:



After the termination of HVISIT(x), since $x \in Finished$, $ismax[U]$ is set to *false*. After that, HVISIT(y) is called, and at Line 29, it can be checked that c_{e_5} has been incremented to $2 = |T(e_5)|$ because $R_{e_5} = \textsc{Find}(r_{e_5}) = \textsc{Find}(w) = U$ and $U \in S$. Therefore, e_5 is pushed to F_U, and the state is:

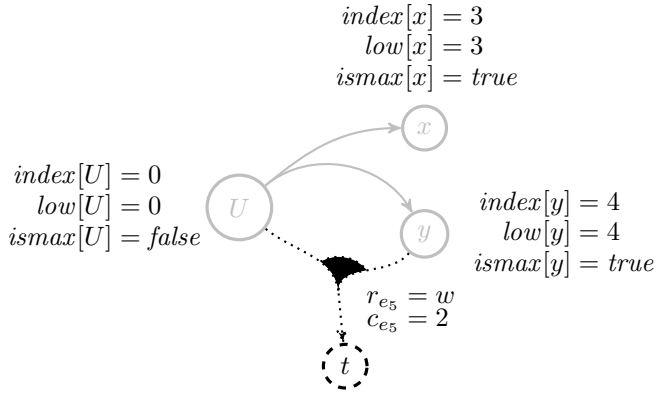


As for the node x, HVISIT(y) terminates by incrementing nb, popping y from S and adding it to *Finished*. Back to the execution of HVISIT(U), at Line 43, the state is:



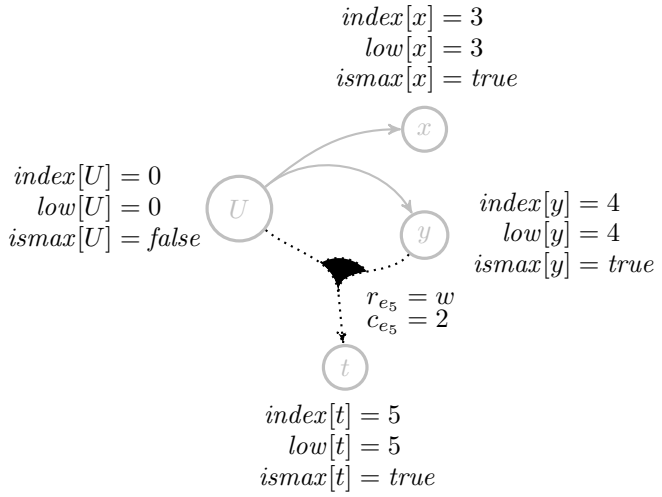
$$
\begin{aligned}
S &= [U] \\
n &= 5 \\
F &= [\,] \\
F_U &= [e_5] \\
U &= \textsc{Find}(u) = \textsc{Find}(v) \\
&= \textsc{Find}(w) \\
nb &= 2 \\
\mathit{Finished} &= \{\, y, x \,\}
\end{aligned}
$$

While $low[U] = index[U]$, $ismax[U]$ is equal to *false*, so that no node merging loop is performed on U. Therefore, e_5 is not popped from F_U and nb is not incremented. Nevertheless, the loop from Lines 57 to 59 is executed, and after that, HVisit(u) is terminated in the state:



$$
\begin{aligned}
S &= [\,] \\
n &= 5 \\
F &= [\,] \\
F_U &= [e_5] \\
U &= \textsc{Find}(u) = \textsc{Find}(v) \\
&= \textsc{Find}(w) \\
nb &= 2 \\
\mathit{Finished} &= \{\, U, y, x \,\}
\end{aligned}
$$

Finally, HVisit(t) is called from HMaxSccCount at Line 9. It can be verified that a trivial node merging loop is performed on t only, and that nb is incremented. After that, t is placed into *Finished*. Therefore, the final state of HMaxSccCount is:



$$
\begin{aligned}
S &= [\,] \\
n &= 6 \\
F_U &= [e_5] \\
U &= \textsc{Find}(u) = \textsc{Find}(v) \\
&= \textsc{Find}(w) \\
nb &= 3 \\
\mathit{Finished} &= \{\, t, U, y, x \,\}
\end{aligned}
$$

Consequently, there is 3 maximal SCCs in the hypergraph. As $ismax[x] = ismax[y] = ismax[t] = true$ and $ismax[\textsc{Find}(z)] = false$ for $z = u, v, w$, they are given by the sets:

$$\{ z \mid \textsc{Find}(z) = x \} = \{ x \},$$
$$\{ z \mid \textsc{Find}(z) = y \} = \{ y \},$$
$$\{ z \mid \textsc{Find}(z) = t \} = \{ t \}.$$

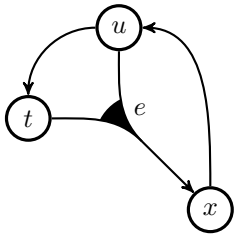
4.3 Conclusion of the chapter

In this chapter, we have developed a quasi-linear algorithm which is able to determine the maximal SCCs in a directed hypergraph. It improves by almost a factor $|N|$ the complexity of the method relying on the reachability algorithm of Gallo *et al.* discussed in Section 4.1. An implementation of this algorithm can be found in the library TPLib [All09].[4]

We consider that this algorithm is of independent interest. It may have some applications outside the context of the combinatorics of tropical polyhedra. However, two interesting questions remain open:

- *can we improve the algorithm to reach a linear complexity?* The fact that the complexity of the algorithm is not strictly linear is due to the necessity to represent partitions of the set of nodes, here with a union/find structures. The SCC algorithm of Gabow on digraphs [Gab00] similarly manipulated partitions of the initial set of nodes (moreover, it also involves a node merging step). And yet, its complexity is linear, because it represents the elements of the partition by intervals. Intuitively, an interval $[n; p]$ encodes the nodes whose index ranges between n and p. This representation can be used because the elements of a same SCC necessarily have consecutive indexes. This property also holds for maximal SCCs in directed hypergraphs. That is why it may be possible to adapt Gabow's idea to our approach, by interfacing the algorithm on directed hypergraphs on its algorithm instead of Tarjan's one. It could allow to reach a linear complexity.

- *can our algorithm be generalized to determine all* SCC*s, without sacrificing the complexity?* First of all, the current algorithm MAXSCCCOUNT is not able to compute all SCCs in directed hypergraphs. Consider the following example:



  Our algorithm determines the unique maximal SCC, which is reduced to the node t. However, the non-maximal SCC formed by u and x is not discovered. Indeed, the non-simple hyperedge e, which allows to reach x from u, is never tranformed into a simple edge, since u and t does not belong to a same SCC of the underlying digraph.

[4] The implementation is provided as an OCaml module named `Hypergraph`, which can be used independently of the rest of the library.

Nevertheless, if the algorithm starts from the node u, the root of the hyperedge e is set to u, and its counter c_e reaches the value 2 after the visit of t. Consequently, the node e is pushed on the stack F_u. A first idea would be to pop this hyperedge from F_u, even if u is not discovered to be in a maximal SCC. Here, this allows to access to the node x from u. In general, this idea is correct: given a hyperedge e, if $c_e = |T(e)|$, then any node of $H(e)$ is reachable from $\textsc{Find}(r_e)$ (see the corresponding invariants in Section 4.2.3).

Unfortunately, this approch misses some hyperedges. For instance, suppose now that the algorithm starts from the node t. In that case, the root r_e is set to t, and since u is not reachable from t, the counter c_e will not be incremented to the value 2 during the visit of the node u. This reveals that our mechanism relying on the auxiliary data of non-simple hyperedges is not adapted to discover all SCCs. In particular, it suggests that the root of a hyperedge should not be the first visited node of its tail, but rather the smallest one in the topological order, if it exists. In our example, after the visit of u, the root r_e would be indeed reassigned to u, and the hyperedge e could be used to reach the node x. Consequently, the challenge would be to maintain this new invariant on the root of the hyperedges, without increasing the time complexity.

```
 1: function HMaxSccCount2(N, E)
 2:     n := 0, nb := 0, S := [ ], Finished := ∅
 3:     for all e ∈ E do collected_e := false
 4:     for all u ∈ N do
 5:         index[u] := undef, low[u] := undef
 6:         Makeset(u)
 7:     done
 8:     for all u ∈ N do
 9:         if low[u] = undef then HVisit2(u)
10:     done
11:     return nb
12: end

13: function HVisit2(u)
14:     local U := Find(u), local F := ∅
15:     index[U] := n, low[U] := n, n := n + 1
16:     ismax[U] := true, push U on the stack S
17:     local no_merge := true
18:     F := { e ∈ E | T(e) = { u } }
19:     for all e ∈ F do collected_e := true
20:     while F is not empty do
21:         pop e from F
22:         for all w ∈ H(e) do
23:             local W := Find(w)
24:             if index[W] = undef then HVisit2(w)
25:             if W ∈ Finished then
26:                 ismax[U] := false
27:             else
28:                 low[U] := min(low[U], low[W])
29:                 ismax[U] := ismax[U] && ismax[W]
30:             end
31:         done
32:     done
33:     if low[U] = index[U] then
34:         if ismax[U] = true then
35:             local i := index[U]
36:             pop V from S
37:             while index[V] > i do
38:                 no_merge := false
39:                 U := Merge(U, V)
40:                 pop V from S
41:             done
42:             push U on S
43:             F := { e ∈ E | collected_e = false
                       and ∀x ∈ T(e), Find(x) = U }
44:             for all e ∈ F do collected_e := true
45:             if no_merge = false then
46:                 n := i, index[U] := n, n := n + 1
47:                 no_merge := true, go to Line 20
48:             else
49:                 nb := nb + 1
50:             end
51:         end
52:         repeat
53:             pop V from S, add V to Finished
54:         until index[V] = index[U]
55:     end
56: end
```

Figure 4.5: First intermediary form of our almost linear algorithm on hypergraphs

4.4 Proving Theorem 4.1

Theorem 4.1 contains two statements, a first one relative to the correctness of HMaxSccCount (*i.e.* it precisely computes the maximal SCCs), and a second one to its time complexity. The first part is proved in Section 4.4.1, the second one in Section 4.4.2.

4.4.1 Correctness of the algorithm

The correctness proof of the algorithm HMaxSccCount turns out to be harder than the one of the classical Tarjan's algorithm, due to the complexity of the invariants which arise in the former algorithm. That is why we propose to show the correctness of two intermediary algorithms, named HMaxSccCount2 (Figure 4.5) and HMaxSccCount3 (Figure 4.6), and then to prove that they are equivalent to HMaxSccCount.

The main difference between the first intermediary form and HMaxSccCount is that it does not use auxiliary data associated to the hyperedges to determine which ones are added to the digraph $G(\mathcal{H}_{cur})$ after a node merging step. Instead, the stack F is directly filled with the right hyperedges (Lines 18 and 43). Besides, a boolean *no_merge* is used to determine whether a node merging step has been executed. The notion of *node merging step* is refined: it now refers to the execution of the instructions between Lines 35 and 44 in which the boolean *no_merge* is set to *false*.

For the sake of simplicity, we will suppose that sequences of assignment or stack manip-

ulations are executed atomically. For instance, the sequences of instructions located in the blocks from Lines 14 and 19, or from Lines 35 and 44, and at from Lines 52 to 54, are considered as elementary instructions. Under this assumption, intermediate complex invariants do not have to be considered.

We first begin with very simple invariants:

Invariant 4.1. *Let U be a node of the current hypergraph $\mathcal{H}_{cur}$. Then $index[U]$ is defined if and only if $index[u]$ is defined for all $u \in N$ such that* $\textsc{Find}(u) = U$.

Proof. It can be shown by induction on the number of node merging steps which has been performed on U.

In the basis case, there is a unique element $u \in N$ such that $\textsc{Find}(u) = U$. Besides, $U = u$, so that the statement is trivial.

After a node merging step yielding the node U, we necessarily have $index[U] \neq undef$. Moreover, all the nodes V which has been merged into U satisfied $index[V] \neq undef$ because they were stored in the stack S. Applying the induction hypothesis terminates the proof. □

Invariant 4.2. *Let $u \in N$. When $index[u]$ is defined, then* $\textsc{Find}(u)$ *belongs either to the stack S, or to the set Finished (both cases cannot happen simultaneously).*

Proof. Initially, $\textsc{Find}(u) = u$, and once $index[u]$ is defined, $\textsc{Find}(u)$ is pushed on S (Line 16). Naturally, $u \notin Finished$, because otherwise, $index[u]$ would have been defined before (see the condition Line 54). After that, $U = \textsc{Find}(u)$ can be popped from S at three possible locations:

- at Lines 36 or 40, in which case U is transformed into a node U' which is immediately pushed on the stack S at Line 42. Since after that, $\textsc{Find}(u) = U'$, the property $\textsc{Find}(u) \in S$ still holds.
- at Line 53, in which case it is directly appended to the set *Finished*. □

Invariant 4.3. *The set Finished is always growing.*

Proof. Once an element is added to *Finished*, it is never removed from it nor merged into another node (the function Merge is always called on elements immediately popped from the stack S). □

Proposition 4.5. *The function* $\textsc{HMaxSccCount2}(\mathcal{H})$ *returns the number of maximal* Scc *of $\mathcal{H}$.*

Besides, the maximal Sccs *are formed by the sets* $\{ v \in N \mid \textsc{Find}(v) = U \text{ and } ismax[U] = true \}$.

Proof. We prove the whole statement by induction on the number of node merging steps.

Basis Case. First, suppose that the hypergraph $\mathcal{H}$ is such that no nodes are merged during the execution of $\textsc{HMaxSccCount2}(\mathcal{H})$, *i.e.* the node merging loop (from Lines 37 to 41) is never executed. Then the boolean *no_merge* is always set to *true*, so that n is never redefined to $i+1$ (Line 46), and there is no back edge to Line 20 in the control-flow graph. It follows that removing all the lines between Lines 35 to 47 does not change the behavior of the algorithm. Besides, since the function Merge is never called, $\textsc{Find}(u)$ always coincides with u. Finally, at Line 18, F is precisely assigned to the set of simple hyperedges leaving u in $\mathcal{H}$, so that

the loop from Lines 20 to 32 iterates on the successors of u in $G(\mathcal{H})$. As a consequence, the algorithm HMaxSccCount2($\mathcal{H}$) behaves exactly like HMaxSccCount($G(\mathcal{H})$). Moreover, under our assumption, the maximal Sccs of $G(\mathcal{H})$ are all reduced to singletons (otherwise, the loop from Lines 37 to 41 would be executed, and some nodes would be merged). Therefore, by Proposition 4.3, the statement in Proposition 4.5 holds.

Inductive Case. Now suppose that the node merging loop is executed at least once, and that its first execution happens during the execution of, say, HVisit2(x). Consider the state of the algorithm at Line 35 just before the execution of the first occurrence of the node merging step. Until that point, Find(v) is still equal to v for all node $v \in N$, so that the execution of HMaxSccCount($\mathcal{H}$) coincides with the execution of HMaxSccCount($G(\mathcal{H})$). Consequently, if C is the set formed by the nodes y located above x in the stack S (including x), C forms a maximal Scc of $G(\mathcal{H})$. In particular, the elements of C are located in a same Scc of the hypergraph $\mathcal{H}$.

Consider the hypergraph $\mathcal{H}'$ obtained by merging the elements of C in the hypergraph $(N, E \setminus \{\, e \mid \exists y \in C \text{ s.t. } T(e) = \{y\} \,\})$, and let X be the resulting node. For now, we may add a hypergraph as last argument of the functions HVisit2, Find, ... to distinguish their execution in the context of the call to HMaxSccCount2($\mathcal{H}$) or HMaxSccCount2($\mathcal{H}'$). We make the following observations:

- the node x is the first element of C to be visited during the execution of HMaxSccCount2($\mathcal{H}$). It follows that the execution of HMaxSccCount2($\mathcal{H}$) until the call to HVisit2($x, \mathcal{H}$) coincides with the execution of HMaxSccCount2($\mathcal{H}'$) until the call to HVisit2($X, \mathcal{H}'$).

- besides, during the execution of HVisit2($x, \mathcal{H}$), the execution of the loop from Lines 20 to 32 only has a local impact, *i.e.* on the $ismax[y]$, $index[y]$, or $low[y]$ for $y \in C$, and not on nb or any information relative to other nodes. Indeed, we claim that the set of the nodes y on which HVisit2 is called during the execution of the loop is exactly $C \setminus \{\, x \,\}$. First, for all $y \in C \setminus \{\, x \,\}$, HVisit2($y$) has necessarily been executed *after* Line 20 (otherwise, by Invariant 4.2, y would be either below x in the stack S, or in *Finished*). Conversely, suppose that after Line 20, there is a call to HVisit2(t) with $t \notin C$. By Invariant 4.2, t belongs to *Finished*, so that for one of the nodes w examined in the loop, either $w \in$ *Finished* or $ismax[w] =$ *false* after the call to HVisit2(w). Hence $ismax[x]$ should be *false*, which contradicts our assumptions.

- finally, from the execution of Line 47 during the call to HVisit2($x, \mathcal{H}$), our algorithm behaves exactly as HMaxSccCount2($\mathcal{H}'$) from the execution of Line 20 in HVisit2($X, \mathcal{H}'$). Indeed, $index[X]$ is equal to i, and the latter is equal to $n - 1$. Similarly, for all $y \in C$, $low[y] = i$ and $ismax[y] =$ *true*. The node X being equal to one of the $y \in C$, we also have $low[X] = i$ and $ismax[X] =$ *true*. Moreover, X is the top element of S.

  Furthermore, it can be verified that at Line 43, the set F contains exactly all the hyperedges of E which generate the simple hyperedges leaving X in $\mathcal{H}'$: they are exactly characterized by

$$\begin{aligned} &\textsc{Find}(z, \mathcal{H}) = X \text{ for all } z \in T(e), \text{ and } T(e) \neq \{\, y \,\} \text{ for all } y \in C \\ \iff &\textsc{Find}(z, \mathcal{H}) = X \text{ for all } z \in T(e), \text{ and } \mathit{collected}_e = \mathit{false} \end{aligned}$$

since at that Line 43, a hyperedge e satisfies $collected_e = true$ if and only if $T(e)$ is reduced to a singleton $\{t\}$ such that $index[t]$ is defined.

Finally, for all $y \in C$, $\textsc{Find}(y, \mathcal{H})$ can be equivalently replaced by $\textsc{Find}(X, \mathcal{H}')$.

As a consequence, HMaxSccCount2($\mathcal{H}$) and HMaxSccCount2($\mathcal{H}'$) return the same result. Besides, both functions perform the same union-find operations, except the first the node merging step executed by HMaxSccCount2($\mathcal{H}$) on C.

Let f be the function which maps all nodes $y \in C$ to X, and any other node to itself. We claim that $\mathcal{H}'$ and $f(\mathcal{H})$ have the same reachability graph, *i.e.* $\rightsquigarrow_{\mathcal{H}'}$ and $\rightsquigarrow_{f(\mathcal{H})}$ are identical relations. Indeed, the two hypergraphs only differ on the images of the hyperedges $e \in E$ such that $T(e) = \{y\}$ for some $y \in C$. For such hyperedges, we have $H(e) \subset C$, because otherwise, $ismax[x]$ would have been set to *false* (*i.e.* the Scc C would not be maximal). It follows that their are mapped to the cycle $(\{X\}, \{X\})$ by f, so that $\mathcal{H}'$ and $f(\mathcal{H})$ clearly have the same reachability graph. In particular, they have the same maximal Sccs.

Finally, since the elements of C are in a same Scc of $\mathcal{H}$, Proposition 4.4 shows that the function f induces a one-to-one correspondence between the Sccs of $\mathcal{H}$ and the Sccs of $f(\mathcal{H})$:

$$\begin{array}{rll} D & \longmapsto f(D) & \\ (D' \setminus \{ X \}) \cup C & \longmapsto\!\shortmid D' & \text{if } X \in D' \\ D' & \longmapsto\!\shortmid D' & \text{otherwise.} \end{array}$$

The action of the function f exactly corresponds to the node merging step performed on C. Since by induction hypothesis, HMaxSccCount2($\mathcal{H}'$) determines the maximal Sccs in $f(\mathcal{H})$, it follows that Proposition 4.5 holds. □

The second intermediary version of our algorithm, HMaxSccCount3, is based on the first one, but it performs the same computations on the auxiliary data r_e and c_e as in HMaxSccCount. However, the latter are never used, because at Line 58, F is re-assigned to the value provided in HMaxSccCount2. It follows that for now, the parts in gray can be ignored. The following lemma states that HMaxSccCount2 and HMaxSccCount3 are equivalent:

Proposition 4.6. *Let $\mathcal{H}$ be a hypergraph. Then* HMaxSccCount3($\mathcal{H}$) *returns the number of maximal* Scc *of $\mathcal{H}$. Besides, the maximal* Sccs *are formed by the sets $\{ v \in N \mid \textsc{Find}(v) = U$ and $ismax[U] = true \}$.*

Proof. When HVisit3(u) is executed, the local stack F is not directly assigned to the set $\{ e \in E \mid T(e) = \{ u \} \}$ (see Line 18 in Figure 4.5), but built by several iterations on the set E_u (Line 21). Since $u \in T(e)$ and $|T(e)| = 1$ holds if and only if $T(e)$ is reduced to $\{ u \}$, HVisit3(u) initially fills F with the same hyperedges as HVisit2(u).

Besides, the condition $no\_merge = false$ in HVisit2 (Line 45) is replaced by $F \neq \emptyset$ (Line 60). We claim that the condition $F \neq \emptyset$ can be safely used in HVisit2 as well. Indeed, in HVisit2, $F \neq \emptyset$ implies $no\_merge = false$. Conversely, suppose that in HVisit2, $no\_merge = false$ and $F = \emptyset$, so that the algorithm goes back to Line 47 after having no_merge to $true$. The loop from Lines 20 to 32 is not executed since $F = \emptyset$, and it directly leads to a new execution of Lines 33 to 45 with $no\_merge = true$. Therefore, going back to Line 47 was useless.

Finally, during the node merging step in HVisit3, n keeps its value, which is greater than or equal to $i + 1$, but is not necessarily equal to $i + 1$ like in HVisit2 (just after Line 46).

```
 1: function HMaxSccCount3(N, E)
 2:     n := 0, nb := 0, S := [], Finished := ∅
 3:     for all e ∈ E do
 4:         r_e := undef, c_e := 0
 5:         collected_e := false
 6:     done
 7:     for all u ∈ N do
 8:         index[u] := undef, low[u] := undef
 9:         Makeset(u), F_u := []
10:     done
11:     for all u ∈ N do
12:         if index[u] = undef then HVisit3(u)
13:     done
14:     return nb
15: end

16: function HVisit3(u)
17:     local U := Find(u), local F := []
18:     index[U] := n, low[U] := n, n := n + 1
19:     ismax[U] := true, push U on the stack S
20:     for all e ∈ E_u do
21:         if |T(e)| = 1 then push e on F
22:         else
23:             if r_e = undef then r_e := u
24:             local R_e := Find(r_e)
25:             if R_e appears in S then
26:                 c_e := c_e + 1
27:                 if c_e = |T(e)| then
28:                     push e on the stack F_{R_e}
29:                 end
30:             end
31:         end
32:     done
33:     for all e ∈ F do collected_e := true
34:     while F is not empty do
35:         pop e from F
36:         for all w ∈ H(e) do
37:             local W := Find(w)
38:             if low[W] = undef then HVisit(w)
39:             if W ∈ Finished then
40:                 ismax[U] := false
41:             else
42:                 low[U] := min(low[U], low[W])
43:                 ismax[U] := ismax[U] && ismax[W]
44:             end
45:         done
46:     done
47:     if low[U] = index[U] then
48:         if ismax[U] = true then
49:             local i := index[U]
50:             pop each e ∈ F_U and push it on F
51:             pop V from S
52:             while index[V] > i do
53:                 pop each e ∈ F_V and push it on F
54:                 U := Merge(U, V)
55:                 pop V from S
56:             done
57:             index[U] := i, push U on S
58:             F := { e ∈ E | collected_e = false
                         and ∀x ∈ T(e), Find(x) = U }
59:             for all e ∈ F do collected_e := true
60:             if F ≠ ∅ then go to Line 34
61:             nb := nb + 1
62:         end
63:         repeat
64:             pop V from S, add V to Finished
65:         until index[V] = index[U]
66:     end
67: end
```

Figure 4.6: Second intermediary form of our linear algorithm on hypergraphs

This is safe because the whole algorithm only need that n take increasing values, and not necessarily consecutive ones.

We conclude by applying Proposition 4.5. □

We make similar assumptions on the atomicity of the sequences of instructions. Note that Invariant 4.1, 4.2, and 4.3 still holds in HVisit3.

Invariant 4.4. *Let $e \in E$ such that $|T(e)| > 1$. If for all $x \in T(e)$, $index[x]$ is defined, then the root r_e is defined.*

Proof. For all $x \in T(e)$, HVisit3(x) has been called. The root r_e has necessarily been defined at the first of these calls (remember that the block from Lines 17 to 33 is supposed to be executed atomically). □

Invariant 4.5. *Consider a state cur of the algorithm in which $U \in Finished$. Then any node reachable from U in $G(\mathcal{H}_{cur})$ is also in Finished.*

Proof. The invariant clearly holds when U is placed in *Finished*. Using the atomicity assumptions, the call to HVisit3(u) is necessarily terminated. Let *old* be the state of the algorithm at that point, and $\mathcal{H}_{old}$ and $Finished_{old}$ the corresponding hypergraph and set of terminated nodes at that state respectively. Since HVisit3(u) has performed a depth-first search from the node U in $G(\mathcal{H}_{old})$, all the nodes reachable from U in $\mathcal{H}_{old}$ stand in $Finished_{old}$.

We claim that the invariant is then preserved by the following node merging steps. The graph edges which may be added by the latter leave nodes in S, and consequently not from elements in *Finished* (by Invariant 4.2). It follows that the set of reachable nodes from elements of $\mathit{Finished}_{old}$ is not changed by future node merging steps. As a result, *all the nodes reachable from U in $G(\mathcal{H}_{cur})$ are elements of $\mathit{Finished}_{old}$*. Since by Invariant 4.5, $\mathit{Finished}_{old} \subset \mathit{Finished}$, this proves the whole invariant in the state *cur*. □

Invariant 4.6. *In the digraph $G(\mathcal{H}_{cur})$, at the call to* HVisit3(u)*, u is reachable from a node W such that $\mathit{index}[W]$ is defined if and only if W belongs to the stack S.*

Proof. The "if" part can be shown by induction. When the function HVisit3(u) is called from Line 12, the stack S is empty, so that this is obvious. Otherwise, it is called from Line 38 during the execution of HVisit3(x). Then $X = \textsc{Find}(x)$ is reachable from any node in the stack, since x was itself reachable from any node in the stack at the call to $\textsc{Find}(X)$ (inductive hypothesis) and that this reachability property is preserved by potential node merging steps (Proposition 4.4). As u is obviously reachable from X, this shows the statement.

Conversely, suppose that $\mathit{index}[W]$ is defined, and W is not in the stack. According to Invariant 4.2, W is necessarily an element of *Finished*. Hence u also belongs to *Finished* by Invariant 4.5, which is a contradiction since this cannot hold at the call to HVisit(u). □

Invariant 4.7. *Let $e \in E$ such that $|T(e)| > 1$. Consider a state cur of the algorithm* HMaxSccCount3 *in which r_e is defined.*

Then c_e is equal to the number of elements $x \in T(e)$ such that $\mathit{index}[x]$ is defined and Find(x) *is reachable from* Find(r_e) *in $G(\mathcal{H}_{cur})$.*

Proof. Since at Line 26, c_e is incremented only if $R_e = \textsc{Find}(r_e)$ belongs to S, we already know using Invariant 4.6 that c_e is equal to the number of elements $x \in T(e)$ such that, at the call to HVisit3(x), x was reachable from $\textsc{Find}(r_e)$.

Now, let $x \in N$, and consider a state *cur* of the algorithm in which r_e and $\mathit{index}[x]$ are both defined, and $\textsc{Find}(r_e)$ appears in the stack S. Since $\mathit{index}[x]$ is defined, HVisit3 has been called on x, and let *old* be the state of the algorithm at that point. Let us denote by $\mathcal{H}_{old}$ and $\mathcal{H}_{cur}$ the current hypergraphs at the states *old* and *cur* respectively. Like previously, we may add a hypergraph as last argument of the function Find to distinguish its execution in the states *old* and *cur*. We claim that $\textsc{Find}(r_e, \mathcal{H}_{cur}) \rightsquigarrow_{G(\mathcal{H}_{cur})} \textsc{Find}(x, \mathcal{H}_{cur})$ if and only if $\textsc{Find}(r_e, \mathcal{H}_{old}) \rightsquigarrow_{G(\mathcal{H}_{old})} x$. The "if" part is due to the fact that reachability in $G(\mathcal{H}_{old})$ is not altered by the node merging steps (Proposition 4.4). Conversely, if x is not reachable from $\textsc{Find}(r_e, \mathcal{H}_{old})$ in $\mathcal{H}_{old}$, then $\textsc{Find}(r_e, \mathcal{H}_{old})$ is not in the call stack S_{old} (Invariant 4.6), so that it is an element of $\mathit{Finished}_{old}$. But $\mathit{Finished}_{old} \subset \mathit{Finished}_{cur}$, which contradicts our assumption since by Invariant 4.2, an element cannot be stored in $\mathit{Finished}_{cur}$ and S_{cur} at the same time. It follows that if r_e is defined and $\textsc{Find}(r_e)$ appears in the stack S, c_e is equal to the number of elements $x \in T(e)$ such that $\mathit{index}[x]$ is defined and $\textsc{Find}(r_e) \rightsquigarrow_{G(\mathcal{H}_{cur})} \textsc{Find}(x)$.

Let *cur* be the state of the algorithm when $\textsc{Find}(r_e)$ is moved from S to *Finished*. The invariant still holds. Besides, in the future states *new*, c_e is not incremented because $\textsc{Find}(r_e, \mathcal{H}_{cur}) \in \mathit{Finished}_{cur} \subset \mathit{Finished}_{new}$ (Invariant 4.3), so that $\textsc{Find}(r_e, \mathcal{H}_{new}) = \textsc{Find}(r_e, \mathcal{H}_{cur})$, and the latter cannot appear in the stack S_{new} (Invariant 4.2). Furthermore, any node reachable from $R_e = \textsc{Find}(r_e, \mathcal{H}_{new})$ in $G(\mathcal{H}_{new})$ belongs to $\mathit{Finished}_{new}$ (Invariant 4.5). It even belongs to $\mathit{Finished}_{cur}$, as shown in the second part of the proof of Invariant 4.5 (emphasized sentence). It follows that the number of reachable nodes from

FIND(r_e) has not changed between states *cur* and *new*. Therefore, the invariant on c_e will be preserved, which completes the proof. □

Proposition 4.7. *In* HVISIT3*, the assignment at Line 58 does not change the value of* F*.*

Proof. It can be shown by strong induction on the number p of times that this line has been executed. Suppose that we are currently at Line 49, and let $X_1, \dots, X_q$ be the elements of the stack located above the root $U = X_1$ of the maximal SCC of $G(\mathcal{H}_{cur})$. Any edge e which will transferred to F from Line 49 to Line 56 satisfies $c_e = |T(e)| > 1$ and FIND(r_e) $= X_i$ for some $1 \leq i \leq q$ (since at 49, F is initially empty). Invariant 4.7 implies that for all elements $x \in T(e)$, FIND(x) is reachable from X_i in $G(\mathcal{H}_{cur})$, so that by maximality of the SCC $C = \{ X_1, \dots, X_q \}$, FIND($x$) belongs to C, *i.e.* there exists $j1$ such that FIND(x) $= X_j$. It follows that at Line 56, FIND(x) $= U$ for all $x \in T(e)$. Then, we claim that $collected_e = false$ at Line 56. Indeed, $e' \in E$ satisfies $collected_{e'} = true$ if and only

- either it has been copied to F at Line 21, in which case $|T(e')| = 1$,
- or if it has been copied to F at the r-th execution of Line 58, with $r < p$. By induction hypothesis, this means that e' has been pushed on a stack F_X and then popped from it strictly before the r-th execution of Line 58.

Observe that a given hyperedge can be popped from a stack F_x at most once during the whole execution of HMAXSCCCOUNT3. Here, e has been popped from F_{X_i} after the p-th execution of Line 58, and $|T(e)| > 1$. It follows that $collected_e = false$.

Conversely, suppose for that, at Line 58, $collected_e = false$, and all the $x \in T(e)$ satisfies FIND(x) $= U$. Clearly, $|T(e)| > 1$ (otherwise, e would have been placed into F at Line 21 and $collected_e$ would be equal to *true*). Few steps before, at Line 49, FIND(x) is equal to one of X_j, $1 \leq j \leq q$. Since $index[X_j]$ is defined (X_j is an element of the stack S), by Invariant 4.1, $index[x]$ is also defined for all $x \in T(e)$, hence, the root r_e is defined by Invariant 4.4. Besides, FIND(r_e) is equal to one of the X_j, say X_k (since $r_e \in T(e)$). As all the FIND(x) are reachable from FIND(r_e) in $G(\mathcal{H}_{cur})$, then $c_e = |T(e)|$ using Invariant 4.7. It follows that e has been pushed on the stack F_{R_e}, where $R_e =$ FIND($r_e, \mathcal{H}_{old}$) in an previous state *old* of the algorithm. As $collected_e = false$, e has not been popped from F_{R_e}, and consequently, the node R_e of $\mathcal{H}_{old}$ has not involved in a node merging step. Therefore, R_e is still equal to FIND()$r_e, \mathcal{H}_{cur}) = X_k$. It follows that at Line 49, e is stored in F_{X_k}, and thus it is copied to F between Lines 49 and 56. This completes the proof. □

We now can prove the correctness of HMAXSCCCOUNT. By Proposition 4.7, Line 58 can be safely removed in HVISIT3. It follows that the booleans $collected_e$ are now useless, to that Lines 5, 33, and 59 can be also removed. After that, we precisely obtain the algorithm HMAXSCCCOUNT. Proposition 4.6 completes the proof. □

4.4.2 Complexity proof

Then analysis of the time complexity HMAXSCCCOUNT depends on the kind of the instructions. We distinguish:

(i) the operations on the global stacks F_u and on the local stacks F,

(ii) the call to the functions FIND, MERGE, and MAKESET,

(iii) and the other operations, referred to as *usual operations*. By extension, their time complexity will be referred to as *usual complexity*.

Also note that the function HVisit(u) is executed exactly once for each $u \in N$ during the execution of HMaxSccCount.

Operations on the stacks F and F_u. Each operation on the stack (pop or push) is in $O(1)$. A given hyperedge is pushed on a stack of the form F_u at most once during the whole execution of HMaxSccCount. Once it is popped from it, it will never be pushed on a stack of the form F_V again. Similarly, a hyperedge is pushed on a local stack F at most once, and after it is popped from it, it will never be pushed on any local stack F' in the following states. Therefore, the total number of stack operations on the local and global stacks F and F_u is bounded by $4|N|$. It follows that the corresponding complexity is $O(|N|)$.

As a consequence, the total number of iterations of the loop from Lines 32 to 41 which occur during the whole execution of HMaxSccCount is bounded by $\sum_{e\in E}|H(e)|$.

Union-find operations. During the whole execution of HMaxSccCount, the function Find is called:

- exactly $|N|$ times at Line 14,
- at most $\sum_{u\in N}|E_u| = \sum_{e\in E}|T(e)|$ times at Line 21 (since during the call to HVisit(u), the loop from Lines 17 to 29 has exactly $|E_u|$ iterations),
- at most $\sum_{e\in E}|H(e)|$ at Line 33 (see above).

Hence it is called at most $\mathsf{size}(\mathcal{H})$ times.

The function Merge is always called to merge two distinct nodes. Let $C_1, \ldots, C_p$ ($p \leq |N|$) be the equivalence classes formed by the elements of N at the end of the execution of HMaxSccCount. Then Merge has been called at most $\sum_{i=1}^{p}(|C_i|-1)$. Since $\sum_i |C_i| = |N|$, Merge is executed at most $|N| - 1$ times.

Finally, MakeSet is called exactly $|N|$ times. It follows that the total time complexity of the operations MakeSet, Find and Merge is $O(\mathsf{size}(\mathcal{H}) \times \alpha(|N|))$.

Usual operations. The analysis of the usual complexity is split into several parts:

- the usual complexity HMaxSccCount without the calls to the function HVisit is clearly $O(|N| + |E|)$.
- during the execution of HVisit(u), the usual complexity of the block from Lines 14 to 29 is $O(1) + O(|E_u|)$. Indeed, we suppose that the test at Line 22 can be performed in $O(1)$ by assuming that the stack S is provided with an auxiliary array of booleans which determines, for each element of N, whether it is stored in S.[5] Then the total usual complexity between Lines 14 and 29 is $O(\mathsf{size}(\mathcal{H}))$ for a whole execution of HMaxSccCount.

[5]Obviously, the push and pop operations on the stack S are still in $O(1)$ under this assumption.

- the usual complexity of the body of loop from Lines 32 to 41, without the recursive calls to HVisit, is clearly $O(1)$. As mentioned above, the total number of iterations of this loop is less than $\sum_{e \in E} |H(e)| \leq \mathsf{size}(\mathcal{H})$. Therefore, the total usual complexity of the loop from Lines 30 to 42 is in $O(\mathsf{size}(\mathcal{H}))$.

- the usual complexity of the loop between Lines 48 and 52 for a whole execution of HMaxSccCount is $O(|N|)$, since in total, it is iterated exactly the number of times the function Merge is called.

- the usual complexity of the loop between Lines 57 and 59 for a whole execution of HMaxSccCount is $O(|N|)$, because a given element is placed at most once into *Finished*.

- if the two previous loops are not considered, less than 10 usual operations are executed in the block from Lines 43 to 61, all of complexity $O(1)$. The execution of this block either follows a call to HVisit or the execution of the goto statement (at Line 54). The latter is executed only if the stack F is not empty. Since each hyperedge can be pushed on a local stack F and then popped from it only once, it is executed $|E|$ in the worst case during the whole execution of HMaxSccCount. It follows that the usual complexity of the block from Lines 43 to 61 is $O(|N| + |E|)$ in total (excluding the loops previously discussed).

Total time complexity. Summing all the complexities above proves that the time complexity of HMaxSccCount is $O(\mathsf{size}(\mathcal{H}) \times \alpha(|N|)$. □

CHAPTER 5

Algorithmics of tropical polyhedra

This chapter is devoted to the following algorithmic problem: *given a tropical polyhedron, how to compute an internal representation from an external description, and inversely?* This is a fundamental problem in the computational aspects of tropical polyhedra. In particular, we will see in Chapter 7 that, in the application to static analysis, the scalability of the numerical abstract domain based on tropical polyhedra is in close correlation with the performance of the algorithms which convert one form of description to the other.

As explained in Chapter 1, the question has not received much attention until now. The first algorithm has been introduced by Butkovič and Hegedus in [BH84] in 1984. It allows to compute a generating representation of a tropical polyhedron defined by a system of inequalities. It has been later rediscovered by Gaubert in [Gau92], and discussed in the survey of Max Plus in [GP97]. On the practical side, this algorithm has been implemented in the Max-Plus toolbox [CGMQ] of the numerical computational software Scilab [Scib] and ScicosLab [Scia].[1] The same technique has been then implemented and optimized in the work of Allamigeon, Gaubert, and Goubault in [AGG08].

In contrast, the analogue problem in classical convex polyhedra has been thoroughly studied. It is known under various names, such as the *vertex enumeration problem*[2], or dually the *facet enumeration problem*[3], depending on the kind of description which is expected in output. Observe that in the classical case, the two problems of passing from the external

[1] The Max-Plus toolbox was available in Scilab until the versions 4.x. It is now integrated into ScicosLab.

[2] also known as *extreme rays enumeration problem* in the case of convex cones.

[3] sometimes called *convex hull problem* when it is relative to polytopes.

to the internal representations, and inversely from the internal to the external descriptions, are computationally equivalent. This is a consequence of the combinatorial duality between cones and polar cones. The problem has been treated in many works, see [ABS97, Zie98] for a survey. However, finding an algorithm which is both polynomial in the size of the input and the output, is still an open problem.

One of the most famous algorithms solving the classical version of the problem has been introduced by Motzkin *et al.* [MRTT53] in 1953. It is known as the *double description method*, and applied on convex cones. It was later rediscovered and generalized to any convex polyhedra by Chernikova in [Che68]. Fukuda and Prodon have revisited this technique in [FP96], providing some theoretical and practical recommandations on the way it should be implementated. Many software distributions are available, among others, CDD [Fuk], PORTA [CL], PPL [BHZ08], APRON [JM], *etc.* While this method is rather ancient, double description method based algorithms are still considered as references. This is probably due to their good average performance in practice, while in comparison, some other techniques, such as those developed by Avis and Fukuda [AF92, AF96], are efficient only on a particular class of polyhedra (see [ABS97]).

In this chapter, we introduce two new algorithms. The first one, presented in Section 5.1, allows to compute a minimal generating representation from a description by a system of constraints. The second one, discussed in Section 5.2, computes a set of constraints, also minimal in a certain sense, from a generating representation. Both algorithms only address the problem on tropical polyhedral cones. However, their generalization to tropical polyhedra is straightforward using the homogenization technique, and is left to the reader. The two algorithms are based on an incremental approach, which is analogous to the classical double description method. We give a precise characterization of their complexity. They are compared theoretically and experimentally to the existing techniques, and to their classical analogues. Finally, in Section 5.3, we discuss the problem of finding an upper bound on the number of extreme elements in tropical polyhedra. In particular, this will allow to derive a more global bound on the complexity of our algorihms in conclusion (Section 5.4).

5.1 From the external description to the internal description

In this section, we introduce an algorithm able to compute a minimal generating set of a polyhedral cone, starting from a system of tropically linear inequalities defining it.

This algorithm is based on the *tropical double description method*, treated in Section 5.1.1, which is the analogue of the double description method of Motzkin *et al.* in the tropical setting. Since this method may return non-minimal representations, it is refined using the combinatorial characterization of extreme element based on directed hypergraphs (Chapter 3) and the related algorithm developed in Chapter 4. This leads to the final version of our algorithm, detailled in Section 5.1.2. We will see that this algorithm is theoretically better than the existing approaches and their implementations in the tropical and classical setting (Sections 5.1.3 and 5.1.4). This is also confirmed experimentally in Section 5.1.5.

5.1.1 The tropical double description method

The *tropical double description method* is an incremental technique based on a successive elimination of inequalities. Given a polyhedral cone $\mathcal{C}$ defined by a system of n constraints, it

computes by induction on k ($0 \leq k \leq n$) a generating set G_k of the intermediate cone defined by the first k constraints. Then G_n forms a generating set of the cone $\mathcal{C}$.

Passing from the set G_k to the set G_{k+1} relies on a result which, given a polyhedral cone $\mathcal{K}$ and a tropical halfspace $\mathcal{H} = \{\, \boldsymbol{x} \mid \boldsymbol{ax} \leq \boldsymbol{bx} \,\}$, allows to build a generating set G' of $\mathcal{K} \cap \mathcal{H}$ from a generating set G of $\mathcal{K}$. This is referred to as the *elementary step* of the method:

Theorem 5.1 (Elementary step of the tropical double description method)**.** *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a closed tropical cone generated by a set G of elements of $\mathbb{R}^d_{\max}$, and let $\mathcal{H}$ be a halfspace $\{\, \boldsymbol{x} \mid \boldsymbol{ax} \leq \boldsymbol{bx} \,\}$, where $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^{1\times d}_{\max}$.*

Then the cone $\mathcal{C} \cap \mathcal{H}$ is generated by the following set:

$$\{\, \boldsymbol{g} \in G \mid \boldsymbol{ag} \leq \boldsymbol{bg} \,\} \;\cup\; \{\, (\boldsymbol{ah})\boldsymbol{g} \oplus (\boldsymbol{bg})\boldsymbol{h} \mid \boldsymbol{g}, \boldsymbol{h} \in G,\ \boldsymbol{ag} \leq \boldsymbol{bg},\ \textit{and}\ \boldsymbol{ah} > \boldsymbol{bh} \,\}. \tag{5.1}$$

Observe that this result holds on any closed tropical cones, and not only on polyhedral cones.

Proof. Let G' be the set given in (5.1). The relation $\mathsf{cone}(G') \subset \mathcal{C} \cap \mathcal{H}$ is obvious.

Now consider $\boldsymbol{x} \in \mathcal{C} \cap \mathcal{H}$. Using Minkowski's theorem for tropical closed cones [GK07, Theorem 3.1], $\boldsymbol{x}$ can be written as a combination of at most $d+1$ elements of G, *i.e.* $\boldsymbol{x} = \bigoplus_{i=1}^{d+1} \lambda_i \boldsymbol{g}^i$ where $\boldsymbol{g}^i \in G$ and $\lambda_i \in \mathbb{R}_{\max}$ for all i. Observe that $\boldsymbol{ax} \leq \boldsymbol{bx}$ implies:

$$\bigoplus_{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i} \lambda_i(\boldsymbol{ag}^i) \quad \oplus \quad \bigoplus_{\boldsymbol{ag}^j > \boldsymbol{bg}^j} \lambda_j(\boldsymbol{ag}^j) \leq \bigoplus_{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i} \lambda_i(\boldsymbol{bg}^i) \quad \oplus \quad \bigoplus_{\boldsymbol{ag}^j > \boldsymbol{bg}^j} \lambda_j(\boldsymbol{bg}^j). \tag{5.2}$$

Suppose that $\bigoplus_{\boldsymbol{ag}^j > \boldsymbol{bg}^j} \lambda_j(\boldsymbol{bg}^j) > \bigoplus_{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i} \lambda_i(\boldsymbol{bg}^i)$. There exists k such that $\lambda_k(\boldsymbol{bg}^k) = \bigoplus_{\boldsymbol{ag}^j > \boldsymbol{bg}^j} \lambda_j(\boldsymbol{bg}^j)$, and necessarily $\lambda_k > \mathbb{0}$. But (5.2) leads to $\lambda_k(\boldsymbol{bg}^k) \geq \lambda_k(\boldsymbol{ag}^k)$ while $\boldsymbol{ag}^k > \boldsymbol{bg}^k$, which is a contradiction. It follows that $\bigoplus_{\boldsymbol{ag}^j > \boldsymbol{bg}^j} \lambda_j(\boldsymbol{bg}^j) \leq \bigoplus_{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i} \lambda_i(\boldsymbol{bg}^i)$, so that, by (5.2),

$$\bigoplus_{\boldsymbol{ag}^j > \boldsymbol{bg}^j} \lambda_j(\boldsymbol{ag}^j) \leq \bigoplus_{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i} \lambda_i(\boldsymbol{bg}^i). \tag{5.3}$$

Let κ be the right member of (5.3). If $\kappa > \mathbb{0}$, then

$$\begin{aligned}
\boldsymbol{x} &= \bigoplus_{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i} \lambda_i \boldsymbol{g}^i \quad \oplus \quad \bigoplus_{\boldsymbol{ag}^j < \boldsymbol{bg}^j} \lambda_j \boldsymbol{g}^j \\
&= \bigoplus_{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i} \lambda_i \boldsymbol{g}^i \quad \oplus \quad \kappa^{-1} \bigoplus_{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i} \Big[\bigoplus_{\boldsymbol{ag}^j > \boldsymbol{bg}^j} \lambda_j(\boldsymbol{ag}^j) \Big] \lambda_i \boldsymbol{g}^i \\
&\quad \oplus \quad \kappa^{-1} \bigoplus_{\boldsymbol{ag}^j > \boldsymbol{bg}^j} \Big[\bigoplus_{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i} \lambda_i(\boldsymbol{bg}^i) \Big] \lambda_j \boldsymbol{g}^j \\
&= \bigoplus_{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i} \lambda_i \boldsymbol{g}^i \quad \oplus \quad \kappa^{-1} \bigoplus_{\substack{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i \\ \boldsymbol{ag}^j > \boldsymbol{bg}^j}} \lambda_i \lambda_j \big[\, (\boldsymbol{ag}^j)\boldsymbol{g}^i \oplus (\boldsymbol{bg}^i)\boldsymbol{g}^j \,\big]
\end{aligned}$$

which shows $\boldsymbol{x} \in \mathsf{cone}(G')$. Otherwise, $\kappa = \mathbb{0}$. By (5.3), $\lambda_j(\boldsymbol{ag}^j) = \mathbb{0}$ for each j such that $\boldsymbol{ag}^j > \boldsymbol{bg}^j$, hence $\lambda_j = \mathbb{0}$. It follows that $\boldsymbol{x} = \bigoplus_{\boldsymbol{ag}^i \leq \boldsymbol{bg}^i} \lambda_i \boldsymbol{g}^i$, thus $\boldsymbol{x} \in \mathsf{cone}(G')$. □

The generating set given in (5.1) is formed by the elements $\boldsymbol{g}$ which belongs to the halfspace $\mathcal{H}$, and their pairwise combinations with elements $\boldsymbol{h}$ which are not located in $\mathcal{H}$. Observe that these combinations not only belong to satisfy the constraint $\boldsymbol{ax} \leq \boldsymbol{bx}$, but also saturate it.

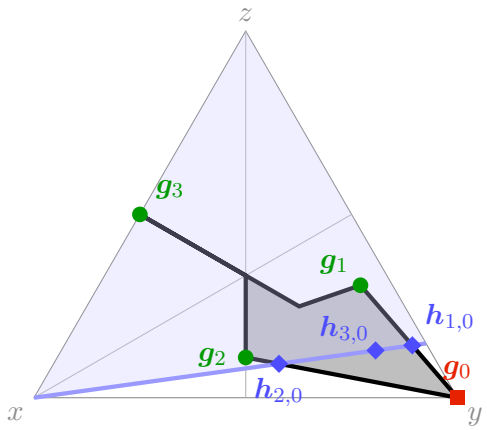


Figure 5.1: The elementary step of the double description method

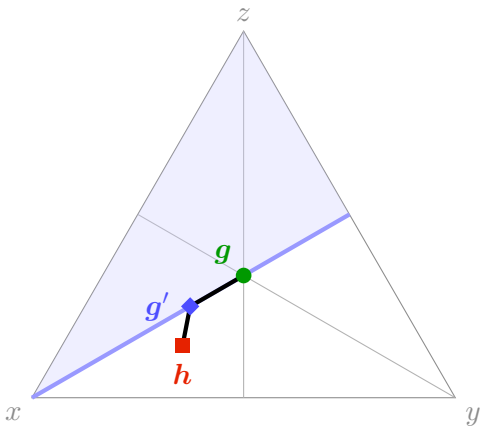


Figure 5.2: Illustration of Remark 5.2

Example 5.1. Figure 5.1 provides an illustration of the elementary step of the double description method on the cone defined in Example 2.10 and the halfspace given by the constraint $y \leq z + 2.5$ (depicted in light blue in Figure 5.1, while the set of elements saturating the inequality is in darker blue). The three elements $\boldsymbol{g}_1$, $\boldsymbol{g}_2$, and $\boldsymbol{g}_3$ satisfy the constraint, while $\boldsymbol{g}_0$ does not. Their combinations are the elements $\boldsymbol{h}_{1,0}$, $\boldsymbol{h}_{2,0}$, and $\boldsymbol{h}_{3,0}$ respectively.

Let us denote by $\boldsymbol{\epsilon}^i$ the element of $\mathbb{R}^d_{\max}$ whose i-th coordinate is equal to $\mathbb{1}$, and the other coordinates to $\mathbb{0}$. Intuitively, the set $(\boldsymbol{\epsilon}^i)_{1\leq i\leq d}$ is a tropical analogue of the canonical basis.

The following theorem describes the whole tropical double description method:

Theorem 5.2 (Tropical double description method)**.** *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a polyhedral cone defined as the set $\{\, \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \leq B\boldsymbol{x} \,\}$, where $A, B \in \mathbb{R}^{p\times d}_{\max}$ (with $p \geq 0$). Let $G_0, \ldots, G_p$ be the sequence of finite subsets of $\mathbb{R}^d_{\max}$ defined as follows:*

$$\begin{cases} G_0 = (\boldsymbol{\epsilon}^i)_{1\leq i\leq d}, \\ G_i = \{\, \boldsymbol{g} \in G_{i-1} \mid A_i\boldsymbol{g} \leq B_i\boldsymbol{g} \,\} \\ \qquad \cup \{\, (A_i\boldsymbol{h})\boldsymbol{g} \oplus (B_i\boldsymbol{g})\boldsymbol{h} \mid \boldsymbol{g}, \boldsymbol{h} \in G_{i-1},\ A_i\boldsymbol{g} \leq B_i\boldsymbol{g},\ \text{and } A_i\boldsymbol{h} > B_i\boldsymbol{h} \,\}, \end{cases}$$

for all $1 \leq i \leq p$, where A_i and B_i are the i-th rows of A and B.

Then $\mathcal{C}$ is generated by the finite set G_p.

Proof. Using Theorem 5.1, it can be easily shown that each G_i forms a generating set of the polyhedral cone $\{\, \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A_j\boldsymbol{x} \leq B_j\boldsymbol{x} \text{ for all } j = 1, \ldots, i \,\}$. □

Observe that Theorem 5.2 provides a constructive proof of the "Minkowski part" of the Minkowski-Weyl theorem (Theorem 2.6), since it shows that all tropical polyhedral cones are generated by finite sets of elements of $\mathbb{R}^d_{\max}$. The other part (*i.e.* the "Weyl part") will be discussed in Section 5.2.1.

Remark 5.2. The tropical double description method looks very similar to the classical one, but they are distinguished by a minor difference: in the elementary step, the combinations

procedure ComputeExtRays(A, B, p) ▷ $A, B \in \mathbb{R}_{\max}^{p\times d}$
  if $p = 0$ **then** ▷ *Base case*
    return $(\boldsymbol{\epsilon}^i)_{1\le i\le d}$
  else ▷ *Inductive case*
    $C := \begin{pmatrix} A_1 \\ \vdots \\ A_{p-1} \end{pmatrix}$, $D := \begin{pmatrix} B_1 \\ \vdots \\ B_{p-1} \end{pmatrix}$, $\boldsymbol{a} := A_p$, $\boldsymbol{b} := B_p$
    $G :=$ ComputeExtRays($C, D, p-1$)
    $G^{\le} := \{\, \boldsymbol{g}^i \in G \mid \boldsymbol{a}\boldsymbol{g}^i \le \boldsymbol{b}\boldsymbol{g}^i \,\}$, $G^{>} := \{\, \boldsymbol{g}^j \in G \mid \boldsymbol{a}\boldsymbol{g}^j > \boldsymbol{b}\boldsymbol{g}^j \,\}$, $H := G^{\le}$
    for all $\boldsymbol{g}^i \in G^{\le}$ and $\boldsymbol{g}^j \in G^{>}$ **do**
      $\boldsymbol{x} := (\boldsymbol{a}\boldsymbol{g}^j)\boldsymbol{g}^i \oplus (\boldsymbol{b}\boldsymbol{g}^i)\boldsymbol{g}^j$
      $\mathcal{H} :=$ BuildHypergraph($\boldsymbol{x}, A, B, p$)
      if HMaxSccCount($\mathcal{H}$) $= 1$ **then** ▷ *Extremality test*
        append $\sigma(\boldsymbol{x})$ to H
      end
    done
  end
  return H
end

Figure 5.3: Computing the extreme rays of tropical cones

$(\boldsymbol{a}\boldsymbol{h})\boldsymbol{g} \oplus (\boldsymbol{b}\boldsymbol{g})\boldsymbol{h}$ with $\boldsymbol{a}\boldsymbol{g} = \boldsymbol{b}\boldsymbol{g}$ and $\boldsymbol{a}\boldsymbol{h} > \boldsymbol{b}\boldsymbol{h}$ do not appear in the classical case, while they are essential in the tropical setting.

For instance, consider the cone of $\mathbb{R}^3_{\max}$ generated by the set G consisting of the elements where $\boldsymbol{g} = (0, 0, 0)$ and $\boldsymbol{h} = (2, 1, 0)$ (in bold black in Figure 5.2). Its intersection with the halfspace $\{\, (x, y, z) \mid y \le z \,\}$ (in light blue) is generated by a minimal set containing: (i) $\boldsymbol{g}$, which is the unique element of G satisfying the condition $y \le z$, and which actually saturates it, (ii) and $\boldsymbol{g}' = (2, 1, 1)$, which results from the combination of $\boldsymbol{g}$ and $\boldsymbol{h}$. The latter element is therefore absolutely necessary here.

5.1.2 Resulting algorithm

We now define an algorithm which implements the tropical double description method. However, Theorem 5.1 and subsequently Theorem 5.2 may return non-extreme elements. For instance, in Example 5.1, only $\boldsymbol{h}^{1,0}$ is extreme, whereas $\boldsymbol{h}^{2,0}$ and $\boldsymbol{h}^{3,0}$ are not. We know by Theorem 2.3 that the extreme rays of a cone form a minimal generating set. Therefore, non-extreme rays are redundant and useless elements, and if they are not eliminated, the approach will not be scalable. Indeed, at each inductive step, the number of elements in the sets G_k grows quadratically in the worst case (because of the pairwise combinations in Theorem 5.2 of the $\boldsymbol{g}$ and $\boldsymbol{h}$). Hence the complexity of the inductive method in total is double exponential ($O(d^{2^p})$), both in time and space, which is clearly untractable.

That is why we propose to eliminate non-extreme elements at each step of the induction, using the combinatorial characterization based on directed hypergraphs (Chapter 3), and the associated quasi-linear algorithm (Chapter 4).

The resulting algorithm ComputeExtRays (Figure 5.3) returns the set of the scaled extreme elements of the cone $\mathcal{C}$. The argument p corresponds to the number of constraints of the system $A\boldsymbol{x} \le B\boldsymbol{x}$:

- when $p = 0$, the cone coincides with $\mathbb{R}^d_{\max}$, so that it is generated by the tropical canonical basis $(\boldsymbol{\epsilon}^i)_{1\le i\le d}$.

- when $p > 0$, the system is split into the system $C\boldsymbol{x} \leq D\boldsymbol{x}$ formed by the $(p-1)$ first inequalities, and the last inequality $\boldsymbol{a}\boldsymbol{x} \leq \boldsymbol{b}\boldsymbol{x}$. Then the elements provided by Theorem 5.2 are computed from the set G of extreme elements of the intermediary cone $\mathcal{D} = \{\, \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid C\boldsymbol{x} \leq D\boldsymbol{x} \,\}$, and stored into the set H.
  The extremality test is implemented at Lines 10-11. First, the tangent hypergraph $\mathcal{H}(\boldsymbol{x}, \mathcal{C})$ is computed thanks to a function BuildHypergraph (a possible implementation of this function is given in Appendix). Then, the function HMaxSccCount that we have defined in Chapter 4 is called. According to Theorem 4.1, its result is equal to 1 if and only if the tangent hypergraph admits a greastest Scc, which is a necessary and sufficient condition of the extremality of $\boldsymbol{x}$. If the test succeeds, the element $\boldsymbol{x}$ is first normalized into a scaled element by σ (see Section 2.2.5), and then appended to the set H. This ensures that H contains only one element of each represented ray.

Observe that the extremality test is applied only on the elements associated to the combinations $(\boldsymbol{a}\boldsymbol{g}^j)\boldsymbol{g}^i \oplus (\boldsymbol{b}\boldsymbol{g}^i)\boldsymbol{g}^j$, and not on the elements $\boldsymbol{g} \in G^{\leq}$ which satisfy $\boldsymbol{a}\boldsymbol{g} \leq \boldsymbol{b}\boldsymbol{g}$. This is a consequence of the following lemma:

Lemma 5.1. *Let $\mathcal{C}, \mathcal{D} \subset \mathbb{R}^d_{\max}$ be two tropical cones such that $\mathcal{C} \subset \mathcal{D}$. Then if $\boldsymbol{x} \in \mathcal{C}$ is extreme in $\mathcal{D}$, then it is also extreme in $\mathcal{C}$.*

Proof. Consider $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{D}$ such that $\boldsymbol{x} = \boldsymbol{u} \oplus \boldsymbol{x}$. Since $\boldsymbol{u}$ and $\boldsymbol{x}$ also belong to $\mathcal{D}$, then $\boldsymbol{x} = \boldsymbol{u}$ or $\boldsymbol{x} = \boldsymbol{v}$, using the extremality of $\boldsymbol{x}$ in $\mathcal{D}$. It follows that $\boldsymbol{x}$ is extreme in $\mathcal{C}$. □

In our setting, any element $\boldsymbol{g} \in G^{\leq}$ belongs to the final $\mathcal{C}$, and is extreme in the intermediary cone $\mathcal{D}$ (by induction hypothesis). Since $\mathcal{C}$ is included into $\mathcal{D}$, it is extreme in $\mathcal{C}$ by Lemma 5.1. As a consequence, it is not necessary to apply the extremality test on the elements of $G^{\leq}$.

Complexity analysis. Each operation in $\mathbb{R}_{\max}$ is supposed to take a unit time. We use hash sets to encode subsets of $\mathbb{R}^d_{\max}$. A *hash set* is a hash table which maps keys to a same meaningless value (for instance *undef*).[4] The keys stored in the hash table correspond to the elements of the represented set. Therefore, the amortized time complexity of adding, searching, and removing an element in the set is bounded by the complexity of hashing a vector of $\mathbb{R}^d_{\max}$, which is supposed to be $O(d)$.

Inductive step. We first study the complexity of the *inductive step*. This step, located from Lines 7 to 14, begins after the termination of the last recursive call to ComputeExtRays. Starting from the last intermediate set G, it consists in (i) computing the set given in (5.1), and (ii) eliminating non-extreme combinations. Its complexity can be precisely characterized in terms of the size of G. It can be easily verified that it is dominated by the extremality tests performed in the loop from Lines 8 to 14. Each test requires to build the hypergraph $\mathcal{H}$ (Line 10). This operation can be done in linear time in its size, which is in $O(p \times d)$. According to Theorem 4.1, HMaxSccCount($\mathcal{H}$) is executed in time $O(\mathsf{size}(\mathcal{H})\alpha(d)) = O(pd\alpha(d))$. The loop is iterated $O(|G|^2)$ in the worst case, so that the following statement holds:

Proposition 5.2. *The worst case time complexity of the inductive step in* ComputeExtRays *is $O(pd\alpha(d)|G|^2)$.*

[4] An implementation for OCaml is provided by Filliâtre in [Fil08].

We also stress that the inductive step is optimal in terms of space complexity, since a non-extreme element is never stored in the resulting set H, even temporarily. It follows that its space complexity is bounded by $O(d\max(|G|,|H|))$.

Remark 5.3. Observe that the construction of the hypergraph $\mathcal{H}$ (Line 10) can be optimized by maintaining some extra information for each element of the intermediate set G.

Indeed, consider a tropical linear form $\boldsymbol{c} \in \mathbb{R}_{\max}^{1\times d}$ and a non-null combination $\boldsymbol{x} = \lambda\boldsymbol{u} \oplus \mu\boldsymbol{v}$ of two elements $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}_{\max}^{d}$. Then the set $\arg\max(\boldsymbol{cx})$ can be computed efficiently from the sets $\arg\max(\boldsymbol{cu})$ and $\arg\max(\boldsymbol{cv})$:

$$\arg\max(\boldsymbol{cx}) = \begin{cases} \arg\max(\boldsymbol{cu}) & \text{if } \lambda(\boldsymbol{cu}) > \mu(\boldsymbol{cv}), \\ \arg\max(\boldsymbol{cv}) & \text{if } \lambda(\boldsymbol{cu}) < \mu(\boldsymbol{cv}), \\ \arg\max(\boldsymbol{cu}) \cup \arg\max(\boldsymbol{cv}) & \text{otherwise.} \end{cases} \tag{5.4}$$

Similarly, the value of $\boldsymbol{cx}$ can be computed in $O(1)$ from $\boldsymbol{cu}$ and $\boldsymbol{cv}$.

Now, let $\boldsymbol{x}$ be an element returned by ComputeExtRays(A, B, p). Using (5.4), the list of the tuples $((A_k\boldsymbol{x}, \arg\max(A_k\boldsymbol{x})), (B_k\boldsymbol{x}, \arg\max(B_k\boldsymbol{x})))$ $(1 \le k \le p)$ can be propagated by induction during the execution of ComputeExtRays(A, B, p). In practice, we have observed that this optimization considerably speeds up the computation of the associated hypergraph.

Overall complexity. The overall complexity of the algorithm ComputeExtRays depends on the maximal size of the sets G_i $(i \le 0 \le p-1)$ returned in the intermediate steps. By Proposition 5.2, we get the following result:

Proposition 5.3. *The worst case time complexity of the* ComputeExtRays *is bounded by:*

$$O\left(\sum_{i=0}^{p-1} pd\alpha(d)|G_i|^2\right),$$

and in particular by $O(p^2 d\alpha(d) G_{\max}^2)$, where $G_{\max}$ is the maximal cardinality of the sets G_i for $i = 0, \ldots, p-1$.

This result can be also expressed in terms of the maximal number $N^{\mathrm{trop}}(p,d)$ of extreme rays of a polyhedral cone of $\mathbb{R}_{\max}^d$ defined by a system of p inequalities. This upper bound will be discussed in detail in Section 5.3. Then we have the following result:

Proposition 5.4. *The worst case time complexity of the* ComputeExtRays *is bounded by:*

$$O\left(\sum_{i=0}^{p-1} pd\alpha(d)(N^{trop}(i,d))^2\right),$$

and in particular, by $O(p^2 d\alpha(d)(N^{trop}(p-1,d))^2)$.

5.1.3 Comparison with the existing approaches

In this section, we describe the existing approaches discussed in the introduction of the chapter, which have been originally defined by Butkovič and Hegedus, then Gaubert, and which are implemented in the Maxplus toolbox [CGMQ] and in [AGG08].

5.1.3.a Main principle. Like ComputeExtRays, these algorithms rely on a successive elimination of inequalities. However, they use a different, albeit equivalent, formulation of the elementary step:

Theorem 5.3 (Elementary step used in [BH84, Gau92, GP97, AGG08])**.** *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical polyhedral cone generated by a set G of elements of $\mathbb{R}^d_{\max}$, and let $\mathcal{H}$ be a halfspace $\{ \boldsymbol{x} \mid \boldsymbol{a}\boldsymbol{x} \leq \boldsymbol{b}\boldsymbol{x} \}$, where $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^{1\times d}_{\max}$.*

Then the cone $\mathcal{C} \cap \mathcal{H}$ is generated by the set $\{ G\boldsymbol{y} \mid \boldsymbol{y} \in G' \}$, where G' is a generating set of the halfspace $\mathcal{H}' = \{ \boldsymbol{y} \in \mathbb{R}^n_{\max} \mid (\boldsymbol{a}G)\boldsymbol{y} \leq (\boldsymbol{b}G)\boldsymbol{y} \}$, and $n = |G|$.

In this statement, the set G is assimilated to the $(d \times n)$-matrix whose columns are given by the elements of G (the order of the columns is meaningless). Then, the inequality $(\boldsymbol{a}G)\boldsymbol{y} \leq (\boldsymbol{b}G)\boldsymbol{y}$ describes a tropical halfspace in $\mathbb{R}^n_{\max}$, where n is the cardinality of the set G (and not the current dimension d). Theorem 5.3 can be associated to following result which provides explicitly a generating set of any tropical halfspace in dimension n:

Proposition 5.5. *Let $\mathcal{H}' = \{ \boldsymbol{x} \in \mathbb{R}^n_{\max} \mid \boldsymbol{a}\boldsymbol{x} \leq \boldsymbol{b}\boldsymbol{x} \}$ be a tropical halfspace ($\boldsymbol{a} = (\boldsymbol{a}_i), \boldsymbol{b} = (\boldsymbol{b}_i) \in \mathbb{R}^{1\times n}_{\max}$). Then a generating family of $\mathcal{H}'$ is given by:*

$$\{ \boldsymbol{e}^i \mid \boldsymbol{a}_i \leq \boldsymbol{b}_i \} \ \cup \ \{ \boldsymbol{a}_j\boldsymbol{e}^i \oplus \boldsymbol{b}_i\boldsymbol{e}^j \mid \boldsymbol{a}_i \leq \boldsymbol{b}_i \text{ and } \boldsymbol{a}_j > \boldsymbol{b}_j \},$$

where $(\boldsymbol{e}^i)_i$ is the tropical canonical basis in dimension n.

It can be checked that combining Theorem 5.3 and Proposition 5.5 yields a generating set of the cone $\mathcal{C} \cap \mathcal{H}$ which coincides with the set given in (5.1). In particular, Proposition 5.5 corresponds to the application of Theorem 5.2 with $G = (\boldsymbol{e}^i)_{1\leq i\leq n}$, and $d = n$.

Nevertheless, Theorem 5.3 is a weaker result than Theorem 5.1, since the former applies only on polyhedral cones, while the latter on any closed tropical cone. Furthermore, the formulation provided in Theorem 5.1 is by far more concise, and has a simple geometric interpretation. It allows to precisely understand on which elements the extremality test has to be executed. As a comparison, the implementation of the Maxplus toolbox of Scilab and of [AGG08] evaluate the extremality test on elements which actually belong to $G^{\leq}$, while it is not needed.

5.1.3.b Elimination of non-extreme elements. The elimination technique of non-extreme elements in the existing algorithms uses a radically different approach than the one defined in Chapter 3. Indeed, it does not rely on the description of the cone by means of halfspaces. It is based on the following statement:

Proposition 5.6. *Let $\mathcal{C}$ be a polyhedral cone generated by a finite set G of scaled elements. Then the scaled extreme generators of $\mathcal{C}$ are precisely the elements $\boldsymbol{x} \in G$ which cannot be expressed as a tropical linear combination of the elements of $G \setminus \{ \boldsymbol{x} \}$, or, equivalently:*

$$\boldsymbol{x} \notin \mathsf{cone}(G \setminus \{ \boldsymbol{x} \}).$$

Proof. Let $G = \{ \boldsymbol{g}^1, \ldots, \boldsymbol{g}^n \}$.

Suppose that $\boldsymbol{x}$ is a scaled extreme generator of $\mathcal{C}$, and that $\boldsymbol{x}$ is equal to $\bigoplus_{\boldsymbol{g}^i \neq \boldsymbol{x}} \lambda_i \boldsymbol{g}^i$ with $\lambda_i \in \mathbb{R}_{\max}$. Then $\boldsymbol{x}$ is equal to one of the $\lambda_i\boldsymbol{g}^i$. Since $\boldsymbol{x}$ and $\boldsymbol{g}^i$ are both scaled, and $\|\lambda_i\boldsymbol{g}^i\| = e^\lambda_i \times \|\boldsymbol{g}^i\| = e^\lambda_i$, we have $\lambda_i = \mathbb{1}$. Hence $\boldsymbol{x}$ is equal to one of the $\boldsymbol{g}^i$, which contradicts the assumption $\boldsymbol{g}^i \neq \boldsymbol{x}$.

Conversely, suppose that $\boldsymbol{x} \neq \bigoplus_{\boldsymbol{g}^i \neq \boldsymbol{x}} \lambda_i \boldsymbol{g}^i$. Let $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{C}$ such that $\boldsymbol{x} = \boldsymbol{u} \oplus \boldsymbol{v}$. The elements $\boldsymbol{u}$ and $\boldsymbol{v}$ can be expressed as tropical combinations of the elements of G:

$$\boldsymbol{u} = \alpha \boldsymbol{x} \oplus \bigoplus_{\substack{1 \leq i \leq n \\ \boldsymbol{g}^i \neq \boldsymbol{x}}} \alpha_i \boldsymbol{g}^i \qquad\qquad \boldsymbol{v} = \beta \boldsymbol{x} \oplus \bigoplus_{\substack{1 \leq i \leq n \\ \boldsymbol{g}^i \neq \boldsymbol{x}}} \beta_i \boldsymbol{g}^i$$

so that:

$$\boldsymbol{x} = (\alpha \oplus \beta)\boldsymbol{x} \oplus \bigoplus_{\substack{1 \leq i \leq n \\ \boldsymbol{g}^i \neq \boldsymbol{x}}} (\alpha_i \oplus \beta_i)\boldsymbol{g}^i$$

If $\boldsymbol{x}$ is neither equal to $\boldsymbol{u}$ nor $\boldsymbol{v}$, then $\alpha < \mathbb{1}$ and $\beta < \mathbb{1}$. It follows that the last equality amounts to

$$\boldsymbol{x} = \bigoplus_{\substack{1 \leq i \leq n \\ \boldsymbol{g}^i \neq \boldsymbol{x}}} (\alpha_i \oplus \beta_i)\boldsymbol{g}^i$$

which shows that $\boldsymbol{x} \in \mathsf{cone}(G \setminus \{ \boldsymbol{x} \})$, which is a contradiction with the initial assumption on $\boldsymbol{x}$. It follows that $\boldsymbol{x} = \boldsymbol{u}$ or $\boldsymbol{x} = \boldsymbol{v}$. □

In the tropical setting, there is a well-known method derived from residuation theory to determine whether a given element is a linear combination of some others. It initially appeared in the work of Vorobyev [Vor67] and Cuninghame-Green [CG76]. See also [BCOQ92, CG95, GP97, BSS07].

Lemma 5.7. *Let $G \subset \mathbb{R}^d_{\max}$ be a finite set. Then $\boldsymbol{x}$ can be expressed as a tropical linear combination of the elements of G if and only if:*

$$\boldsymbol{x} = \bigoplus_{\boldsymbol{g} \in G} (\boldsymbol{g} \backslash \boldsymbol{x}) \boldsymbol{g}$$

where $\boldsymbol{g} \backslash \boldsymbol{x} \stackrel{def}{=} \min_{1 \leq i \leq d} (\boldsymbol{x}_i - \boldsymbol{g}_i)$ (with the convention $-\infty + \infty = +\infty$).

Intuitively, the scalar $\boldsymbol{g} \backslash \boldsymbol{x}$ corresponds to the largest λ such that $\lambda \boldsymbol{g} \leq \boldsymbol{x}$. When the vector $\boldsymbol{g}$ is not identically null, then $\boldsymbol{g} \backslash \boldsymbol{x}$ necessarily belongs to $\mathbb{R}_{\max}$.

An equivalent criterion, studied in various works [Vor67, CG79, AGK05, BSS07], relies on a set covering property:

Lemma 5.8. *Let $G \subset \mathbb{R}^d_{\max}$ be a finite set. Then $\boldsymbol{x}$ belongs to the tropical cone generated by G if and only if $\cup_{\boldsymbol{g} \in G} N_{\boldsymbol{x}}(\boldsymbol{g}) = \mathsf{supp}(\boldsymbol{x})$, where $N_{\boldsymbol{x}}(\boldsymbol{g}) \subset \mathsf{supp}(\boldsymbol{x})$ is defined by:*

$$N_{\boldsymbol{x}}(\boldsymbol{g}) \stackrel{def}{=} \begin{cases} \{ i \in \mathsf{supp}(\boldsymbol{x}) \mid \boldsymbol{x}_i - \boldsymbol{g}_i = \boldsymbol{g} \backslash \boldsymbol{x} \} & \text{if } \boldsymbol{g} \backslash \boldsymbol{x} < +\infty \\ \emptyset & \text{otherwise.} \end{cases} \tag{5.5}$$

Lemmas 5.7 and 5.8 provide a very efficient method to determine whether $\boldsymbol{x}$ is a tropical linear combination of the elements of the set G, in time $O(d|G|)$. Observe that this is an important difference with the classical case, in which the analogue of this problem, *i.e.* determining whether $\boldsymbol{x}$ is a positive linear combination of given elements, is usually solved by linear programming (hence in polynomial time, but with a worse complexity).

The extremality test used in the existing algorithms is based on the characterization provided by Proposition 5.6, in conjunction with Lemmas 5.7 or 5.8. This is illustrated by the algorithm OldComputeExtRays given in Figure 5.4.

1: **procedure** OldComputeExtRays(A, B, p) ▷ $A, B \in \mathbb{R}_{\max}^{p\times d}$
2: **if** $p = 0$ **then** ▷ *Base case*
3: **return** $(\boldsymbol{\epsilon}^i)_{1\le i\le d}$
4: **else** ▷ *Inductive case*
5: $C := \begin{pmatrix} A_1 \\ \vdots \\ A_{p-1} \end{pmatrix}$, $D := \begin{pmatrix} B_1 \\ \vdots \\ B_{p-1} \end{pmatrix}$, $\boldsymbol{a} := A_p$, $\boldsymbol{b} := B_p$
6: $G :=$ OldComputeExtRays($C, D, p-1$)
7: $\boldsymbol{a}' := \boldsymbol{a}G$, $\boldsymbol{b}' := \boldsymbol{b}G$
8: $(\boldsymbol{e}^i)_{1\le i\le n} :=$ tropical canonical basis of $\mathbb{R}_{\max}^n$, where $n = |G|$
9: $G' := \{\, \boldsymbol{e}^i \mid \boldsymbol{a}'_i \le \boldsymbol{b}'_i \,\} \cup \{\, \boldsymbol{a}'_j \boldsymbol{e}^i \oplus \boldsymbol{b}'_i \boldsymbol{e}^j \mid \boldsymbol{a}'_i \le \boldsymbol{b}'_i \text{ and } \boldsymbol{a}'_j > \boldsymbol{b}'_j \,\}$
10: $H := \{\, G\boldsymbol{y} \mid \boldsymbol{y} \in G' \,\}$
11: **for all** $\boldsymbol{x} \in H$ **do**
12: **if** $\boldsymbol{x} = \bigoplus_{\boldsymbol{h}\in H\setminus\{\boldsymbol{x}\}} (\boldsymbol{g}\backslash\boldsymbol{x})\boldsymbol{g}$ **then** remove $\boldsymbol{x}$ from H
13: **done**
14: **end**
15: **return** H
16: **end**

Figure 5.4: Scheme of the most efficient existing methods to compute the extreme rays of tropical cones

Remark 5.4. Since this approach has also been implemented by the author (see [AGG08]), here are some recommandations to make the algorithm as scalable as possible.

First of all, it is not recommended to build the whole set H provided by Theorem 5.3 in a first step, and then apply the test on each element in a second step, as described in Figure 5.4. Indeed, the number of elements provided by Theorem 5.3 may be very large, so that the implementation can run out of memory.[5] That is why an incremental method has been introduced in the refined implementation of the paper [AGG08]: each time an element $\boldsymbol{x}$ is appended to the set H, it is checked whether some element in H can be proved to be non-extreme.

In practice, each element $\boldsymbol{y}$ of H is associated to the subset $\cup_{\boldsymbol{h}\in H\setminus\{\boldsymbol{y}\}} N_{\boldsymbol{y}}(\boldsymbol{h})$, where $N_{\boldsymbol{y}}(\boldsymbol{h})$ is defined as in (5.5). It is also updated in an incremental way, *i.e.* each time an element is appended to H. As soon as it is detected that it is equal to $\mathsf{supp}(\boldsymbol{y})$ (or more efficiently, that its cardinality reaches the threshold $|\mathsf{supp}(\boldsymbol{y})|$), the element $\boldsymbol{y}$ is removed from H.

From our experiments, it is the most efficient strategy of the elimination test by residuation, and the only one which can compete with the extremality test based on directed hypergraphs. Observe however that it does not improve the worst case time and space complexity, as discussed below.

5.1.3.c Complexity analysis and comparison with our algorithm. As for ComputeExtRays, we focus on an accurate estimation of the time complexity of the inductive step (*i.e.* from Lines 7 to 13). It is still dominated by the extremality tests. In the worst case, the $O(|G|^2)$ elements provided by Theorem 5.3 are extreme. In that case, each test has a time complexity of order of $O(d|G|^2)$. Note that the complexity is still the same when the optimizations of Remark 5.4 are used. Then, it can be verified that the total complexity of the inductive step is $O(d|G|^4)$.

[5] This has been experimented with the Maxplus toolbox of Scilab.

Table 5.1: Comparison of the complexity

		ComputeExtRays	OldComp.	classical DDM	
				comb.	algebraic
extremality test		$O(pd\alpha(d))$	$O(d\|G\|^2)$	$O(p\|G\|)$	$O(pd^2)$
inductive step	time	$O(pd\alpha(d)\|G\|^2)$	$O(d\|G\|^4)$	$O(p\|G\|^3)$	$O(pd^2\|G\|^2)$
	space	optimal	not optimal	optimal	
ratio		$O(1)$	$O\left(\frac{\|G\|^2}{p\alpha(d)}\right)$	$O\left(\frac{\|G\|}{d\alpha(d)}\right)$	$O\left(\frac{d}{\alpha(d)}\right)$
overall		$O(p^2d\alpha(d)G_{\max}^2)$	$O(pdG_{\max}^4)$	$O(p^2G_{\max}^3)$	$O((pd)^2G_{\max}^2)$

As a consequence, the overall complexity of OldComputeExtRays is bounded by $O(pdG_{\max}^4)$, where $G_{\max}$ is the maximal cardinality of the intermediate generating sets.

These results are compared to the complexity analysis of ComputeExtRays in Table 5.1. The ratio of the worst case time complexity of the existing methods and ours is also provided. Both for the extremality test and the inductive step, it is of order of $|G|^2/(p\alpha(d))$. As discussed in Section 5.3, and confirmed by the experiments, the size of G, which is the number of extreme rays in the intermediary cone $\mathcal{D}$, is much larger than $p\alpha(d)$ in general.[6] As a result, the performance of ComputeExtRays is significantly better than the performance of the existing methods.

Contrary to ComputeExtRays, the algorithm OldComputeExtRays may temporarily store non-extreme elements of $\mathcal{C}$ in the set H during the inductive step (even if the incremental elimination is used). As a consequence, the space complexity is not optimal, and it can only be bounded by $O(d|G|^2)$. This non-optimality may be harmful to the scalability of OldComputeExtRays.

5.1.4 Comparison with the classical double description method

The principle of the classical double description method is very close to our algorithm. It is also based on successive elimination of inequalities, and its elementary step (see [FP96, Lemmas 3 and 8]) is almost identical to Theorem 5.1.

The elimination of redundant elements is based on a characterization of the adjacency of the elements of G in the intermediary cone $\mathcal{D}$. Two possible criteria can be used. The first one relies on an algebraic property on the rank of a matrix whose size is $(p \times d)$ in the worst case.[7] It can be evaluated in $O(p^2d)$ arithmetical operations, using Gaussian elimination algorithm. The second one is based on a combinatorial property on the set of the inequalities saturated by each ray. It can be checked in $O(p|G|^3)$, where G is the set of extreme rays of the intermediate cone $\mathcal{D}$. In the benchmarks of [FP96], it has been verified that the second criterion is more efficient than the former. This may be due to an explosion of the size of the coefficients of the matrices involved in the algebraic tests, which can severely slow down the evaluation of their rank.

Therefore, suppose that the classical double description method uses the combinatorial

[6] This may also happen even if the cone $\mathcal{C}$ has few extreme rays.

[7] This matrix is formed by the saturated inequalities.

Table 5.2: Execution time benchmarks on a single core of a 3 GHz Intel Xeon with 3 Gb RAM

	d	p	# final	# inter.	T (s)	T' (s)	T/T'
rnd100	12	15	32	59	0.24	6.72	0.035
rnd100	15	10	555	292	2.87	321.78	$8.9 \cdot 10^{-3}$
rnd100	15	18	152	211	6.26	899.21	$7.0 \cdot 10^{-3}$
rnd30	17	10	1484	627	15.2	4667.9	$3.3 \cdot 10^{-3}$
rnd10	20	8	5153	1273	49.8	50941.9	$9.7 \cdot 10^{-4}$
rnd10	25	5	3999	808	9.9	12177.0	$8.1 \cdot 10^{-4}$
rnd10	25	10	32699	6670	3015.7	—	—
cyclic	10	20	3296	887	25.8	4957.1	$5.2 \cdot 10^{-3}$
cyclic	15	7	2640	740	8.1	1672.2	$5.2 \cdot 10^{-3}$
cyclic	17	8	4895	1589	44.8	25861.1	$1.7 \cdot 10^{-3}$
cyclic	20	8	28028	5101	690	~ 45 days	$1.8 \cdot 10^{-4}$
cyclic	25	5	25025	1983	62.6	~ 8 days	$9.1 \cdot 10^{-5}$
cyclic	30	5	61880	3804	261	—	—
cyclic	35	5	155040	7695	1232.6	—	—

test. Since in general, the size of G, both in the classical and the tropical settings, is much larger than $d\alpha(d)$, the extremality test and the inductive step of our algorithm have a better complexity than their classical analogues (see the corresponding ratio in Table 5.1).

5.1.5 Benchmarks

The algorithm ComputeExtreme has been implemented in the library TPLib [All09]. Table 5.2 reports some experiments for different classes of tropical cones: (i) samples formed by several cones chosen randomly (referred to as rnd$x$ where x is the size of the sample), (ii) and *signed cyclic cones* which are known to have a very large number of extreme elements. The latter will be studied in Section 5.3. For each, the first columns respectively report the dimension d, the number of constraints p, the size of the final set of extreme rays, the mean size of the intermediary sets, and the execution time T (for samples of "random" cones, we give average results).

A common point between our algorithm and the classical double description method is that the result does not depend on the order of the inequalities in the initial system. This order may impact the size of the intermediary sets and subsequently the execution time. In our experiments, inequalities are dynamically ordered during the execution: at each step of the induction, the inequality $\boldsymbol{ax} \leq \boldsymbol{bx}$ is chosen so as to minimize the number of combinations $(\boldsymbol{ag}^j)\boldsymbol{g}^i \oplus (\boldsymbol{bg}^i)\boldsymbol{g}^j$. Note that this strategy does not guarantee that the size of the intermediate sets of extreme elements is smaller. However, it reports better results than without ordering.

We compare our algorithm with OldComputeExtRays, whose execution time T' is given in the seventh column. The ratio T/T' shows that our algorithm brings a huge breakthrough in terms of execution time. When the number of extreme rays is of order of 10^4, the second algorithm needs several days to terminate. For instance, the execution of OldComputeExtRays have lasted 45 days on the signed cyclic cone with the parameters $d = 20$ and $p = 8$, while our algorithm ComputeExtRays has returned in only 690 seconds. Therefore, for some extreme cases (for instance $d \geq 30$), the comparison could not be made in practice.

5.2 From the internal description to the external description

This section deals with the dual of the problem addressed in Section 5.1, *i.e.* determining a system of inequalities of a polyhedral cone from a finite generating set.

To get a better understanding of this problem, the notion of *polar* of tropical cones is introduced (Section 5.2.1). Intuitively, the polar of a tropical cone $\mathcal{C}$ is the tropical cone formed by all the inequalities satisfied by $\mathcal{C}$. Section 5.2.2 is focused on the polar of polyhedral cones. It shows that the set of the extreme rays of the polar of a cone $\mathcal{C}$ forms a canonical representation by inequalities of $\mathcal{C}$.

We also show that there is no combinatorial and algorithmic duality between convex cones and their polar in the tropical setting. In particular, we establish a strict refinement of the combinatorial criterion of Chapter 3 to characterize extreme elements in polar cones (Section 5.2.3). In Section 5.2.4, we will see how to algorithmically evaluate with a better complexity than the algorithm of Chapter 4. We then derive the main algorithm, which computes the set of the scaled extreme elements of the polar of a cone $\mathcal{C}$, from a description by a generating set of $\mathcal{C}$. Finally, we compare the algorithm with the alternative approaches (Section 5.2.5).

5.2.1 Polar of tropical cones

We denote by ${}^t\cdot$ the transposition operator.

The polar of a tropical cone $\mathcal{C}$ represents the linear inequalities which are satisfied by all elements of $\mathcal{C}$:

Proposition-Definition 5.9. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical cone. Then the set $\mathcal{C}^\circ \subset (\mathbb{R}^d_{\max})^2$, defined by:*

$$\mathcal{C}^\circ \stackrel{def}{=} \{\, (\boldsymbol{a}, \boldsymbol{b}) \in (\mathbb{R}^d_{\max})^2 \mid {}^t\boldsymbol{a}\boldsymbol{x} \leq {}^t\boldsymbol{b}\boldsymbol{x} \text{ for all } \boldsymbol{x} \in \mathcal{C} \,\},$$

is a tropical cone, and is called the polar cone *of $\mathcal{C}$.*

Note that we implicitly use the isomorphism between $(\mathbb{R}^d_{\max})^2$ and $\mathbb{R}^{2d}_{\max}$.

Proof. Consider $(\boldsymbol{a}, \boldsymbol{b}), (\boldsymbol{c}, \boldsymbol{d}) \in \mathcal{C}^\circ$. Clearly, for all $\lambda, \mu \in \mathbb{R}_{\max}$ and all $\boldsymbol{x} \in \mathcal{C}$, we have:

$${}^t(\lambda\boldsymbol{a} \oplus \mu\boldsymbol{c})\boldsymbol{x} = \lambda\, {}^t\boldsymbol{a}\boldsymbol{x} \oplus \mu\, {}^t\boldsymbol{c}\boldsymbol{x} \leq \lambda\, {}^t\boldsymbol{b}\boldsymbol{x} \oplus \mu\, {}^t\boldsymbol{d}\boldsymbol{x} = {}^t(\lambda\boldsymbol{b} \oplus \mu\boldsymbol{d})\boldsymbol{x}.$$

Hence, $\lambda(\boldsymbol{a}, \boldsymbol{b}) \oplus \mu(\boldsymbol{c}, \boldsymbol{d})$ belongs to $\mathcal{C}^\circ$. □

A dual notion of polar cones of $(\mathbb{R}^d_{\max})^2$ can be introduced:

Definition 5.1. Let $\mathscr{C} \subset (\mathbb{R}^d_{\max})^2$ be a tropical cone. Then the *dual polar cone* of $\mathscr{C}$ is the tropical cone $\mathscr{C}^\diamond \subset \mathbb{R}^d_{\max}$ defined by:

$$\mathscr{C}^\diamond \stackrel{def}{=} \{\, \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid {}^t\boldsymbol{a}\boldsymbol{x} \leq {}^t\boldsymbol{b}\boldsymbol{x} \text{ for all } (\boldsymbol{a}, \boldsymbol{b}) \in \mathscr{C} \,\}.$$

Clearly, any tropical cone $\mathcal{C}$ is included into its *bipolar*, defined as $(\mathcal{C}^\circ)^\diamond$. A separation theorem [Zim77, Gau92, CGQS05] allows to show that both coincide when $\mathcal{C}$ is closed:

Theorem 5.4. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a closed tropical cone. Then the cone $\mathcal{C}$ and its bipolar are identical:*

$$\mathcal{C} = (\mathcal{C}^\circ)^\diamond.$$

This implies that a closed tropical cone can be equivalently represented by the set of inequalities which are satisfied by all of its elements.

5.2.2 Polar of finitely generated cones

When the cone $\mathcal{C}$ is finitely generated, its polar is also finitely generated:

Proposition 5.10. *Let $G = \{ \boldsymbol{g}^1, \dots, \boldsymbol{g}^n \}$ be a finite subset of $\mathbb{R}^d_{\max}$. Then the polar of $\mathsf{cone}(G)$ is the following polyhedral cone:*

$$\left\{ (\boldsymbol{a}, \boldsymbol{b}) \in (\mathbb{R}^d_{\max})^2 \mid {}^t G\boldsymbol{a} \leq {}^t G\boldsymbol{b} \right\}, \tag{5.6}$$

where G is assimilated to the matrix of size $d \times n$ whose columns are the vectors $\boldsymbol{g}^1, \dots, \boldsymbol{g}^n$.

Proof. First note that $(\boldsymbol{a}, \boldsymbol{b})$ satisfies ${}^t G\boldsymbol{a} \leq {}^t G\boldsymbol{b}$ if and only if ${}^t\boldsymbol{a}\boldsymbol{g}^i \leq {}^t\boldsymbol{b}\boldsymbol{g}^i$ for all i.

We then conclude by observing that an inequality holds for all elements of $\mathsf{cone}(G)$ if and only if is satisfied by each vector of G. □

Theorem 5.4 has a remarkable formulation when $\mathcal{C}$ is a finitely generated cone, since $\mathcal{C}$ can be exactly expressed as the solution of the system of inequalities associated to the extreme rays of $\mathcal{C}^\circ$:

Corollary 5.11. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a finitely generated tropical cone. Consider a sequence $(\boldsymbol{a}_1, \boldsymbol{b}_1), \dots, (\boldsymbol{a}_p, \boldsymbol{b}_p)$ formed by one representative of each extreme ray of the polar cone $\mathcal{C}^\circ$.*

Then the following statement holds:

$$\mathcal{C} = \{ \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid {}^t\boldsymbol{a}_j\boldsymbol{x} \leq {}^t\boldsymbol{b}_j\boldsymbol{x} \text{ for all } j = 1, \dots, p \}. \tag{5.7}$$

Proof. Let S be the set given in the right member of (5.7). The inclusion $\mathcal{C} \subset S$ is a straightfoward application of the definition of the polar of $\mathcal{C}$.

Conversely, any element $\boldsymbol{x} \in S$ belongs to the dual polar of $\mathcal{C}^\circ$. Indeed, if $(\boldsymbol{a}, \boldsymbol{b}) \in \mathcal{C}^\circ$, then there exist $\lambda_1, \dots, \lambda_p$ such that $(\boldsymbol{a}, \boldsymbol{b}) = \bigoplus_{j=1}^p \lambda_j(\boldsymbol{a}_j, \boldsymbol{b}_j)$ by Theorem 2.2, so that

$${}^t\boldsymbol{a}\boldsymbol{x} = \bigoplus_{j=1}^p \lambda_j {}^t\boldsymbol{a}_j\boldsymbol{x} \leq \bigoplus_{j=1}^p \lambda_j {}^t\boldsymbol{b}_j\boldsymbol{x} = {}^t\boldsymbol{b}\boldsymbol{x}.$$

Since $\mathcal{C}$ is a closed cone (Lemma 2.5), applying Theorem 5.4 shows that $\boldsymbol{x} \in \mathcal{C}$. □

Corollary 5.11 completes the proof of the Minkowski-Weyl theorem (Theorem 2.6), since it shows that every finitely generated cone is a polyhedral cone.

Remark 5.5. Any sequence $(\boldsymbol{a}_1, \boldsymbol{b}_1), \dots, (\boldsymbol{a}_p, \boldsymbol{b}_p)$ as in Corollary 5.11 forms a minimal set of elements of the polar cone $\mathcal{C}^\circ$, according to Theorem 2.3. However, some of the corresponding inequalities ${}^t\boldsymbol{a}_i\boldsymbol{x} \leq {}^t\boldsymbol{a}_i\boldsymbol{b}$ may be redundant. We say that an inequality is *redundant* in a system of inequalities S defining a cone $\mathcal{C}$ if the set of the solutions of the system S minus this inequality still coincides with the cone $\mathcal{C}$.

This redundancy follows from the tropical bipolar theorem established by Gaubert and Katz in [GK09a]:

> A subset $\mathscr{C} \subset (\mathbb{R}^d_{\max})^2$ is the polar of a tropical cone if and only if the following conditions hold:
>
> (i) $\mathscr{C}$ is a tropical cone,
>
> (ii) $(\boldsymbol{u}, \boldsymbol{v}) \in \mathscr{C}$ for all $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d_{\max}$ such that $\boldsymbol{u} \leq \boldsymbol{v}$,

(iii) and if $(\boldsymbol{u}, \boldsymbol{v}), (\boldsymbol{v}, \boldsymbol{w}) \in \mathscr{C}$, then $(\boldsymbol{u}, \boldsymbol{w}) \in \mathscr{C}$.

Consequently, the polar $\mathcal{C}^\circ$ necessarily contains all the tautologies, and is closed by transitivity. It follows that some of its extreme rays are redundant. For instance, it can be easily verified that any element of the form $(\mathbb{0}, \boldsymbol{\epsilon}^i)$, which corresponds to the tautology $\boldsymbol{x}_i \geq \mathbb{0}$ $(1 \leq i \leq d)$, is extreme in $\mathcal{C}^\circ$.

This is a major difference with the classical case, in which the extremality and non-redundancy properties are equivalent, as a consequence of Farkas Lemma. Indeed, Farkas Lemma (see *e.g.* [Zie98, Proposition 1.9]) states that an inequality is redundant in a system if and only if it can be expressed as a positive linear combination of the other inequalities of the system.

Despite it is not minimal in the sense of redundancy, the set of the extreme rays of the polar cone $\mathcal{C}^\circ$ forms a canonical external representation of the cone $\mathcal{C}$. It provides an acceptable solution in the absence of a well-defined notion of facets (see the work of Develin and Yu in which some conjectures on the definition of faces have been studied [DY07]).

5.2.3 Efficient characterization of extreme elements of the polar of a polyhedral cone

Observe that the inequalites which define the polar cone of $\mathcal{C} = \mathsf{cone}(G)$ have a very particular form. Suppose that $G = \{ \boldsymbol{g}^1, \ldots, \boldsymbol{g}^p \}$. Then recall that $(\boldsymbol{a}, \boldsymbol{b})$ belongs to $\mathcal{C}^\circ$ if and only if:

$$\bigoplus_{1 \leq i \leq d} \boldsymbol{g}_i^k \boldsymbol{a}_i \leq \bigoplus_{1 \leq j \leq d} \boldsymbol{g}_j^k \boldsymbol{b}_j \quad \text{for all } 1 \leq k \leq p,$$

where $\boldsymbol{a} = (\boldsymbol{a}_i)$, $\boldsymbol{b} = (\boldsymbol{b}_i)$, and $\boldsymbol{g}^k = (\boldsymbol{g}_i^k)$. These inequalities have a very special form: their left member involves only the first component $\boldsymbol{a}$ of $(\boldsymbol{a}, \boldsymbol{b})$, while their right member, only $\boldsymbol{b}$.

As a consequence, the tangent hypergraph $\mathcal{H}((\boldsymbol{a}, \boldsymbol{b}), \mathcal{C}^\circ)$ is also of a special kind: each hyperedge necessary leaves a subset of $\mathsf{supp}(\boldsymbol{b})$, and enters a subset of $\mathsf{supp}(\boldsymbol{a})$. In the graph terminology, we say that the hypergraph admits a cut $(\mathsf{supp}(\boldsymbol{b}), \mathsf{supp}(\boldsymbol{a}))$, and every hyperedge crosses it, *i.e.* the cutset consists of all the hyperedges. This configuration is illustrated in Figure 5.5, where we suppose that $\mathsf{supp}(\boldsymbol{a}) = \{ 3, 4, 6, 8 \}$ and $\mathsf{supp}(\boldsymbol{b}) = \{ 1, 2, 5, 7, 9, 10 \}$. In such hypergraphs, the existence of a greastest Scc can be characterized elementary:

Lemma 5.12. *Let $\mathcal{H} = (N, E)$ be a directed hypergraph such that there exists a partition (N_1, N_2) of N and for all $e \in E$, $T(e) \subset N_1$ and $H(e) \subset N_2$. Then $\mathcal{H}$ admits a greatest* Scc *if and only if one of the two following conditions is satisfied:*

- *either N_1 is reduced to a singleton and N_2 is empty,*
- *or N_2 is reduced to a singleton $\{ v \}$, $v \notin N_1$, and each node $u \in N_1$ is bound to v by a simple hyperedge $(\{ u \}, \{ v \})$.*

Proof. Suppose that $\mathcal{H}$ admits a greatest Scc. If $N_2 = \emptyset$, then each node of N_1 forms a maximal Scc. Therefore, N_1 has to be reduced to a singleton. Otherwise, if $N_2 \neq \emptyset$, then every node of N_2 forms a maximal Scc. Hence N_2 contains only one node v. Since (N_1, N_2) forms a cut of $\mathcal{H}$, v does not belong to N_1. Besides, v must be reachable from any node $u \in N_1$. For any two nodes $u, u' \in N_1$, we clearly have $u \not\rightsquigarrow_{\mathcal{H}} u'$. Thus, v is reachable from j only if there exists a hyperedge of the form $(\{ u \}, \{ v \})$ in $\mathcal{H}$.

Conversely, any hypergraph satisfying one of the two conditions admits a greatest Scc. □

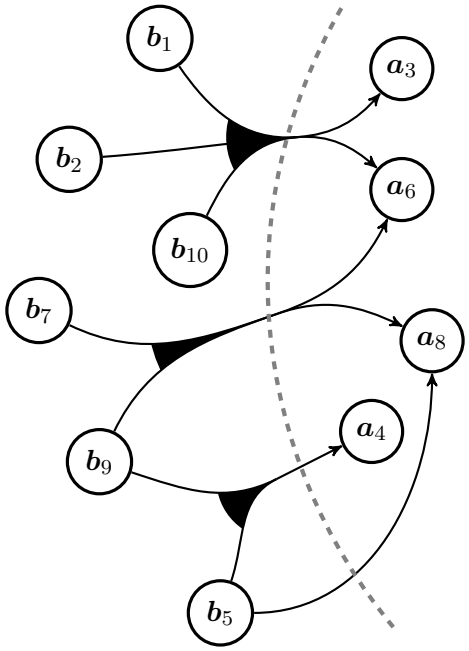


Figure 5.5: The cut $(\mathsf{supp}(\boldsymbol{b}), \mathsf{supp}(\boldsymbol{a}))$ in the directed hypergraph $\mathcal{H}((\boldsymbol{a}, \boldsymbol{b}), \mathcal{C}^\circ)$

Applying Lemma 5.12 and the characterization of extremality provided by Theorem 3.3 yields the following statement.[8]

Proposition 5.13. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical polyhedral cone generated by a finite set G, and $(\boldsymbol{a}, \boldsymbol{b}) \in \mathcal{C}^\circ$. Then $(\boldsymbol{a}, \boldsymbol{b})$ is extreme in $\mathcal{C}^\circ$ if and only if one of the two conditions holds:*

(1) either $\boldsymbol{a} = \mathbb{0}$ and $\mathsf{supp}(\boldsymbol{b}) = \{\, j \,\}$ for some $1 \leq j \leq d$,

(2) or $\mathsf{supp}(\boldsymbol{a}) = \{\, i \,\}$ for some $1 \leq i \leq d$, $i \notin \mathsf{supp}(\boldsymbol{b})$, and for all $j \in \mathsf{supp}(\boldsymbol{b})$, there exists $\boldsymbol{g} \in G$ such that

$$\boldsymbol{a}_i \boldsymbol{g}_i = \boldsymbol{b}_j \boldsymbol{g}_j > \mathbb{0} \text{ and } \boldsymbol{b}_j \boldsymbol{g}_j > \boldsymbol{b}_k \boldsymbol{g}_k \text{ for all } k \neq j. \tag{5.8}$$

Observe that the elements of the form (1) are precisely the tautologies $\boldsymbol{x}_j \geq \mathbb{0}$.

5.2.4 Resulting algorithm

We now describe an algorithm which computes the extreme rays of the polar of a cone $\mathcal{C} = \mathit{cone}(G)$ from a generating set G. It is a defined in a dual way to the algorithms discussed in Section 5.1, *i.e.* by induction on the number of vectors in G. The elementary step of this induction is described in Section 5.2.4.a, the extremality test in Sections 5.2.4.b and 5.2.4.c, and the main algorithm is provided in Section 5.2.4.d.

5.2.4.a Elementary step. We begin with a dual formulation of the elementary step given in Theorem 5.1:

[8] This result has also been established by Gaubert and Katz in [GK09b], but with a different proof.

Corollary 5.14. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical polyhedral cone generated by a finite set G, and $\boldsymbol{g} \in \mathbb{R}^d_{\max}$. Let $\mathscr{G} \subset (\mathbb{R}^d_{\max})^2$ be a generating set of the polar $\mathcal{C}^\circ$.*

Then the polar of $\mathsf{cone}(G \cup \{\, \boldsymbol{g} \,\})$ *is generated by:*

$$\{\, (\boldsymbol{a},\boldsymbol{b}) \in \mathscr{G} \mid {}^t\boldsymbol{a}\boldsymbol{g} \leq {}^t\boldsymbol{b}\boldsymbol{g} \,\} \cup \{\, ({}^t\boldsymbol{c}\boldsymbol{g})(\boldsymbol{a},\boldsymbol{b}) \oplus ({}^t\boldsymbol{b}\boldsymbol{g})(\boldsymbol{c},\boldsymbol{d}) \mid (\boldsymbol{a},\boldsymbol{b}),(\boldsymbol{c},\boldsymbol{d}) \in \mathscr{G},\ {}^t\boldsymbol{a}\boldsymbol{g} \leq {}^t\boldsymbol{b}\boldsymbol{g},\ \textit{and}\ {}^t\boldsymbol{c}\boldsymbol{g} > {}^t\boldsymbol{d}\boldsymbol{g} \,\}.$$

Proof. Observe that the polar of $\mathsf{cone}(G \cup \{\, \boldsymbol{g} \,\})$ concides with the intersection of $\mathcal{C}^\circ$ and the halfspace $\mathscr{H} \stackrel{def}{=} \{\, (\boldsymbol{a},\boldsymbol{b}) \in (\mathbb{R}^d_{\max})^2 \mid {}^t\boldsymbol{a}\boldsymbol{g} \leq {}^t\boldsymbol{b}\boldsymbol{g} \,\}$, according to Proposition 5.10. Therefore, Theorem 5.1 can be applied on the set $\mathscr{G}$ and the halfspace $\mathscr{H}$. □

5.2.4.b Extremality test. Following Proposition 5.13, we introduce a particular algorithm to evaluate the extremality of an element in the polar of the cone $\mathcal{C} = cone(G)$.

This function, called CheckExtInPolar, is presented in Figure 5.6. It takes as input an element $(\boldsymbol{a},\boldsymbol{b}) \in \mathcal{C}^\circ$ to be tested, and also a list L of additional data, called the max/arg max information. This list is formed by the quadruples $({}^t\boldsymbol{a}\boldsymbol{g}, \arg\max({}^t\boldsymbol{a}\boldsymbol{g}), {}^t\boldsymbol{b}\boldsymbol{g}, \arg\max({}^t\boldsymbol{b}\boldsymbol{g}))$ for all the elements $\boldsymbol{g} \in G$.

```
1:  procedure CheckExtInPolar((a,b), L)
2:     if a = 𝟘 and |supp(b)| = 1 then    ▷ Form (1)?
3:        return true
4:     else if |supp(a)| = 1 then    ▷ Form (2)?
5:        let i such that supp(a) = { i }
6:        if i ∉ supp(b) then
7:           R := ∅
8:           for all (v_l, arg_l, v_r, arg_r) ∈ L do
                        ▷ the loop iterates over the quadruples (ᵗag, arg max(ᵗag), ᵗbg, arg max(ᵗbg))
9:              if v_l = v_r > 𝟘 and |arg_r| = 1 then
10:                let j such that arg_r = { j }
11:                add j to R
12:             end
13:          done
14:          if |R| = |supp(b)| then
15:             return true
16:          end
17:       end
18:    end
19:    return false
20: end
```

Figure 5.6: Evaluation of the extremality in a polar cone, provided the max/arg max information

Let us discuss the principle of CheckExtInPolar. The function first checks whether $(\boldsymbol{a},\boldsymbol{b})$ is of the form (1) (Line 2) in the sense of Proposition 5.13. If not, it tries to determine whether it is of the form (2) (from Lines 4 to 18). The set R stores the elements $j \in \mathsf{supp}(\boldsymbol{b})$ which satisfies the condition given in (5.8). If it is equal to $\mathsf{supp}(\boldsymbol{b})$, then the element $(\boldsymbol{a},\boldsymbol{b})$ is of the form (2) (Line 15). Otherwise, it is not extreme in $\mathcal{C}^\circ$ (Line 19).[9]

[9]Note that the test $R = \mathsf{supp}(\boldsymbol{b})$ is replaced by the equality $|R| = |\mathsf{supp}(\boldsymbol{b})|$, since by construction, R is always a subset of $\mathsf{supp}(\boldsymbol{b})$.

We suppose that subsets of $[d]$ are encoded by hash sets. Thus the operations of adding an element or evaluating the cardinality can be supposed to be in $O(1)$.[10] Then it can be verified that the time complexity of CheckExtInPolar is in $O(d+p)$ where $p = |G|$. Indeed, computing the support of $\boldsymbol{a}$ and $\boldsymbol{b}$ can be done in $O(d)$. Besides, the loop from Lines 8 to 13 is iterated $O(p)$ times, and the amortized complexity of each iteration is in $O(1)$.

The complexity of CheckExtInPolar in $O(d+p)$ is a very important speed-up over the evaluation of the extremality criterion in the general case (which is in $O(pd\alpha(d))$). However, we have supposed that the max/arg max information has been already computed. Without optimization, they can be determined in time $O(d \times p)$. Fortunately, they can be much more efficiently determined by induction, using an optimization similar to Remark 5.3. Indeed, as in the dual algorithms, the extremality test will be applied only on combinations of elements previously computed.

5.2.4.c Computing the max/arg max **information by induction.** We now describe how to efficiently compute the max/arg max information of an element of the form $(\boldsymbol{a}, \boldsymbol{b}) = \lambda(\boldsymbol{a}', \boldsymbol{b}') \oplus \mu(\boldsymbol{a}'', \boldsymbol{b}'')$, from the max/arg max information of $(\boldsymbol{a}', \boldsymbol{b}')$ and $(\boldsymbol{a}'', \boldsymbol{b}'')$.

First observe that the extremality characterization of Proposition 5.13 *never* uses the exact value of the sets $\arg\max({}^t\boldsymbol{a}\boldsymbol{g})$ and $\arg\max({}^t\boldsymbol{b}\boldsymbol{g})$ when their cardinality is strictly greater than 1. That is why such sets will be represented by the special value $\top$. By convention, $|\top|$ evaluates to an integer greater than 2.[11]

The function Combine, presented in Figure 5.7, returns the combination $\lambda(\boldsymbol{a}', \boldsymbol{b}') \oplus \mu(\boldsymbol{a}'', \boldsymbol{b}'')$ and its max/arg max information, starting from the elements $(\boldsymbol{a}', \boldsymbol{b}')$ and $(\boldsymbol{a}'', \boldsymbol{b}'')$ and their max/arg max information L' and L''.

1: **procedure** Combine(λ, $((\boldsymbol{a}', \boldsymbol{b}'), L')$, μ, $((\boldsymbol{a}'', \boldsymbol{b}''), L'')$)
2: $\quad \boldsymbol{a} := \lambda\boldsymbol{a}' \oplus \mu\boldsymbol{a}''$, $\boldsymbol{b} := \lambda\boldsymbol{b}' \oplus \mu\boldsymbol{b}''$
3: $\quad L := [\,]$
4: $\quad$ **for all** $(v'_l, \arg'_l, v'_r, \arg'_r), (v''_l, \arg''_l, v''_r, \arg''_r) \in (L', L'')$ **do**
5: $\qquad (v_l, \arg_l) := \begin{cases} (\lambda v'_l, \arg'_l) & \text{if } \lambda v'_l > \mu v''_l \\ (\mu v''_l, \arg''_l) & \text{if } \lambda v'_l < \mu v''_l \\ (\mu v''_l, \textsc{Union}(\arg'_l, \arg''_l)) & \text{otherwise} \end{cases}$
6: $\qquad (v_r, \arg_r) := \begin{cases} (\lambda v'_r, \arg'_r) & \text{if } \lambda v'_r > \mu v''_r \\ (\mu v''_r, \arg''_r) & \text{if } \lambda v'_r < \mu v''_r \\ (\mu v''_r, \textsc{Union}(\arg'_r, \arg''_r)) & \text{otherwise} \end{cases}$
7: $\qquad$ append $(v_l, \arg_l, v_r, \arg_r)$ at the end of the list L
8: $\quad$ **done**
9: $\quad$ **return** $((\boldsymbol{a}, \boldsymbol{b}), L)$
10: **end**

Figure 5.7: Combining elements and the associated information

It traverses the list denoted by (L', L''), which is formed by the couples (l'_i, l''_i), where l'_i and l'_i are the i-th elements of the lists L' and L'' respectively. The quadruples $(v_l, \arg_l, v_r, \arg_r)$

[10] Like in the implementation of [Fil08], we assume that each set is provided with a internal counter representing its cardinality.

[11] In the abstract interpretation jargon, we would say that subsets of the arg max are over-approximated using the abstract domain of constants [Kil73].

are built at Lines 5 and 6, using the following principle: for all $\boldsymbol{g} \in G$,

$$\arg\max({}^t\boldsymbol{a}\boldsymbol{g}) = \begin{cases} \arg\max({}^t\boldsymbol{a}'\boldsymbol{g}) & \text{if } \lambda({}^t\boldsymbol{a}'\boldsymbol{g}) > \mu({}^t\boldsymbol{a}''\boldsymbol{g}), \\ \arg\max({}^t\boldsymbol{a}''\boldsymbol{g}) & \text{if } \lambda({}^t\boldsymbol{a}'\boldsymbol{g}) < \mu({}^t\boldsymbol{a}''\boldsymbol{g}), \\ \arg\max({}^t\boldsymbol{a}'\boldsymbol{g}) \cup \arg\max({}^t\boldsymbol{a}''\boldsymbol{g}) & \text{otherwise.} \end{cases}$$

A similar identity holds for $\arg\max({}^t\boldsymbol{b}\boldsymbol{g})$, replacing $\boldsymbol{a}'$ and $\boldsymbol{a}''$ by $\boldsymbol{b}'$ and $\boldsymbol{b}''$ respectively. The union (third case) is computed thanks to the function UNION (Figure 5.8). The latter precisely computes the result except when one of the arguments has more than 2 elements (in which case it is encoded by the value $\top$, see Line 4). Its complexity is in $O(1)$.

```
1: procedure UNION(S_1, S_2)
2:     if S_1 = ∅ then return S_2
3:     if S_2 = ∅ then return S_1
4:     if S_1 = ⊤ or S_2 = ⊤ then return ⊤
5:     let (i, j) such that S_1 = { i } and S_2 = { j }
6:     if i ≠ j then return ⊤
7:     else return { i }
8: end
```

Figure 5.8: $O(1)$ union function

The time complexity of COMBINE is in $O(d+p)$, where p is the number of elements in G (also equal to the length of the lists L' and L''). Indeed, $\lambda\boldsymbol{a}' \oplus \mu\boldsymbol{a}''$ and$\lambda\boldsymbol{b}' \oplus \mu\boldsymbol{b}''$ are computed in $O(d)$ time. Besides, the loop over the list (L', L'') performs precisely p iterations, each of which has a complexity in $O(1)$.

5.2.4.d Main algorithm. The resulting algorithm COMPUTEEXTRAYSPOLAR is given in Figure 5.9. Given a finite set $G \subset \mathbb{R}^d_{\max}$ of p vectors, encoded as a $(d \times p)$-matrix, COMPUTEEXTRAYSPOLAR(G, p) recursively computes the set of the scaled extreme elements of the polar of $\mathsf{cone}(G)$ and their associated max/arg max information.

The structure is akin to COMPUTEEXTRAYS:

(i) when $p = 0$, then $\mathsf{cone}(G)$ is reduced to the null vector $\mathbb{0}$, hence its polar is equal to the whole set $(\mathbb{R}^d_{\max})^2$. Therefore, the canonical basis of $(\mathbb{R}^d_{\max})^2$ is returned. It consists of the elements of the form $(\boldsymbol{\epsilon}^i), \mathbb{0})$ and $(\mathbb{0}, \boldsymbol{\epsilon}^i)$, for $1 \leq i \leq d$. The associated lists are all empty since G is empty.

(ii) when $p > 0$, the set G is first split into a set H of $(p-1)$ vectors and a vector $\boldsymbol{g}$. The function COMPUTEEXTRAYSPOLAR is called recursively on H. For each couple $((\boldsymbol{a}, \boldsymbol{b}), L)$ returned, the max/arg max information relative to $\boldsymbol{g}$ is added to L (Line 8), in time $O(d)$.

Then the combinations provided by Corollary 5.14 are examined from Lines 11 to 18. Each combination is built thanks to the function COMBINE in time $O(d+p)$ (Line 14), and the extremality is checked at Line 15 in time $O(d+p)$.

The normalization of the coefficients λ, μ of the combination into λ', μ' allows to ensure that the combination $\lambda'(\boldsymbol{a}, \boldsymbol{b}) \oplus \mu'(\boldsymbol{c}, \boldsymbol{d})$ is a scaled element (provided that $(\boldsymbol{a}, \boldsymbol{b})$ and $(\boldsymbol{c}, \boldsymbol{d})$ are also scaled).

```
1: procedure ComputeExtRaysPolar(G, p)    ▷ G ∈ ℝ_max^{d×p}
2:    if p = 0 then    ▷ Base case
3:       return the list formed by the elements ((ε^i), 0), [ ]) and ((0, ε^i), [ ]) for i = 1, ..., d
4:    else    ▷ Inductive case
5:       H := (p − 1) first columns of G, g := last column of G
6:       𝒢 := ComputeExtRaysPolar(H, p − 1)
7:       for all ((a, b), L) ∈ 𝒢 do
8:          append (ag, arg max(ag), bg, arg max(bg)) to the list L
9:       done
10:      𝒢^≤ := { ((a, b), L) ∈ 𝒢 | ^t ag ≤ ^t bg }, 𝒢^> := { ((c, d), L′) ∈ 𝒢 | ^t cg > ^t dg }, ℋ := 𝒢^≤
11:      for all ((a, b), L) ∈ 𝒢^≤ and ((c, d), L) ∈ 𝒢^> do
12:         λ := cg, μ := bg
13:         λ′ := (λ ⊕ μ)^{−1}λ, μ′ := (λ ⊕ μ)^{−1}μ
14:         ((α, β), Λ) := Combine(λ′, ((a, b), L), μ′, ((c, d), L′))
15:         if CheckExtInPolar((α, β), Λ) = true then    ▷ Extremality test
16:            append ((α, β), Λ) to ℋ
17:         end
18:      done
19:   end
20:   return ℋ
21: end
```

Figure 5.9: Computing the extreme rays of the polar of a tropical cone

Like in ComputeExtRays, it can be verified that the elements of $\mathscr{G}^{\leq}$ are all extreme in the polar of $\mathsf{cone}(G)$.

We can now characterize the time complexity of the inductive step (from Lines 7 to 18), in terms of the size of the intermediate set $\mathscr{G}$:

Proposition 5.15. *The worst case time complexity of the inductive step of* ComputeExtRaysPolar *is* $O((d+p)|\mathscr{G}|^2)$.

This must be compared to the complexity bound in $O(pd\alpha(d)|G|^2)$ of the same step in the dual algorithm ComputeExtRays. As a result, we have shown that computing an external representation of a polyhedral cone from an internal one can be performed more efficiently than computing an internal description from an external one. This is a major algorithmic difference with the classical algorithms, which have exactly the same theoretical complexity.

Following Remark 5.5, the algorithm ComputeExtRaysPolar necessarily returns elements $(\boldsymbol{a}, \boldsymbol{b})$ which represent redundant inequalities.

This algorithm is also implemented in the library TPLib.

5.2.5 Comparison with alternative approaches

As far as we know, the only existing algorithms to pass from an external representation to an internal one rely on dual algorithms performing the inverse operation. More precisely, given an algorithm ExternalToInternal which passes from the external to the internal descriptions, we can define its dual InternalToExternal as follows:

```
1: procedure InternalToExternal(G, p)    ▷ G ∈ ℝ_max^{d×p}
2:     A := (ᵗG  𝟘_{p×d}), B := (𝟘_{p×d}  ᵗG)
3:     R := ExternalToInternal(A, B, p)
4:     return { (a, b) | (a; b) ∈ R }
5: end
```

The argument G is a generating set, assimilated to the matrix formed by its elements, and $\mathbb{0}_{p\times d}$ denoted the identically null matrix of size $p \times d$.

The correctness of this construction is ensured by Proposition 5.10, which states that $(\boldsymbol{a}, \boldsymbol{b})$ belongs to the polar of the cone $\mathsf{cone}(G)$ if and only if it is a solution of the system:

$$\begin{pmatrix} {}^tG & \mathbb{0}_{p\times d} \end{pmatrix} \begin{pmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{pmatrix} \leq \begin{pmatrix} \mathbb{0}_{p\times d} & {}^tG \end{pmatrix} \begin{pmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{pmatrix}.$$

When ExternalToInternal is instantiated by the algorithms ComputeExtRays or OldComputeExtRays, the function InternalToExternal indeed returns the set of the scaled extreme elements of the polar of $\mathsf{cone}(G)$.[12] Table 5.3 provides the complexity of InternalToExternal using these two instantiations, and compare it to ComputeExtRaysPolar.[13] In both cases, InternalToExternal is less efficient, since it does not use the particular extremality criterion established in Section 5.2.3.

Table 5.3: Comparison of the complexity

	Comp.ExtRaysPolar	InternalToExternal	
		ComputeExtRays	OldComp.ExtRays
inductive step	$O((p+d)\lvert\mathscr{G}\rvert^2)$	$O(pd\alpha(2d)\lvert\mathscr{G}\rvert^2)$	$O(d\lvert\mathscr{G}\rvert^4)$
ratio	$O(1)$	$O\left(\frac{pd\alpha(2d)}{p+d}\right)$	$O\left(\frac{d\lvert\mathscr{G}\rvert^2}{p+d}\right)$
overall	$O(p(p+d)\mathscr{G}_{\max}^2)$	$O(p^2d\alpha(2d)\mathscr{G}_{\max}^2)$	$O(pd\mathscr{G}_{\max}^4)$

5.3 The number of extreme elements in tropical polyhedra

The complexity of the algorithms that we have developed in the two previous sections depends on the size of the set of the extreme elements in the intermediate cones. In this section, we propose to give some elements about the order of magnitude of the size of these sets in the worst case.

An entire chapter could be dedicated to the problem of determining an upper bound on the number of extreme elements in tropical polyhedra or polyhedral cones. It has been thoroughly discussed by Allamigeon, Gaubert, and Katz in [AGK10]. In Sections 5.3.1 to 5.3.4, we include a summary of the main results:

- Section 5.3.1 shows that the number of extreme rays in tropical polyhedral cones can be bounded by a result akin to its classical analogue.

[12] In particular, Allamigeon *et al.* used the instantiation by OldComputeExtRays in [AGG08].

[13] The notation $\mathscr{G}_{\max}$ represents the maximal cardinality of the intermediate generating sets.

- in Section 5.3.2, we introduce the *signed cyclic polyhedral cones*, which appear as natural candidates to maximize the number of extreme rays.
- in Section 5.3.3, we show that the classical versions of the signed cyclic polyhedral cones have in comparison much more extreme rays, and that they reach the bound established in Section 5.3.1.
- Section 5.3.4 provides some lower and upper bound on the number of extreme rays in tropical signed cyclic polyhedral cones. This shows in particular that the bound of Section 5.3.1 is tight for a fixed number of constraints, and a dimension tending to $+\infty$.

Finally, we develop in Section 5.3.5 an original result on the maximal number of extreme rays in polar cones.

5.3.1 A first McMullen-type bound

The classical version of the upper bound problem is known to be solved by a remarkable result of McMullen [McM70]:

Theorem 5.5 (Upper bound theorem [McM70])**.** *Among all polytopes in $\mathbb{R}^d$ with p extreme points, the cyclic polytope maximizes the number of faces of each dimension.*

Remember that the cyclic polytope with p extreme points in $\mathbb{R}^d$ is the convex hull of p points of the form $(t_i, t_i^2, \dots, t_i^d)$ $(i = 1, \dots, p)$ for a given choice of p pairwise distinct elements $t_1, \dots, t_p$.

In particular, the number of facets (*i.e.* the faces of dimension $d-1$) is known to be at most

$$U(p,d) \stackrel{def}{=} \begin{cases} \binom{p - \lfloor d/2 \rfloor}{\lfloor d/2 \rfloor} + \binom{p - \lfloor d/2 \rfloor - 1}{\lfloor d/2 \rfloor - 1} & \text{for } d \text{ even,} \\ 2\binom{p - \lfloor d/2 \rfloor - 1}{\lfloor d/2 \rfloor} & \text{for } d \text{ odd.} \end{cases}$$

By duality, the same upper bound applies to the number of extreme points of a d-dimensional polytope defined by a system of p inequalities.

In the tropical setting, we have established the correctness of the McMullen-type following bound:

Theorem 5.6 ([AGK10, Theorem 1])**.** *The number of extreme rays of a tropical polyhedral cone in $\mathbb{R}^d_{\max}$ defined by a system of p inequalities cannot exceed $U(p+d, d-1)$.*

The number $p+d$ instead of p for the number of constraints can be explained intuitively by the saturation of at most d implicit inequalities $\boldsymbol{x}_i \geq \mathbb{0}$ for each $i \notin \mathsf{supp}(\boldsymbol{x})$. The number $d-1$ instead of d for the dimension reflects the fact that the result is relative to cones. However, tropical homogenization allows to derive a similar result for tropical polyhedra.

Since faces are not yet clearly defined in the tropical setting, the proof of Theorem 5.6 does not rely on the f-vector theory (unlike its classical counterpart). Instead, a different approach is used, based on a deformation argument in which the tropical polyhedron is seen as a degenerate limit of a sequence of classical polyhedra. The bound is then obtained by applying the classical upper bound theorem on the polyhedra of the sequence.

5.3.2 Signed cyclic polyhedral cones

We may now ask whether the bound $U(p+d, d-1)$ is attained. In the classical case, the dual polar of a cyclic polytope with p extreme points maximizes the number of extreme points among all the polytopes of dimension d defined by p inequalities. We can define by analogy natural candidates to maximize the number of extreme elements in the tropical case. The notion of polar of cyclic polytopes is generalized to the two-sided nature of tropical linear inequalities. To this aim, a *sign pattern* (ϵ_{ij}) determines the distribution of the coefficients between the left and the right members.

Definition 5.2. Let $t_1, \ldots, t_p$ be p scalars $\mathbb{0} < t_1 < \ldots < t_p$, and let $\epsilon = (\epsilon_{ij})$ be a $(p \times d)$-*sign pattern*, *i.e.* a collection of signs $\epsilon_{ij} \in \{+, -\}$ $(1 \leq i \leq p,\ 1 \leq j \leq d)$.

The *signed cyclic polyhedral cone* with sign pattern (ϵ_{ij}) is the tropical cone of $(\mathbb{R}^d_{\max})^2$ generated by the couples (C_k^+, C_k^-), where $C_k^+ = (C_{ki}^+)$ and $C_k^- = (C_{kj}^-)$ are elements of $\mathbb{R}^d_{\max}$ defined by:

$$C_{ki}^+ \stackrel{def}{=} \begin{cases} t_k^i & \text{if } \epsilon_{ki} = +, \\ \mathbb{0} & \text{otherwise,} \end{cases} \qquad C_{kj}^- \stackrel{def}{=} \begin{cases} t_k^j & \text{if } \epsilon_{kj} = -, \\ \mathbb{0} & \text{otherwise.} \end{cases}$$

The notation x^i refers to the tropical exponentiation.

The dual polar of the signed cyclic polyhedral cone will be denoted by $\mathcal{K}(\epsilon)$ (or $\mathcal{K}$ for the sake of brevity). It is defined as the set of the solution $\boldsymbol{x} \in \mathbb{R}^d_{\max}$ of the system of p inequalities:

$$C_k^- \boldsymbol{x} \leq C_k^+ \boldsymbol{x}, \quad k = 1, \ldots, p.$$

Example 5.6. Suppose that $d = 4$, $p = 3$, $t_1 = 0$, $t_2 = 1$, $t_3 = 2$, and that the sign pattern ϵ is defined as $\left[\begin{smallmatrix} + & - & + & - \\ + & - & + & - \\ + & - & + & - \end{smallmatrix}\right]$. Then the cone $\mathcal{K}(\epsilon)$ is the set of the elements $(x, y, z, t) \in \mathbb{R}^4_{\max}$ such that:

$$\begin{pmatrix} -\infty & 0 & -\infty & 0 \\ -\infty & 1 & -\infty & 3 \\ -\infty & 2 & -\infty & 6 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix} \leq \begin{pmatrix} 0 & -\infty & 0 & -\infty \\ 0 & -\infty & 2 & -\infty \\ 0 & -\infty & 4 & -\infty \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix}.$$

Given a sign pattern ϵ, the number of extreme rays of the dual polar of signed cyclic polyhedral cones does not depend on the choice of the scalars $t_1, \ldots, t_p$. More precisely, such extreme rays can be shown to be in one-to-one correspondence with tropically allowed paths for the pattern ϵ:

Definition 5.3. Let $\epsilon = (\epsilon_{ij})$ be $(p \times d)$-sign pattern. A lattice path is said to be *tropically allowed* for the sign pattern ϵ if the following conditions are satisfied:

(i) every sign occurring on the initial vertical segment, except possibly the sign at the bottom of the segment, is positive.

(ii) every sign occurring on the final vertical segment, except possibly the sign at the top of the segment, is positive.

(iii) every sign occurring in some other vertical segment, except possibly the signs at the top and bottom of this segment, is positive.

(iv) for every horizontal segment, the pair of signs consisting of the signs of the leftmost and rightmost positions of the segment is of the form $(+, -)$ or $(-, +)$.

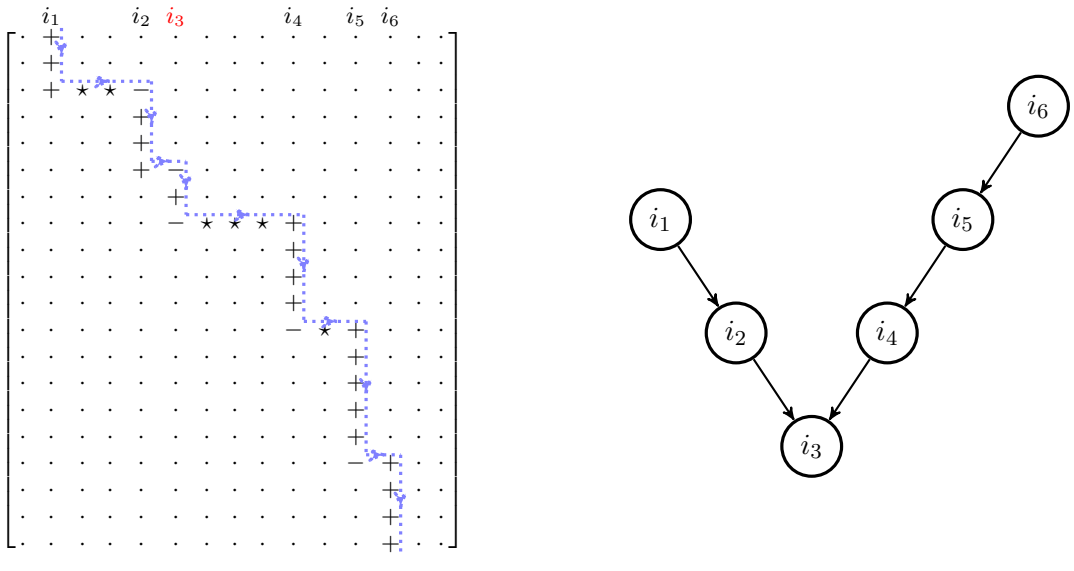


Figure 5.10: A tropically allowed lattice path (left), and the corresponding directed hypergraph (right)

(v) as soon as a pair $(-,+)$ occurs as the pair of extreme signs of some horizontal segment, the pairs of signs corresponding to all the horizontal segments below this one must also be equal to $(-,+)$.

Example 5.7. Figure 5.10 provides an example of tropically allowed lattice path. The symbol "$\star$" indicates positions of the path whose sign is irrelevant. The positions which do not belong to the path are indicated by the symbol "$\cdot$".

Theorem 5.7 ([AGK10, Theorem 2])**.** *The extreme rays of the polar of a signed cyclic polyhedral cone are in one to one correspondence with tropically allowed lattice paths.*

Example 5.8. Figure 5.11 provides two examples of polars of signed cyclic polyhedral cones for $d = 3$. The two cones are defined by $p = 2$ and $p = 5$ inequalities respectively, and, for all $1 \leq i \leq p$, $t_i = i - 1$ and $\epsilon_{ij} = -$ if and only if $j = 2$. In other words, the first cone is associated to the sign pattern $\left[\begin{smallmatrix} + & - & + \\ + & - & + \end{smallmatrix}\right]$, and its polar is defined as the set of elements $(x, y, z) \in \mathbb{R}^3_{\max}$ such that:

$$\begin{pmatrix} -\infty & 0 & -\infty \\ -\infty & 1 & -\infty \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq \begin{pmatrix} 0 & -\infty & 0 \\ 0 & -\infty & 2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

The extreme rays are depicted by blue points. For the first cone, a representative of each extreme ray is provided, and the corresponding tropically allowed path is given beside.

5.3.3 Comparison with the classical case

In the classical case, given p real scalars $t_1 < \ldots < t_p$ and a $(p \times d)$-sign pattern, the polar of the signed cyclic polyhedral cones can be defined as well:

$$\mathcal{K}'(\epsilon) \stackrel{def}{=} \left\{ \boldsymbol{x} \in \mathbb{R}^d \mid \boldsymbol{x} \geq 0 \,, C\boldsymbol{x} \geq 0 \right\} \quad \text{where } C_{ij} = \epsilon_{ij} t_i^{j-1}.$$

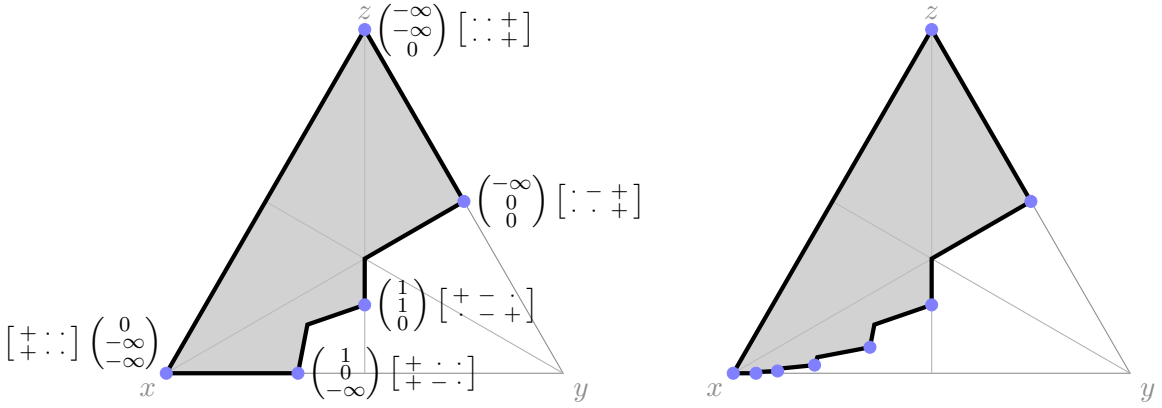


Figure 5.11: The polars of two signed cyclic polyhedral cones in $\mathbb{R}^3_{\max}$.

The number of extreme rays of this classical cone is greater than the number of extreme rays of the corresponding tropical cone. Indeed, the extreme rays of the former can be shown to be in one-to-one correspondence with the non-tropically allowed paths in the sign pattern:

Definition 5.4. Let $\epsilon = (\epsilon_{ij})$ be $(p \times d)$-sign pattern. A lattice path is said to be *non-tropically allowed* for the sign pattern ϵ if Conditions (i)-(iv) of Definition 5.3 are satisfied.

Theorem 5.8 ([AGK10, Theorem 5])**.** *If the ratios $t_2/t_1, \ldots, t_{p+1}/t_p$ are sufficiently large, the extreme rays of the classical polar $\mathcal{K}'(\epsilon)$ are in one to one correspondence with the non-tropically allowed lattice paths for the sign pattern (ϵ_{ij}).*

This result is a major difference with Theorem 5.7. It can be informally explained by the fact that the characterization of the extremality of a ray in tropical cones not only relies on the saturated inequalities like in the classical case, but also on the existence of a type in the sense of Proposition 3.1, and subsequently of a greatest Scc in the tangent directed hypergraph.

More precisely, consider an extreme ray of a tropical polar $\mathcal{K}(\epsilon)$, and a tropically allowed path in the sign pattern ϵ. Let $i_1 < \ldots < i_q$ be the indexes of the columns of the signs $+$ or $-$ located at the left or right ends of the horizontal segments of the path (see the top line of the sign pattern in Figure 5.10). We claim that the support of the ray coincides with the set of the i_j. Now, consider the first occurence of the pair $(-,+)$ as in Condition (v), and let t be index of the column of the sign $-$.[14] In Figure 5.10, t is equal to the index i_3. We claim that the form of the hypergraph associated to the extreme ray is a "V", where the bottom of the V is precisely the node t. This is illustrated in the right side of Figure 5.10. This constraint on the form of the hypergraph, which is provided by Condition (v) explains the combinatorial difference between the results of Theorem 5.7 and Theorem 5.8.

Furthermore, in the classical setting, the polar of the signed cyclic polyhedral cone can be proved to reach the bound of Theorem 5.6:

Theorem 5.9 ([AGK10, Theorem 6])**.** *The number of extreme rays of the classical polar $\mathcal{K}'(\epsilon)$, with sign pattern $\epsilon_{ij} = -$ if and only if j is even, is precisely $U(p+d, d-1)$.*

[14] If no such pair occurs, t can be equivalently defined as the index of the column of $-$ of the last $(+,-)$ horizontal pair.

5.3.4 The number of extreme rays in signed cyclic polyhedral cones

Let us denote by $N^{\mathrm{tpath}}(\epsilon)$ (resp. $N^{\mathrm{path}}(\epsilon)$) the number of tropically (resp. non-tropically) allowed lattice paths for the sign pattern ϵ. Recall that $N^{\mathrm{trop}}(p,d)$ denotes the maximal number of extreme rays of a tropical cone in dimension d defined by p inequalities. We have shown the following relations:

$$\max_{\epsilon\in\{\pm1\}^{p\times d}} N^{\mathrm{tpath}}(\epsilon) \leq N^{\mathrm{trop}}(p,d) \leq U(p+d,d-1) = \max_{\epsilon\in\{\pm1\}^{p\times d}} N^{\mathrm{path}}(\epsilon). \tag{5.9}$$

It has been conjectured in [AGK10] that the polar of signed cyclic polytopes maximize the number of extreme rays (*i.e.* the two leftmost terms in (5.9) are equal). Albeit this conjecture could have not been proved, it has been established that the maximal number of extreme rays is reached by polyhedral cones which are defined as the intersection of halfspaces in general position (in particular, like in signed cyclic polyhedral cones).

Upper and lower bounds on the number of extreme rays of the polar of signed cyclic polyhedral cones have been established. They confirm our assumption on the worst-case size of the set G of extreme rays of the intermediary cone made in Sections 5.1 and 5.2.

Proposition 5.16 ([AGK10, Propositions 2, 3, and 4])**.** *For every p, d,*

$$N^{tpath}(p,d) \leq (p(d-1)+1)2^{d-1}.$$

For $p \geq 2d$,

$$N^{tpath}(p,d) \geq (p-2d+7)(2^{d-2}-2). \tag{5.10}$$

For $d \geq 2p+1$, we have

$$N^{tpath}(p,d) \geq U(d,d-p-1). \tag{5.11}$$

Equation (5.11) implies that the bound $U(p+d,d-1)$ of Theorem 5.6 is asymptotically reached when $d \to +\infty$ and p is fixed, since in that case, $U(p+d,d-1) \sim U(d,d-p-1)$. This lower bound is obtained using the alternate sign pattern ϵ defined by $\epsilon_{ij} = -$ if and only if $i+j$ is odd. The other lower bound provided in (5.10) is obtained with a pattern in which the $+$ form a kind of natural symbol shape (♮). Finding a general formula for maximizing patterns is still an open problem.

Example 5.9. An example of signed cyclic polyhedral cone equipped with the "natural" pattern for $d = 4$ and $p = 10$ is provided in Figure 5.12. The fourth coordinate is assimilated to an affine dimension, so that each element (x,y,z,t) is represented by the point $(x-t, y-t, z-t)$ in $\mathbb{R}^3$ when $t \neq -\infty$.

The cone contains 24 extreme rays, depicted by red points. It has been drawn using PolyMake, which also generates all the pseudo-vertices (*i.e.* the vertices of the corresponding polyhedral complex in the sense of [DS04]), see [Jos08] for further details. The pseudo-vertices which are not extreme rays are represented by yellow points. As explained in Chapter 1, they are much more numerous: here, 1215 on top of the 24 extreme elements.

5.3.5 The number of extreme rays in polar cones

Polar cones are defined as tropical cones in $(\mathbb{R}^n_{\max})^2$. However, as stated in Proposition 5.13, their extreme rays are of a very special form, so that they do not have as many extreme rays as arbitrary cones of $(\mathbb{R}^n_{\max})^2$:

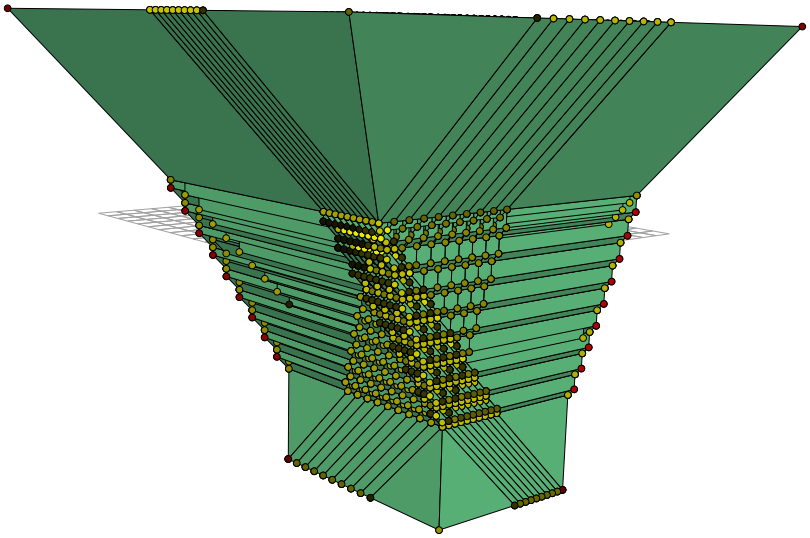


Figure 5.12: Signed cyclic polyhedral cone equipped with the "natural" pattern, for $d = 4$ and $p = 10$

Theorem 5.10. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ ($d \geq 2$) be a tropical polyhedral cone generated by a finite set G. Then the number of extreme rays in the polar $\mathcal{C}^\circ$ is bounded by $d \cdot N^{trop}(p, d)$, where $p = |G|$.*

Proof. According to Proposition 5.13, extreme elements $(\boldsymbol{a}, \boldsymbol{b})$ of the polar $\mathcal{C}^\circ$ have two possible forms:

- either $\boldsymbol{a} = \mathbb{0}$ and $\mathsf{supp}(\boldsymbol{b})$ is reduced to a singleton. Such elements generate at most d rays.
- or there exists i such that $\mathsf{supp}(\boldsymbol{a}) = \{\, i \,\}$ and $i \notin \mathsf{supp}(\boldsymbol{b})$. They will be said to be *of kind i*. Let ϕ_i be the function which maps such elements to $\boldsymbol{x} \in \mathbb{R}^d_{\max}$ defined by $\boldsymbol{x}_j = \boldsymbol{b}_j$ if $j \neq i$, and $\boldsymbol{x}_i = \boldsymbol{a}_i$. Let $\mathcal{D} \subset \mathbb{R}^d_{\max}$ be the polyhedral cone defined by the following system of p inequalities:

$$\boldsymbol{g}^k_i \boldsymbol{x}_i \leq \bigoplus_{j \neq i} \boldsymbol{g}^k_j \boldsymbol{x}_j \text{ for all } 1 \leq k \leq p, \tag{5.12}$$

  where the $\boldsymbol{g}^k$ are the elments of G, and $\boldsymbol{g}^k = (\boldsymbol{g}^k_i)$ for all k. Then clearly ϕ_i is an injective function from the elements of $\mathcal{C}^\circ$ of kind i to $\mathcal{D}$. Since ϕ_i is also a morphism for the addition in $(\mathbb{R}^d_{\max})^2$, then $(\boldsymbol{a}, \boldsymbol{b})$ is extreme in $\mathcal{C}^\circ$ if and only if $\phi_i(\boldsymbol{a}, \boldsymbol{b})$ is extreme in $\mathcal{D}$. Let $\boldsymbol{x} = \phi_i(\boldsymbol{a}, \boldsymbol{b})$ for some extreme element $(\boldsymbol{a}, \boldsymbol{b})$ of kind i. The cone $\mathcal{D}$ admits $d-1$ elements extreme rays generated by the elements $\boldsymbol{\epsilon}^j$ for $j \neq i$. Since $\boldsymbol{x}_i \neq \mathbb{0}$, these extreme rays cannot be generated by $\boldsymbol{x}$. Consequently, the extreme elements $(\boldsymbol{a}, \boldsymbol{b})$ of kind i cannot generate more than $N^{\mathrm{trop}}(p, d) - (d-1)$ extreme rays in $\mathcal{C}^\circ$.

As a result, the number of extreme rays in $\mathcal{C}^\circ$ is bounded by:

$$d + d \cdot (N^{\mathrm{trop}}(p, d) - (d-1)) \leq d \cdot N^{\mathrm{trop}}(p, d). \qquad \square$$

Remark 5.10. Observe that the system given in (5.12) has a particular form. Thus it may be possible to refine the result of Theorem 5.10 using a tighter bound on the tropical cones defined by such systems.

5.4 Conclusion of the chapter

Summary of the contributions. We have first defined the algorithm ComputeExtRays, which computes the set of the scaled extreme elements of a polyhedral cone defined by means of a system of inequalities, and is based on the incremental *tropical double description method*. Thanks to the extremality criterion of Chapter 3 and the associated algorithm of Chapter 4, it is theoretically more efficient than the existing solutions. Besides, the experiments shows that it is able to scale up to instances which were previously by far inaccessible.

We have then developed the second algorithm, ComputeExtRaysPolar, which determines the set of the scaled extreme elements of the polar of a cone defined by means of generating set. In particular, we have derived from the results of Chapter 3 a new characterization of extremality in polar cones, which is simpler than in the general case. As a consequence, ComputeExtRaysPolar is equipped with a more efficient extremality test, so that its complexity is better that the alternative approaches.

Both algorithms have been implemented in the Tropical Polyhedra Library TPLib (in the OCaml module `Tplib_core`).

Moreover, we have studied the problem of determining an upper bound on the number of extreme elements in tropical polyhedra and polyhedral cones. While this problem is still open, we have made important advances, first by establishing a McMullen-type bound, and second by identifying a class of tropical polyhedral cones with a large number of extreme rays. This class allows to theoretically confirm the superiority of the algorithms ComputeExtRays and ComputeExtRaysPolar over the existing methods. Besides, the cones of this class have fewer extreme rays than their analogues in the classical setting.

The order of magnitude of the classical upper bound $U(p,d)$ is known as $O(p^{\lfloor d/2 \rfloor})$. Using the results of Section 5.3, we can now give upper bounds on the complexity of the algorithms ComputeExtRays and ComputeExtRaysPolar:

Corollary 5.17. *Given $A, B \in \mathbb{R}_{\max}^{p\times d}$, the time complexity of* ComputeExtRays(A,B,p) *is bounded by:*

$$\begin{cases} O(p^2 d\alpha(d)(p+d)^{d-2}) & \text{if } d \text{ is even,} \\ O(p^2 d\alpha(d)(p+d)^{d-1}) & \text{if } d \text{ is odd.} \end{cases}$$

Given $G \in \mathbb{R}_{\max}^{d\times p}$, the time complexity of ComputeExtRaysPolar(G,p) *is bounded by:*

$$\begin{cases} O(pd^2(p+d)^{d-1}) & \text{if } d \text{ is even,} \\ O(pd^2(p+d)^{d}) & \text{if } d \text{ is odd.} \end{cases}$$

When p is fixed and $d \to +\infty$, the bound of Theorem 5.6 is tight. In that case, the bounds of Corollary 5.17 can be seen as representative of the worst case complexity of the algorithms ComputeExtRays and ComputeExtRaysPolar. However, this may not be true for a choice of d and p such that $d \ll p$.

Future work. Even if the algorithms ComputeExtRays and ComputeExtRaysPolar represent positive advances, the algorithmics of tropical polyhedra is still in its first stages. In particular, all the approaches discussed here are incremental, and follows the principle of the double description method. We may ask whether other classical approaches, such as the Beneath-and-Beyond algorithm, pivoting algorithms (such as [AF92, AF96]), or optimal algorithms (see for instance [Cha93]) can be adapted to the tropical setting. However, it is probable that further advances will require to have a better understanding of the notion of faces in tropical polyhedra (see the conjectures of Develin and Yu on this topic [DY07]).

The question of the "tropical upper bound" is also a big challenge. Following the idea of Theorem 5.10, it could be interesting to study in details the relation between the upper bound on polyhedral cones, and the upper bound on the polar cones. Besides, the latter may lead to a different class of maximizing candidates, which could maybe help to identify another class in the general case.

Part II

Application to static analysis by abstract interpretation

CHAPTER 6

Introduction to static analysis by abstract interpretation

This chapter provides a practical introduction to static program analysis by abstract interpretation. In Section 6.1, we define the target programming language. It is rather simple, but includes some of the essential features of the languages C or C++ relative to memory manipulations, such as dynamic allocations of arrays and manipulations of strings. In Section 6.2, the semantics of the language is introduced. Section 6.3 presents the theoretical framework of abstract interpretation. It is then applied to the construction of two abstract semantics, in Sections 6.4 and 6.5.

6.1 Kernel language

Programs written in real (high-level) programming languages are hard to analyze because of the complexity and the richness of the languages. For instance, the C language has many diverse constructs, some with implicit semantics information, some with unspecified or architecture and compiler-dependent behaviors, and some other being redundant. This is the reason why, for the sake of concision, we restrict ourselves to a simpler kernel language, which nevertheless embodies some of C essential paradigms. Translation from the C language to our kernel language is not explained here, as it is fairly obvious (although quite tedious). Some intermediate forms such as [NMRW02] and especially [HL08], and their associated compilers are good starting points for such a translation. In Section 6.1.1, we describe the fundamental

principles of the kernel language which we will analyze. We define its syntax in Section 6.1.2.

6.1.1 Principles of the language

In our kernel language, data are of two kinds: either integers, or array of integers. Type castings are forbidden, so that it can be assumed that the type of each variable is statically known, and that the set of integer variables and of arrays are disjoint. The (finite) set of all variables, and the (disjoint) sets of integer variables and arrays are respectively denoted by Vars, IntVars, and ArrayVars.

We suppose that there exists a function read(), which returns an integer provided by an external source, such as standard input, a file, or a network. It behaves similarly to the function `getchar` in the C language. In the absence of further information on the source, the returned value is assumed to be a random integer.

Following standard C compilation conventions, we suppose that the memory is split in two disjoint parts: the stack and the heap. The stack stores the values of integer variables, which are supposed to be statically allocated data (this corresponds to variable declaration in C). In contrast, the heap stores the content of arrays. The latter are allocated dynamically by a call to the function malloc. We assume that newly allocated data start with arbitrary values (this is slightly different from the C language where global variables are, by default, initialized to the value 0 with most compilers).

For the sake of simplicity, function definitions and function calls are not considered (except calls to the functions read() and malloc).

6.1.2 Syntax of the language

The syntax of the language is given in Figure 6.1. The symbol x refers to an element of IntVars, while t, t_1, and t_2 belong to ArrayVars.

A program is defined as a sequence of array allocations and commands. Commands allow to control the execution flow, for instance using conditionals or loops. They also include *instructions*, whose role is to manipulate the content of the memory. Observe that the syntax forbids allocations in the branches of a conditional or in the body of a loop.[1]

Instructions are distinguished according as they manipulate the stack or the heap. Both kinds include non-deterministic assignments, which store in the corresponding memory area a value returned by the function read().

The conditions involved in loop, assertions, or conditionals, are either comparison tests on the content of the memory, or conjunction/disjunction of conditions.[2]

Finally, expressions are built upon the integer variables. The symbol op denotes arithmetic binary operations, such as the addition, the substraction, or the multiplication.

[1] This restriction is purely technical, in order to get a simpler formalism for the sake of concision. The general case can be handled using the formalism developed by Allamigeon and Hymans in [AH08]. However, note that the present restricted form of allocation cannot be assimilated to static allocation. Indeed, in our setting, the size of the allocated arrays may not be statically known, *e.g.* when malloc(p) is called after a non-deterministic assignment $p \coloneqq$ read(). In contrast, with static allocation, the size of the allocated data is always known, as in malloc(10) for instance.

[2] Note that the sets of conditions is preserved by logical negation. For instance, the negation of $e_1 \leq e_2$ is $e_1 \geq e_2 + 1$ since e_i are integer expressions. Similarly, the negation of $e_1 = e_2$ is $(e_1 \leq e_2 - 1) \vee (e_1 \geq e_2 + 1)$, the negation of $t[e_1] = e_2$ is $t[e_1] \neq e_2$, *etc.*

$prog$	::=	$alloc$	allocation
	|	cmd	command
	|	$prog_1$; $prog_2$	sequence
$alloc$	::=	$t \coloneqq \mathsf{malloc}(e)$	array allocation
cmd	::=	$instr_{stack}$	stack instruction
	|	$instr_{heap}$	heap instruction
	|	while $cond$ do cmd done	loop
	|	if $cond$ then cmd_1 else cmd_2 end	conditional
	|	cmd_1 ; cmd_2	sequence
	|	assume $cond$	assertion
	|	skip	no-op
$cond$	::=	e_1 $(\leq \mid = \mid \geq)$ e_2	integer comparison
	|	$t[e_1]$ $(= \mid \neq)$ e_2	array comparison
	|	$cond_1 \wedge cond_2$	conjunction
	|	$cond_1 \vee cond_2$	disjunction
$instr_{stack}$	::=	$x \coloneqq e$	arithmetic assignment
	|	$x \coloneqq \mathsf{read}()$	non-deterministic assignment
	|	$x \coloneqq t[e]$	array lookup
$instr_{heap}$	::=	$t[e_1] \coloneqq e_2$	arithmetic array assignment
	|	$t[e] \coloneqq \mathsf{read}()$	non-deterministic array assignment
	|	$t_1[e_1] \coloneqq t_2[e_2]$	array copy
e	::=	c	constant
	|	x	variable
	|	$\mathsf{op}(e_1, e_2)$	binary operation

Figure 6.1: Syntax of the language

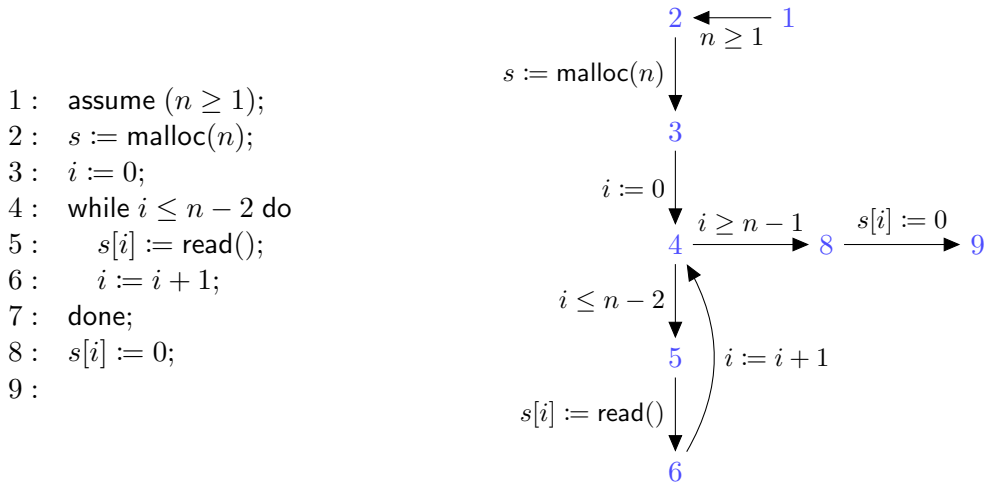


Figure 6.2: A first example of program (left), with its associated control-flow graph (right)

Note that for the sake of simplicity, the integer variables are supposed to be implicitly allocated before the execution of the program. In particular, they all have a global scope.

Example 6.1. Figure 6.2 provides an example of program written in our kernel language.

In this program, an array s is first allocated. Its size is equal to the integer n, which can be seen as an argument of the program. It is then filled by $n-2$ successive calls to the function read() (Lines 4 to 7). Finally, its last element is assigned to the value 0.

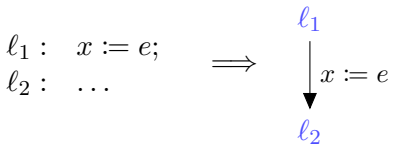
6.2 Semantics of the language

In this section, we assign a formal semantics to programs written in the kernel language. We first recall some of the necessary requirements on control-flow graphs, in Section 6.2.1. In Section 6.2.2, we formally define the way memory states will be modelled as mathematial objects. In Section 6.2.3, we then describe a small-step structural operational semantics [Plo81] of the language. This then leads to the definition of the collecting semantics of a program, in Section 6.2.4. Finally, the formal characterization of the class of errors formed by the heap overflows is discussed in Section 6.2.5.

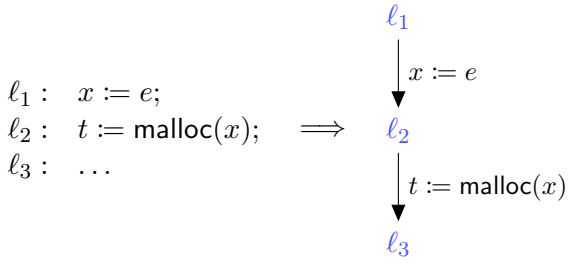
6.2.1 Control-flow graph

The semantics defined in the following sections uses a representation of the programs by their control-flow graph. The *control-flow graph* of a program is a description of the layout of the elementary steps of the program, by means of a graph. Its nodes are formed by some of the control points of the program, while its edges are labelled by allocations, instructions, or conditions. An edge from ℓ to ℓ' represents the possibility to go from the control point ℓ to the control point ℓ', while respectively executing an allocation, an instruction, or satisfying a condition. Let us give some informal illustrations of the construction of such a control-flow graph:

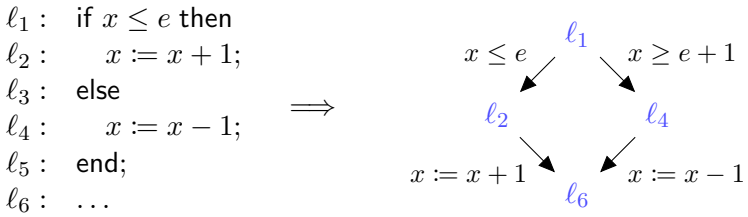
- a simple instruction, for instance $x := e$, and which starts at the control point ℓ_1 and ends at ℓ_2, is represented as follows:



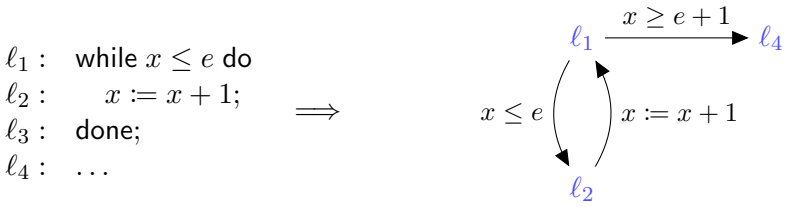
- a sequence is represented as the concatenation of the two corresponding subgraphs:



- a conditional statement if *cond* then ... else ... end starting from ℓ_1 consists of two branches respectively labelled by the condition *cond* and its negation, which later are joined at the end of the statement:



- finally, a loop in the program is transformed in a loop in the control-flow graph:



A more complete example can be found in the right part of Figure 6.2, which provides the control-flow graph of the program introduced in Example 6.1.

The control-flow graph can be statically and automatically built from the source code of the program. This task is very classic, implemented in most of compilers, thus we do not give further details here. For a given program, we denote by $\langle \ell, \mathit{step}, \ell' \rangle$ the fact that there exists an edge from ℓ to ℓ' labelled by *step* in its control-flow graph. Each program P is supposed to have an entry control point $\mathit{entry}(P)$, which is not reached by any incoming edge. For the program of Example 6.1, it is the control point ℓ_1.

$\mathsf{States} \stackrel{def}{=} \mathsf{Stack} \times \mathsf{Heap}$	$\mathsf{Stack} \stackrel{def}{=} \mathbb{Z}^{\mathsf{Vars}}$
$\Sigma = s \setminus h$	s function from Vars to $\mathbb{Z}$
$\mathsf{Heap} \stackrel{def}{=} \mathbb{Z}^{(\mathsf{Addr})}$	$\mathsf{Addr} \stackrel{def}{=} \mathsf{ArrayVars} \times \mathbb{N}$
h partial function from Addr to $\mathbb{Z}$	$a = (t, i)$

Figure 6.3: Summary of the choices to model memory states.

6.2.2 Memory model

Following the principles of the language introduced in Section 6.1.1, a memory state is entirely described the states s and h of the stack and the heap respectively. In that case, the memory state is denoted by $s \setminus h$. The set of the memory states will be denoted by Mem. In this section, the choices made for the modelization of the stack and the heap are discussed. They are recapitulated in Figure 6.3.

Stack. The stack s contains the values of the integer variables. It is therefore represented by a function from $\mathsf{IntVars}$ to $\mathbb{Z}$, which maps each variable to its value. The set of all possible stacks is denoted by Stack.

Remark 6.2. Note that, for the sake of simplicity, the set of the values taken by integer variables is supposed to be the whole set of integers $\mathbb{Z}$. In computers, such values are however finitely encoded (usually on less than 64 bits), so that they should range over a finite set. Our assumption is valid only when it is shown that the program does not lead to any integer overflow during its execution. Automatically detecting all such overflows in a given program has been the topic of active research (see for instance [BCC+03]), and is handled by many static analyzers (for instance [Ast, Pol, Pen]).

Heap. Similarly, the heap h maps addresses to integers. An address is a valid memory location in an allocated array. It consists in a couple (t, i), where $t \in \mathsf{ArrayVars}$, and $i \in \mathbb{N}$. When i is strictly less than the size of the array, the address (t, i) refers to the $(i + 1)$-th element of the array t.[3] The set of the addresses is denoted by Addr.

With this choice, the heap is represented by a *partial* function from Addr to $\mathbb{Z}$, mapping the allocated addresses to their value. Indeed, the heap is defined only on the addresses which have been previously allocated by some call to malloc. We will see in Section 6.2.5, that any attempt to write to an address that is not allocated in the heap causes an error called *heap overflow*. The domain of definition of a such partial function h is denoted by $\mathsf{dom}(h)$. The set of all heaps is denoted by Heap.

Example 6.3. Consider the following very simple program:

$$
\begin{array}{ll}
1: & x \coloneqq 0; \\
2: & t \coloneqq \mathsf{malloc}(2); \\
3: & t[0] \coloneqq 2; \\
4: & t[1] \coloneqq 1; \\
5: &
\end{array}
$$

[3] Remember that in a array, the first element is stored at the index 0.

$$\frac{c \in \mathbb{Z}}{s \vdash c \Rightarrow c}$$

$$\frac{x \in \mathsf{IntVars}}{s \vdash x \Rightarrow s(x)}$$

$$\frac{s \vdash e_1 \Rightarrow v_1 \qquad s \vdash e_2 \Rightarrow v_2 \qquad \mathsf{op}(v_1, v_2) = v}{s \vdash \mathsf{op}(e_1, e_2) \Rightarrow v}$$

Figure 6.4: Evaluation of expressions

Then the state of the memory at the end of this program (*i.e.* at the control point 5) is $s \dagger h$, where $s : x \mapsto 0$, and $h : (t, 0) \mapsto 2, (t, 1) \mapsto 1$.

6.2.3 Operational semantics

This section deals with the operational semantics of the kernel language. It is defined by means of a *transition relation*, denoted by $\rightarrow$. Thus we will have $\Sigma_1 \rightarrow \Sigma_2$ when the machine can pass from the state Σ_1 to the state Σ_2 by an elementary program step (*i.e.* the execution of an allocation, an instruction or a condition).

The transition relation will be defined thanks to *inference rules*, denoted under the form

$$\frac{C_1 \quad \dots \quad C_n}{D}.$$

The C_i are called the *premises*, and D the *conclusion*. The meaning of such a rule is the following: "if all the conditions C_i are true, then the conclusion D holds".

We first describe the mathematical model of the semantic states (Section 6.2.3.a), then the evaluation of integer expressions in a given state (Section 6.2.3.b), the semantics of instructions and allocations (Section 6.2.3.c), and the semantics of conditions (Section 6.2.3.d).

6.2.3.a Semantic states. The semantic state describes the current state of the machine. In our setting, it is defined as a couple $(\ell, s \dagger h)$ formed by the current control point ℓ, and a memory state $s \dagger h$. It means that before the execution of the step located at ℓ, the memory is in state $s \dagger h$.

If Ctrl corresponds to the set of the control points appearing in the control-flow graph, the set of semantic states is thus defined by $\mathsf{States} \stackrel{def}{=} \mathsf{Ctrl} \times \mathsf{Mem}$.

6.2.3.b Evaluation of expressions. We first define the evaluation of an integer expression e in a given memory state $s \dagger h$. Since e involves only integer variables, this evaluation is defined only in the context of the stack. We will denote by $s \vdash e \Rightarrow v$ the fact that e is evaluated to the integer $v \in \mathbb{Z}$ in the stack s. The relation $s \vdash e \Rightarrow v$ is defined by the inference rules given in Figure 6.4.

6.2.3.c Semantics of allocations and instructions. The semantics of allocations and instructions is given by the following rule: for a given allocation or instruction *step*,

$$\frac{\langle \ell, step, \ell' \rangle \qquad s \dagger h \vdash step : s' \dagger h'}{(\ell, s \dagger h) \rightarrow (\ell', s' \dagger h')} \tag{6.1}$$

$$\frac{s \vdash e \Rightarrow v}{s \shortmid h \vdash x \coloneqq e : s[x \mapsto v] \shortmid h}$$

$$\frac{v \in \mathbb{Z}}{s \shortmid h \vdash x \coloneqq \mathsf{read}() : s[x \mapsto v] \shortmid h}$$

$$\frac{s \vdash e \Rightarrow i \qquad (t,i) \in \mathsf{dom}(h)}{s \shortmid h \vdash x \coloneqq t[e] : s[x \mapsto h(t,i)] \shortmid h}$$

$$\frac{s \vdash e_1 \Rightarrow i \qquad s \vdash e_2 \Rightarrow v \qquad (t,i) \in \mathsf{dom}(h)}{s \shortmid h \vdash t[e_1] \coloneqq e_2 : s \shortmid h[(t,i) \mapsto v]}$$

$$\frac{s \vdash e \Rightarrow i \qquad (t,i) \in \mathsf{dom}(h) \qquad v \in \mathbb{Z}}{s \shortmid h \vdash t[e] \coloneqq \mathsf{read}() : s \shortmid h[(t,i) \mapsto v]}$$

$$\frac{s \vdash e_1 \Rightarrow i \qquad s \vdash e_2 \Rightarrow j \qquad (t_1,i) \in \mathsf{dom}(h) \qquad (t_2,j) \in \mathsf{dom}(h)}{s \shortmid h \vdash t_1[e_1] \coloneqq t_2[e_2] : s \shortmid h[(t_1,i) \mapsto h(t_2,j)]}$$

$$\frac{s \vdash e \Rightarrow n \qquad n > 0 \qquad h' = h_{|\mathsf{dom}(h) \setminus (\{t\} \times \mathbb{N})} \qquad w_0, \ldots, w_{n-1} \in \mathbb{Z}}{s \shortmid h \vdash t \coloneqq \mathsf{malloc}(e) : s \shortmid h'[(t,0) \mapsto w_0] \ldots [(t,n-1) \mapsto w_{n-1}]}$$

Figure 6.5: Side effect of the instructions on memory (Given a map f, $g \stackrel{def}{=} f[x \mapsto v]$ is the function which coincides with f on its domain, except on x where $g(x) = v$. Given a function f and a set S, $f_{|S}$ denotes the restriction of f on S.)

where the relation $s \shortmid h \vdash \mathit{step} : s' \shortmid h'$, defined in Figure 6.5, describes the side effect of the instruction or the allocation *step* on memory.

The interpretation of these rules is the following:

- the stack assignments $x \coloneqq e$, $x \coloneqq \mathsf{read}()$, and $x \coloneqq t[e]$ consist in replacing in the stack s the value of x by the value of the right member. For a simple assignment, it is equal to the evaluation of the expression e in the stack s. If the assignment is non-deterministic, the value is an arbitrary element $v \in \mathbb{Z}$. Finally, if the assignment is an array lookup, it is first ensured that the element $t[e]$ is allocated, before getting the corresponding value in the heap.

- for the heap assignment $t[e_1] \coloneqq e_2$, the expression e_2 is first evaluated to a value v. This value is then stored into the heap at the address corresponding to $t[e_1]$, if this address is indeed allocated (condition $(t,i) \in \mathsf{dom}(h)$). The semantics of the assignments $t[e] \coloneqq \mathsf{read}()$ and $t_1[e_1] \coloneqq t_2[e_2]$ is defined similarly.

- the allocation $t \coloneqq \mathsf{malloc}(e)$ first evaluates the value n of the expression e. The integer n is required to be strictly positive.[4] Then all the previously allocated addresses of the form (t,i) in the heap h are removed, as if they were collected by a garbage collector.

[4]The case where n is negative is considered as an error which stops the execution (according to the standard Ansi C, the behavior of $\mathsf{malloc}(0)$ is implementation-dependent [fS99, Section 7.20.3, Paragraph 1]).

$$\frac{s \vdash e_1 \Rightarrow v_1 \qquad s \vdash e_2 \Rightarrow v_2 \qquad v_1 \diamond v_2}{s \restriction h \vdash e_1 \diamond e_2 : s \restriction h} \qquad \text{where } \diamond \in \{\leq, =, \geq\}$$

$$\frac{s \vdash e_1 \Rightarrow i \qquad (t,i) \in \mathsf{dom}(h) \qquad s \vdash e_2 \Rightarrow v \qquad h(t,i) \diamond v}{s \restriction h \vdash t[e_1] \diamond e_2 : s \restriction h} \qquad \text{where } \diamond \in \{=, \neq\}$$

$$\frac{s \restriction h \vdash \mathit{cond}_1 : s \restriction h \qquad s \restriction h \vdash \mathit{cond}_2 : s \restriction h}{s \restriction h \vdash \mathit{cond}_1 \wedge \mathit{cond}_2 : s \restriction h}$$

$$\frac{s \restriction h \vdash \mathit{cond}_i : s \restriction h}{s \restriction h \vdash \mathit{cond}_1 \vee \mathit{cond}_2 : s \restriction h} \qquad \text{where } i = 1 \text{ or } 2$$

Figure 6.6: Selection of memory states with respect to the conditions

This yields the restriction h' of the partial function h on the domain $\mathsf{dom}(h) \setminus (\{t\} \times \mathbb{N})$. Finally, the addresses $(t,0), \ldots, (t,n-1)$ are freshly allocated in h', and mapped to arbitrary values.

6.2.3.d Semantics of conditions. The semantics of conditions is defined similarly to instructions: if *cond* is a condition,

$$\frac{\langle \ell, \mathit{cond}, \ell' \rangle \qquad s \restriction h \vdash \mathit{cond} : s' \restriction h'}{(\ell, s \restriction h) \to (\ell', s' \restriction h')} \tag{6.2}$$

where the relation $s \restriction h \vdash \mathit{cond} : s' \restriction h'$, defined in Figure 6.6, consists in selecting the memory states $s \restriction h$ which satisfy the condition *cond*. The state $s' \restriction h'$ is the same as initially, since the conditions do not have any side effect on memory.

Example 6.4. Consider the simple program of Example 6.3. Let $s_i \restriction h_i$ $(1 \leq i \leq 5)$ be the memory states defined as follows:

$$\begin{cases} s_1 : x \mapsto 314159265 \\ h_1 : \end{cases} \qquad \begin{cases} s_2 : x \mapsto 0 \\ h_2 : \end{cases}$$

$$\begin{cases} s_3 : x \mapsto 0 \\ h_3 : (t,0) \mapsto 271828183, (t,1) \mapsto 161803399 \end{cases} \qquad \begin{cases} s_4 : x \mapsto 0 \\ h_4 : (t,0) \mapsto 2, (t,1) \mapsto 161803399 \end{cases}$$

$$\begin{cases} s_5 : x \mapsto 0 \\ h_5 : (t,0) \mapsto 2, (t,1) \mapsto 1 \end{cases}$$

They form a sequence of possible memory states during the execution of the program, *i.e.*

$$(1, s_1 \restriction h_1) \to (2, s_2 \restriction h_2) \to (3, s_3 \restriction h_3) \to (4, s_4 \restriction h_4) \to (5, s_5 \restriction h_5).$$

Observe that initially, the variable x has an arbitrary value (state $(1, s_1 \restriction h_1)$), and the heap is empty. Just after the allocation of the array t (state $(3, s_3 \restriction h_3)$), the heap is defined on the addresses $(t,0)$ and $(t,1)$ with arbitrary values.

6.2.4 Collecting semantics of a program

The collecting semantics $C(P)$ of a program P consists of all the machine states reachable during *any* execution of P. In particular, it is a subset of the set States. It is defined by means of the function $F : \wp(\mathsf{States}) \to \wp(\mathsf{States})$,[5] which corresponds to the execution of an additional step in the program:

$$F(X) \stackrel{def}{=} \{(entry(P), s_0 \mid \emptyset) \mid s_0 \in \mathsf{Stack}\} \cup \bigcup_{(\ell, s \mid h) \in X} \left\{(\ell', s' \mid h') \mid (\ell, s \mid h) \to (\ell', s' \mid h')\right\}.$$

Intuitively, the set $\{(entry(P), s_0 \mid \emptyset) \mid s_0 \in \mathsf{Stack}\}$ consists of the possible initial states of the machine: as mentioned in Section 6.1.1, the variables in the stack are not initialized, and the heap does not contain any data (the symbol $\emptyset$ refers here to the partial function from Addr to $\mathbb{Z}$ with empty domain). Besides, the set $\{(\ell', s' \mid h') \mid (\ell, s \mid h) \to (\ell', s' \mid h')\}$ corresponds to the states reachable from states $(\ell, s \mid h) \in X$ after the execution of an instruction, an allocation, or a condition.

Then, the collecting semantics $C(P)$ is defined as the smallest fixpoint of F, which is denoted by $C(P) \stackrel{def}{=} \mathsf{lfp}\, F$. Since $\wp(\mathsf{States})$, partially order by $\subset$ and endowed with the join and meet operators $\cup$ and $\cap$ respectively, is a complete lattice and F a monotone map, the existence of such a fixpoint is ensured by the Knaster-Tarski theorem [Tar55]. Besides, the function F is continuous, so that by Kleene's theorem:

$$C(P) = \bigcup_{n \geq 0} F^n(\emptyset), \tag{6.3}$$

where F^n is the n-th iterate of F.[6] Intuitively, this means that the collecting semantics consists of all the states reachable after a finite number n of execution steps, for any $n \geq 0$.

The collecting semantics can be equivalently defined as a function which maps the set Ctrl to the set $\wp(\mathsf{Mem})$, and associates each $\ell \in \mathsf{Ctrl}$ to the set of memory states possibly arising at the control point ℓ. Then, F can be redefined as the application which maps each $X : \mathsf{Ctrl} \to \wp(\mathsf{Mem})$ to the function $F(X) : \mathsf{Ctrl} \to \wp(\mathsf{Mem})$ defined by: for each $\ell' \in \mathsf{Ctrl}$,

$$F(X)(\ell') \stackrel{def}{=} \begin{cases} \{s_0 \mid \emptyset \mid s_0 \in \mathsf{Stack}\} & \text{if } \ell' = entry(P), \\ \displaystyle\bigcup_{s \mid h \in X(\ell)} \left\{s' \mid h' \mid (\ell, s \mid h) \to (\ell', s' \mid h')\right\} & \text{otherwise.} \end{cases} \tag{6.4}$$

We still have $C(P) = \mathsf{lfp}\, F$, and for the same reasons than before,

$$C(P) = \bigcup_{n \geq 0} F^n(\emptyset).$$

Note that this presentation is equivalent to the previous one, using the one-to-one correspondence between $\wp(\mathsf{States})$ and $\wp(\mathsf{Mem})^{\mathsf{Ctrl}}$ defined by:

$$\begin{array}{rcl} \wp(\mathsf{States}) & \longleftrightarrow & \wp(\mathsf{Mem})^{\mathsf{Ctrl}} \\ X & \longmapsto & X' \text{ such that } X'(\ell) \stackrel{def}{=} \{\, \Sigma \mid (\ell, \Sigma) \in X \,\} \\ \{\, (\ell, \Sigma) \mid \Sigma \in X'(\ell), \ell \in \mathsf{Ctrl} \,\} & \longmapsfrom & X' \end{array}$$

[5] Given a set S, $\wp(S)$ refers to the powerset of S.

[6] Recall that a function F from a complete partial order to itself is said to be *continuous* if, for any non-empty ascending chain $(X_i)_i$, we have $F(\sup X_i) = \sup F(X_i)$ (where $\sup X_i$ refers to the supremum of the chain $(X_i)_i$)

Example 6.5. Using the second formalism, the collecting semantics $C(P)$ of the simple program of Example 6.3 is defined as follows:

$$\ell_1 \mapsto \{\, s \shortmid \emptyset \mid s \in \mathsf{Stack} \,\}$$

$$\ell_2 \mapsto \{\, s \shortmid \emptyset \mid s(x) = 0 \,\}$$

$$\ell_3 \mapsto \big\{\, s \shortmid h \mid s(x) = 0 \text{ and } \mathsf{dom}(h) = \{\, (t,0), (t,1) \,\} \,\big\}$$

$$\ell_4 \mapsto \big\{\, s \shortmid h \mid s(x) = 0,\ \mathsf{dom}(h) = \{\, (t,0), (t,1) \,\},\ h(t,0) = 2 \,\big\}$$

$$\ell_5 \mapsto \big\{\, s \shortmid h \mid s(x) = 0,\ \mathsf{dom}(h) = \{\, (t,0), (t,1) \,\},\ h(t,0) = 2,\ h(t,1) = 1 \,\big\}$$

It can be verified that, for each control point, it indeed represents the whole set of the possible memory states, taking for instance into account the non-determinism introduced by allocations.

6.2.5 Proving the absence of heap overflows

The collecting semantics yields the set of all the possible states of the machine during any execution of a program. It can be used to verify that no state leads to the class of errors formed by the heap overflows.

A *heap overflow* occurs if the program reads in or writes to an address which is not allocated in the heap. It may happen when executing any instruction or condition which accesses to the heap. Formally, there is a heap overflow if such an instruction or a condition involves an expression of the form $t[e]$ in a memory state $s \shortmid h$ such that $s \vdash e \Rightarrow i$ and $(t,i) \notin \mathsf{dom}(h)$.

Heap overflows can be modelled by an *error state* $\square$, added to the set States, and defined by the following inference rules:

$$\frac{\langle \ell, x \coloneqq t[e], \ell_2 \rangle \qquad s \vdash e \Rightarrow i \qquad (t,i) \notin \mathsf{dom}(h)}{(\ell, s \shortmid h) \to \square}$$

$$\frac{\langle \ell, t[e_1] \coloneqq e_2, \ell_2 \rangle \qquad s \vdash e_1 \Rightarrow i \qquad (t,i) \notin \mathsf{dom}(h)}{(\ell, s \shortmid h) \to \square}$$

$$\frac{\langle \ell, t[e] \coloneqq \mathsf{read}(), \ell_2 \rangle \qquad s \vdash e \Rightarrow i \qquad (t,i) \notin \mathsf{dom}(h)}{(\ell, s \shortmid h) \to \square}$$

$$\frac{\langle \ell, t_1[e_1] \coloneqq t_2[e_2], \ell_2 \rangle \qquad s \vdash e_1 \Rightarrow i \qquad (t_1,i) \notin \mathsf{dom}(h)}{(\ell, s \shortmid h) \to \square}$$

$$\frac{\langle \ell, t_1[e_1] \coloneqq t_2[e_2], \ell_2 \rangle \qquad s \vdash e_2 \Rightarrow j \qquad (t_2,j) \notin \mathsf{dom}(h)}{(\ell, s \shortmid h) \to \square}$$

$$\frac{\langle \ell, t[e_1] \diamond e_2, \ell_2 \rangle \qquad s \vdash e_1 \Rightarrow i \qquad (t,i) \notin \mathsf{dom}(h)}{(\ell, s \shortmid h) \to \square} \qquad \text{where } \diamond \in \{\, =, \neq \,\}$$

Then, the absence of heap overflows is equivalent to the condition $\square \notin C(P)$.

Nevertheless, for the sake of simplicity, we prefer to encode the error state by the absence of transition, as if the machine were stopping whenever the program tries to read in or write to an invalid part of the heap. Then, the error state is assimilated to an unreachable state. Such a semantics is said to be *blocking*. This formalism avoids the introduction of a special

error state, and will allow to adopt a simpler formalism in the following parts. It can be shown to be equivalent to the previous choice, see for instance [CDNB08].

Then, to ensure that a program does not cause any heap overflow, it suffices to check that the following property on $C(P)$ is satisfied: for any state $(\ell, s \restriction h) \in C(P)$ and any edge $\langle \ell, \mathit{step}, \ell' \rangle$,

$$\begin{cases} s \vdash e \Rightarrow i \text{ and } (t,i) \in \mathsf{dom}(h) & \text{if } \mathit{step} \text{ is of the form } x \coloneqq t[e] \text{ or } t[e] \coloneqq \mathsf{read}() \\ s \vdash e_1 \Rightarrow i \text{ and } (t,i) \in \mathsf{dom}(h) & \text{if } \mathit{step} \text{ is of the form } t[e_1] \coloneqq e_2 \text{ or } t[e_1] \diamond e_2 \\ s \vdash e_1 \Rightarrow i,\ s \vdash e_2 \Rightarrow j, \\ \text{and } (t_1,i),(t_2,j) \in \mathsf{dom}(h) & \text{if } \mathit{step} \text{ is of the form } t_1[e_1] \coloneqq t_2[e_2] \end{cases} \tag{6.5}$$

6.3 Abstract interpretation

A naive method to statically analyze programs could consist in determining their collecting semantics. Unfortunately, as discussed in Chapter 1, Rice's theorem [Ric56] implies that the collecting semantics is not computable in general. Subsequently, showing that a given program satisfies, for instance, the non-trivial property given in (6.5), is a non-decidable problem.

More precisely, several difficulties would have to be tackled to "compute" $C(P)$ for any program P:

- the subsets of $\wp(\mathsf{States})$ may not be representable in machine,
- the function F may not be computable,
- and the iteration sequence to compute $C(P)$ by iterations (see (6.3)) may not necessarily converge after a finite number of steps.

The principle of abstract interpretation is to determine an over-approximation of the semantics. Using over-approximations allows to overcome all the difficulties previously mentioned:

- semantics states can be represented by *computer-representable* over-approximations, called "abstract semantic states",
- similarly, the transfer function can be over-approximated by a computable "abstract" function,
- and finally, up to losing some precision, the convergence of the corresponding "abstract" sequence in a finite number of iterations can be ensured.

This allows to compute an abstraction of the collecting semantics. Since every approximation is sound, no semantic state is forgotten. Therefore, it can be used to determine whether a program satisfies a given property. However, because of the non-exactness of the approximations in the general case, the analysis may sometimes fail to prove a property satisfied by the program.

This section is organized as follows. Section 6.3.1 presents the theoretical framework of abstract interpretation in a general setting. Section 6.3.2 then focuses on abstraction of numerical properties.

6.3.1 Theoretical framework

Different formalisms have been developed for abstract interpretation (see [CC92b] for a survey). The presentation is here restricted to one of the most general formalism, because it is

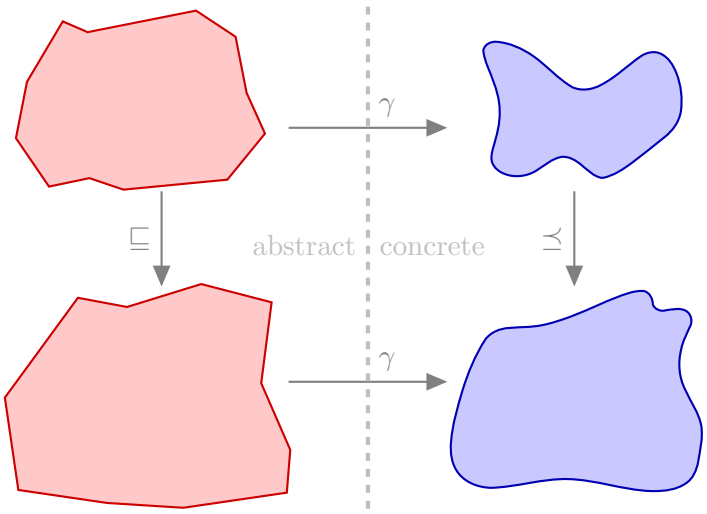


Figure 6.7: Diagram illustrating the notion of precision

adapted to the contributions of Chapter 7. Some other possible formalisms are discussed in Section 6.3.1.d.

6.3.1.a Abstract domain. Let D be the set of the elements to be abstracted. We assume that D is a complete lattice $(D, \preceq, \bot, \top, \curlyvee, \curlywedge)$. The relation $\preceq$ is a partial order on the elements of D, $\bot$ and $\top$ are respectively the least and greatest elements of D, and $\curlyvee$ and $\curlywedge$ are join and meet operators. The set D is called the *concrete domain*. It contains *concrete* elements, as opposed to abstract ones.

Example 6.6. In the semantics introduced in Section 6.2, the concrete domain corresponds to the complete lattice $(\wp(\mathsf{States}), \subset, \emptyset, \mathsf{States}, \cup, \cap)$.

An abstract domain is basically defined as follows:

Definition 6.1. An *abstract domain* over the set D is a couple $(\mathcal{D}, \gamma)$, where $\mathcal{D}$ is the set of the *abstract elements*, and γ is a function from $\mathcal{D}$ to D, called the concretization operator.

The concretization operator represents the relation between abstract and concrete elements: it maps any abstract element $\mathcal{X} \in \mathcal{D}$ to a concrete element which it represents.

Usually, abstract domains are equipped with a preorder which allows to compare two different abstract elements in term of precision:

Definition 6.2. Let $(\mathcal{D}, \gamma)$ be an abstract domain over D. An *abstract preorder* $\sqsubseteq$ over $\mathcal{D}$ is a preorder which satisfies: for all $\mathcal{X}, \mathcal{Y} \in \mathcal{D}$,

$$\mathcal{X} \sqsubseteq \mathcal{Y} \implies \gamma(\mathcal{X}) \preceq \gamma(\mathcal{Y}).$$

Intuitively, if $\mathcal{X} \sqsubseteq \mathcal{Y}$, then $\mathcal{X}$ is more precise than $\mathcal{Y}$ (possibly as precise), since it represents a smaller (or equal) concrete element for the order $\preceq$ (see Figure 6.7). Note that the condition on the abstract preorder $\sqsubseteq$ is equivalent to the monotonicity of the concretization operator w.r.t. $\sqsubseteq$ and $\preceq$. It is also sometimes called the *soundness* of the preorder.

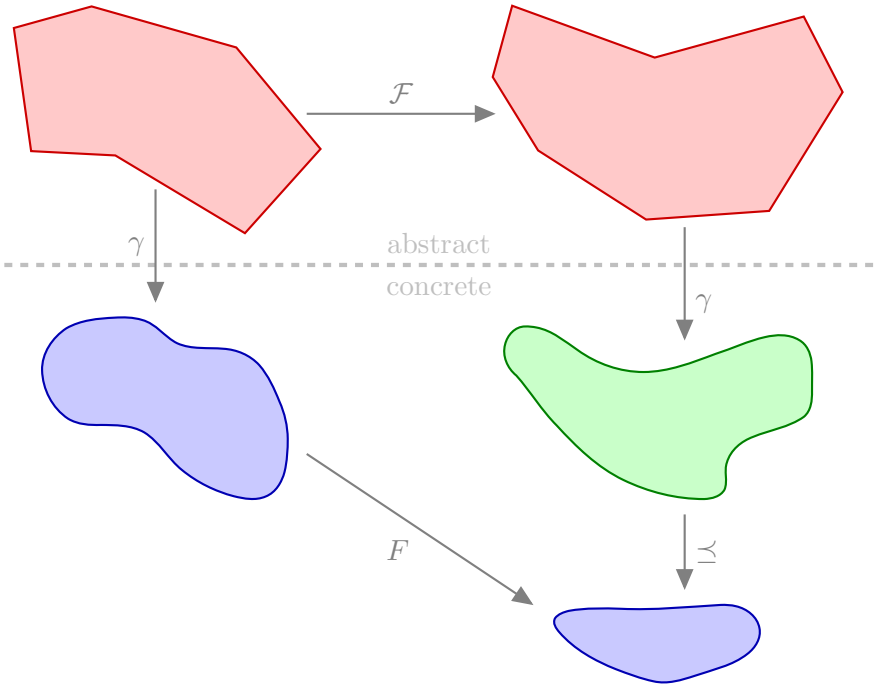


Figure 6.8: Diagram representing the soundness of an abstract function

6.3.1.b Sound abstract functions. Let F be a function manipulating concrete elements. An abstraction of F is a function which performs the same manipulations, but on the corresponding abstract elements. This abstract function has however to preserve the property of over-approximation, in which case it is said to be sound:

Definition 6.3. Let $F : D^p \to D$ $(p \geq 1)$. Then $\mathcal{F} : \mathcal{D}^p \to \mathcal{D}$ is said to be a *sound abstraction* of F if it satisfies the following property: for all $\mathcal{X}_1, \ldots, \mathcal{X}_p \in \mathcal{D}$,

$$F(\gamma(\mathcal{X}_1), \ldots, \gamma(\mathcal{X}_p)) \preceq \gamma(\mathcal{F}(\mathcal{X}_1, \ldots, \mathcal{X}_p)).$$

Intuitively, starting from over-approximations $\mathcal{X}_i$ of $\gamma(\mathcal{X}_i)$ for each i, $\mathcal{F}(\mathcal{X}_1, \ldots, \mathcal{X}_p)$ is still an over-approximation of $F(\gamma(\mathcal{X}_1), \ldots, \gamma(\mathcal{X}_p))$. Then, the application of $\mathcal{F}$ on the tuple $(\mathcal{X}_1, \ldots, \mathcal{X}_p)$ does not forget any concrete elements. The case $p = 1$ is illustrated in Figure 6.8.

In the general case, a same concrete function admits several sound abstractions, whose precision may vary. The following definition characterizes two particular levels of precision:

Definition 6.4. Let $F : D^p \to D$ $(p \geq 1)$, and $\mathcal{F} : \mathcal{D}^p \to \mathcal{D}$ be a sound abstraction of F. Then $\mathcal{F}$ is said to be *exact* when the following property holds: for all $\mathcal{X}_1, \ldots, \mathcal{X}_p \in \mathcal{D}$,

$$F(\gamma(\mathcal{X}_1), \ldots, \gamma(\mathcal{X}_p)) = \gamma(\mathcal{F}(\mathcal{X}_1, \ldots, \mathcal{X}_p)).$$

Besides, $\mathcal{F}$ is said to be a *best possible abstraction* of F when for any $\mathcal{X}_1, \ldots, \mathcal{X}_p, \mathcal{Y} \in \mathcal{D}$,

$$F(\gamma(\mathcal{X}_1), \ldots, \gamma(\mathcal{X}_p)) \preceq \gamma(\mathcal{Y}) \implies \gamma(\mathcal{F}(\mathcal{X}_1, \ldots, \mathcal{X}_p)) \preceq \gamma(\mathcal{Y}).$$

In the former case, the application of the abstract function $\mathcal{F}$ does not introduce further approximations. In the latter case, $\mathcal{F}$ returns an abstract element which is as accurate as possible.

An abstract domain is usually provided with sound abstractions of the join and meet operations of the concrete lattice:

- an abstract join operator $\sqcup : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$, such that for all $\mathcal{X}, \mathcal{Y} \in \mathcal{D}$,
$$\gamma(\mathcal{X}) \curlyvee \gamma(\mathcal{Y}) \preceq \gamma(\mathcal{X} \sqcup \mathcal{Y}),$$
- an abstract meet operator $\sqcap : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$, verifying for all $\mathcal{X}, \mathcal{Y} \in \mathcal{D}$
$$\gamma(\mathcal{X}) \curlywedge \gamma(\mathcal{Y}) \preceq \gamma(\mathcal{X} \sqcap \mathcal{Y});$$

Similarly, in most abstract domains, it can be supposed that the least and greatest elements $\pm$ and $\mp$ have abstract counterparts $\bot$ and $\top$ respectively. Both are required to be over-approximations, *i.e.*:
$$\pm \preceq \gamma(\bot), \qquad \mp \preceq \gamma(\top).$$
and to satisfy $\bot \sqsubseteq \mathcal{X} \sqsubseteq \top$ for all $\mathcal{X} \in \mathcal{D}$. The relation $\pm \preceq \gamma(\bot)$ is a tautology, and most often, $\bot$ is an exact approximation of $\pm$, *i.e.* $\gamma(\bot) = \pm$.

6.3.1.c Abstract fixpoint computation. Let F be a continuous function on the complete lattice $(D, \preceq, \pm, \mp, \curlyvee, \curlywedge)$. By Kleene's theorem, we know that F admits a least fixpoint, which is given by:
$$\mathsf{lfp}\, F = \bigvee_{n \geq 0} F^n(\pm).$$

Let $\mathcal{F}$ be a sound abstraction of F. Then, each iterate $F^n(\pm)$ can be over-approximated by $\mathcal{F}^n(\bot)$. Indeed, $\pm \preceq \gamma(\bot)$ by hypothesis, and by recursion, if for a given n, $F^n(\pm) \preceq \gamma(\mathcal{F}^n(\bot))$, then

$$\begin{aligned} F^{n+1}(\pm) &= F(F^n(\pm)) \\ &\preceq F(\gamma(\mathcal{F}^n(\bot))) && \text{by recursion hypothesis and monotonicity of } F \\ &\preceq \gamma(\mathcal{F}(\mathcal{F}^n(\bot))) && \text{by soundness of } F \\ &\preceq \gamma(\mathcal{F}^{n+1}(\bot)). \end{aligned}$$

Moreover, as soon as $\mathcal{F}$ is monotone, the $\mathcal{F}^n(\bot)$ form an increasing sequence. Then, if this sequence eventually stabilizes, *i.e.* if there exists an index N from which all the terms of the sequence are equal, we have:
$$\mathsf{lfp}\, F \preceq \gamma(\mathcal{F}^N(\bot)). \tag{6.6}$$

This may happen in particular when the abstract domain $\mathcal{D}$ forms a partially ordered set satisfying the ascending chain condition.[7] Nevertheless, this does not hold in the general case. In order to obtain an over-approximation $\mathsf{lfp}\, F$ as a sequence of abstract elements, a widening operator has to be used:

[7]The *ascending chain condition* states that increasing chain $\mathcal{X}_1 \sqsubseteq \ldots \sqsubseteq \mathcal{X}_n \sqsubseteq \ldots$ eventually stabilizes.

Definition 6.5. A function $\nabla : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is said to be a *widening operator* if it satisfies the following conditions:

- (*soundness*) $\mathcal{X}, \mathcal{Y} \sqsubseteq \mathcal{X} \nabla \mathcal{Y}$ for all $\mathcal{X}, \mathcal{Y} \in \mathcal{D}$,
- (*ascending chain condition*) for any increasing sequence of elements $\mathcal{X}_0 \sqsubseteq \ldots \sqsubseteq \mathcal{X}_n \sqsubseteq \ldots$, the sequence defined by

$$\begin{cases} \mathcal{Y}_0 \stackrel{def}{=} \mathcal{X}_0 \\ \mathcal{Y}_{n+1} \stackrel{def}{=} \mathcal{Y}_n \nabla \mathcal{X}_{n+1} \end{cases} \tag{6.7}$$

  eventually stabilizes.

Intuitively, a widening operator maps any increasing sequence of abstract elements $\mathcal{X}_n$ into an eventually stabilizing one, whose limit over-approximates every $\mathcal{X}_n$.

Then the following statement holds:

Proposition 6.1. *Let $F : D \to D$ be a continuous function, and $\mathcal{F} : \mathcal{D} \to \mathcal{D}$ a sound and monotone abstraction of F. If $(\mathcal{X}_n)_n$ is the sequence defined by:*

$$\begin{cases} \mathcal{X}_0 \stackrel{def}{=} \bot \\ \mathcal{X}_{n+1} \stackrel{def}{=} \begin{cases} \mathcal{X}_n & \text{if } \mathcal{F}(\mathcal{X}_n) \sqsubseteq \mathcal{X}_n \\ \mathcal{X}_n \nabla \mathcal{F}(\mathcal{X}_n) & \text{otherwise} \end{cases} \end{cases},$$

then this sequence eventually stabilizes, and its limit $\mathcal{X}_\infty$ verifies:

$$\mathsf{lfp}\, F \preceq \gamma(\mathcal{X}_\infty).$$

Besides, this limit is a post-fixpoint of $\mathcal{F}$.

Proof. Let us first show that the sequence of the $\mathcal{X}_n$ eventually stabilizes. If not, then for all n, we have $\mathcal{X}_{n+1} = \mathcal{X}_n \nabla \mathcal{F}(\mathcal{X}_n)$. The sequence $(\mathcal{F}(\mathcal{X}_n))_n$ is increasing because $\mathcal{F}$ is monotone and the widening operator is sound, so that $\mathcal{X}_n \sqsubseteq \mathcal{X}_{n+1}$. Since the widening operator satisfies the ascending chain condition, the assumption leads to a contradiction.

Now, let us show by induction that for all n, $F^n(\bot) \preceq \gamma(\mathcal{X}_n)$. The case $n = 0$ is obvious. Besides, if $F^n(\bot) \preceq \gamma(\mathcal{X}_n)$, then

$$\begin{aligned} F^{n+1}(\bot) &\preceq F(\gamma(\mathcal{X}_n)) && \text{since } F \text{ is monotone} \\ &\preceq \gamma(\mathcal{F}(\mathcal{X}_n)) && \text{by soundness of } \mathcal{F} \\ &\preceq \gamma(\mathcal{X}_{n+1}) && \text{by soundness of } \nabla. \end{aligned}$$

Since the sequence of the $F^n(\bot)$ is increasing, this shows that $\cup_{i=0}^{n} F^n(\bot) \preceq \gamma(\mathcal{X}_n)$. Taking the limit when $n \to +\infty$ implies that $\mathsf{lfp}\, F \preceq \gamma(\mathcal{X}_\infty)$.

Finally, let N such that $\mathcal{X}_\infty = \mathcal{X}_N = \mathcal{X}_{N-1}$. Then $\mathcal{F}(\mathcal{X}_N) \sqsubseteq \mathcal{X}_N$ because either $\mathcal{F}(\mathcal{X}_{N-1}) \sqsubseteq \mathcal{X}_{N-1}$ and $\mathcal{X}_N = \mathcal{X}_{N-1}$, or $\mathcal{X}_N = \mathcal{X}_{N-1} \nabla \mathcal{F}(\mathcal{X}_{N-1}) = \mathcal{X}_N \nabla \mathcal{F}(\mathcal{X}_N)$, so that $\mathcal{F}(\mathcal{X}_N) \sqsubseteq \mathcal{X}_N$ by soundness of ∇. In both cases, $\mathcal{X}_\infty$ is a post-fixpoint of $\mathcal{F}$. □

In order to enforce the convergence of any sequence, the widening operator may perform rough over-approximations. As an example, the function which maps any couple $(\mathcal{X}, \mathcal{Y})$ to the element $\top$ is a correct widening operator, but is extremely imprecise. Nevertheless, since the limit $\mathcal{X}_\infty$ of Proposition 6.1 is a post-fixpoint of $\mathcal{F}$, then $\mathcal{F}^n(\mathcal{X}_\infty)$ is always more precise than $\mathcal{X}_\infty$ for all n, and by soundness of $\mathcal{F}$, it is still an over-approximation of $F^n(\mathsf{lfp}\, F) = \mathsf{lfp}\, F$. As a consequence, the result provided by Proposition 6.1 can be refined thanks to the decreasing sequence of the $\mathcal{F}^n(\mathcal{X}_\infty)$. Obviously, the latter may naturally not stabilize ultimately. That is why a narrowing operator can be used to enforce the convergence of such a sequence in a finite number of steps:

Definition 6.6. A function $\Delta : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is said to be a *narrowing operator* if it satisfies the following conditions:

- (*soundness*) $\mathcal{Y} \sqsubseteq \mathcal{X} \,\Delta\, \mathcal{Y} \sqsubseteq \mathcal{X}$ for all $\mathcal{X}, \mathcal{Y} \in \mathcal{D}$ such that $\mathcal{Y} \sqsubseteq \mathcal{X}$,
- (*descending chain condition*) for any decreasing sequence of elements $\mathcal{X}_0 \sqsupseteq \ldots \sqsupseteq \mathcal{X}_n \sqsupseteq \ldots$, the sequence defined by

$$\begin{cases} \mathcal{Y}_0 \stackrel{def}{=} \mathcal{X}_0 \\ \mathcal{Y}_{n+1} \stackrel{def}{=} \mathcal{Y}_n \,\Delta\, \mathcal{X}_{n+1} \end{cases} \tag{6.8}$$

  eventually stabilizes.

Proposition 6.2. *Let $F : D \to D$ be a continuous function, and $\mathcal{F} : \mathcal{D} \to \mathcal{D}$ a sound and monotone abstraction of F. Let $\mathcal{X}_\infty$ be the limit of the sequence $(\mathcal{X}_n)_n$ of Proposition 6.1. If $(\mathcal{Y}_n)_n$ is the sequence defined by:*

$$\begin{cases} \mathcal{Y}_0 \stackrel{def}{=} \mathcal{X}_\infty \\ \mathcal{Y}_{n+1} \stackrel{def}{=} \begin{cases} \mathcal{Y}_n & \text{if } \mathcal{Y}_n = \mathcal{F}(\mathcal{Y}_n) \\ \mathcal{Y}_n \,\Delta\, \mathcal{F}(\mathcal{Y}_n) & \text{otherwise} \end{cases} \end{cases},$$

then this sequence eventually stabilizes, and its limit $\mathcal{Y}_\infty$ verifies:

$$\mathsf{lfp}\, F \preceq \gamma(\mathcal{Y}_\infty).$$

Proof. We claim that for all n, $\mathcal{F}(\mathcal{Y}_n) \sqsubseteq \mathcal{Y}_{n+1} \sqsubseteq \mathcal{Y}_n$.

This property holds when $n = 0$, since $\mathcal{Y}_0 = \mathcal{X}_\infty$, $\mathcal{F}(\mathcal{Y}_0) \sqsubseteq \mathcal{Y}_0$ by Proposition 6.1, and $\mathcal{F}(\mathcal{Y}_0) \sqsubseteq \mathcal{Y}_0 \,\Delta\, \mathcal{F}(\mathcal{Y}_0) \sqsubseteq \mathcal{Y}_0$ by soundness of Δ.

If the same property holds at the rank $n-1$ ($n \geq 1$), then:

- either $\mathcal{Y}_{n+1} = \mathcal{Y}_n$ and $\mathcal{Y}_n = \mathcal{F}(\mathcal{Y}_n)$, so that $\mathcal{F}(\mathcal{Y}_n) = \mathcal{Y}_{n+1} = \mathcal{Y}_n$,
- or $\mathcal{Y}_{n+1} = \mathcal{Y}_n \,\Delta\, \mathcal{F}(\mathcal{Y}_n)$. By induction hypothesis, $\mathcal{F}(\mathcal{Y}_{n-1}) \sqsubseteq \mathcal{Y}_n \sqsubseteq \mathcal{Y}_{n-1}$, so that $\mathcal{F}(\mathcal{Y}_n) \sqsubseteq \mathcal{F}(\mathcal{Y}_{n-1}) \sqsubseteq \mathcal{Y}_n$ since $\mathcal{F}$ is monotone. By soundness of Δ, we consequently have $\mathcal{F}(\mathcal{Y}_n) \sqsubseteq \mathcal{Y}_{n+1} \sqsubseteq \mathcal{Y}_n$.

This proof by induction implies that the sequence of the $\mathcal{Y}_n$, and thus of the $\mathcal{F}(\mathcal{Y}_n)$, is decreasing. Since Δ satisfies the descending chain condition, the sequence of the $\mathcal{Y}_n$ can be shown to be eventually stabilizing.

Now let us show by induction that for all n, $\mathsf{lfp}\, F \preceq \gamma(\mathcal{Y}_n)$. The case $n = 0$ is ensured by Proposition 6.1. Now, if $n \geq 0$, we have:

$$\begin{array}{rrl} & \mathcal{F}(\mathcal{Y}_n) \sqsubseteq \mathcal{Y}_{n+1} & \\ \iff & \gamma(\mathcal{F}(\mathcal{Y}_n)) \preceq \gamma(\mathcal{Y}_{n+1}) & \text{as } \gamma \text{ is monotone} \\ \iff & F(\gamma(\mathcal{Y}_n)) \preceq \gamma(\mathcal{Y}_{n+1}) & \text{because } \mathcal{F} \text{ is sound} \\ \iff & F(\mathsf{lfp}\, F) \preceq \gamma(\mathcal{Y}_{n+1}) & \text{since } F \text{ is monotone and using inductive hypothesis} \\ \iff & \mathsf{lfp}\, F \preceq \gamma(\mathcal{Y}_{n+1}). & \end{array}$$

It follows that $\mathsf{lfp}\, F \preceq \gamma(\mathcal{Y}_\infty)$. □

6.3.1.d Alternative formalisms. The theory of abstract interpretation has been initially introduced in [CC77] with a formalism based on Galois connections.

In this formalism, the abstract domain $\mathcal{D}$ is not only provided with concretization operator γ, but also with an abstraction operator $\alpha : D \to \mathcal{D}$, dual of the former. Like γ, the function α is required to be monotone, and the two operators have to satisfy the following property: for all $X \in D$ and $\mathcal{Y} \in \mathcal{D}$,

$$\alpha(X) \sqsubseteq \mathcal{Y} \iff X \preceq \gamma(Y).$$

This condition means that for any concrete element $X \in D$, $\alpha(X)$ represents the most precise abstraction in $\mathcal{D}$.

Consider a function $F : D^p \to D$. Then the abstract function $\mathcal{F}$ defined by:

$$\mathcal{F}(\mathcal{X}_1, \ldots, \mathcal{X}_p) \stackrel{def}{=} \alpha(F(\gamma(\mathcal{X}_1), \ldots, \gamma(\mathcal{X}_p))), \tag{6.9}$$

can be proved to be sound, and even the best possible abstraction of F. As a consequence, the abstract operator allows to systematically design sound and precise abstract functions.[8]

Given an abstraction operator α, the concretization operator γ is entirely determined as the upper adjoint of α, *i.e.* for all $\mathcal{X} \in \mathcal{D}$, $\gamma(\mathcal{X}) \stackrel{def}{=} \bigvee_{\alpha(X) \sqsubseteq \mathcal{X}} X$. That is why some works in abstract interpretation (*e.g.* [SRW98]) define only an abstraction operator.

The reason why we decided not to introduce this formalism at first is that for some abstract domains, including the one developed in Chapter 7, there is no abstraction operator. In other words, there may not be a best possible abstraction of a given concrete element $X \in D$. An example of an existing abstract domain having the same property is given in Section 6.3.2.a.

6.3.2 Numerical abstract domains

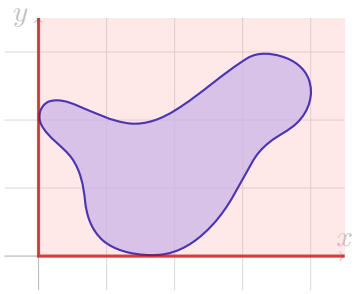
An abstract domain is said to be *numerical* when its elements over-approximate subsets of $\mathcal{K}^{\mathsf{Vars}}$, where $\mathsf{Vars} = \{ \boldsymbol{v}_1, \ldots, \boldsymbol{v}_d \}$ is a given set of variables, and $\mathcal{K}$ is a set of numerical values, typically $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$. In other words, the abstract elements represent sets of numerical environments $\nu : \mathsf{Vars} \to \mathcal{K}$. Observe that any such environment can be assimilated to the element $(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d))$ of $\mathcal{K}^d$. Under this convention, abstract elements of a numerical domain can be seen as representing subsets of $\mathcal{K}^d$. As an illustration, we choose $\mathcal{K} = \mathbb{R}$.

In Section 6.3.2.a, we present the most popular numerical abstract domains. In Section 6.3.2.b, we give further details on the abstract domains of zones and octagons. Finally, in Section 6.3.2.c, we deal with additional primitives which usually equip numerical abstract domains.

[8] However, nothing ensures that the abstract function defined in (6.9) is computable.

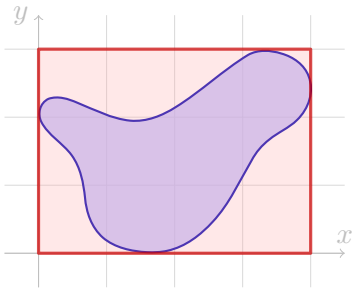
6.3.2.a Overview of the existing abstract domains. Numerical abstract domains have been thoroughly studied in the literature, probably because the first properties addressed in software verification are of numerical nature (integer overflows, array out of bounds,...). They now form a various collection of tools. We give a small description of the most used of them, including the numerical properties that they are able to express, a geometric illustration of their abstract elements for $d = 2$, and the complexity of the abstract primitives:

(a) the *abstract domain of signs* [CC77] infer properties on the signs of the variables $\boldsymbol{v}_i$. Its elements represent intersections of half-spaces or lines of the form $\{\, \boldsymbol{x} = (\boldsymbol{x}_i) \mid \boldsymbol{x}_i \diamond 0 \,\}$, with $\diamond \in \{\, <, \leq, =, \geq, > \,\}$.



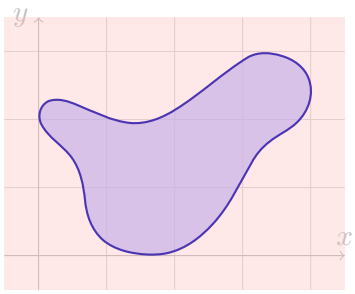
The worst-case complexity of its primitives is $O(d)$.

(b) the *abstract domain of intervals* [CC77] expresses interval invariants $a \leq \boldsymbol{v}_i \leq b$, $a, b \in \mathbb{R} \cup \{\, -\infty, +\infty \,\}$. Each of its elements is concretized to a hyperrectangle of $(\mathbb{R} \cup \{\, -\infty, +\infty \,\})^d$.



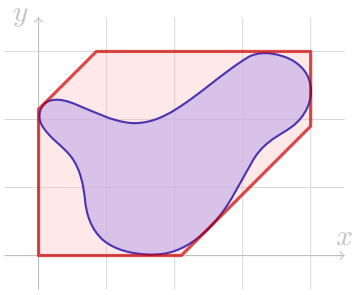
Like for the domain of signs, the worst-case complexity of the primitives on intervals is $O(d)$.

(c) the *abstract domain of affine equalities* has been developed by Karr in [Kar76], and represents affine equalities over the $\boldsymbol{v}_i$, *i.e.* conjunctions of relationships of the form $a_1\boldsymbol{v}_1 + \ldots + a_d\boldsymbol{v}_d = b$, for some $a_1, \ldots, a_d, b \in \mathbb{R}$. Its elements consequently correspond to affine linear varieties of $\mathbb{R}^d$.



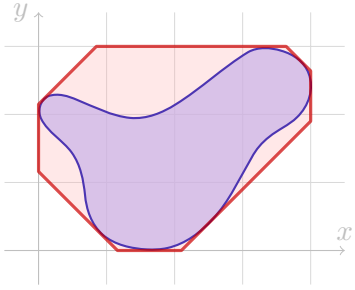
The complexity of this abstract domain is dominated by the complexity of the Gaussian elimination algorithm, in $O(d^3)$. Indeed, some abstract primitives manipulate the abstract elements as a system of linear inequalities, while some others consider them as affine subspaces of $\mathbb{R}^d$ generated by a point and some vectors. The Gaussian elimination algorithm allows to pass from one of the form to the other.

(d) the *abstract domain of zones*, introduced by Mine in [Min01a], is able to express properties of the form $\boldsymbol{v}_i - \boldsymbol{v}_j \geq a$, and $b \leq \boldsymbol{v}_i \leq -c$, with $a, b, c \in \mathbb{R} \cup \{ -\infty \}$. They represent closed convex polyhedra such that the slope of each edge is a vector of $\{ 0, 1 \}^d$.



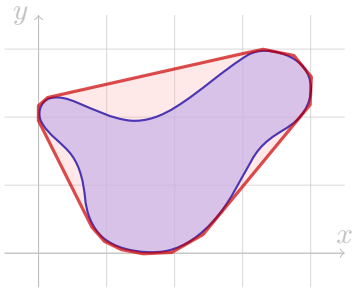
We give further details on this abstract domain in Section 6.3.2.b. The complexity of the abstract primitives is also dominated by $O(d^3)$.

(e) the *abstract domain of octagons* [Min01b] is an extension of the latter domain to inequalities of the form $\pm\boldsymbol{v}_i \pm \boldsymbol{v}_j \geq a$ and $b \leq \boldsymbol{v}_i \leq -c$. Geometrically, the only difference is that edges can also have a slope with some coordinates equal to -1.



The complexity in the worst case is $O(d^3)$.

(f) the *abstract domain of convex polyhedra* has been introduced by Cousot and Halbwachs in [CH78]. Its abstract elements represent closed convex polyhedra, and represents affine inequality invariants over the variables $\boldsymbol{v}_i$, *i.e.* of the form $a_1\boldsymbol{v}_1 + \ldots + a_d\boldsymbol{v}_d \leq b$.



Similarly to Karr's domain, the complexity of the abstract domain of convex polyhedra is dominated by the complexity of the algorithms passing from the description by systems of inequalities to the representation by system of generators (*i.e.* vertices and rays), *i.e.* exponential in d in the worst case. Contrary to the previous abstract domains, there is in general no best possible abstraction of a given set in the domain of convex polyhedra. For instance, in dimension $d = 2$, a circle can be over-approximated by several and incomparable polyhedra whose facets are tangent to the circle.

We have omitted several other domains, such as the domain of *constants* [Kil73], *congruences over rationals* [Gra97], *octahedrons* [CC04], *affine inequalities with at most two variables* [SKH03], *linear congruence equalities* [Gra91], *linear programming templates* [SSM05], *etc.*

Each numerical abstract domain corresponds to a certain trade-off between precision and scalability: in general, the more precise the over-approximation is, the worse its computational complexity is. One of the most typical representative of this principle is the abstract domain of convex polyhedra. In practice, it provides very precise over-approximations, since it is able to capture any affine inequality invariants. Unfortunately, scaling up to problems with more than 20 variables is almost impossible because of its exponential complexity. Historically, this is the reason why more efficient subaffine abstract domains, such as those of zones or octagons, have been introduced.

As discussed in Chapter 1, most of these numerical abstract domains are formed by elements representing convex subsets of $\mathbb{R}^d$, since they basically infer conjunctions of (sub)affine (in)equalities over the variables $\boldsymbol{v}_i$. As a consequence, they can not precisely over-approximate disjunctive invariants, such as properties corresponding to non-convex subsets of $\mathbb{R}^d_{\max}$. A systematic way to lift non-disjunctive abstract domains to disjunctive ones is to use disjunctive completion [CC79]. Using the notations of Section 6.3.1, the disjunctive completion of an abstract domain $(\mathcal{D}, \gamma)$ is defined as the pair $(\wp(\mathcal{D}), \gamma')$, where

$$\gamma'(\{\, \mathcal{X}_1, \ldots, \mathcal{X}_n \,\}) \stackrel{def}{=} \bigvee_{i=1}^{n} \gamma(\mathcal{X}_i).$$

In other words, disjunctions are represented as sets of abstract elements of $\mathcal{D}$. Unfortunately, the complexity of this abstract domain is most often prohibitive, since the size of its abstract

elements may grow arbitrarily (in particular, when computing a fixpoint following the iterative scheme of Kleene's Theorem). Some heuristics can be used to bound it, however they usually lead to a uncontrolled loss of precision.

6.3.2.b More details on the abstract domains of zones and octagons. As mentioned above, the numerical abstract domain of zones, denoted by Zone, allows to express relations of the form $\boldsymbol{v}_i - \boldsymbol{v}_j \geq A_{ij}$, and $\boldsymbol{b}_i \leq \boldsymbol{v}_i \leq -\boldsymbol{c}_i$, where $A = (A_{ij})$ is a matrix, and $\boldsymbol{b} = (\boldsymbol{b}_i)$ and $\boldsymbol{c} = (\boldsymbol{c}_i)$ are vectors with coefficients in $\mathbb{R} \cup \{ -\infty \}$.[9]

Each abstract element is usually encoded as a matrix of $(\mathbb{R} \cup \{ -\infty \})^{(d+1)\times(d+1)}$, where the $(d+1)$-column and row are respectively the vectors $\boldsymbol{b}$ and ${}^t\boldsymbol{c}$. The concretization of an abstract element is defined by:

$$\gamma_{\mathsf{Zone}}\begin{pmatrix} A & \boldsymbol{b} \\ {}^t\boldsymbol{c} & d \end{pmatrix} \stackrel{def}{=} \begin{cases} \left\{ \nu \in \mathbb{R}^{\mathsf{Vars}} \;\middle|\; \begin{array}{ll} A_{ij} \leq \nu(\boldsymbol{v}_i) - \nu(\boldsymbol{v}_j) & \text{for } i,j = 1,\ldots,d \\ \boldsymbol{b}_i \leq \nu(\boldsymbol{v}_i) \leq -\boldsymbol{c}_i & \text{for } i = 1,\ldots,d \end{array} \right\} & \text{if } d \leq 0, \\ \emptyset & \text{otherwise.} \end{cases} \tag{6.10}$$

Abstract elements of Zone can be partially ordered by the point-wise extension of the order $\geq$ on each coefficient of the matrices: $M \sqsubseteq_{\mathsf{Zone}} M'$ if and only if $M_{ij} \geq M'_{ij}$ for all $(i,j) \in [d]^2$.

Observe that two abstract elements M and M' can represent the same concrete elements, *i.e.* $\gamma_{\mathsf{Zone}}(M) = \gamma_{\mathsf{Zone}}(M')$, while M is smaller than M' w.r.t. $\sqsubseteq_{\mathsf{Zone}}$.

Example 6.7. When $d = 3$, consider the two following abstract elements:

$$M = \begin{pmatrix} 0 & 1 & 3 & -\infty \\ -\infty & 0 & 2 & -\infty \\ -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & 0 \end{pmatrix}, \quad M' = \begin{pmatrix} 0 & 1 & 0 & -\infty \\ -\infty & 0 & 2 & -\infty \\ -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & 0 \end{pmatrix}.$$

Then the two abstract elements exactly represent the sets of environments ν such that $\nu(\boldsymbol{v}_1) - \nu(\boldsymbol{v}_2) \geq 1$, $\nu(\boldsymbol{v}_2) - \nu(\boldsymbol{v}_3) \geq 2$, and $\nu(\boldsymbol{v}_1) - \nu(\boldsymbol{v}_3) \geq 3$. Note that the third constraint is implied by the two first ones, and is more precise than the weaker constraint $\nu(\boldsymbol{v}_1) - \nu(\boldsymbol{v}_3) \geq 0$ in the abstract element M'. The latter can be safely refined into M, while preserving the over-approximation property.

Given an abstract element $M \in \mathsf{Zone}$ such that $\gamma_{\mathsf{Zone}}(M)$ is not empty, there exists a least abstract element representing the same subset of $\mathbb{R}^{\mathsf{Vars}}$. It is provided thanks to a closure operator η_{Zone}, which maps any M to the element of Zone defined by:

$$(\eta_{\mathsf{Zone}}(M))_{ij} \stackrel{def}{=} \begin{cases} 0 & \text{if } i = j, \\ \max\left\{ \sum_{l=1}^{n-1} M_{k_l k_{l+1}} \mid k_1 = i, k_2, \ldots, k_{n-1}, k_n = j \right\} & \text{otherwise.} \end{cases} \tag{6.11}$$

The reader will notice that this closure operator is similar to the maximal weight algorithm applied on the associated graph G, in which each edge $i \to j$ is assigned with the weight M_{ij}. It can be also written as the limit of the sum $\bigoplus_{n \geq 0} M^n$, where M is seen as a matrix in $\mathbb{R}_{\max}^{(n+1)\times(n+1)}$.

[9] Note in their initial presentation [Min01a], zone invariants were given under the form $\boldsymbol{v}_i - \boldsymbol{v}_j \leq A'_{ij}$, with $A'_{ij} \in \mathbb{R} \cup \{ +\infty \}$. Up to considering the opposite inequalities, this presentation is naturally equivalent to ours.

In contrast, the emptyness of $\gamma_{\mathsf{Zone}}(M)$ can be characterized by the existence of a circuit with a strictly positive weight in the graph previously mentioned.[10] Discovering such a circuit can be implemented in the operator η_{Zone}. In that case, η_{Zone} maps the element M to a special element $\bot_{\mathsf{Zone}}$. The latter is a canonical represention of the emptyset: $\gamma_{\mathsf{Zone}}(\bot_{\mathsf{Zone}}) \stackrel{def}{=} \emptyset$.

The closure operator has a complexity in $O(d^3)$. An abstract element M which satisfies $\eta_{\mathsf{Zone}}(M) = M$ is said to be *under closed form*. We refer the reader to Mine's thesis [Min04] for a discussion on the algorithmic details.

The abstract domain of octagons, denoted by Oct, is a generalization of the domain Zone, since it can be seen as expressing zone invariants over the variables $\boldsymbol{v}_i$ and their opposite $-\boldsymbol{v}_i$. Its elements are therefore encoded by elements of the abstract domain Zone in dimension $2d$, the last d coordinates representing the opposite variables $-\boldsymbol{v}_1, \ldots, -\boldsymbol{v}_d$:

$$\gamma_{\mathsf{Oct}}(M) \stackrel{def}{=} \{\, \nu \in \mathbb{R}^{\mathsf{Vars}} \mid (\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d), -\nu(\boldsymbol{v}_1), \ldots, -\nu(\boldsymbol{v}_d)) \in \gamma_{\mathsf{Zone}}(M) \,\}.$$

A closure operator on Oct is also defined. It extends its analogue η_{Zone}, by enabling a communication between the coordinates encoding the $\boldsymbol{v}_i$ and those representing their opposite.

6.3.2.c Additional primitives on numerical abstract domains. Following the principles discussed in Section 6.3.1, numerical abstract domains are equipped with an abstract (pre)order, abstract join and meet, least and greatest abstract elements, and widening and narrowing operators. They are also provided with primitives over-approximating the side-effect of conditions and assignments on environments, and which will be used to define abstract counterparts of the rules of the concrete semantics given in Figures 6.5 and 6.6.

Assignments. An assignment $\boldsymbol{v}_i \leftarrow e$ on an environment $\nu : \mathsf{Vars} \to \mathbb{R}$ consists in replacing the value of $\boldsymbol{v}_i$ in ν by the value v obtained when evaluating the expression e in the environment ν. Expressions over the set Vars of variables are defined similarly to expressions in our kernel language (see Figure 6.1), *i.e.* by a syntactic rule of the form:

$$\begin{array}{rcll} e & ::= & c & \text{constant} \\ & | & \boldsymbol{v}_i & \text{variable} \\ & | & \mathsf{op}(e_1, e_2) & \text{binary operation} \end{array}$$

where op denotes usual binary operations over $\mathbb{R}$, such as addition, substraction, multiplication, *etc*. The value v of an expression in a given environment is therefore given by the relation $\nu \vdash e \Rightarrow v$ defined in Figure 6.4. Given a numerical abstract domain $(\mathcal{D}, \gamma)$, it will be therefore supposed that there exists an abstract assignment primitive, denoted by $(\!|\boldsymbol{v}_i \leftarrow e|\!)$, from $\mathcal{D}$ to itself, and which satisfies the following soundness property:

$$\{\, \nu[\boldsymbol{v}_i \mapsto v] \mid \nu \in \gamma(\mathcal{X}),\ \nu \vdash e \Rightarrow v \,\} \subset \gamma((\!|\boldsymbol{v}_i \leftarrow e|\!)(\mathcal{X}))$$

for every $\mathcal{X} \in \mathcal{D}$.

Example 6.8. In the abstract domain of intervals, an abstract element is represented by a tuple of d intervals, each of them representing lower and upper bounds of the variables $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d$.

[10] That is why in particular the concretization of M in (6.10) is empty when the diagonal coefficient d is strictly positive.

Starting from an abstract element $\mathcal{X}$, $(\!|\boldsymbol{v}_i \leftarrow e|\!)(\mathcal{X})$ can be obtained by: (i) evaluating the expression e as an interval, *i.e.* replacing each variable $\boldsymbol{v}_j$ in e by the corresponding interval and then evaluating it in interval arithmetics, (ii) and then updating in $\mathcal{X}$ the interval associated to $\boldsymbol{v}_i$ by the result.

Assignments can also be *parallel*. In that case, they represent several assignments $\boldsymbol{v}_{i_1} \leftarrow e_1$, $\ldots$, $\boldsymbol{v}_{i_n} \leftarrow e_n$ performed simultaneously on an environment. The corresponding abstract primitive will be denoted by $(\!|\boldsymbol{v}_{i_1} \leftarrow e_1, \ldots, \boldsymbol{v}_{i_n} \leftarrow e_n|\!)$, and must satisfy the soundness property: for any $\mathcal{X} \in \mathcal{D}$,

$$\{\, \nu[\boldsymbol{v}_{i_k} \mapsto v_k]_{k=1,\ldots,n} \mid \nu \in \gamma(\mathcal{X}),\ \nu \vdash e_k \Rightarrow v_k \text{ for all } k = 1, \ldots, n \,\} \subset \gamma((\!|\boldsymbol{v}_{i_1} \leftarrow e_1, \ldots, \boldsymbol{v}_{i_n} \leftarrow e_n|\!)(\mathcal{X})).$$

A last class of assignments are *non-deterministic assignments*, which correspond to the update of a variable $\boldsymbol{v}_i$ by an arbitrary value $v \in \mathbb{R}$. The abstract primitive will be denoted by $(\!|\boldsymbol{v}_i \leftarrow ?|\!)$, and will have to be such that, for all $\mathcal{X} \in \mathcal{D}$,

$$\{\, \nu[\boldsymbol{v}_i \mapsto v] \mid \nu \in \gamma(\mathcal{X}),\ v \in \mathbb{R} \,\} \subset \gamma((\!|\boldsymbol{v}_i \leftarrow ?|\!)(\mathcal{X})).$$

Example 6.9 (Continuing Example 6.8). In interval domain, the abstract non-deterministic assignment $(\!|\boldsymbol{v}_i \leftarrow ?|\!)(\mathcal{X})$ on the abstract element $\mathcal{X}$ can be defined as replacing the interval associated to $\boldsymbol{v}_i$ in $\mathcal{X}$ by the interval $]-\infty; +\infty[$.

Conditions. Conditions are also built under similar rules to those defined in Figure 6.1, typically:

$$\begin{array}{rcll} \mathit{cond} & ::= & e_1\ (\leq \mid = \mid \geq)\ e_2 & \text{comparison of expressions} \\ & \mid & \mathit{cond}_1 \wedge \mathit{cond}_2 & \text{conjunction} \\ & \mid & \mathit{cond}_1 \vee \mathit{cond}_2 & \text{disjunction} \end{array}$$

The fact that an environment ν satisfies a condition *cond* is denoted by $\nu \models \mathit{cond}$, and is defined by the following inference rules, analogous to the corresponding rules in the concrete semantics (Figure 6.6):

$$\frac{\nu \vdash e_1 \Rightarrow v_1 \qquad \nu \vdash e_2 \Rightarrow v_2 \qquad v_1 \diamond v_2}{\nu \models e_1 \diamond e_2} \quad \text{where } \diamond \in \{\leq, =, \geq\}$$

$$\frac{\nu \models \mathit{cond}_1 \qquad \nu \models \mathit{cond}_2}{\nu \models \mathit{cond}_1 \wedge \mathit{cond}_2}$$

$$\frac{\nu \models \mathit{cond}_i}{\nu \models \mathit{cond}_1 \vee \mathit{cond}_2} \quad \text{for } i = 1, 2$$

The abstract primitive $(\!|\mathit{cond}|\!)$ associated to the condition *cond* must satisfy the following soundness property: for all $\mathcal{X} \in \mathcal{D}$,

$$\{\, \nu \in \gamma(\mathcal{X}) \mid \nu \models \mathit{cond} \,\} \subset \gamma((\!|\mathit{cond}|\!)(\mathcal{X})).$$

Observe that the abstract primitives $(\!|\mathit{cond}_1 \wedge \mathit{cond}_2|\!)$ and $(\!|\mathit{cond}_1 \vee \mathit{cond}_2|\!)$ can be generally built from each abstract function $(\!|\mathit{cond}_i|\!)$ using abstract meet and join operations, as follows:

for all $\mathcal{X} \in \mathcal{D}$,

$$\begin{aligned}(\!|cond_1 \wedge cond_2|\!)(\mathcal{X}) &\stackrel{def}{=} (\!|cond_1|\!)(\mathcal{X}) \sqcap (\!|cond_2)(\mathcal{X}),\\ (\!|cond_1 \vee cond_2|\!)(\mathcal{X}) &\stackrel{def}{=} (\!|cond_1|\!)(\mathcal{X}) \sqcup (\!|cond_2)(\mathcal{X}).\end{aligned}$$

The construction of the abstract primitives $(\!|e_1 \diamond e_2|\!)$ depends on the nature of the abstract domain.

6.4 A first possible abstract semantics

In this section, we give a first example of abstraction on the kernel language of Section 6.1. The principle of the abstraction is the following:

- for the stack, integer variables are over-approximated by means of a numerical abstract domain, such as one of those presented in Section 6.3.2.a.
- for the heap, the exact content of arrays is totally forgotten. The only numerical information which is kept in the abstraction is the size of the arrays.

This abstraction is rough regarding the content of the arrays, but is nevertheless able to infer properties between integer variables and the size of the arrays. Therefore, it will be maybe able to prove the absence of some heap overflows.

Abstract domain. Formally, the abstraction is parametrized by a numerical abstract domain Num, whose aim is to express numerical information over the integer variables of $\mathsf{IntVars}$, and *ghost* variables representing the size of arrays. The latter form the set denoted by $\mathsf{SizeVars}$. By convention, the size of the array $t \in \mathsf{ArrayVars}$ will be represented by the variable $sz_t \in \mathsf{SizeVars}$.

We suppose that the numerical abstract domain Num is provided with a concretization operator $\gamma_{\mathsf{Num}} : \mathsf{Num} \to \wp(\mathbb{Z}^{\mathsf{IntVars} \cup \mathsf{SizeVars}})$, an abstract order $\sqsubseteq$, and abstract primitives $\sqcup$, $\sqcap$, $\bot$, $\top$, ∇, and Δ, which satisfy all the properties given in Section 6.3.1. As explained in Section 6.3.2.c, we also suppose that Num is provided with abstract operations which soundly over-approximate the side-effect of assignments and conditions. Finally, we assume that $\sqcup$, $\sqcap$, and the assignment and condition abstract primitives are all monotone.

The abstract domain over-approximating memory states is denoted by AMem. The set AMem is formed by the abstract elements of the domain Num, and its concretization operator $\gamma_{\mathsf{Mem}} : \mathsf{AMem} \to \wp(\mathsf{Mem})$ gives the meaning of an abstract memory state $\mathcal{M}$:

$$\gamma_{\mathsf{Mem}}(\mathcal{M}) \stackrel{def}{=} \left\{ s \natural h \in \mathsf{Mem} \;\middle|\; \begin{array}{l} \nu \in \gamma_{\mathsf{Num}}(\mathcal{M}),\ \forall x \in \mathsf{IntVars}, s(x) = \nu(x), \\ \text{for all } t \in \mathsf{ArrayVars},\ \nu(sz_t) \geq 0, \\ \text{and } (t,i) \in \mathsf{dom}(h) \text{ iff } 0 \leq i \leq \nu(sz_t) - 1 \end{array} \right\}. \tag{6.12}$$

Example 6.10. Assume that $\mathsf{IntVars}$ is reduced to a single integer variable x, and that $\mathsf{ArrayVars}$ consists of only one array t. Suppose that Mem is instantiated with the abstract domain of convex polyhedra. Then the abstract element $\mathcal{M}$ represented by the system of inequalities:

$$\begin{cases} 2 \leq x \leq 5 \\ 3 \leq sz_t \leq x + 1 \end{cases}$$

represents the set of the memory states $s \shortmid h$ in which the value of $s(x)$ is bounded between 2 and 5, and the array t has a size between 3 and $x+1$. For instance, if $s(x) = 3$, then the heap h is an arbitrary function defined over the addresses $(t,0), \ldots, (t,2)$, and also possibly over $(t,3)$.

The abstract domain of semantics states is denoted by $\mathsf{AStates}$. It is formed by the functions from Ctrl to AMem, and its concretization operator $\gamma : \mathsf{AStates} \to \wp(\mathsf{States})$ is defined as follows:

$$\gamma(\mathcal{X}) \stackrel{def}{=} \left\{ \begin{array}{rcl} \mathsf{Ctrl} & \to & \wp(\mathsf{Mem}) \\ \ell & \mapsto & \gamma_{\mathsf{Mem}}(\mathcal{X}(\ell)) \end{array} \right. .^{11} \tag{6.13}$$

The partial ordering $\mathsf{AStates}$, denoted by $\dot{\sqsubseteq}$, is defined as the lift of the ordering $\sqsubseteq$ over Ctrl: $\mathcal{X} \dot{\sqsubseteq} \mathcal{Y}$ if and only if for any $\ell \in \mathsf{Ctrl}$, $\mathcal{X}(\ell) \sqsubseteq \mathcal{Y}(\ell)$.

Abstract join, meet, widening, and narrowing operators, as well as least and top elements on $\mathsf{AStates}$ can be defined using the corresponding primitives in Num: given $\mathcal{X}, \mathcal{Y} \in \mathsf{AStates}$ and $\ell \in \mathsf{Ctrl}$,

$$\dot{\bot}(\ell) \stackrel{def}{=} \bot, \qquad \dot{\top}(\ell) \stackrel{def}{=} \top,$$
$$(\mathcal{X} \mathbin{\dot{\sqcup}} \mathcal{Y})(\ell) \stackrel{def}{=} \mathcal{X}(\ell) \sqcup \mathcal{Y}(\ell), \qquad (\mathcal{X} \mathbin{\dot{\sqcap}} \mathcal{Y})(\ell) \stackrel{def}{=} \mathcal{X}(\ell) \sqcap \mathcal{Y}(\ell),$$
$$(\mathcal{X} \mathbin{\dot{\nabla}} \mathcal{Y})(\ell) \stackrel{def}{=} \mathcal{X}(\ell) \nabla \mathcal{Y}(\ell), \qquad (\mathcal{X} \mathbin{\dot{\Delta}} \mathcal{Y})(\ell) \stackrel{def}{=} \mathcal{X}(\ell) \Delta \mathcal{Y}(\ell).$$

Abstract transfer function. Now, given a program P, we are going to build a sound abstraction $\mathcal{F}$ of the transfer function F defined by (6.4): for each $\mathcal{X} \in \mathsf{AStates}$, and for any $\ell' \in \mathsf{Ctrl}$, we define

$$\mathcal{F}(\mathcal{X})(\ell') \stackrel{def}{=} \begin{cases} (\!| sz_t \leftarrow 0 \text{ for all } t \in \mathsf{ArrayVars} |\!)(\top) & \text{if } \ell' = \mathit{entry}(P), \\ \bigsqcup\limits_{\langle \ell, \mathit{step}, \ell' \rangle} [\![\mathit{step}]\!](\mathcal{X}(\ell)) & \text{otherwise.} \end{cases}$$

By convention, when an array is not allocated, its size is supposed to be equal to 0. Therefore, the abstract memory state $(\!| sz_t \leftarrow 0 \text{ for all } t \in \mathsf{ArrayVars} |\!)(\top)$ is a sound approximation of the set of initial memory states. It can be verified that its concretization is actually formed by memory states in which the variables in the stack have arbitrary values, while the heap is empty.

If *step* is an allocation, an instruction, or a condition, then the function $[\![\mathit{step}]\!] : \mathsf{AMem} \to \mathsf{AMem}$, defined in Figure 6.9, is an abstraction of the effect of the allocation, instruction, or condition *step*:

Let us comment these definitions:

- the assignment $x := e$ modifies the value of x in the abstract state $\mathcal{M}$ using the corresponding primitive defined in the domain Num.
- the non-deterministic assignment $x := \mathsf{read}()$ is translated into the corresponding abstract primitive on Num.
- for the assignment $x := t[e]$, it is first verified that the array element $t[e]$ is indeed allocated. Then a non-deterministic assignment on x is performed, since we do not have any information on the exact value of $t[e]$.

[11] We here implicitly use the correspondence between $\wp(\mathsf{States})$ and $\wp(\mathsf{Mem})^{\mathsf{Ctrl}}$.

$$
\begin{aligned}
[\![x \coloneqq e]\!](\mathcal{M}) &\stackrel{def}{=} (\!|x \leftarrow e|\!)(\mathcal{M}) \\
[\![x \coloneqq \mathsf{read}()]\!](\mathcal{M}) &\stackrel{def}{=} (\!|x \leftarrow ?\,|\!)(\mathcal{M}) \\
[\![x \coloneqq t[e]]\!](\mathcal{M}) &\stackrel{def}{=} (\!|x \leftarrow ?|\!) \circ (\!|0 \leq e \leq sz_t - 1|\!)(\mathcal{M}) \\
\\
[\![t[e_1] \coloneqq e_2]\!](\mathcal{M}) &\stackrel{def}{=} (\!|0 \leq e_1 \leq sz_t - 1|\!)(\mathcal{M}) \\
[\![t[e] \coloneqq \mathsf{read}()]\!](\mathcal{M}) &\stackrel{def}{=} (\!|0 \leq e \leq sz_t - 1|\!)(\mathcal{M}) \\
[\![t_1[e_1] \coloneqq t_2[e_2]]\!](\mathcal{M}) &\stackrel{def}{=} (\!|(0 \leq e_1 \leq sz_{t_1} - 1) \wedge (0 \leq e_2 \leq sz_{t_2} - 1)|\!)(\mathcal{M}) \\
[\![t[e_1] \diamond e_2]\!](\mathcal{M}) &\stackrel{def}{=} (\!|0 \leq e_1 \leq sz_t - 1|\!)(\mathcal{M}) \qquad \text{for } \diamond \in \{=, \neq\} \\
\\
[\![t \coloneqq \mathsf{malloc}(e)]\!](\mathcal{M}) &\stackrel{def}{=} (\!|sz_t \leftarrow e|\!) \circ (\!|e \geq 1|\!)(\mathcal{M}) \\
\\
[\![e_1 \diamond e_2]\!](\mathcal{M}) &\stackrel{def}{=} (\!|e_1 \diamond e_2|\!)(\mathcal{M}) \qquad \text{for } \diamond \in \{\leq, =, \geq\} \\
[\![cond_1 \wedge cond_2]\!](\mathcal{M}) &\stackrel{def}{=} (\!|cond_1|\!)(\mathcal{M}) \sqcap (\!|cond_2|\!)(\mathcal{M}) \\
[\![cond_1 \vee cond_2]\!](\mathcal{M}) &\stackrel{def}{=} (\!|cond_1|\!)(\mathcal{M}) \sqcup (\!|cond_2|\!)(\mathcal{M})
\end{aligned}
$$

Figure 6.9: Abstract primitives for allocations, intructions, and conditions

- the instructions $t[e_1] \coloneqq e_2$, $t[e] \coloneqq \mathsf{read}()$, $t_1[e_1] \coloneqq t_2[e_2]$, and the condition $t[e_1] \diamond e_2$ only filter the memory states for which the involved indexes are whithin the bounds of the corresponding arrays.
- the allocation $t \coloneqq \mathsf{malloc}(e)$ filters the values of e which are strictly positive (since according to our convention, $\mathsf{malloc}(0)$ stops the machine). It then overwrites the size of the array t to the value of e.
- the conditions $e_1 \diamond e_2$, $cond_1 \wedge cond_2$, and $cond_1 \vee cond_2$ are translated into the corresponding operations on Num.

The soundness of the abstract operator $[\![step]\!]$ is ensured by the following lemma:

Lemma 6.3. *Let $\mathcal{M} \in \mathsf{AMem}$. Then:*

$$
\left\{ s' \restriction h' \mid s \restriction h \in \gamma_{\mathsf{Mem}}(\mathcal{M}) \text{ and } s \restriction h \vdash step : s' \restriction h' \right\} \subset \gamma_{\mathsf{Mem}}\left([\![step]\!](\mathcal{M})\right). \tag{6.14}
$$

It can be also verified that the function $\mathcal{F}$ is monotone, as a combination of the monotone abstract primitives $\sqcup$, $\sqcap$, $(\!|\cdot \leftarrow \cdot|\!)$, and $(\!|cond|\!)$. It follows that the function $\mathcal{F}$ is itself a sound abstraction of the concrete transfer function F.

Proposition 6.4. *Let $\mathcal{X} \in \mathsf{AStates}$. Then we have:*

$$
F(\gamma(\mathcal{X})) \subset \gamma(\mathcal{F}(\mathcal{X})). \tag{6.15}
$$

Proof. First note that when $\ell' = entry(P)$, we clearly have:

$$\begin{aligned} F(\gamma(\mathcal{X}))(\ell') &= \{\, s_0 \natural \emptyset \mid s_0 \in \mathsf{Stack} \,\} \\ &\subset \gamma_{\mathsf{Mem}}(\!|sz_t \leftarrow 0 \text{ for all } t \in \mathsf{ArrayVars}|\!)(\top)) \\ &= \gamma_{\mathsf{Mem}}(\mathcal{F}(\mathcal{X})(\ell')). \end{aligned}$$

Besides, if $\ell' \neq entry(P)$,

$$\begin{aligned} F(\gamma(\mathcal{X}))(\ell') &= \bigcup_{s \natural h \in \gamma(\mathcal{X})(\ell)} \{\, s' \natural h' \mid (\ell, s \natural h) \rightarrow (\ell', s' \natural h') \,\} \\ &= \bigcup_{\substack{\langle \ell, step, \ell' \rangle \\ s \natural h \in \gamma_{\mathsf{Mem}}(\mathcal{X}(\ell))}} \{\, s' \natural h' \mid s \natural h \vdash step : s' \natural h' \,\} && \text{using (6.1), (6.2), and (6.13)} \\ &\subset \bigcup_{\langle \ell, step, \ell' \rangle} \gamma_{\mathsf{Mem}}\left([\![step]\!](\mathcal{X}(\ell))\right) && \text{according to Lemma 6.3} \\ &\subset \gamma_{\mathsf{Mem}}\left(\bigsqcup_{\langle \ell, step, \ell' \rangle} [\![step]\!](\mathcal{X}(\ell) \right) && \text{by soundness of } \sqcup \\ &= \gamma_{\mathsf{Mem}}(\mathcal{F}(\mathcal{X})(\ell')). \end{aligned}$$

As a consequence, we have $F(\gamma(\mathcal{X})) \subset \gamma(\mathcal{F}(\mathcal{X}))$. □

Propositions 6.1 and 6.2 then allow to compute a sound abstraction of the collecting semantics $C(P)$. This element of $\mathsf{AStates}$ will be denoted $\mathcal{C}(P)$:

$$C(P) \subset \gamma(\mathcal{C}(P)).$$

Example 6.11. Consider the program defined in Example 6.1. Supposing that Num is instantiated by the abstract domain of closed convex polyhedra, its abstract collecting semantics is depicted in Figure 6.10.

Ensuring the absence of heap overflows. Once the abstract collecting semantics has been computed, the abstract memory states $\mathcal{C}(P)(\ell)$ represent program invariants at each control point $\ell \in \mathsf{Ctrl}$. These invariants can be used to prove that there is no heap overflow.

Indeed, consider a program control point ℓ of an instruction or a condition involving an expression of the form $t[e]$. If the abstract memory states at ℓ satisfies

$$(\!|e \leq -1 \vee e \geq sz_t|\!)(\mathcal{C}(P)(\ell)) = \bot,$$

then all the memory states represented by $\mathcal{C}(P)(\ell)$ necessarily satisfy $0 \leq e \leq sz_t - 1$. Indeed, the concretization of the memory states satisfying $e \leq -1 \vee e \geq sz_t$ is equal to $\emptyset$. Since $\mathcal{C}(P)(\ell)$ over-approximates all the possible memory states at the control point ℓ, this proves that the array lookup $t[e]$ is safe.

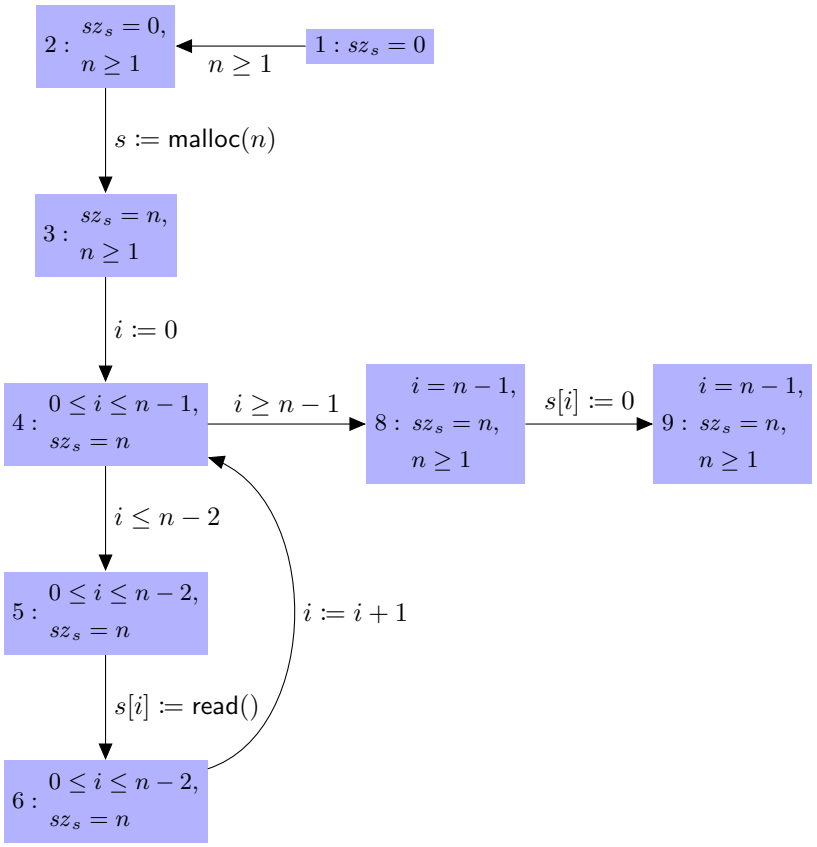


Figure 6.10: The abstract collecting semantics of our running example, using the first abstraction

Using this principle, the following properties on the abstract semantics ensure that there is no heap overflow during any execution of the program P:

$$\begin{cases} (\!|e \leq -1 \vee e \geq sz_t|\!)(\mathcal{C}(P)(\ell)) = \bot & \text{if } \textit{step} \text{ is of the form } x := t[e] \text{ or } t[e] := \mathsf{read}() \\ (\!|e_1 \leq -1 \vee e_1 \geq sz_t|\!)(\mathcal{C}(P)(\ell)) = \bot & \text{if } \textit{step} \text{ is of the form } t[e_1] := e_2 \text{ or } t[e_1] \diamond e_2 \\ (\!|e_1 \leq -1 \vee e_1 \geq sz_{t_1}|\!)(\mathcal{C}(P)(\ell)) = \bot \\ (\!|e_2 \leq -1 \vee e_2 \geq sz_{t_2}|\!)(\mathcal{C}(P)(\ell)) = \bot & \text{if } \textit{step} \text{ is of the form } t_1[e_1] := t_2[e_2] \end{cases} \tag{6.16}$$

Example 6.12. In the abstract collecting semantics provided in Example 6.1, it can be verified that for any access $t[i]$ at index i, the invariants ensure that the property $i \leq -1 \vee i \geq sz_t$ never holds. As a consequence, there is no heap overflow.

6.5 An abstraction on strings

The abstraction which has been defined in the previous section is not precise regarding the heap, since its whole content is forgotten. In this section, we propose a tighter abstraction which allows to track information on the length of the strings which are stored in the heap.

In programming languages like C, strings refer to a sequence of characters. Assimilating here integers and characters (supposing for instance that characters are encoded by their ASCII code) allows us to consider that the arrays of our kernel language store strings. The *null character*, which is given by the integer 0, plays the role of the delimiter of the end of strings, and any character stored after it is meaningless, and may have an arbitary value. The *length* of a string is defined as the position of the first null character in the array (starting from the index 0). To avoid confusion between the null character and the integer encoding the character 0 (*i.e.* 48 in the ASCII standard), the latter will be denoted by '0'.

The length of a string terminated by a null character is always strictly less than the size of the array. If the array does not contain any null character, we use the convention that the length of the string is equal to the size of the array. Informally, for a given array $t \in \mathsf{ArrayVars}$, the length of the string is therefore given by $\min(\{\, i \mid t[i] = 0 \,\} \cup \{\, sz_t \,\})$.

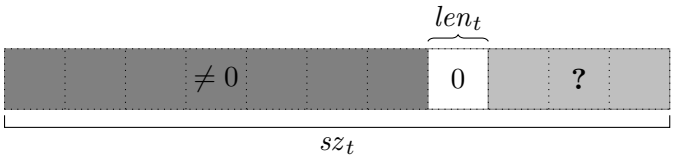
Example 6.13. Consider the following array:

'E'	'X'	'A'	'M'	'P'	'L'	'E'	0	'm'	'&'	'/'	'2'	'('	'%'	'P'

Its size is equal to 15. It contains the string EXAMPLE, whose length is equal to 7. All the characters located after the null terminal character are irrelevant.

Example 6.14. Consider the program defined in Example 6.1 as manipulating strings. The fact that the last character is assigned to 0 at Line 8 allows to ensure that the length of the string stored in the array t is less than or equal to $n - 1$.

Let us discuss the principle of the string abstraction. For each array t, we associate a new ghost variable, denoted by len_t, which will represent the length of the string stored in the array t. The set of such variables is denoted by $\mathsf{LenVars}$, and is supposed to be disjoint from $\mathsf{IntVars}$ and $\mathsf{SizeVars}$. The principle of the abstraction is now to approximate the content of the arrays by inferring numerical relations between their size, the length of the string which is stored in, and the integer variables of the program. Intuitively, the content of an array in the abstract world will be seen as follows:



The dark gray area contains only non-null characters, while the white one represents the first null character (its index is equal to len_t). Besides, we do not have precise information on the rest of the array (in light gray): it may contain both non-null and null characters.

This abstraction is not really new. It follows the principles of the string analyses developed by Simon *et al.* [SK02], Dor *et al.* [DRS03], and Allamigeon *et al.* [AGH06]. The major novelty is that ours is described in a general setting, *i.e.* for any underlying numerical abstract domain. In contrast, the existing abstractions are all presented with a particular instantiation (convex polyhedra for the two first ones, intervals for the last one).

Abstract domain. Our abstraction is still parametrized by a numerical abstract domain Num, provided with a concretization operator γ_{Num} which maps elements of Num to subsets of $\mathbb{Z}^{\mathsf{IntVars} \cup \mathsf{SizeVars} \cup \mathsf{LenVars}}$.

The abstract domain of memory states AMem is still formed by elements of Num, but its concretization operator $\gamma_{\mathsf{Mem}} : \mathsf{AMem} \to \wp(\mathsf{Mem})$ is now refined thanks to the information of the length of the string stored in each array:

$$\gamma_{\mathsf{Mem}}(\mathsf{Mem}) \stackrel{def}{=} \left\{ s \shortmid h \in \mathsf{Mem} \;\middle|\; \begin{array}{l} \nu \in \gamma_{\mathsf{Num}}(\mathcal{M}),\ \forall x \in \mathsf{IntVars}, s(x) = \nu(x), \\ \text{for all } t \in \mathsf{ArrayVars},\ \nu(sz_t) \geq 0, \\ (t,i) \in \mathsf{dom}(h) \text{ iff } 0 \leq i \leq \nu(sz_t), \\ \text{and } \nu(len_t) = \min(\{\, i \mid h(t,i) = 0 \,\} \cup \{\, \nu(sz_t) \,\}) \end{array} \right\}. \quad (6.17)$$

For each array $t \in \mathsf{ArrayVars}$, the array elements whose index is strictly less than the length len_t must be non-null, while the element of index len_t must contain the value 0 (if it is allocated).

Example 6.15. Suppose that Num is the abstract domain of convex polyhedra, and consider the abstract memory state $\mathcal{M}$ given by the following system of affine constraints:

$$\begin{cases} 1 \leq n \\ n = sz_t \\ 0 \leq len_t \leq sz_t - 1 \end{cases}$$

It represents the set of memory states $s \shortmid h$ in which the value of $s(n)$ is greater than 1, the heap is precisely defined on the addresses $(t,0), \ldots, (t, s(n)-1)$, and is such that there exists at least one index i between 0 and $s(n)-1$ such that $h(t,i)$ is equal to 0.

The abstract domain of semantics states $\mathsf{AStates}$ and the abstract primitives $\dot{\sqsubseteq}$, $\dot{\sqcup}$, $\dot{\sqcap}$, $\dot{\nabla}$, $\dot{\Delta}$, $\dot{\bot}$, and $\dot{\top}$ are defined as previously.

Abstract transfer functions. The abstract transfer function $\mathcal{F}$ is slightly modified, since in the initial memory state, the length of every string is equal to 0 (*i.e.* following the convention that the size of non-allocated arrays is 0):

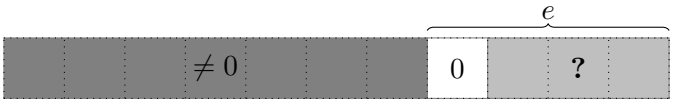
$$\mathcal{F}(\mathcal{X})(\ell') \stackrel{def}{=} \begin{cases} (\!| sz_t \leftarrow 0 \text{ and } len_t \leftarrow 0 \text{ for all } t \in \mathsf{ArrayVars} |\!)(\top) & \text{if } \ell' = entry(P), \\ \displaystyle\bigsqcup_{\langle \ell, step, \ell' \rangle} [\![step]\!](\mathcal{X}(\ell)) & \text{otherwise.} \end{cases}$$

When *step* does not manipulate the content of the heap, the definition of $[\![step]\!]$ is not modified. Otherwise, $[\![step]\!]$ is built thanks to four abstract primitives, $\mathsf{guard\_null}$, $\mathsf{guard\_non\_null}$, $\mathsf{assign\_null}$, and $\mathsf{assign\_non\_null}$. These primitives are defined as follows:

- $\mathsf{guard\_null}(\mathcal{M}, t, e)$ filters the memory states abstracted by $\mathcal{M}$, such that the array element $t[e]$ contains the value 0. This happens only if e is greater than or equal to the length len_t. Indeed, the elements whose index is strictly less than the length are necessarily non-null. Thus:

  $$\mathsf{guard\_null}(\mathcal{M}, t, e) \stackrel{def}{=} (\!| e \geq len_t \wedge 0 \leq e \leq sz_t - 1 |\!)(\mathcal{M}),$$

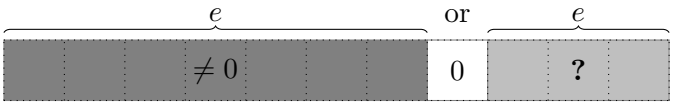
  which can be represented by the following figure:



- on the contrary, $\mathsf{guard\_non\_null}(\mathcal{M}, t, e)$ yields an approximation of the memory states abstracted by $\mathcal{M}$, such that the array element $t[e]$ does not store the null character. This naturally happens when $e < len_t$. It may also happen when $e > len_t$, since in that case, our abstraction does not have any precise information on the value of the array element:

$$\begin{aligned}\mathsf{guard\_non\_null}(\mathcal{M}, t, e) \stackrel{def}{=} &(\!|e \leq len_t - 1 \wedge 0 \leq e \leq sz_t - 1|\!)(\mathcal{M}) \\ &\sqcup (\!|e \geq len_t + 1 \wedge 0 \leq e \leq sz_t - 1|\!)(\mathcal{M}),\end{aligned}$$

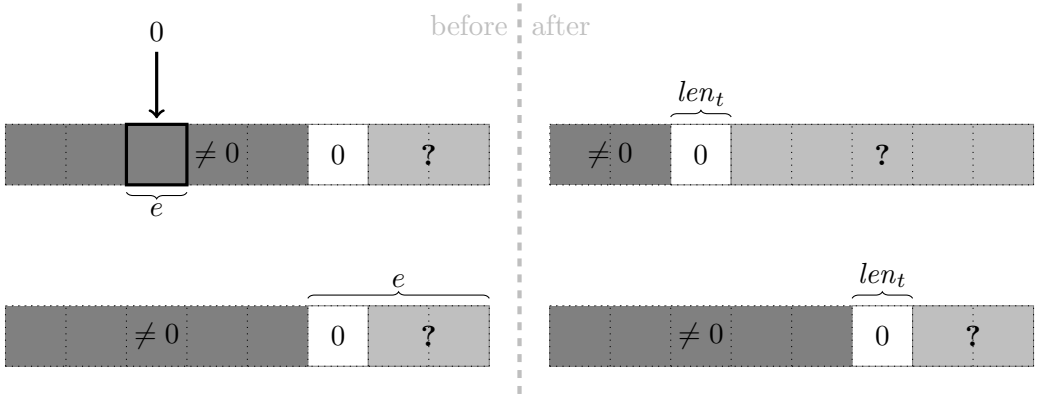
  which can be illustrated by:



- $\mathsf{assign\_null}(\mathcal{M}, t, e)$ is an abstraction of the assignment $t[e] := 0$ on the abstract state $\mathcal{M}$. Two cases can be distinguished: (i) either e is strictly less than len_t, so that the length has to be updated to the value of e, (ii) or $e \geq len_t$, in which case the length of the string does not change. Therefore:

$$\begin{aligned}\mathsf{assign\_null}(\mathcal{M}, t, e) \stackrel{def}{=} &(\!|len_t \leftarrow e|\!) \circ (\!|e \leq len_t - 1 \wedge 0 \leq e \leq sz_t - 1|\!)(\mathcal{M}) \\ &\sqcup (\!|e \geq len_t \wedge 0 \leq e \leq sz_t - 1|\!)(\mathcal{M}).\end{aligned}$$

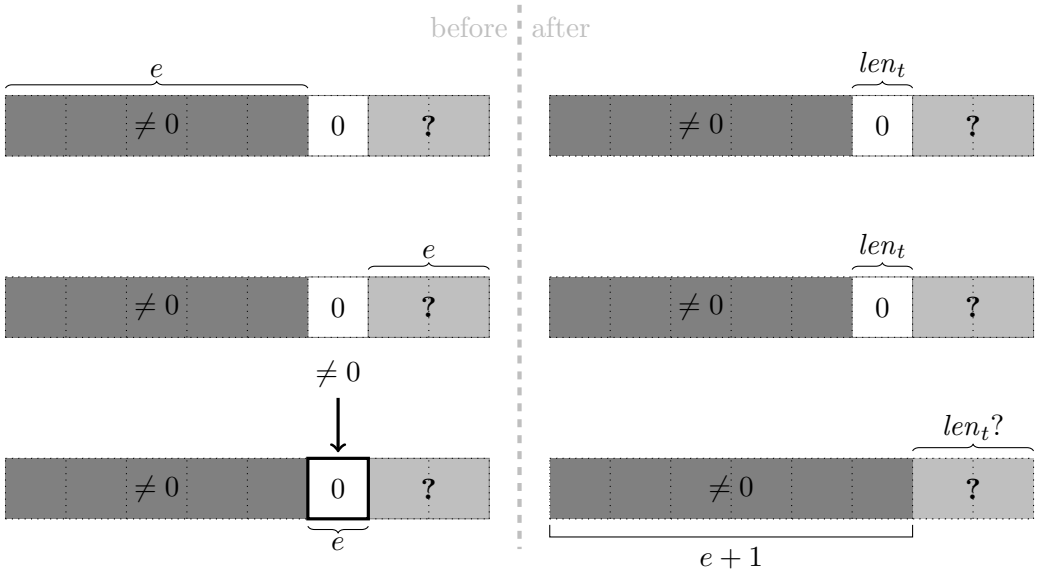
  The two cases can be represented as follows:



- $\mathsf{assign\_non\_null}(\mathcal{M}, t, e)$ is an abstraction of the assignment $t[e] := c$ where c is a non-null character. Three cases can be distinguished: $e \leq len_t - 1$, $e \geq len_t + 1$, or $e = len_t$. In the two first ones, the length of the string is not altered. In the last one, the null terminal character is overwritten by a non-null character, so that the length is updated to an arbitrary value between $e + 1$ and sz_t:

$$\begin{aligned}&\mathsf{assign\_non\_null}(\mathcal{M}, t, e) \\ &\stackrel{def}{=} (\!|e \leq len_t - 1 \wedge 0 \leq e \leq sz_t - 1|\!)(\mathcal{M}) \sqcup (\!|e \geq len_t + 1 \wedge 0 \leq e \leq sz_t - 1|\!)(\mathcal{M}) \\ &\quad \sqcup (\!|e + 1 \leq len_t \leq sz_t|\!) \circ (\!|len_t \leftarrow ?|\!) \circ (\!|e = len_t \wedge 0 \leq e \leq sz_t - 1|\!)(\mathcal{M}).\end{aligned}$$

  The three cases can be illustrated as follows:



The following lemma states that these primitives are sound:

Lemma 6.5. *Let* $\mathcal{M} \in \mathsf{Mem}$. *Then the following relations holds:*

$$\left\{ s \shortmid h \in \gamma_{\mathsf{Mem}}(\mathcal{M}) \;\middle|\; \begin{array}{l} s \vdash e \Rightarrow i,\ (t,i) \in \mathsf{dom}(h), \\ \textit{and } h(t,i) = 0 \end{array} \right\} \subset \gamma_{\mathsf{Mem}}(\mathsf{guard\_null}(\mathcal{M}, t, e)),$$

$$\left\{ s \shortmid h \in \gamma_{\mathsf{Mem}}(\mathcal{M}) \;\middle|\; \begin{array}{l} s \vdash e \Rightarrow i,\ (t,i) \in \mathsf{dom}(h), \\ \textit{and } h(t,i) \neq 0 \end{array} \right\} \subset \gamma_{\mathsf{Mem}}(\mathsf{guard\_non\_null}(\mathcal{M}, t, e)),$$

$$\left\{ s' \shortmid h' \mid s \shortmid h \in \gamma_{\mathsf{Mem}}(\mathcal{M}),\ s \shortmid h \vdash t[e] \coloneqq 0 : s' \shortmid h' \right\} \subset \gamma_{\mathsf{Mem}}(\mathsf{assign\_null}(\mathcal{M}, t, e)),$$

$$\left\{ s' \shortmid h' \;\middle|\; \begin{array}{l} c \in \mathbb{Z} \setminus \{0\},\ s \shortmid h \in \gamma_{\mathsf{Mem}}(\mathcal{M}), \\ \textit{and } s \shortmid h \vdash t[e] \coloneqq c : s' \shortmid h' \end{array} \right\} \subset \gamma_{\mathsf{Mem}}(\mathsf{assign\_non\_null}(\mathcal{M}, t, e)).$$

We can now modify the definition of $[\![step]\!]$ when *step* performs operations on the heap:

- for the stack assignment $x \coloneqq t[e]$, the cases $t[e] \neq 0$ and $t[e] = 0$ are distinguished thanks to the functions $\mathsf{guard\_non\_null}$ and $\mathsf{guard\_null}$. In the former case, x is updated to an arbitrary value which is then filtered to non-zero values, while in the latter case, x is directly updated to the value 0:

$$\begin{aligned} [\![x \coloneqq t[e]]\!](\mathcal{M}) \overset{def}{=} &\big((\!|x \leq -1 \vee x \geq 1|\!) \circ (\!|x \leftarrow ?|\!)\big)(\mathsf{guard\_non\_null}(\mathcal{M}, t, e)) \\ &\sqcup (\!|x \leftarrow 0|\!)(\mathsf{guard\_null}(\mathcal{M}, t, e)). \end{aligned}$$

- for the heap assignment $t[e_1] \coloneqq e_2$, the primitives $\mathsf{assign\_non\_null}$ and $\mathsf{assign\_null}$ are used according to the value of e_2:

$$\begin{aligned} [\![t[e_1] \coloneqq e_2]\!](\mathcal{M}) \overset{def}{=} &\ \mathsf{assign\_non\_null}((\!|e_2 \leq -1 \vee e_2 \geq 1|\!)(\mathcal{M}), t, e_1) \\ &\sqcup \mathsf{assign\_null}((\!|e_2 = 0|\!)(\mathcal{M}), t, e_1). \end{aligned}$$

- the non-deterministic heap assignment $t[e] \coloneqq \mathsf{read}()$ is obtained by joining the cases where the assigned character is null or not:

$$[\![t[e] \coloneqq \mathsf{read}()]\!](\mathcal{M}) \overset{def}{=} \mathsf{assign\_non\_null}(\mathcal{M}, t, e) \sqcup \mathsf{assign\_null}(\mathcal{M}, t, e).$$

- similarly, for the array copy $t_1[e_1] \coloneqq t_2[e_2]$, two cases can be distinguished: (i) a non-null value is assigned to $t_1[e_1]$ when $e_2 \leq len_{t_2} - 1$, or may be assigned when $e_2 \geq len_{t_2} + 1$, (ii) or the null character may be assigned as soon as $e_2 \leq len_{t_2}$. The primitives $\mathsf{assign\_non\_null}(\cdot, t_1, e_1)$ and $\mathsf{assign\_null}(\cdot, t_1, e_1)$ are used accordingly:

$$\begin{aligned}[\![t_1[e_1] \coloneqq t_2[e_2]]\!](\mathcal{M}) \overset{def}{=}\ & \mathsf{assign\_non\_null}(\!(e_2 \leq len_{t_2} - 1 \wedge 0 \leq e_2 \leq sz_{t_2} - 1)\!)(\mathcal{M}, t_1, e_1) \\ & \sqcup \mathsf{assign\_non\_null}(\!(e_2 \geq len_{t_2} + 1 \wedge 0 \leq e_2 \leq sz_{t_2} - 1)\!)(\mathcal{M}, t_1, e_1) \\ & \sqcup \mathsf{assign\_null}(\!(e_2 \geq len_{t_2} \wedge 0 \leq e_2 \leq sz_{t_2} - 1)\!)(\mathcal{M}, t_1, e_1).\end{aligned}$$

- the condition $t[e_1] = e_2$ on $t[e_1]$ is implemented using $\mathsf{guard\_non\_null}$ (resp. $\mathsf{guard\_null}$) when e_2 is not null (resp. null):

$$\begin{aligned}[\![t[e_1] = e_2]\!](\mathcal{M}) \overset{def}{=}\ & (\mathsf{guard\_null}(\mathcal{M}, t, e_1) \sqcap (\!(e_2 = 0)\!)(\mathcal{M})) \\ & \sqcup (\mathsf{guard\_non\_null}(\mathcal{M}, t, e_1) \sqcap (\!(e_2 \leq -1 \vee e_2 \geq 1)\!)(\mathcal{M})).\end{aligned}$$

- for the condition $t[e_1] \neq e_2$, $\mathsf{guard\_non\_null}$ can be applied when e_2 is null, while when it is not null, no further information can be obtained:

$$\begin{aligned}[\![t[e_1] \neq e_2]\!](\mathcal{M}) \overset{def}{=}\ & \big(\mathsf{guard\_non\_null}(\mathcal{M}, t, e_1) \sqcap (\!(e_2 = 0)\!)(\mathcal{M})\big) \\ & \sqcup (\!(e_2 \leq -1 \vee e_2 \geq 1)\!)(\mathcal{M})\end{aligned}$$

- the allocation $t \coloneqq \mathsf{malloc}(e)$ not only updates the value of sz_t to e, but also sets the length len_t to an arbitrary value between 0 and e:

$$[\![t \coloneqq \mathsf{malloc}(e)]\!](\mathcal{M}) \overset{def}{=} (\!(0 \leq len_t \leq e)\!) \circ (\!(len_t \leftarrow ?\,, sz_t \leftarrow e)\!) \circ (\!(e \geq 1)\!)(\mathcal{M}).$$

Then the following statement holds:

Lemma 6.6. *Let $\mathcal{M} \in \mathsf{AMem}$. Then:*

$$\left\{s' \shortmid h' \mid s \shortmid h \in \gamma_{\mathsf{Mem}}(\mathcal{M}) \text{ and } s \shortmid h \vdash step : s' \shortmid h'\right\} \subset \gamma_{\mathsf{Mem}}\left([\![step]\!](\mathcal{M})\right). \tag{6.18}$$

As a consequence, $\mathcal{F}$ can be shown to be a sound abstraction of the concrete transfer function F:

Proposition 6.7. *Let $\mathcal{X} \in \mathsf{AStates}$. Then we have:*

$$F(\gamma(\mathcal{X})) \subset \gamma(\mathcal{F}(\mathcal{X})).$$

It can be also verified that $\mathcal{F}$ is still monotone. Thus the abstract semantics $\mathcal{C}(P)$ can be then computed using Propositions 6.1 and 6.2.

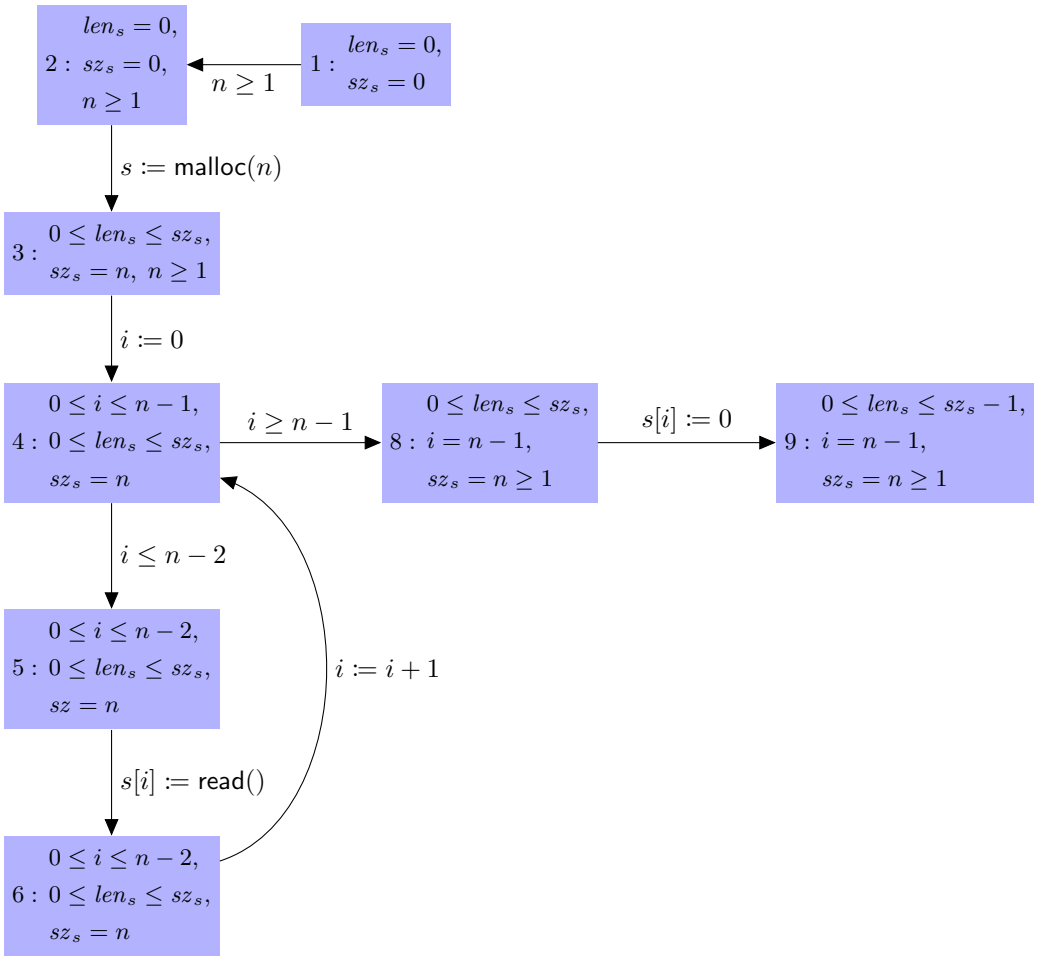


Figure 6.11: The abstract collecting semantics of our running example, using the string abstraction

Example 6.16. Using the string abstraction, the abstract collecting semantics $\mathcal{C}(P)$ provides more precise invariants on the array s of the program of Example 6.1.

Indeed, at control point 8, the length of s is still unknown, being between 0 and $n = sz_s$. After the assignment $s[i] := 0$, the abstract primitive assign_null is applied on $\mathcal{M} = \mathcal{C}(P)(8)$. It yields the abstract memory states:

$$\begin{aligned}\mathsf{assign\_null}(\mathcal{M}, s, i) = (\!|len_s \leftarrow i|\!) \circ (\!|i \leq len_s - 1 \wedge 0 \leq i \leq sz_s - 1|\!)(\mathcal{M}) \\ \sqcup\, (\!|i \geq len_s \wedge 0 \leq i \leq sz_s - 1|\!)(\mathcal{M})\end{aligned}$$

In the first argument of the union, the invariant $len_s = i = n - 1$ holds because of the assignment $len_s \leftarrow i$. The second argument over-approximates the states in which $i = n-1 \geq len_s$. In both cases, and therefore in the result of the abstract join operator, the invariant $len_s \leq n-1$ holds. Therefore, at control point 9, it is ensured that the length of s is bounded by 0 and $n - 1$.

6.6 Conclusion of the chapter

In this chapter, we have introduced the basics of the theory of abstract interpretation, and applied it to the construction of an analysis able to infer invariants over the length of the strings in a kernel language equipped with dynamic allocation. In particular, this analysis has been able to show the absence of heap overflows in an small example. Thus, for the moment, the precision of the analysis is satisfactory.

Now, let us complete the program of Example 6.1 in the following way:

```
 1 : assume (n ≥ 1);                12 :     upp[i] := s[i];
 2 : s := malloc(n);                13 :     i := i + 1;
 3 : i := 0;                        14 : done;
 4 : while i ≤ n − 2 do             15 : i := 0;
 5 :     s[i] := read();            16 : while upp[i] ≠ 0 do
 6 :     i := i + 1;                17 :     c := upp[i];
 7 : done;                          18 :     if (c ≥ 97) ∧ (c ≤ 122) then
 8 : s[i] := 0;                     19 :         upp[i] := c − 32;
 9 : upp := malloc(n);              20 :     end;
10 : i := 0;                        22 :     i := i + 1;
11 : while i ≤ n − 1 do             23 : done;
```

This version of the program performs the following additionnal manipulations:

- from Lines 9 to 14, it copies the whole content of the array s into a newly allocated array upp. This is equivalent to a call to the function `memcpy`(upp, s, n) in C.
- from Lines 15 to 20, it transforms any character c of upp between 97 and 122, *i.e.* ranging over the letters 'a',...,'z' in ASCII code, into the character $c - 32$, which is the corresponding upper case letter. This transformation is iterated until the null terminal character of upp is encountered.

In other words, the program first reads a string from an external source, and then creates the corresponding string in capital letters. This is a typical representative of the string manipulations which can be found in most software.

This complete version is still correct. Indeed, after the "`memcpy` part", the length of the string stored in upp is precisely equal to len_s, and in particular less than or equal to $n - 1$. Thus, the loop from Lines 16 to 20 can not cause a heap overflow since it stops at the index $i = len_{upp}$, which is strictly less than the size of the array upp (equal to n).

However, as discussed in Chapter 1, convex numerical abstract domains are not able to precisely analyze the "`memcpy` part". For instance, we have implemented the abstraction on strings with convex polyhedra. The computed invariant on len_{upp} at control point 15 is then $0 \leq len_{upp} \leq n$ and $len_{upp} + len_s \leq 2n - 2$. In particular, it is not able to infer the relation $len_{upp} \leq n - 1$. This means that in the abstract memory state, we may have $len_{upp} = n = sz_{upp}$, which corresponds to the fact that the array upp *does not* contain any null terminal character. Then, the loop from Lines 16 to 20 can iterate until i reaches the value of n, which causes a heap overflow at Line 16. This is naturally a false alarm, which is entirely due to the lack of precision of the numerical domain.

As a consequence, using such a non-disjunctive numerical abstract domain, our string analysis is not able to show the absence of heap overflows in the second loop.

CHAPTER 7

Numerical abstract domains based on tropical polyhedra

In this chapter, we introduce three new numerical abstract domains based on tropical polyhedra. They respectively allow to infer max-invariants, min-invariants, and min- and max-invariants, over a given set of variables. The first class of invariants is formed by systems of tropically affine inequalities over these variables, *i.e.* with usual notations:

$$\max(\alpha_0, \max(\alpha_1 + \boldsymbol{v}_1, \ldots, \alpha_d + \boldsymbol{v}_d)) \leq \max(\beta_0, \max(\beta_1 + \boldsymbol{v}_1, \ldots, \beta_d + \boldsymbol{v}_d)),$$

where $\alpha_0, \ldots, \alpha_d$ and $\beta_0, \ldots, \beta_d$ range over the set $\mathbb{R}_{\max}$. The second class of invariants is obtained by replacing the operator max by min in the previous inequality. The third class is able to express, in particular, both min- and max-invariants. The first two classes of invariants contain zone invariants, of the form $\boldsymbol{v}_i - \boldsymbol{v}_j \geq \alpha$, and express in fact some disjunctions of such invariants. Similarly, the third class is able to express some disjunctions of octagonal invariants $\pm\boldsymbol{v}_i \pm \boldsymbol{v}_j \geq \alpha$.

As far as we know, the only other existing domain inferring similar invariants is due by Gulavani and Gulwani in [GG08].[1] Their domain is paramaterized by two fixed disjoint sets U and V of variables, and an integer $K > 0$. It is able to express invariants of the form $e \leq \max(f_1, \ldots, f_p)$, where e is a (classical) linear expression over the variables of U,

[1] Purely coincidentally, it was introduced as the very same time as a first version of our approach was published in [AGG08], while min-/max-invariants had been never discussed before in the literature.

$f_1, \ldots, f_p$ are linear expressions over the variables of V, and $p \leq K$. Their approach is based on a kind of disjunctive completion, and is absolutely not related to tropical polyhedra nor tropical convex sets. In particular, albeit it is sound, it may be very imprecise because of the use of heuristics.[2] This abstract domain is applied to timing analysis, which allows to statically determine timing bounds on programs. Also note that it is cannot express the invariants on memory manipulations which have discussed in Chapter 1.

The chapter is organized as follows. In Sections 7.1, 7.2, and 7.3, we define the three numerical abstract domains. They all rely on tropical polyhedra to over-approximate sets of environments. Each polyhedron is represented by a double description formed by a generating set and a system of inequalities, following the equivalence of the two representations stated in Theorem 2.5. We also define abstract primitives which soundly over-approximate their concrete analogues. Most of the abstract primitives involve only one of the two components of the double descriptions. The algorithms that we have defined in Chapter 5 will be therefore of critical importance to obtain the full double descriptions when needed. Finally, in Section 7.4, we combine the abstract semantics defined in Chapter 6 with the three domains, and evaluates the resulting analysis on memory manipulating programs such as algorithms on arrays or strings.

7.1 Inferring max-invariants: the abstract domain MaxPoly

We introduce the numerical abstract domain MaxPoly, which will be used to infer max-invariants. We first discuss in Section 7.1.1 the choice of the abstract representation and the corresponding concretization operator. We also express the max-invariants in terms of disjunctions of zone invariants.

Section 7.1.2 to 7.1.5 define the usual primitives over-approximating the order, the union, the intersection, and the assignments. Section 7.1.6 is devoted to the definition of two widening operators. Section 7.1.7 introduces a primitive which, given an abstract element of MaxPoly, allows to extract the smallest zone containing it. For each abstract primitive, the soundness, the level of precision, and the complexity is discussed.

7.1.1 Definition of the abstract domain

The abstract domain MaxPoly over-approximates sets of numerical environments. The latter are functions from Vars to $\mathbb{R}$, where Vars is a set of variables d pairwise distinct variables $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d$. The set Vars acts as a parameter of the abstract domain, and can be naturally replaced by another set of variables. That is why the abstract domain may also be denoted by MaxPoly(Vars), to highlight the fact that the computed invariants range over the variables of Vars.

The elements of MaxPoly are tropical polyhedra. Obviously, a tropical polyhedron is not represented by the set of its points, which may be not finite. Instead, every non-empty tropical polyhedron is given under the form of a *double representation*, consisting in a minimal generating representation (*i.e.* its extreme points and the extreme rays of its recession cone), and a system of constraints which precisely defines it. Formally, the abstract domain MaxPoly is defined as follows:

[2] The main heuristics ensures that the expressions $\max(f_1, \ldots, f_p)$ do not contain more than K arguments.

Definition 7.1. We define MaxPoly as the set formed by the element $\bot$ and by the *double representations* $((P,R),(A,\boldsymbol{c},B,\boldsymbol{d}))$, which have to satisfy the following conditions:

(i) the following identity holds

$$\mathsf{co}(P) \oplus \mathsf{cone}(R) = \{\, \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d} \,\}, \tag{7.1}$$

(ii) (P,R) is precisely formed by the extreme points and the scaled extreme rays of the tropical polyhedron given in (7.1),

(iii) P is not empty, and there is no $i \in [d]$ satisfying $\boldsymbol{p}_i = \mathbb{0}$ and $\boldsymbol{r}_i = \mathbb{0}$ for all $\boldsymbol{p} = (\boldsymbol{p}_i) \in P$ and $\boldsymbol{r} = (\boldsymbol{r}_i) \in R$,

(iv) $A, B \in \mathbb{R}^{p\times d}_{\max}$, $\boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}^d_{\max}$, and the inequalities of the system $A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d}$ are linearly independent.

Let us give some details about Definition 7.1. The special element $\bot$ is meant to represent the empty tropical polyhedron. Any other abstract element is a double representation: its first component is said to be *by generators*, while the second one is referred to as *by constraints*. Condition (i) ensures the equivalence of the two components. Each abstract element $\mathcal{X} \in$ MaxPoly is therefore associated to the tropical polyhedron $\overline{\mathcal{X}}$ which it represents:

$$\begin{aligned} \overline{\bot} &\stackrel{def}{=} \emptyset \\ \overline{((P,R),(A,\boldsymbol{c},B,\boldsymbol{d}))} &\stackrel{def}{=} \mathsf{co}(P) \oplus \mathsf{cone}(R) \\ \text{or, equivalently,} \qquad &\stackrel{def}{=} \{\, \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d} \,\} \end{aligned}$$

Condition (ii) on the extremality of the elements of P and R ensures that (P,R) forms a minimal generating representation of the tropical polyhedron $\mathcal{P} = \overline{\mathcal{X}}$ (Theorem 2.4). Observe that it amounts to the fact that $(P \times \{\, \mathbb{1} \,\}) \cup (R \times \{\, \mathbb{0} \,\})$ contains exactly one representative of each extreme ray of the homogenized cone $\widehat{\mathcal{P}}$ (Corollary 2.3). Equivalently, no element of $(P \times \{\, \mathbb{1} \,\}) \cup (R \times \{\, \mathbb{0} \,\})$ can be expressed as a tropical linear combination of the others (Proposition 5.6). These equivalent formulations of Condition (ii) will be often used in the sequel.

Similarly, Condition (iv) expresses that the constraints component is under minimal form. Indeed, the inequalities of the system $A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d}$ must be linearly independent, which means that no inequality can be expressed as a tropical linear combination of the others (Condition (iv)). Consider $H \subset (\mathbb{R}^{(d+1)}_{\max})^2$ the set formed by the elements

$$\left(\begin{pmatrix} {}^tA_k & \boldsymbol{c}_k \end{pmatrix}, \begin{pmatrix} {}^tB_k & \boldsymbol{d}_k \end{pmatrix}\right),$$

where A_k and B_k are the k-th row of the matrices A and B respectively, and $\boldsymbol{c} = (\boldsymbol{c}_i)$, $\boldsymbol{d} = (\boldsymbol{d}_i)$). Then Condition (iv) ensures that H is indeed a minimal generating set of the tropical cone $\mathsf{cone}(H)$ of $(\mathbb{R}^{(d+1)}_{\max})^2$. However, for the same reason than those discussed in Remark 5.5, the system $A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d}$ may contain some redundant inequalities.

Finally, since we are interested in over-approximating subsets of $\mathbb{R}^{\mathsf{Vars}}$, which is isomorphic to $\mathbb{R}^d$, the intersection of $\overline{\mathcal{X}}$ with $\mathbb{R}^d$ should not be empty, unless $\mathcal{X} = \bot$. This is equivalent to Condition (iii). Indeed, $\overline{\mathcal{X}} \cap \mathbb{R}^d$ is empty if and only if there exists $i \in [d]$ such that $\boldsymbol{x}_i = \mathbb{0}$ for all $\boldsymbol{x} = (\boldsymbol{x}_i) \in \mathbb{R}^d_{\max}$. This precisely happens in one of the two following cases:

- $\overline{\mathcal{X}}$ is empty, which is equivalent to $P = \emptyset$,
- or for a given $i \in [d]$, the i-th coordinate of all elements of P and R is equal to $\mathbb{0}$. Intuitively, this corresponds to the fact that $\boldsymbol{x}_i = \mathbb{0}$ for all $\boldsymbol{x} \in \overline{\mathcal{X}}$.

7.1.1.a Concretization operator. We can now define the concretization operator of the abstract domain MaxPoly. It maps any abstract element to the set of environments $\nu : \mathsf{Vars} \to \mathbb{R}$ such that the point $(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d))$ belongs to the associated polyhedron: given $\mathcal{X} \in \mathsf{MaxPoly}$,

$$\gamma_{\mathsf{MaxPoly}}(\mathcal{X}) \stackrel{def}{=} \{\, \nu \in \mathbb{R}^{\mathsf{Vars}} \mid (\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) \in \overline{\mathcal{X}} \,\}.$$

Observe that according to Definition 7.1, $\bot$ is the unique abstract element whose concretization is empty. The operator $\gamma_{\mathsf{MaxPoly}}$ will be simply denoted by γ when it is clear from context.

7.1.1.b Full representations. In the following sections, we are going to define the abstract primitives which manipulate the elements of the abstract domain MaxPoly, and are sound approximations of their concrete counterparts.

Most of these primitives return only one of the components of the double representation. For this reason, we introduce two functions allowing to obtain a full representation from one of the components. They are naturally based on the algorithms which have been defined in Chapter 5, and on the properties on homogenization (Corollary 2.10).

Recall that ι refers to the one-to-one correspondence allowing to pass from the scaled minimal generating representation of a tropical polyhedron to the scaled set of extreme elements of its homogenized cone (defined in Corollary 2.10).

The function ofCons builds a full representation from a constraint component. It is defined on quadruples $(A, \boldsymbol{c}, B, \boldsymbol{d})$ satisfying the requirement (iv):

$$\mathsf{ofCons}(A, \boldsymbol{c}, B, \boldsymbol{d}) \stackrel{def}{=} \begin{cases} ((P,R),(A,\boldsymbol{c},B,\boldsymbol{d})) & \text{if } (P,R) \text{ satisfies Condition (iii) of Def. 7.1,} \\ \bot & \text{otherwise,} \end{cases}$$

where $(P,R) = \iota^{-1}\left(\textsc{ComputeExtRays}\left(\begin{pmatrix} A & \boldsymbol{c} \end{pmatrix}, \begin{pmatrix} B & \boldsymbol{d} \end{pmatrix}, p\right)\right)$, and p is the number of constraints in the constraint component.[3] Observe that $\bot$ is returned if (P,R) does not satisfy Condition (iii) of Definition 7.1, which characterizes the emptyness of the intersection of the represented tropical polyhedron with $\mathbb{R}^d$. Corollary 2.10 and Theorem 5.2 ensure that the returned element indeed belongs to MaxPoly. Using the complexity bound of Corollary 5.17, we know that the worst-case complexity of ofCons is bounded by:

$$\begin{cases} O(p^2 d\alpha(d+1)(p+d+1)^{d-1}) & \text{if } d \text{ is odd,} \\ O(p^2 d\alpha(d+1)(p+d+1)^{d}) & \text{if } d \text{ is even.} \end{cases}$$

Inversely, starting from a generating component (P,R) verifying (ii), the function ofGen is defined by:

$$\mathsf{ofGen}(P,R) \stackrel{def}{=} \begin{cases} ((P,R),(A,\boldsymbol{c},B,\boldsymbol{d})) & \text{if } (P,R) \text{ satisfies Condition (iii) of Definition 7.1,} \\ \bot & \text{otherwise,} \end{cases}$$

[3] *i.e.* $A, B \in \mathbb{R}_{\max}^{p \times d}$, $\boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}_{\max}^{p}$.

where ${}^t\begin{pmatrix} A & \boldsymbol{c} & B & \boldsymbol{d}\end{pmatrix} = \textsc{ComputeExtRaysPolar}(\iota(P,R),q)$, and $q = |P| + |R|$.[4] Using Corollary 5.17, the worst-case complexity of ofGen is bounded by:

$$\begin{cases} O(pd^2(p+d+1)^d) & \text{if } d \text{ is odd,} \\ O(pd^2(p+d+1)^{d+1}) & \text{if } d \text{ is even.} \end{cases}$$

In practice, we will see in Section 7.4 that full representations are computed lazily so as to optimize the performance of the whole static analysis. Therefore, we have chosen to not include the cost of ofCons or ofGen in the presentation of the complexity results on the abstract primitives defined in the following sections.

7.1.1.c Disjunctions of zones. Each max-invariant expresses a disjunction of zone invariants over the variables $\boldsymbol{v}_i$. Indeed,

$$\begin{aligned} & \max(\alpha_0, \max(\alpha_1 + \boldsymbol{v}_1, \ldots, \alpha_d + \boldsymbol{v}_d)) \leq \max(\beta_0, \max(\beta_1 + \boldsymbol{v}_1, \ldots, \beta_d + \boldsymbol{v}_d)) \\ \iff & \bigvee_{\substack{1 \leq i \leq d \\ \beta_i \neq -\infty}} \left[\Big(\bigwedge_{1 \leq j \leq d} \alpha_j - \beta_i \leq \boldsymbol{v}_i - \boldsymbol{v}_j \Big) \wedge (\alpha_0 - \beta_i \leq \boldsymbol{v}_i) \right] \\ & \vee \left[\bigwedge_{\substack{1 \leq i \leq d \\ \alpha_i \neq -\infty}} \boldsymbol{v}_i \leq \beta_0 - \alpha_i \right] \quad \text{(this term appears only when } \alpha_0 \leq \beta_0) \end{aligned}$$

As conjunctions of max-invariants, the invariants represented by the abstract elements of MaxPoly can also been expressed as disjunctions of zone invariants.

Note that this disjunctive point of view is reminiscent of the approach of Develin and Sturmfels in [DS04, Theorem 15], in which tropical polytopes are expressed as unions of a finite number of bounded *cells*. The latter are precisely defined by zone invariants. They can be enumerated thanks to combinatorial considerations on the generating sets of the polytopes.

7.1.1.d No best possible abstraction. In general, there is no best possible abstraction of a given $X \subset \mathbb{R}^{\mathsf{Vars}}$ in the abstract domain MaxPoly. Consider for instance the set

$$X = \left\{ \nu \in \mathbb{R}^{\mathsf{Vars}} \;\middle|\; \begin{array}{l} \nu(\boldsymbol{v}_i) \geq 0 \text{ for all } i, \\ (\nu(\boldsymbol{v}_1))^2 + \ldots + (\nu(\boldsymbol{v}_d))^2 \geq 1 \end{array} \right\}.$$

It contains the environments ν such that $\nu(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d)$ is in the complement of the open d-sphere in $(\mathbb{R}_+)^d$. The case $d = 2$ is illustrated in Figure 7.1 (left side, with a radius equal to 4 instead of 1), with two incomparable abstract elements of MaxPoly (middle and right side).

7.1.2 Abstract preorder

In this section, we define a preorder $\sqsubseteq$ on the domain MaxPoly, which over-approximates the partial order $\subset$ over the powerset $\wp(\mathbb{R}^{\mathsf{Vars}}_{\max})$.

First of all, we define

$$\begin{cases} \bot \sqsubseteq \mathcal{X} & \text{for all } \mathcal{X} \in \mathsf{MaxPoly}, \\ \mathcal{X} \sqsubseteq \bot & \text{if and only if } \mathcal{X} = \bot. \end{cases}$$

[4] The integer q is also equal to the cardinality of $\iota(P,R)$.

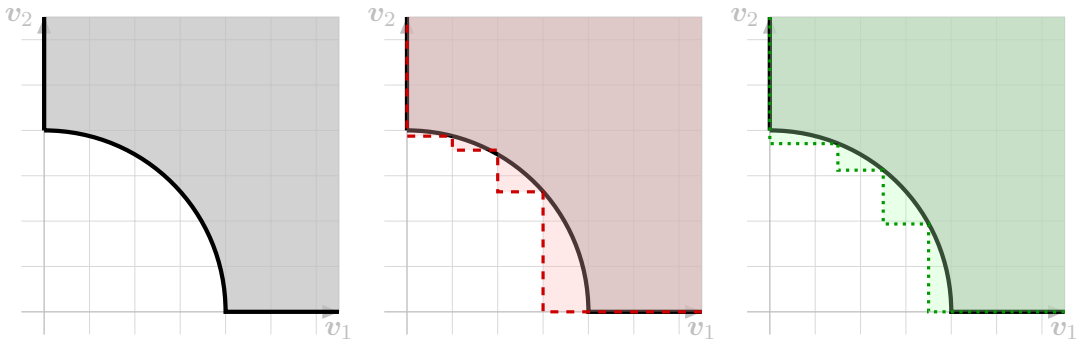


Figure 7.1: A set which has no best possible abstraction in MaxPoly

Now, if the two operands of $\sqsubseteq$ are provided by double representations, the abstract order is defined using the generator component of the first operand, and equivalently one of the components of the second operand: given $\mathcal{X} = ((P,R),\cdot)$ and $\mathcal{X}' = ((P',R'),(A',\boldsymbol{c}',B',\boldsymbol{d}'))$ in MaxPoly,

$$\mathcal{P} \sqsubseteq \mathcal{P}' \overset{def}{\iff} \boldsymbol{g} = \bigoplus_{\boldsymbol{g}' \in \iota(P',R')} (\boldsymbol{g}' \backslash \boldsymbol{g}) \boldsymbol{g}' \text{ for all } \boldsymbol{g} \in \iota(P,R), \tag{7.2}$$

or, equivalently,

$$\overset{def}{\iff} \begin{cases} A'\boldsymbol{p} \oplus \boldsymbol{c}' \leq B'\boldsymbol{p} \oplus \boldsymbol{d}' & \text{for all } \boldsymbol{p} \in P, \\ A'\boldsymbol{r} \leq B'\boldsymbol{r} & \text{for all } \boldsymbol{r} \in R. \end{cases} \tag{7.3}$$

These definitions can be shown to be equivalent because of the following lemma:

Lemma 7.1. *The two definitions* (7.2) *and* (7.3) *are both equivalent to the inclusion of* $\overline{\mathcal{X}}$ *into* $\overline{\mathcal{X}'}$.

Proof. Let $\mathcal{P} = \overline{\mathcal{X}}$ and $\mathcal{P}' = \overline{\mathcal{X}'}$. We first claim that $\mathcal{P} \subset \mathcal{P}'$ if and only if $\widehat{\mathcal{P}} \subset \widehat{\mathcal{P}'}$. Indeed, if $\mathcal{P} \subset \mathcal{P}'$, then

$$\{\, (\alpha \boldsymbol{x}, \alpha) \mid \boldsymbol{x} \in \mathcal{P}, \alpha \in \mathbb{R}_{\max} \,\} \subset \{\, (\alpha \boldsymbol{x}, \alpha) \mid \boldsymbol{x} \in \mathcal{P}', \alpha \in \mathbb{R}_{\max} \,\}$$

so by applying cl on both sides,

$$\widehat{\mathcal{P}} \subset \widehat{\mathcal{P}'}$$

Conversely, if $\widehat{\mathcal{P}} \subset \widehat{\mathcal{P}'}$, then using Proposition 2.1, we have $\mathcal{P} = \{\, \boldsymbol{x} \mid (\boldsymbol{x}, \mathbb{1}) \in \widehat{\mathcal{P}} \,\} \subset \{\, \boldsymbol{x} \mid (\boldsymbol{x}, \mathbb{1}) \in \widehat{\mathcal{P}'} \,\} = \mathcal{P}'$.

On top of that, $G = \iota(P,R)$ and $G' = \iota(P',R')$ form a generating set of the cones $\widehat{\mathcal{P}}$ and $\widehat{\mathcal{P}'}$ respectively, so that $\widehat{\mathcal{P}} \subset \widehat{\mathcal{P}'}$ is equivalent to the fact that $\boldsymbol{g} \in \mathsf{cone}(G')$ for all $\boldsymbol{g} \in G$. Using Lemma 5.7, this happens if and only if $\boldsymbol{g} = \bigoplus_{\boldsymbol{g}' \in \iota(P',R')} (\boldsymbol{g}' \backslash \boldsymbol{g}) \boldsymbol{g}'$. This shows that the first definition is equivalent to the fact that $\mathcal{P} \subset \mathcal{P}'$.

Now, $\mathcal{P} \subset \mathcal{P}'$ if and only if $\boldsymbol{p} \in \mathcal{P}'$ for all $\boldsymbol{p} \in P$ and $\boldsymbol{r} \in \mathsf{rec}(\mathcal{P}')$ for all $\boldsymbol{r} \in R$. Using Proposition 2.4, this happens if and only if the inequalities of the second definition holds. □

As a consequence, the relation $\sqsubseteq$ can be shown to be sound and exact:

Proposition 7.2. *Let* $\mathcal{X}, \mathcal{X}' \in$ MaxPoly. *Then* $\gamma(\mathcal{X}) \subset \gamma(\mathcal{X}')$ *if and only if* $\mathcal{X} \sqsubseteq \mathcal{X}'$.

Proof. When $\mathcal{X}, \mathcal{X}' \neq \bot$, the equivalence is obtained by applying Lemma 7.1. When $\mathcal{X}$ or $\mathcal{X}'$ is equal to $\bot$, this is straightforward. □

Proposition 7.3. *Let* $\mathcal{X}, \mathcal{X}' \in \mathsf{MaxPoly} \setminus \{ \bot \}$.

Suppose that $\mathcal{X} = ((P, R), \cdot)$ *and* $\mathcal{X}' = ((P', R'), (A', \boldsymbol{c}', B', \boldsymbol{d}'))$, *with* $p = |P'| + |R'|$ *and* $A', B' \in \mathbb{R}_{\max}^{q \times d}$, $\boldsymbol{c}', \boldsymbol{d}' \in \mathbb{R}_{\max}^{q}$.

Then the complexity of the evaluation of $\mathcal{X} \sqsubseteq \mathcal{X}'$ *is* $O(dp(|P| + |R|))$ *if the definition* (7.2) *of* $\sqsubseteq$ *is used, and* $O(dq(|P| + |R|))$ *if* (7.3) *is used.*

Remark 7.1. Observe that $\sqsubseteq$ is not a partial order over MaxPoly. Indeed, because of the non-canonicity of the constraint component of the abstract elements, we may have $\mathcal{P} \sqsubseteq \mathcal{P}' \sqsubseteq \mathcal{P}$ while $\mathcal{P}$ and $\mathcal{P}'$ are distinct.

Note that the canonicity of the constraint component could be enforced by requiring that the constraints correspond to the scaled extreme elements of the polar cone of $\widehat{\widehat{\mathcal{X}}}$. However, it could introduce additional redundant inequalities, such as tautologies (see Remark 5.5) in the system.

We can also define least and greatest elements of the abstract domain MaxPoly. The least element is naturally equal to $\bot$, while we define $\top$ by:

$$\top \stackrel{def}{=} ((\{ \mathbb{0} \}, \{ \boldsymbol{\epsilon}^1, \ldots, \boldsymbol{\epsilon}^d \}), \emptyset)$$

where, here, $\emptyset$ represents the empty system of inequalities.[5] Then we have:

$$\gamma(\bot) = \emptyset, \qquad \gamma(\top) = \mathbb{R}_{\max}^{\mathsf{Vars}}.$$

7.1.3 Abstract union operator

The abstract union operator $\sqcup$ over-approximates the union operator $\cup$ on $\wp(\mathbb{R}^{\mathsf{Vars}})$. It is defined by means of the generator components of its two operands, and yields a result under the same form. Given $\mathcal{X}, \mathcal{X}' \in \mathsf{MaxPoly}$, it is defined by:

$$\mathcal{X} \sqcup \mathcal{X}' \stackrel{def}{=} \begin{cases} \mathcal{X} & \text{if } \mathcal{P}' = \bot, \\ \mathcal{X}' & \text{if } \mathcal{P} = \bot, \\ \mathsf{ofGen}(\textsc{MinimizePoly}(P \cup P', R \cup R')) & \text{if } \mathcal{X} = ((P, R), \cdot), \mathcal{X}' = ((P', R'), \cdot). \end{cases} \tag{7.4}$$

The function MinimizePoly eliminates redundant elements in the generating representation (P, R) given as input (Figure 7.2). It relies on an analogue function, Minimize (Figure 7.3), defined on generating sets of tropical cones. The correctness of the two functions is ensured by Proposition 5.6, Lemma 5.7, and Corollary 2.10. In particular, for any $G \subset \mathbb{R}_{\max}^d$, the following relation holds:

$$\mathsf{cone}(G) = \mathsf{cone}(\textsc{Minimize}(G)),$$

since the set Minimize(G) consists of exactly one representative of each extreme ray of $\mathsf{cone}(G)$. As a consequence, we also have:

$$\mathsf{co}(P) \oplus \mathsf{cone}(R) = \mathsf{co}(Q) \oplus \mathsf{cone}(S) \quad \text{where } (Q, S) = \textsc{MinimizePoly}(P, R),$$

```
1: procedure MinimizePoly(P, R)
2:     ι^{-1}(Minimize(ι(G)))
3: end
```

Figure 7.2: Eliminating redundant elements in a generating representation of a tropical polyhedron

```
1: procedure Minimize(G)
2:     H := ∅
3:     for all g ∈ G do
4:         if g ≠ ⨁_{h∈H} (h\g)h then
5:             append g to H
6:         end
7:     done
8:     return H
9: end
```

Figure 7.3: Eliminating redundant elements in a generating set of a tropical polyhedral cone

thanks to Corollary 2.10. The time complexity of a call to Minimize(G) is $O(d|G|^2)$. Note that the linear independence criterion provided by Lemma 5.8 can be used as well, with the same worst-case time complexity.

The operator $\sqcup$ is sound, and is even the most precise abstract union operator:

Proposition 7.4. *Let $\mathcal{X}, \mathcal{X}' \in \mathsf{MaxPoly}$. Then the following properties holds:*

- $\gamma(\mathcal{X}) \cup \gamma(\mathcal{X}') \subset \gamma(\mathcal{X} \sqcup \mathcal{X}')$.
- *for every $\mathcal{Y} \in \mathsf{MaxPoly}$ such that $\gamma(\mathcal{X}) \cup \gamma(\mathcal{X}') \subset \gamma(\mathcal{Y})$, we have $\gamma(\mathcal{X} \sqcup \mathcal{X}') \subset \gamma(\mathcal{Y})$.*

Proof. First consider two non-empty tropical polyhedra $\mathcal{P} = \mathsf{co}(P) \oplus \mathsf{cone}(R)$ and $\mathcal{P}' = \mathsf{co}(P') \oplus \mathsf{cone}(R')$, and let us define $\mathcal{Q} = \mathsf{co}(P \cup P') \oplus \mathsf{cone}(R \cup R')$. We claim that $\mathcal{Q} = cl(\mathsf{co}(\mathcal{P} \cup \mathcal{P}'))$.

Indeed, we have $\mathsf{co}(P) \subset \mathsf{co}(P \cup P')$ and $\mathsf{cone}(R) \subset \mathsf{cone}(R \cup R')$, so that $\mathcal{P} = \mathsf{co}(P) \oplus \mathsf{cone}(R) \subset \mathcal{Q}$. Similarly, $\mathcal{P}' \subset \mathcal{Q}$. As a consequence, $\mathcal{P} \cup \mathcal{P}' \subset \mathcal{Q}$, and it follows that any affine combination of the elements of $\mathcal{P} \cup \mathcal{P}'$ also belongs to $\mathcal{Q}$. Therefore, $\mathsf{co}(\mathcal{P} \cup \mathcal{P}') \subset \mathcal{Q}$. Since $\mathcal{Q}$ is closed (Lemma 2.5), this implies that $cl(\mathsf{co}(\mathcal{P} \cup \mathcal{P}')) \subset \mathcal{Q}$.

Conversely, let $\boldsymbol{x} \in \mathcal{Q}$. Supposing $P = (\boldsymbol{p}_i)$, $R = (\boldsymbol{r}_i)$, $P' = (\boldsymbol{p}'_i)$, and $R' = (\boldsymbol{r}'_i)$, there exists $(\alpha_i)_i, (\beta_j)_j, (\lambda_i)_i, (\mu_j)_j$ such that $\bigoplus_i \alpha_i \oplus \bigoplus_j \beta_j = \mathbb{1}$, and:

$$\boldsymbol{x} = \bigoplus_i \alpha_i \boldsymbol{p}_i \oplus \bigoplus_i \lambda_i \boldsymbol{r}_i \oplus \bigoplus_j \beta_j \boldsymbol{p}'_j \oplus \bigoplus_j \mu_j \boldsymbol{r}'_j.$$

Let $\kappa = \bigoplus_i \alpha_i$ and $\kappa' = \bigoplus_j \beta_j$. Suppose without loss of generality that $\kappa = \mathbb{1}$. We distinguish two cases:

- first suppose that $\kappa = \mathbb{1}$ and $\kappa' > \mathbb{0}$. Then

  $$\boldsymbol{x} = \kappa \boldsymbol{y} \oplus \kappa' \boldsymbol{z}$$

  with $\boldsymbol{y} = \bigoplus_i \alpha_i \boldsymbol{p}_i \oplus \bigoplus_i \lambda_i \boldsymbol{r}_i$, and $\boldsymbol{z} = \bigoplus_j \kappa^{-1} \beta_j \boldsymbol{p}'_j \bigoplus \bigoplus_j \mu_j \boldsymbol{r}'_j$. Since $\bigoplus_i \alpha_i = \kappa = \mathbb{1}$ and $\bigoplus_j \kappa^{-1} \beta_j = \mathbb{1}$, then $\boldsymbol{y} \in \mathcal{P}$ and $\boldsymbol{z} \in \mathcal{P}'$. Besides, $\kappa \oplus \kappa' = \mathbb{1}$, so that $\boldsymbol{x} \in \mathsf{co}(\mathcal{P} \cup \mathcal{P}')$.

[5] Recall that the $(\boldsymbol{\epsilon}^i)_i$ are the d elements of the canonical basis of $\mathbb{R}^d_{\max}$.

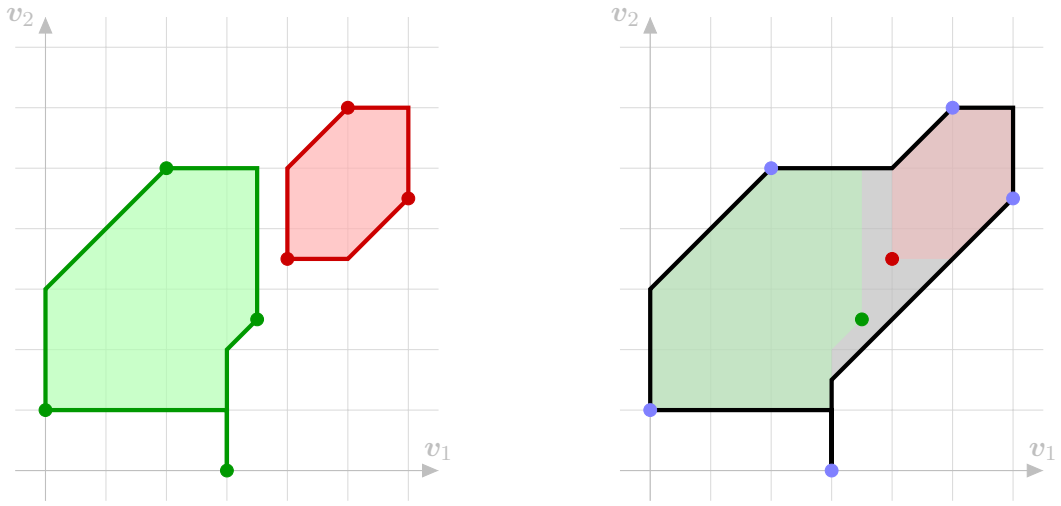


Figure 7.4: Abstract union of two elements of MaxPoly

- now suppose $\kappa = \mathbb{1}$ and $\kappa' = \mathbb{0}$. Let $\boldsymbol{x}_n$ $(n \geq 0)$ be defined by:

$$\boldsymbol{x}_n = \bigoplus_i \alpha_i \boldsymbol{p}_i \oplus \bigoplus_i \lambda_i \boldsymbol{r}_i \oplus \bigoplus_j (-n)\boldsymbol{p}'_j \oplus \bigoplus_j \mu_j \boldsymbol{r}'_j.$$

  Since $\bigoplus_i \alpha_i \oplus \bigoplus_j (-n) = \bigoplus_i \alpha_i = \mathbb{1}$, we have $\boldsymbol{x}_n \in \mathcal{Q}$. Besides, $\bigoplus_j (-n) = -n > \mathbb{0}$, so that using the previous case, we have $\boldsymbol{x}_n \in \mathsf{co}(\mathcal{P} \cup \mathcal{P}')$. As a consequence, $\boldsymbol{x} = \lim_{n \to +\infty} \boldsymbol{x}_n \in cl(\mathsf{co}(\mathcal{P} \cup \mathcal{P}'))$.

Now, consider $\mathcal{X}, \mathcal{X}' \in \mathsf{MaxPoly}$. If $\mathcal{X}$ or $\mathcal{X}'$ is equal to $\bot$, then clearly the statement holds.

Otherwise, using (7.1.3) and the previous identity, supposing $\mathcal{X} = ((P, R), \cdot)$ and $\mathcal{X} = ((P', R'), \cdot)$, we have $cl(\mathsf{co}(\overline{\mathcal{X}} \cup \overline{\mathcal{X}'})) = \mathsf{co}(P \cup P') \oplus \mathsf{cone}(R \cup R') = \overline{\mathcal{X} \sqcup \mathcal{X}'}$. Then $\overline{\mathcal{X}}, \overline{\mathcal{X}'} \subset \overline{\mathcal{X} \sqcup \mathcal{X}'}$, thus $\gamma(\mathcal{X}), \gamma(\mathcal{X}') \subset \gamma(\mathcal{X} \sqcup \mathcal{X}')$. Besides, if $\mathcal{Y} \in \mathsf{MaxPoly}$ satisfies $\gamma(\mathcal{X}), \gamma(\mathcal{X}') \subset \gamma(\mathcal{Y})$, then $\overline{\mathcal{X} \sqcup \mathcal{X}'} = cl(\mathsf{co}(\overline{\mathcal{X}} \cup \overline{\mathcal{X}'})) \subset \overline{\mathcal{Y}}$, so that $\gamma(\mathcal{X} \sqcup \mathcal{X}') \subset \gamma(\mathcal{Y})$. □

Example 7.2. Figure 7.4 provides an illustration of the abstract union of two abstract elements, whose corresponding tropical polyhedra are depicted in green and red with their extreme elements (left side). The resulting abstract element is represented in the right side. The associated tropical polyhedron is indeed the least polyhedron which contains the two initial polyhedra. Note that only the points depicted in blue are extreme, while the red and green ones are eliminated by the call to MinimizePoly.

Proposition 7.5. *Let* $\mathcal{X} = ((P, R), \cdot), \mathcal{X}' = ((P', R'), \cdot) \in \mathsf{MaxPoly}$. *Then* $\mathcal{X} \sqcup \mathcal{X}'$ *can be computed in time* $O(d \times (|P| + |P'| + |R| + |R'|)^2)$.

Remark 7.3. The function MinimizePoly can be optimized, by observing that an element of $\iota(P, R)$ of the form $(\boldsymbol{r}, \mathbb{0})$ where $\boldsymbol{r} \in R$, cannot be expressed as a tropical combination involving elements $(\boldsymbol{p}, \mathbb{1})$ with $\boldsymbol{p} \in P$. Indeed, the $(d+1)$-th entries of the latter are non-null, while the $(d+1)$-th entry of the former is null. Consequently, MinimizePoly could be implemented as follows:

```
1: procedure MinimizePoly((P, R))
2:    Q := ∅, S := ∅
3:    for all r ∈ R do
4:       if r ≠ ⨁_{s∈S}(s\r)s then
5:          append r to S
6:       end
7:    done
8:    for all p ∈ P do
9:       if (p, 𝟙) ≠ ⨁_{q∈Q}((q, 𝟙)\(p, 𝟙))(q, 𝟙) ⊕ ⨁_{s∈S}((s, 𝟘)\(p, 𝟙))(s, 𝟘) then
10:         append p to Q
11:      end
12:   done
13:   return (Q, S)
14: end
```

The resulting time complexity is $O(d|R|(|P|+|R|))$, instead of $O(d(|P|+|R|)^2)$.

7.1.4 Abstract intersection primitives

7.1.4.a Intersection operator. The abstract intersection operator $\sqcap$ over-approximates the intersection $\cap$ on $\wp(\mathbb{R}^{\mathsf{Vars}})$. It is defined dually to the abstract union. It uses the constraints forms of its operands, concatenates them, and eliminates the inequalities which can be expressed as tropical linear combinations of the others:

$$\mathcal{X} \sqcap \mathcal{X}' \stackrel{def}{=} \begin{cases} \bot & \text{if } \mathcal{X} = \bot \text{ or } \mathcal{X}' = \bot, \\ \mathsf{ofCons}(A'', \boldsymbol{c}'', B'', \boldsymbol{d}'') & \text{if } \mathcal{X} = (\cdot, (A, \boldsymbol{c}, B, \boldsymbol{d})), \mathcal{X}' = (\cdot, (A', \boldsymbol{c}', B', \boldsymbol{d}')), \end{cases} \tag{7.5}$$

where the constraint component $(A'', \boldsymbol{c}'', B'', \boldsymbol{d}'')$ is given by:

$${}^t\begin{pmatrix} A'' & \boldsymbol{c}'' & B'' & \boldsymbol{d}'' \end{pmatrix} = \textsc{Minimize}\left({}^t\begin{pmatrix} A & \boldsymbol{c} & B & \boldsymbol{d} \\ A' & \boldsymbol{c}' & B' & \boldsymbol{d}' \end{pmatrix} \right).$$

(We assimilate matrices to the set formed by their columns, and inversely.) The call to the function Minimize allows to eliminate constraints of the concatenated system which are combinations of the others.

This intersection operator is both sound and exact:

Proposition 7.6. *Let $\mathcal{X}, \mathcal{X}' \in \mathsf{MaxPoly}$. Then we have:*

$$\gamma(\mathcal{X}) \cap \gamma(\mathcal{X}') = \gamma(\mathcal{X} \sqcap \mathcal{X}').$$

Proof. The statement obviously holds if $\mathcal{X}$ or $\mathcal{X}'$ is equal to $\bot$.

Otherwise, we have:

$$\begin{aligned} &\nu \in \gamma(\mathcal{X}) \cap \gamma(\mathcal{X}') \\ &\quad \iff \begin{cases} A(\nu(\boldsymbol{v}_1), \dots, \nu(\boldsymbol{v}_d)) \oplus \boldsymbol{c} \leq B(\nu(\boldsymbol{v}_1), \dots, \nu(\boldsymbol{v}_d)) \oplus \boldsymbol{d} \\ A'(\nu(\boldsymbol{v}_1), \dots, \nu(\boldsymbol{v}_d)) \oplus \boldsymbol{c}' \leq B'(\nu(\boldsymbol{v}_1), \dots, \nu(\boldsymbol{v}_d)) \oplus \boldsymbol{d}' \end{cases} \\ &\quad \iff A''(\nu(\boldsymbol{v}_1), \dots, \nu(\boldsymbol{v}_d)) \oplus \boldsymbol{c}'' \leq B''(\nu(\boldsymbol{v}_1), \dots, \nu(\boldsymbol{v}_d)) \oplus \boldsymbol{d}'' \qquad \text{using (7.1.3)} \\ &\quad \iff \nu \in \gamma(\mathcal{X} \sqcap \mathcal{X}'). \end{aligned}$$

□

Proposition 7.7. *Let $\mathcal{X} = (\cdot, (A, \boldsymbol{c}, B, \boldsymbol{d})), \mathcal{X}' = (\cdot, (A', \boldsymbol{c}', B', \boldsymbol{d}')) \in \mathsf{MaxPoly}$. Suppose that the systems $A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d}$ and $A'\boldsymbol{x} \oplus \boldsymbol{c}' \leq B'\boldsymbol{x} \oplus \boldsymbol{d}'$ are respectively formed by p and q inequalities.*

Then $\mathcal{X} \sqcap \mathcal{X}'$ can be computed in time $O(d(p+q)^2)$.

7.1.4.b Tropically affine conditions. A *tropically affine condition* refers to a system of tropical affine inequalities over the elements of Vars, *i.e.* of the form $A(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d) \oplus \boldsymbol{c} \leq B(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d) \oplus \boldsymbol{d}$ where $A, B \in \mathbb{R}_{\max}^{p \times d}$ and $\boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}_{\max}^{p}$. We now define an abstract primitive which over-approximates the effect of a condition on an abstract element (see Section 6.3.2.c).

Using the constraint form. A first abstract primitive can be defined by means of the constraint component of the abstract element:

$$(\!|A(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d) \oplus \boldsymbol{c} \leq B(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d) \oplus \boldsymbol{d}|\!)(\mathcal{X}) \stackrel{def}{=} \mathcal{X} \sqcap \mathsf{ofCons}(A', \boldsymbol{c}', B', \boldsymbol{d}').$$

where $A', B' \in \mathbb{R}_{\max}^{q \times d}$ and $\boldsymbol{c}', \boldsymbol{d}' \in \mathbb{R}_{\max}^{p}$ are defined by:

$${}^{t}\begin{pmatrix} A' & \boldsymbol{c}' & B' & \boldsymbol{d}' \end{pmatrix} = \textsc{Minimize}\left({}^{t}\begin{pmatrix} A & \boldsymbol{c} & B & \boldsymbol{d} \end{pmatrix}\right).$$

Proposition 7.6 allows to show that this abstract primitive is both sound and exact:

Proposition 7.8. *Let $\mathcal{X} \in \mathsf{MaxPoly}$, $A, B \in \mathbb{R}_{\max}^{p \times d}$ and $\boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}_{\max}^{p}$. Then we have:*

$$\{\, \boldsymbol{v} \in \gamma(\mathcal{X}) \mid A(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) \oplus \boldsymbol{c} \leq B(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) \oplus \boldsymbol{d} \,\} = \gamma((\!|A(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d) \oplus \boldsymbol{c} \leq B(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d) \oplus \boldsymbol{d}|\!)(\mathcal{X})).$$

Using the generator form. Alternatively, the abstract primitive can be defined so as to use the generator component. Suppose that $\mathcal{X} = ((P, R), \cdot)$, and let $P = (\boldsymbol{p}^i)$, $R = (\boldsymbol{r}^j)$, $p = |P|$, and $r = |R|$. Then $\boldsymbol{x} \in \overline{\mathcal{X}}$ satisfies the constraints $A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d}$ if and only if there exist $\boldsymbol{\lambda} = (\boldsymbol{\lambda}_i) \in \mathbb{R}_{\max}^{p}$ and $\boldsymbol{\mu} = (\boldsymbol{\mu}_i) \in \mathbb{R}_{\max}^{r}$ such that:

$$\begin{cases} \boldsymbol{x} = \bigoplus_{i=1}^{p} \boldsymbol{\lambda}_i \boldsymbol{p}^i \oplus \bigoplus_{j=1}^{r} \boldsymbol{\mu}_j \boldsymbol{r}^j \\ \bigoplus_{i=1}^{p} \boldsymbol{\lambda}_i = \mathbb{1} \\ A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d} \end{cases}$$

It can be seen as a system of tropical linear inequalities over the unknown $(\boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{x})$. The homogenized system can be solved using the algorithm ComputeExtRays, which yields a minimal generating set $G \subset \mathbb{R}_{\max}^{p+r+d+1}$. Let $H \subset \mathbb{R}_{\max}^{d+1}$ be the system obtained by projecting each element of G on its $(d+1)$ last coordinates, and let $(Q, S) = \iota^{-1}(G)$. We can now define:

$$\{\, \boldsymbol{v} \in \gamma(\mathcal{X}) \mid A(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) \oplus \boldsymbol{c} \leq B(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) \oplus \boldsymbol{d} \,\}(\mathcal{X}) \stackrel{def}{=} \begin{cases} \mathsf{ofGen}(\textsc{MinimizePoly}(Q, S)) & \text{if } (Q, S) \text{ satisfies Condition (iii) of Definition 7.1,} \\ \bot & \text{otherwise.} \end{cases}$$

It can be shown that Proposition 7.8 still holds with this definition.

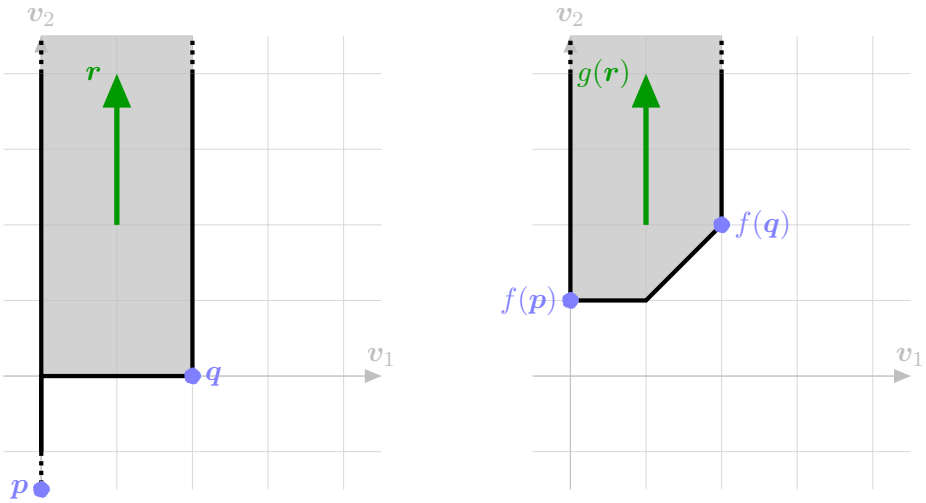


Figure 7.5: Illustration of the assignment $\boldsymbol{v}_2 \leftarrow \max(\boldsymbol{v}_1, \boldsymbol{v}_2, 1)$

7.1.5 Abstract assignment operators

7.1.5.a Tropically affine assignments. A *tropically affine assignment* is an assignment of the form $\boldsymbol{v}_k \leftarrow \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l \boldsymbol{v}_l$ for some $\lambda_0, \lambda_1, \ldots, \lambda_d \in \mathbb{R}_{\max}$ (where the λ_i are not all equal to $\mathbb{0}$).

Intuitively, the effect of such an assignment can be seen as the application of a tropical affine map on the tropical polyhedra. The corresponding abstract primitive $(\!|\boldsymbol{v}_k \leftarrow \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l \boldsymbol{v}_l|\!)$ takes as input and returns generators components, and is defined as follows:

$$(\!|\boldsymbol{v}_k \leftarrow \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l \boldsymbol{v}_l|\!)(\mathcal{X}) \stackrel{def}{=} \begin{cases} \bot & \text{if } \mathcal{X} = \bot, \\ \mathsf{ofGen}(\textsc{MinimizePoly}(f(P), (\sigma \circ g)(R))) & \text{if } \mathcal{X} = ((P,R),\cdot) \end{cases}$$

where $f, g : \mathbb{R}^d_{\max} \to \mathbb{R}^d_{\max}$ are given by:

$$[f(\boldsymbol{x})]_i = \begin{cases} \boldsymbol{x}_i & \text{if } i \neq k, \\ \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l \boldsymbol{x}_l & \text{if } i = k, \end{cases} \qquad [g(\boldsymbol{x})]_i = \begin{cases} \boldsymbol{x}_i & \text{if } i \neq k, \\ \bigoplus_{l=1}^{d} \lambda_l \boldsymbol{x}_l & \text{if } i = k. \end{cases}$$

The function g can be seen as a linear version of the function f. The functions f and g are respectively affine and linear maps, which represent the side-effect of the assignment on the points and rays of the tropical polyhedron. The function σ is then applied to provide scaled representative of rays.[6] Finally, the call to function MinimizePoly ensures that the resulting generator component is under minimal form.

Example 7.4. Consider the abstract element represented by the tropical polyhedron given in the left side of Figure 7.5. Its generator component is formed by the vertices $\boldsymbol{p} = (0, -\infty)$ and $\boldsymbol{q} = (2, 0)$, and by the ray $\boldsymbol{r} = (-\infty, 0)$.

Its image by the abstract assignment operation $(\!|\boldsymbol{v}_2 \leftarrow \max(\boldsymbol{v}_1, \boldsymbol{v}_2, 1)|\!)$ is depicted in the right side. Its generator component consists of the vertices $f(\boldsymbol{p}) = (0, 1)$, $f(\boldsymbol{q}) = (2, 2)$, and the ray $g(\boldsymbol{r}) = \boldsymbol{r} = (-\infty, 0)$.

[6] Recall that σ has been introduced in Section 2.2.5.

This abstract operator can be shown to be sound and exact:

Proposition 7.9. *Let $\mathcal{X} \in \mathsf{MaxPoly}$. Then we have:*

$$\left\{ \nu[\boldsymbol{v}_k \mapsto \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l \nu(\boldsymbol{v}_l)] \mid \nu \in \gamma(\mathcal{X}) \right\} = \gamma(\langle\!|\boldsymbol{v}_k \leftarrow \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l \boldsymbol{v}_l|\!\rangle(\mathcal{X})).$$

Proof. The property is straightforward when $\mathcal{X} = \bot$.

Otherwise, let $(P', R') = \textsc{MinimizePoly}(f(P), (\sigma \circ g)(R))$ as above. Using (7.1.3), we have:

$$\mathsf{co}(P') \oplus \mathsf{cone}(R') = \mathsf{co}(f(P)) \oplus \mathsf{cone}((\sigma \circ g)(R)) = \mathsf{co}(f(P)) \oplus \mathsf{cone}(g(R)).$$

Let us define $\mathcal{Y} = \langle\!|\boldsymbol{v}_k \leftarrow \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_j \boldsymbol{v}_l|\!\rangle(\mathcal{X})$.

Let $\nu \in \gamma(\mathcal{X})$, and $\nu' = \nu[\boldsymbol{v}_k \mapsto \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_j \nu(\boldsymbol{v}_l)]$. There exists $(\alpha_i)_i, (\beta_j)_j$ such that:

$$(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) = \bigoplus_i \alpha_i \boldsymbol{p}^i \oplus \bigoplus_j \beta_j \boldsymbol{r}^j,$$

with $\bigoplus_i \alpha_i = \mathbb{1}$. As a consequence,

$$\begin{aligned}
\nu'(\boldsymbol{v}_k) &= \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l (\bigoplus_i \alpha_i \boldsymbol{p}_l^i \oplus \bigoplus_j \beta_j \boldsymbol{r}_l^j) \\
&= \lambda_0 \oplus \bigoplus_{i,l} (\alpha_i \lambda_l) \boldsymbol{p}_l^i \oplus \bigoplus_{j,l} (\beta_j \lambda_l) \boldsymbol{r}_l^j \\
&= \bigoplus_i \alpha_i (\lambda_0 \oplus \bigoplus_l \lambda_l \boldsymbol{p}_l^i) \oplus \bigoplus_j \beta_j (\bigoplus_l \lambda_l \boldsymbol{r}_l^j) \qquad \text{since } \bigoplus_i \lambda_0 \alpha_i = \lambda_0 \\
&= \bigoplus_i \alpha_i (f(\boldsymbol{p}^i))_k \oplus \bigoplus_j \beta_j (g(\boldsymbol{r}^j))_k
\end{aligned}$$

and trivially, $\nu'(\boldsymbol{v}_l) = \bigoplus_i \alpha_i (f(\boldsymbol{p}^i))_l \oplus \bigoplus_j \beta_j (g(\boldsymbol{r}^j))_l$ for $l \neq k$. Thus, $(\nu'(\boldsymbol{v}_1), \ldots, \nu'(\boldsymbol{v}_d)) \in \mathsf{co}(f(P)) \oplus \mathsf{cone}(g(R))$, so that $\nu' \in \gamma(\mathcal{Y})$.

Conversely, supposing that $\nu' \in \gamma(\mathcal{Y})$, we have:

$$(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) = \bigoplus_i \alpha_i f(\boldsymbol{p}^i) \oplus \bigoplus_j \beta_j g(\boldsymbol{r}^j)$$

for some $(\alpha_i)_i, (\beta_j)_j$ such that $\bigoplus_i \alpha_i = \mathbb{1}$. Let us introduce $\nu \in \gamma(\mathcal{X})$ defined by:

$$(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) \stackrel{def}{=} \bigoplus_i \alpha_i \boldsymbol{p}^i \oplus \bigoplus_j \beta_j \boldsymbol{r}^j.$$

Using the same sequence of identities, we can prove that $\nu' = \nu[\boldsymbol{v}_k \mapsto \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_j \nu(\boldsymbol{v}_l)]$, which terminates the proof. □

Parallel tropically affine assignments. We can extend this method to *parallel tropically affine assignments*, which are assignments $\boldsymbol{v}_k \leftarrow \lambda_0^k \oplus \bigoplus_{l=1}^d \lambda_l^k \boldsymbol{v}_l$ on different coordinates $k \in K$, where $K \subset [d]$. We define:

$$(\!|\boldsymbol{v}_k \leftarrow \lambda_0^k \oplus \bigoplus_{l=1}^{d} \lambda_l^k \boldsymbol{v}_l \text{ for each } k \in K|\!)(\mathcal{X}) \stackrel{def}{=} \begin{cases} \bot & \text{if } \mathcal{X} = \bot \\ \mathsf{ofGen}(\textsc{MinimizePoly}(f'(P), (\sigma \circ g')(R))) & \text{if } \mathcal{X} = ((P,R),\cdot) \end{cases}$$

where $f', g' : \mathbb{R}_{\max}^d \to \mathbb{R}_{\max}^d$ are given by:

$$[f'(\boldsymbol{x})]_i = \begin{cases} \boldsymbol{x}_i & \text{if } i \notin K, \\ \lambda_0^i \oplus \bigoplus_{l=1}^d \lambda_l^i \boldsymbol{x}_l & \text{if } i \in K, \end{cases} \qquad [g'(\boldsymbol{x})]_i = \begin{cases} \boldsymbol{x}_i & \text{if } i \notin K, \\ \bigoplus_{l=1}^d \lambda_l^i \boldsymbol{x}_l & \text{if } i \in K. \end{cases}$$

The soundness and exactness of the abstract primitives still holds:

Proposition 7.10. *Let $\mathcal{X} \in \mathsf{MaxPoly}$. Then we have:*

$$\left\{ \nu[\boldsymbol{v}_k \mapsto \lambda_0^k \oplus \bigoplus_{l=1}^{d} \lambda_l^k \nu(\boldsymbol{v}_l)]_{k \in K} \mid \nu \in \gamma(\mathcal{X}) \right\} = \gamma((\!|\boldsymbol{v}_k \leftarrow \lambda_0^k \oplus \bigoplus_{l=1}^{d} \lambda_l^k \boldsymbol{v}_l \text{ for each } k \in K|\!)(\mathcal{X})).$$

Proposition 7.11. *Let $\mathcal{X} = ((P,R),\cdot) \in \mathsf{MaxPoly}$.*

The time complexity of the assignments $(\!|\boldsymbol{v}_k \leftarrow \lambda_0 \oplus \bigoplus_{l=1}^d \lambda_l \boldsymbol{v}_l|\!)(\mathcal{X})$ and $(\!|\boldsymbol{v}_k \leftarrow \lambda_0^k \oplus \bigoplus_{l=1}^d \lambda_l^k \boldsymbol{v}_l$ for each $k \in K|\!)(\mathcal{X})$ is $O(d(|P|+|R|)^2)$ and $O(d(|P|+|R|)(|K|+|P|+|R|))$ respectively.

7.1.5.b Non-deterministic assignments. Similarly, the abstract primitive $(\!|\boldsymbol{v}_j \leftarrow ?|\!)$ also relies on the generator component of the abstract elements:

$$(\!|\boldsymbol{v}_k \leftarrow ?|\!)(\mathcal{X}) \stackrel{def}{=} \begin{cases} \bot & \text{if } \mathcal{X} = \bot \\ \mathsf{ofGen}(\textsc{MinimizePoly}(h(P), (\sigma \circ h)(R) \cup \{\, \boldsymbol{\epsilon}^k \,\})) & \text{if } \mathcal{X} = ((P,R),\cdot) \end{cases}$$

where $\boldsymbol{\epsilon}^k$ is the k-th element of the canonical basis in $\mathbb{R}_{\max}^d$, and $h : \mathbb{R}_{\max}^d \to \mathbb{R}_{\max}^d$ is given by:

$$[h(\boldsymbol{x})]_i = \begin{cases} \boldsymbol{x}_i & \text{if } i \neq k, \\ \mathbb{0} & \text{otherwise.} \end{cases}$$

Intuitively, the function h deletes the information relative to the variable $\boldsymbol{v}_k$ in every generator of (P,R). Then the ray $\boldsymbol{\epsilon}^k$ is added so as the variable $\boldsymbol{v}_k$ takes arbitrary values.

Example 7.5. The principle of the non-deterministic assignment abstract operator $(\!|\boldsymbol{v}_2 \leftarrow ?|\!)$ is illustrated in Figure 7.6.

Consider the abstract element whose generator component consists of the vertices $\boldsymbol{p}^1 = (0, -\infty)$, $\boldsymbol{p}^2 = (2,2)$, $\boldsymbol{p}^3 = (2, 3.5)$, and $\boldsymbol{p}^4 = (0, 3.5)$ (left side). The operator $(\!|\boldsymbol{v}_2 \leftarrow ?|\!)$

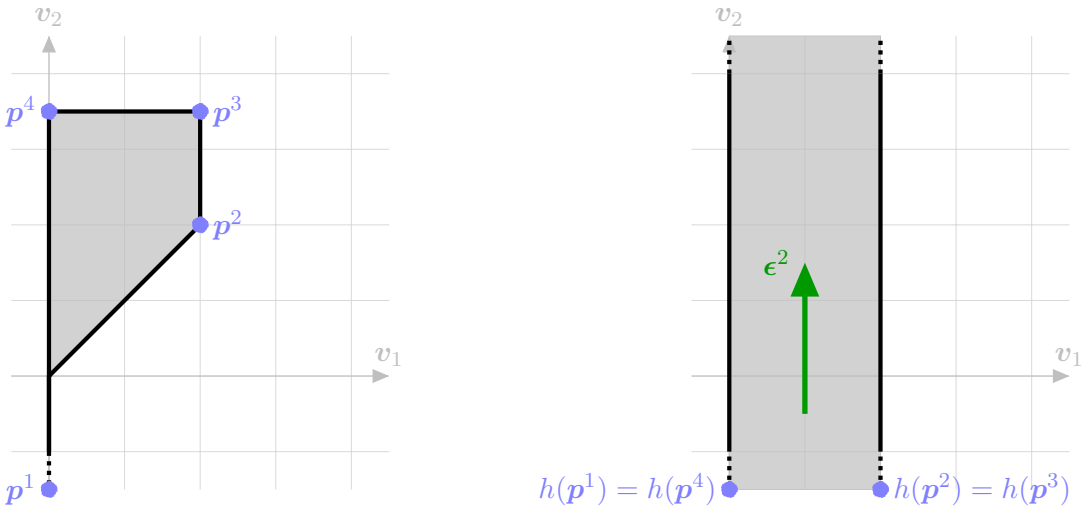


Figure 7.6: Illustration of the non-deterministic assignment $\boldsymbol{v}_2 \leftarrow ?$

annihilates the coordinate corresponding to $\boldsymbol{v}_2$ in the vertices, so that $\boldsymbol{p}^1$ and $\boldsymbol{p}^4$ are mapped to $\boldsymbol{p}^1$, and $\boldsymbol{p}^2$ and $\boldsymbol{p}^3$ to $(2, -\infty)$.

The resulting abstract element is thus generated by the vertices $\boldsymbol{p}^1$ and $(2, -\infty)$, and by the ray $\boldsymbol{\epsilon}^2$ (right side).

This operator is both sound and exact:

Proposition 7.12. *Let $\mathcal{X} \in \mathsf{MaxPoly}$. Then we have:*

$$\{\, \nu[\boldsymbol{v}_k \mapsto x] \mid \nu \in \gamma(\mathcal{X}), x \in \mathbb{R} \,\} = \gamma(\llparenthesis \boldsymbol{v}_k \leftarrow ? \rrparenthesis(\mathcal{X})).$$

Proof. The case $\mathcal{X} = \bot$ is trivial. Now suppose that $\mathcal{X} = ((P, R), \cdot)$. Let $\mathcal{Y} = \llparenthesis \boldsymbol{v}_k \leftarrow ? \rrparenthesis(\mathcal{X})$ and $(P', R') = \textsc{MinimizePoly}(h(P), (\sigma \circ h)(R) \cup \{\, \boldsymbol{\epsilon}^k \,\}))$. By (7.1.3), we have:

$$\mathsf{co}(P') \oplus \mathsf{cone}(R') = \mathsf{co}(h(P)) \oplus \mathsf{cone}(h(R) \cup \{\, \boldsymbol{\epsilon}^k \,\}).$$

Consider $\nu' = \nu[\boldsymbol{v}_k \mapsto x]$ with $\nu \in \gamma(\mathcal{X})$ and $x \in \mathbb{R}$. Then for some $(\alpha_i), (\beta_j)$ such that $\bigoplus_i \alpha_i = \mathbb{1}$,

$$(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) = \bigoplus_i \alpha_i \boldsymbol{p}^i \oplus \bigoplus_j \beta_j \boldsymbol{r}^j$$

so that:

$$\begin{aligned}(\nu'(\boldsymbol{v}_1), \ldots, \nu'(\boldsymbol{v}_d)) &= h(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) \oplus (x\boldsymbol{\epsilon}^j) \\ &= \bigoplus_i \alpha_i h(\boldsymbol{p}^i) \oplus \bigoplus_j \beta_j h(\boldsymbol{r}^j) \oplus (x\boldsymbol{\epsilon}^k),\end{aligned}$$

which proves that $(\nu'(\boldsymbol{v}_1), \ldots, \nu'(\boldsymbol{v}_d))$ belongs to $\gamma(\mathcal{Y})$.

Conversely, consider $\nu' \in \gamma(\mathcal{Y})$. Let $(\alpha_i), (\beta_j)$ such that $\bigoplus_i \alpha_i = \mathbb{1}$ and

$$(\nu'(\boldsymbol{v}_1), \ldots, \nu'(\boldsymbol{v}_d)) = \bigoplus_i \alpha_i h(\boldsymbol{p}^i) \oplus \bigoplus_j \beta_j h(\boldsymbol{r}^j) \oplus \nu'(\boldsymbol{v}_k)\boldsymbol{\epsilon}^k,$$

since the k-th coordinate of the $h(\boldsymbol{p}^i)$ and $h(\boldsymbol{r}^j)$ is equal to $\mathbb{0}$. Now let us define $\nu \in \gamma(\mathcal{X})$ by:

$$(\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) = \bigoplus_i \alpha_i \boldsymbol{p}^i \oplus \bigoplus_j \beta_j \boldsymbol{r}^j.$$

Then $\nu' = \nu[\boldsymbol{v}_j \mapsto x]$ with $x = \nu'(\boldsymbol{v}_j)$, which completes the proof. □

Proposition 7.13. *Let $\mathcal{X} = ((P,R),\cdot) \in \mathsf{MaxPoly}$. The time complexity of the assignment $(\!|\boldsymbol{v}_k \leftarrow ?|\!)(\mathcal{X})$ is $O(d \times (|P| + |R|)^2)$.*

7.1.6 Widening operators

As soon as $d \geq 2$, infinite ascending chains of tropical polyhedra of $\mathbb{R}^d_{\max}$ can be built.[7] As a result, widening operators are defined to enforce convergence.

In this section, we define two possible widenings. The first (Section 7.1.6.a) is defined in the same vein as the widening operator on classical convex polyhedra. The second one (Section 7.1.6.b) is radically new. It is based on a projection operator on tropical cones, and only needs the generator components of the operands.

7.1.6.a "Standard" widening based on stable constraints. We first introduce a widening operator which is analogue to the widening initially defined on classical convex polyhedra in [CH78]: given $\mathcal{X}, \mathcal{X}' \in \mathsf{MaxPoly}$,

$$\mathcal{X} \nabla \mathcal{X}' \stackrel{def}{=} \begin{cases} \mathcal{X}' & \text{if } \mathcal{X} = \bot \\ \mathcal{X} & \text{if } \mathcal{X}' = \bot \\ \mathsf{ofCons}(A'', \boldsymbol{c}'', B'', \boldsymbol{d}'') & \text{if } \mathcal{X} = (\cdot, (A, \boldsymbol{c}, B, \boldsymbol{d})) \text{ and } \mathcal{X} = ((P', R'), \cdot) \end{cases}$$

where the system of constraints $A''\boldsymbol{x} \oplus \boldsymbol{c}'' \leq B''\boldsymbol{x} \oplus \boldsymbol{d}''$ is formed by the inequalities $\boldsymbol{a}\boldsymbol{x} \oplus c \leq \boldsymbol{b}\boldsymbol{x} \oplus d$ of the system $A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d}$ which are satisfied for any element of the tropical polyhedron $\overline{\mathcal{X}'}$, *i.e.*

$$\begin{cases} \boldsymbol{a}\boldsymbol{p}' \oplus c \leq \boldsymbol{b}\boldsymbol{p}' \oplus d & \text{for all } \boldsymbol{p}' \in P', \\ \boldsymbol{a}\boldsymbol{r}' \leq \boldsymbol{b}\boldsymbol{r}' & \text{for all } \boldsymbol{r}' \in R'. \end{cases}$$

Such constraints are said to be stable. The principle of the operator ∇ is therefore to keep only the constraints of the abstract element $\mathcal{X}$ which are stable in $\mathcal{X}'$.

The following proposition ensures that ∇ is indeed a widening operator:

Proposition 7.14. *The following statements hold:*

(i) for all $\mathcal{X}, \mathcal{X}' \in \mathsf{MaxPoly}$, $\mathcal{X} \sqcup \mathcal{X}' \sqsubseteq \mathcal{X} \nabla \mathcal{X}'$,

(ii) for any increasing sequence of elements $\mathcal{X}_0 \sqsubseteq \ldots \sqsubseteq \mathcal{X}_n \sqsubseteq \ldots$, the sequence defined by

$$\begin{cases} \mathcal{Y}_0 \stackrel{def}{=} \mathcal{X}_0 \\ \mathcal{Y}_{n+1} \stackrel{def}{=} \mathcal{Y}_n \nabla \mathcal{X}_{n+1} \end{cases}$$

eventually stabilizes.

[7] For instance, consider the sequence formed by the tropical analogue of cyclic polytopes defined in Remark 2.8, for $n = 1, 2, \ldots$.

Proof. The first statement is obvious.

Now consider two sequences $(\mathcal{X}_n)_n$ and $(\mathcal{Y}_n)$ as in (ii). Up to extracting a subsequence, let us suppose that $\mathcal{X}_0 \neq \bot$ (unless $\mathcal{X}_n = \bot$ for all n, in which case it is obvious that $(\mathcal{Y}_n)_n$ converges in a finite number of steps). In that case, $\mathcal{X}_n \neq \bot$ for all n, and subsequently, $\mathcal{Y}_n \neq \bot$ for all n. The convergence of the sequence of the $(\mathcal{Y}_n)_n$ is ensured by the fact the number of constraints in the system defining each $\mathcal{Y}_n$ is strictly decreasing. □

Nevertheless, this widening operator depends on the choice on the system of constraints of the second operand, while, as discussed in Remark 7.1, there is no canonical form for the constraint component of abstract elements of MaxPoly. As a consequence, $\mathcal{X} \triangledown \mathcal{X}'$ and $\mathcal{X} \triangledown \mathcal{X}''$ may be different while $\mathcal{X}'$ and $\mathcal{X}''$ are equivalent (*i.e.* $\mathcal{X}' \sqsubseteq \mathcal{X}''$ and $\mathcal{X}'' \sqsubseteq \mathcal{X}'$). In the classical case, this difficulty has been overcomed in [Hal79] by adding to the resulting system of constraints $A''\boldsymbol{x} \oplus \boldsymbol{c}'' \leq B''\boldsymbol{x} \oplus \boldsymbol{d}''$ the constraints of $A'\boldsymbol{x} \oplus \boldsymbol{c}' \leq B'\boldsymbol{x} \oplus \boldsymbol{d}'$ which are mutually redundant with some constraints of $\mathcal{X}$. Such constraints can be identified using combinatorial properties (see [BHRZ03, Proposition 1]). Unfortunately, these properties do not have yet any tropical analogues, so that this method cannot be used in our setting.

Proposition 7.15. *Let $\mathcal{X} = (\cdot, (A, \boldsymbol{c}, B, \boldsymbol{d}))$ and $\mathcal{X} = ((P', R'), \cdot)$ be two abstract elements of* MaxPoly *(with $A, B \in \mathbb{R}_{\max}^{p\times d}$ and $\boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}_{\max}^{p}$). The time complexity of $\mathcal{X} \triangledown \mathcal{X}'$ is $O(dp(|P'| + |R'|))$.*

7.1.6.b Widening on generator components. We now define a widening which only uses the generator components of the abstract elements. Since the generator components of the elements of MaxPoly are canonical representations, the problem previously discussed on the standard widening is avoided.

We first define a widening operator on tropical cones. We will then derive a similar primitive on abstract elements of MaxPoly using homogenization.

Widening on cones. This widening $\triangledown_{cone}$ is based on a projection operator on tropical polyhedral cones. Given a tropical polyhedral cone $\mathcal{C} = \mathsf{cone}(G) \subset \mathbb{R}_{\max}^d$, we define:

$$\mathit{\Pi}_{\mathcal{C}}(\boldsymbol{x}) \stackrel{def}{=} \bigoplus_{\boldsymbol{g}\in G} (\boldsymbol{g} \backslash \boldsymbol{x})\boldsymbol{g}.$$

This operator appeared for instance in [CGQ04, DS04]. As a consequence of Lemma 5.7, it satisfies the following requirements: (i) $\mathit{\Pi}_{\mathcal{C}}(\boldsymbol{x}) \in \mathcal{C}$ for all $\boldsymbol{x} \in \mathbb{R}_{\max}^d$, (ii) $\mathit{\Pi}_{\mathcal{C}} \circ \mathit{\Pi}_{\mathcal{C}} = \mathit{\Pi}_{\mathcal{C}}$, and (iii) $\mathit{\Pi}_{\mathcal{C}}(\boldsymbol{x}) = \boldsymbol{x}$ if and only if $\boldsymbol{x} \in \mathcal{C}$. On top of that, we have $\mathit{\Pi}_{\mathcal{C}}(\boldsymbol{x}) \leq \boldsymbol{x}$ for all $\boldsymbol{x} \in \mathbb{R}_{\max}^d$. More precisely, $\mathit{\Pi}_{\mathcal{C}}(\boldsymbol{x})$ can be shown to be the greatest element of $\mathcal{C}$ which is less than or equal to $\boldsymbol{x}$. In particular, it is independent of the choice of the generating set G.

Definition 7.2. Let $\mathcal{C}, \mathcal{C}' \subset \mathbb{R}_{\max}^d$ be two tropical polyhedral cones. Let G, G' be respectively the set of their scaled extreme elements.

Then $\mathcal{C} \triangledown_{cone} \mathcal{C}'$ is defined as the tropical cone generated by the set $G \cup H$, where H is given by:

$$H = \left\{ \boldsymbol{h} \;\middle|\; \boldsymbol{g}' \in G' \text{ and for all } i \in [d], \boldsymbol{h}_i = \begin{cases} \boldsymbol{g}'_i & \text{if } (\mathit{\Pi}_{\mathcal{C}}(\boldsymbol{g}'))_i < \boldsymbol{g}'_i \\ \mathbb{0} & \text{otherwise} \end{cases} \right\}.$$

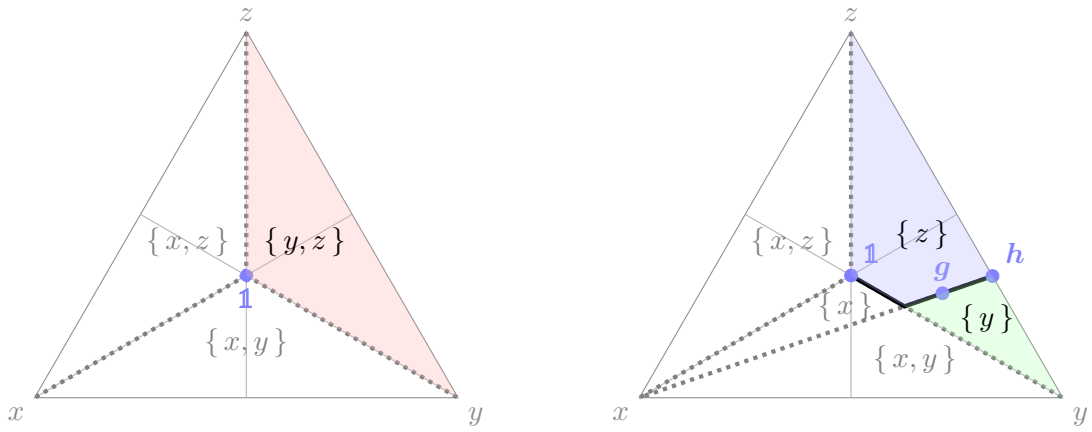


Figure 7.7: Widening tropical polyhedral cones

Intuitively, H is formed by elements $\boldsymbol{h}$ which allows to reach each $\boldsymbol{g}' \in G'$ from its projection $\Pi_{\mathcal{C}}(\boldsymbol{g}')$ on $\mathcal{C}$. Indeed, it can be shown that the equality $\boldsymbol{g}' = \Pi_{\mathcal{C}}(\boldsymbol{g}') \oplus \boldsymbol{h}$ holds.

Example 7.6. Consider the cone $\mathcal{C}$ reduced to the element $\mathbb{1}$ (left side of Figure 7.7), and the cone $\mathcal{C}'$ generated by the elements $\mathbb{1}$ and $\boldsymbol{g} = (-2, 0, -1)$ (right side).

It can be easily verified that $\Pi_{\mathcal{C}}(\boldsymbol{g}) = (-2, -2, -2)$ (it represents the same ray as $\mathbb{1}$). This is why the element $\boldsymbol{h} = (\mathbb{0}, 0, -1)$ is introduced in $\mathcal{D} = \mathcal{C} \nabla_{cone} \mathcal{C}'$ (see right side of Figure 7.7).

This operator can be shown to be a widening operator on the tropical cones:

Proposition 7.16. *The operator ∇_{cone} satisfies the following properties:*

(i) for any tropical polyhedral cones $\mathcal{C}$ and $\mathcal{C}'$, $\mathcal{C} \cup \mathcal{C}' \subset \mathcal{C} \nabla_{cone} \mathcal{C}'$.

(ii) given a sequence of tropical polyhedral cones $\mathcal{C}_0 \subset \cdots \subset \mathcal{C}_n \subset \ldots$, the increasing sequence defined by:

$$\begin{cases} \mathcal{D}_0 \stackrel{def}{=} \mathcal{C}_0 \\ \mathcal{D}_{i+1} \stackrel{def}{=} \mathcal{D}_i \nabla_{cone} \mathcal{C}_{i+1} \end{cases}, \tag{7.6}$$

eventually stabilizes.

The proof of (ii) relies on a combinatorial argument based on a partitioning of the space $\mathbb{R}^d_{\max}$ induced by the projector $\Pi_{\mathcal{C}}$. Given $\mathcal{C}$ a tropical polyhedral cone, we define, for $I \subset [d]$:

$$S_{\mathcal{C}}(I) \stackrel{def}{=} \{ \boldsymbol{x} \in \mathbb{R}^d_{\max} \mid (\Pi_{\mathcal{C}}(\boldsymbol{x}))_i < \boldsymbol{x}_i \text{ iff } i \in I \},$$

Let $sectors(\mathcal{C})$ be the set of the $I \subset [d]$ such that $S_{\mathcal{C}}(I) \neq \emptyset$. The sets $S_{\mathcal{C}}(I)$ for $I \in sectors(\mathcal{C})$ indeed form a partition of $\mathbb{R}^d_{\max}$. In particular, the cone $\mathcal{C}$ coincides with $S_{\mathcal{C}}([d])$.

Example 7.7 (Continuing Example 7.6). If $\mathcal{C}$ is reduced to the element $\mathbb{1}$, the set $\mathbb{R}^d_{\max} \setminus \mathcal{C}$ is split into three sectors $R_{\mathcal{C}}(\{ x, y \})$, $R_{\mathcal{C}}(\{ y, z \})$, and $R_{\mathcal{C}}(\{ x, z \})$ (see Figure 7.7, left-hand side).

Given a set $S \subset \wp([d])$, we define by $\max S$ the antichain formed by the maximal elements of S, *i.e.*:

$$\max S \stackrel{def}{=} \{\, I \in S \mid \forall J \in S, I \subset J \implies I = J \,\}.$$

Let *Antichain* be the set formed by the antichains of elements of $\wp([d])$. Recall that *Antichain* can be partially ordered by the following relation:

$$S_1 \preceq S_2 \stackrel{def}{\iff} \text{for all } I_1 \in S_1, \text{there exists } I_2 \in S_2 \text{ s.t. } I_1 \subset I_2.$$

Also note that since *Antichain* is finite, the partial order $\preceq$ is well-founded.

Lemma 7.17. *Let $\mathcal{C}, \mathcal{D}$ be two tropical polyhedral cones such that $\mathcal{C} \subset \mathcal{D}$. Then we have:*

$$\max \mathit{sectors}(\mathcal{D}) \preceq \max \mathit{sectors}(\mathcal{C}).$$

Proof. Let $J \in \max \mathit{sectors}(\mathcal{D})$, and $\boldsymbol{x} \in S_{\mathcal{D}}(J)$. For any $j \in J$,

$$(\Pi_{\mathcal{D}}(\boldsymbol{x}))_j < \boldsymbol{x}_j.$$

It can be shown that $\Pi_{\mathcal{C}}(\boldsymbol{x}) \leq \Pi_{\mathcal{D}}(\boldsymbol{x})$ since $\mathcal{C} \subset \mathcal{D}$. Thus for every $j \in J$,

$$(\Pi_{\mathcal{C}}(\boldsymbol{x}))_j < \boldsymbol{x}_j.$$

Let $I' \in \mathit{sectors}(\mathcal{C})$ such that $\boldsymbol{x} \in S_{\mathcal{C}}(I')$. Then $J \subset I'$. Thus there exists $I \in \max \mathit{sectors}(\mathcal{C})$ such that $J \subset I$. It follows that $\max \mathit{sectors}(\mathcal{D}) \preceq \max \mathit{sectors}(\mathcal{C})$. □

It follows that if $(\mathcal{C}_n)$ and $(\mathcal{D}_n)$ are two sequences as defined in Proposition 7.16 (ii), then the sequence of the $\max \mathit{sectors}(\mathcal{D}_n)$ is decreasing for the order $\preceq$.

Example 7.8 (Continuing Example 7.6). Consider the cone $\mathcal{D}$ generated by the elements $\mathbb{1}$ and $\boldsymbol{h}$, depicted in the right hand side of Figure 7.7. We have

$$\max \mathit{sectors}(\mathcal{D}) = \{\{\, x, y \,\}, \{\, x, z \,\}, \{\, y \,\}, \{\, z \,\}\},$$

which is indeed less than:

$$\max \mathit{sectors}(\mathcal{C}) = \{\{\, x, y \,\}, \{\, x, z \,\}, \{\, y, z \,\}\}.$$

We are now going to show that when the sequence of the $\mathcal{D}_n$ is strictly increasing at index n, then $\max \mathit{sectors}(\mathcal{D}_n) \neq \max \mathit{sectors}(\mathcal{D}_{n+1})$, so that the sequence $\max \mathit{sectors}(\mathcal{D}_n)$ is also strictly decreasing at this index.

Lemma 7.18. *Let $\mathcal{C}, \mathcal{C}'$ be two tropical polyhedral cones, and $\mathcal{D} = \mathcal{C} \bigtriangledown_{cone} \mathcal{C}'$. If $\mathcal{C} \neq \mathcal{D}$, then we have:*

$$\max \mathit{sectors}(\mathcal{C}) \neq \max \mathit{sectors}(\mathcal{D}).$$

Proof. Let G and G' be respectively the set of the scaled extreme elements of $\mathcal{C}$ and $\mathcal{C}'$. If $\mathcal{C} \neq \mathcal{D}$, then $\mathcal{C}'$ cannot be included into $\mathcal{C}$. As a consequence, there exists $\boldsymbol{g}' \in G'$ which is not in $\mathcal{C}$.

Let $I \in \mathit{sectors}(\mathcal{C})$ such that $\boldsymbol{g}' \in S_{\mathcal{C}}(I)$. Let us define $\boldsymbol{h} \in \mathbb{R}^d_{\max}$ such that:

$$\boldsymbol{h}_i = \begin{cases} \boldsymbol{g}'_i & \text{if } i \in I, \\ \mathbb{0} & \text{otherwise.} \end{cases}$$

Then $\boldsymbol{h}$ is not identically null, and by definition, it belongs to $\mathcal{D}$.

We claim that there is no $J \in \mathit{sectors}(\mathcal{D})$ which contains I. Indeed, suppose that $\boldsymbol{x} \in S_{\mathcal{D}}(J)$ with $J \supset I$. Then for all $i \in I$,

$$(\boldsymbol{x}\backslash\boldsymbol{h})\boldsymbol{x}_i \leq \mathit{\Pi}_{\mathcal{D}}(\boldsymbol{x}) < \boldsymbol{x}_i.$$

But $\boldsymbol{x}\backslash\boldsymbol{h} = \min_i(\boldsymbol{x}_i - \boldsymbol{v}_i) = \min_{i\in I}(\boldsymbol{x}_i - \boldsymbol{h}_i)$, because $\boldsymbol{h}_i = \mathbb{0}$ for $i \notin I$. Hence, there exists $i_0 \in I$ such that $\boldsymbol{x}\backslash\boldsymbol{h} = \boldsymbol{x}_{i_0} - \boldsymbol{h}_{i_0}$. Necessarily, $\boldsymbol{h}_{i_0} \neq \mathbb{0}$ (since $\boldsymbol{h}_{i_0} = \boldsymbol{g}_{i_0} > (\mathit{\Pi}_{\mathcal{C}}(\boldsymbol{g}))_{i_0} \geq \mathbb{0}$), thus $(\boldsymbol{x}\backslash\boldsymbol{h})\boldsymbol{x}_{i_0} = \boldsymbol{x}_{i_0}$, which is a contradition. It follows that there is no $J \in \max \mathit{sectors}(\mathcal{D})$ such that $I \subset J$.

Now, let I' be the unique element of $\max \mathit{sectors}(\mathcal{C})$ such that $I \subset I'$. Then necessarily $I' \notin \max \mathit{sectors}(\mathcal{D})$. □

Example 7.9 (Continuing Example 7.6). Introducing the element $\boldsymbol{h}$ splits the set $\{ y, z \} \in \max \mathit{sectors}(\mathcal{C})$ into two smaller sets $\{ x \}$ and $\{ y \}$ (see Figure 7.7), as expected by the proof of Lemma 7.18.

We can now prove Proposition 7.16.

Proof of Proposition 7.16. (i) let $\mathcal{D} = \mathcal{C} \nabla_{\mathit{cone}} \mathcal{C}$. Clearly, any element of G belongs to $\mathcal{D}$, so that $\mathcal{C} \subset \mathcal{D}$. Besides, any element $\boldsymbol{g}' \in G'$ can be expressed as the sum of $\mathit{\Pi}_{\mathcal{C}}(\boldsymbol{g}') \in \mathcal{D}$ and an element of $H \subset \mathcal{D}$, so that it also belongs to $\mathcal{D}$, which shows that $\mathcal{C}' \subset \mathcal{D}$.

(ii) straightforward using Lemmas 7.17, 7.18, and the well-foundness of the order $\preceq$. □

Remark 7.10. We claim that the result of the widening $\mathcal{C} \nabla_{\mathit{cone}} \mathcal{C}'$ does not depend on the choice of the representing set G and G'. In that case, the assumptions on G and G' in Definition 7.2 could be removed.

Back to tropical polyhedra. The widening defined on tropical polyhedral cones can be adapted to tropical polyhedra using homogenization. Formally, we define:

$$\mathcal{X} \nabla_{\mathit{gen}} \mathcal{X}' \stackrel{\mathit{def}}{=} \begin{cases} \mathcal{X}' & \text{if } \mathcal{X} = \bot \\ \mathcal{X} & \text{if } \mathcal{X}' = \bot \\ \mathsf{ofGen}(P'', R'') & \text{if } \mathcal{X} = ((P, R), \cdot), \mathcal{X}' = ((P', R'), \cdot) \end{cases}$$

where $G = \iota(P, R)$, $G' = \iota(P', R')$, and

$$(P'', R'') = \iota^{-1}(\textsc{Minimize}(G \cup \sigma(H)))$$
$$H = \left\{ \boldsymbol{h} \;\middle|\; \boldsymbol{g}' \in G' \text{ and for all } i \in [d], \boldsymbol{h}_i = \begin{cases} \boldsymbol{g}'_i & \text{if } (\mathit{\Pi}_{\mathsf{cone}(G)}(\boldsymbol{g}'))_i < \boldsymbol{g}'_i \\ \mathbb{0} & \text{otherwise} \end{cases} \right\}.$$

Remark 7.11. According to Condition (iii) of Definition 7.1, note that there exists $\boldsymbol{g} = (\boldsymbol{g}_i) \in G$ such that $\boldsymbol{g}_{d+1} = \mathbb{1}$, so that $\mathsf{cone}(\textsc{Minimize}(G \cup \sigma(H))) = \mathsf{cone}(G \cup \sigma(H))$ contains an element $\boldsymbol{z} \in \mathbb{R}^{d+1}_{\max}$ verifying $\boldsymbol{z}_{d+1} = \mathbb{1}$. Using Proposition 2.9, this ensures that the tropical polyhedron $\overline{\mathcal{X} \nabla_{\mathit{gen}} \mathcal{X}'}$ is non-empty, and that its homogenized cone is precisely $\mathsf{cone}(G \cup \sigma(H))$.

Proposition 7.19. *The following statements hold:*

(i) for all $\mathcal{X}, \mathcal{X}' \in \mathsf{MaxPoly}$, $\mathcal{X} \sqcup \mathcal{X}' \sqsubseteq \mathcal{X} \nabla_{gen} \mathcal{X}'$,

(ii) for any increasing sequence of elements $\mathcal{X}_0 \sqsubseteq \ldots \sqsubseteq \mathcal{X}_n \sqsubseteq \ldots$, *the sequence defined by*

$$\begin{cases} \mathcal{Y}_0 \stackrel{def}{=} \mathcal{X}_0 \\ \mathcal{Y}_{n+1} \stackrel{def}{=} \mathcal{Y}_n \nabla_{gen} \mathcal{X}_{n+1} \end{cases}$$

converges in a finite number of steps.

Proof. Let $\mathcal{X}, \mathcal{X}' \in \mathsf{MaxPoly}$, and $\mathcal{X}'' = \mathcal{X} \nabla_{gen} \mathcal{X}'$. Let $\mathcal{C} = \widehat{\overline{\mathcal{X}}}$, $\mathcal{C}' = \widehat{\overline{\mathcal{X}'}}$, and $\mathcal{C}'' = \widehat{\overline{\mathcal{X}''}}$. We claim that $\mathcal{C}'' = \mathcal{C} \nabla_{cone} \mathcal{C}'$.

If $\mathcal{X}$ or $\mathcal{X}'$ is equal to $\bot$, then one of the cones $\mathcal{C}$ or $\mathcal{C}'$ is empty, in which case the statement can be proved straightforwardly.

Otherwise, let $\mathcal{X} = ((P, R), \cdot)$ and $\mathcal{X}' = ((P', R'), \cdot)$. Let $G = \iota(P, R)$, $G' = \iota(P', R')$, and H' defined as above. Thanks to Corollary 2.10, we indeed have $\mathcal{C} = \mathsf{cone}(G)$ and $\mathcal{C}' = \mathsf{cone}(G')$, and G and G' are respectively the set of the scaled extreme rays of $\mathcal{C}$ and $\mathcal{C}'$. Similarly, $\mathcal{C}'' = \mathsf{cone}(G \cup \sigma(H)) = \mathsf{cone}(G \cup H)$, which proves that $\mathcal{C}'' = \mathcal{C} \nabla_{cone} \mathcal{C}'$.

Now let us show the two statements of Proposition 7.19.

(i) if $\mathcal{X}, \mathcal{X}' \in \mathsf{MaxPoly}$, and $\mathcal{X}'' = \mathcal{X} \nabla_{gen} \mathcal{X}'$. Let $\mathcal{C} = \widehat{\overline{\mathcal{X}}}$, $\mathcal{C}' = \widehat{\overline{\mathcal{X}'}}$, and $\mathcal{C}'' = \widehat{\overline{\mathcal{X}''}}$. We know that $\mathcal{C}'' = \mathcal{C} \nabla_{cone} \mathcal{C}'$. By Proposition 7.16, we have:

$$\mathcal{C} \cup \mathcal{C}' \subset \mathcal{C}''$$

hence, by Proposition 2.1,

$$\overline{\mathcal{X}} \cup \overline{\mathcal{X}'} \subset \overline{\mathcal{X}''}$$

or, equivalently,

$$\gamma(\mathcal{X}) \cup \gamma(\mathcal{X}') \subset \gamma(\mathcal{X}'')$$

which shows that $\mathcal{X} \sqcup \mathcal{X}'' \sqsubseteq \mathcal{X}''$ by Proposition 7.2.

(ii) now consider $(\mathcal{X}_n)$ and $(\mathcal{Y}_n)$. Let $\mathcal{C}_n = \widehat{\overline{\mathcal{X}_n}}$ and $\mathcal{D}_n = \widehat{\overline{\mathcal{Y}_n}}$.

Clearly, $\mathcal{D}_{n+1} = \mathcal{D}_n \nabla_{cone} \mathcal{C}_{n+1}$ for all n. Besides, the sequence of the $\mathcal{C}_n$ is increasing, since the sequence of the $\gamma(\mathcal{X}_n)$ is increasing according to Proposition 7.2. It follows that the sequence of the $\mathcal{D}_n$ converges after a finite number of steps using Proposition 7.16.

As a consequence, if for all n, G_n is the set of the scaled extreme generators of $\mathcal{D}_n$, then there exists N such that for all $n \geq N$, $G_{n+1} = G_n$. It that case, it can be verified that for all $n \geq N+1$, $\mathcal{Y}_n$ is defined as $\mathsf{ofGen}(\iota^{-1}(\textsc{Minimize}(G_n)))$, so that for all $n \geq N+1$, we also have $\mathcal{Y}_{n+1} = \mathcal{Y}_n$. □

Example 7.12. Consider the abstract elements $\mathcal{X}$ and $\mathcal{X}'$ given by:

$$\mathcal{X} = \left((\{\, \mathbb{1} \,\}, \emptyset), \begin{bmatrix} x \leq 0 & 0 \leq x \\ y \leq 0 & 0 \leq y \end{bmatrix} \right)$$

$$\mathcal{X}' = \left((\{\, \mathbb{1}, \boldsymbol{p} \,\}, \emptyset), \begin{bmatrix} x \leq 0 & 0 \leq \max(x, y-1) \\ y \leq 1 & y \leq x+2 \\ 0 \leq y & \end{bmatrix} \right)$$

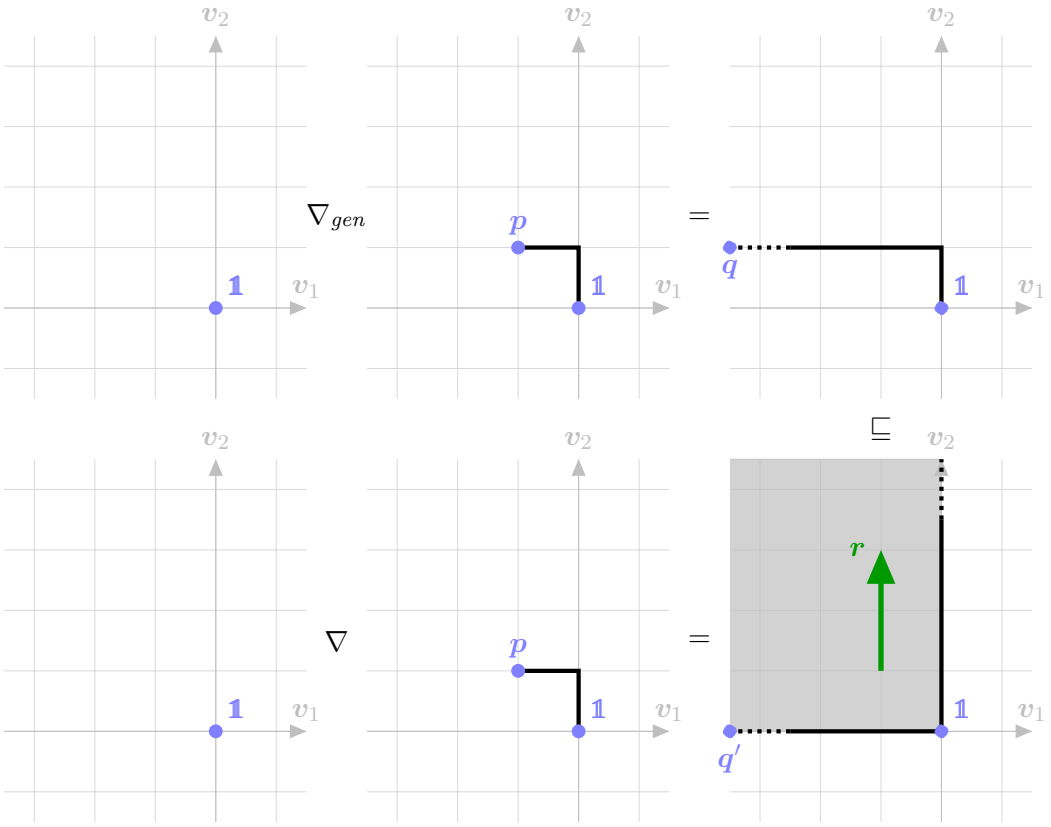


Figure 7.8: Comparing the two widening operators on MaxPoly

where $\boldsymbol{p} = (-1, 1)$. They are respectively represented by the left and middle polyhedra in the top of Figure 7.8. Their homogenized cones are respectively the cones $\mathcal{C}$ and $\mathcal{C}'$ introduced in Example 7.6. The element $\mathcal{X}'' = \mathcal{X} \nabla \mathcal{X}'$ (top right) is therefore generated by the points $\mathbb{1}$ and $\boldsymbol{q} = \iota^{-1}(\boldsymbol{h}) = (-\infty, 1)$.

In contrast, the standard widening applied on $\mathcal{X}$ and $\mathcal{X}'$ is less precise, since it yields the element corresponding to the bottom right polyhedron, which strictly contains $\mathcal{X}''$. Indeed, the inequalities $x \leq 0$ and $0 \leq y$ are the only constraints of $\mathcal{X}$ which are stable.

However, the widening ∇_{gen} can also be less precise than the standard widening. Consider for instance the following abstract elements:

$$\mathcal{Y} = \left((\{\boldsymbol{p}, \boldsymbol{q}\}, \{\boldsymbol{r}\}), \begin{bmatrix} \max(y, 0) \leq x \\ x \leq 1 \end{bmatrix} \right)$$
$$\mathcal{Y}' = \left((\{\boldsymbol{p}, \boldsymbol{q}'\}, \{\boldsymbol{r}\}), \begin{bmatrix} \max(y, 0) \leq x \\ x \leq 2 \end{bmatrix} \right)$$

representing the tropical polyhedra in the left and the middle of Figure 7.9. The inequality $\max(y, 0) \leq x$ is obviously stable, so that it is kept by the standard widening (right top in Figure 7.9). The represented polyhedron is clearly included into the set represented by the abstract element provided by the widening ∇_{gen} (right bottom).

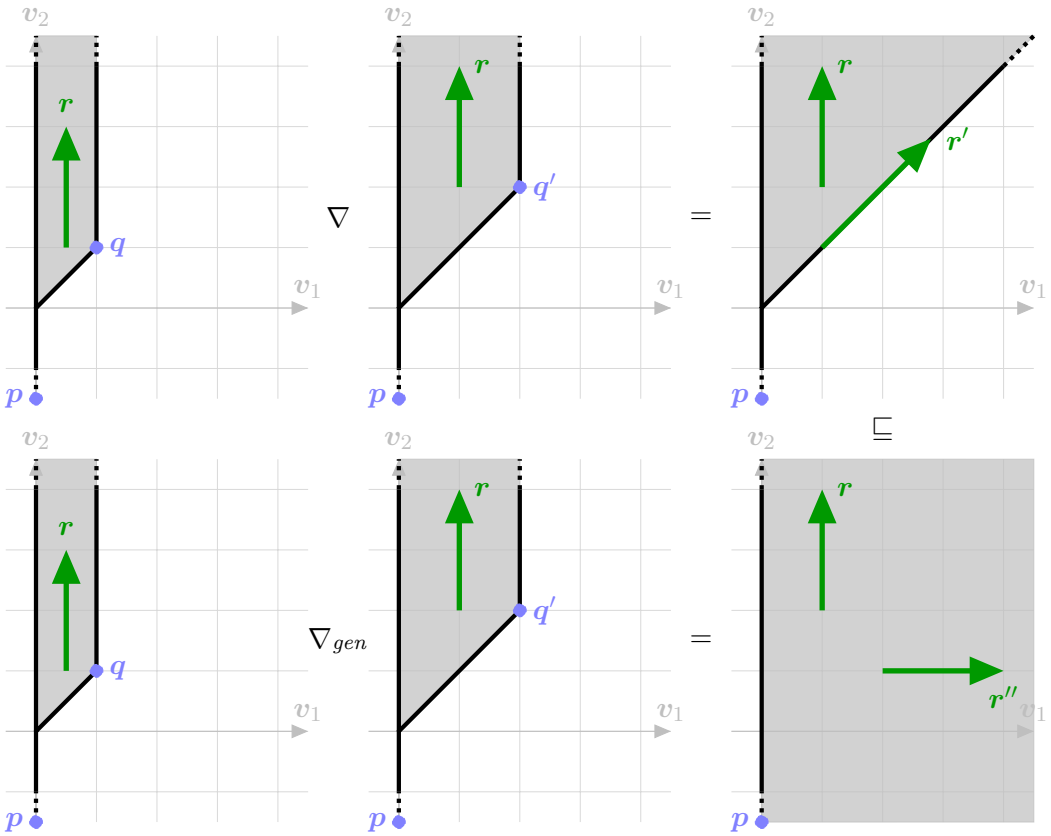


Figure 7.9: Comparing the two widening operators on MaxPoly (cont.)

Proposition 7.20. *Let* $\mathcal{X} = ((P', R'), \cdot)$ *and* $\mathcal{X} = ((P', R'), \cdot)$ *be two abstract elements of* MaxPoly. *The time complexity of* $\mathcal{X} \nabla_{gen} \mathcal{X}'$ *is* $O(d(|P| + |R| + |P'| + |R'|)^2)$.

7.1.7 Reduction with zones

7.1.7.a Zones and tropical polyhedra. Recall that zones are sets defined by bound constraints on the differences $\boldsymbol{x}_i - \boldsymbol{x}_j$.[8] In this section, we show that zones form a particular class of tropical polyhedra. Besides, we introduce an abstract primitive allowing to extract from an abstract element of MaxPoly the smallest abstract element of Zone containing it.

Let us first introduce formal definitions of zones in the tropical setting.

Definition 7.3. Let $\mathcal{Z} \subset \mathbb{R}^d_{\max}$. The set $\mathcal{Z}$ is said to be a *linear zone* if it is equal to the set of the solutions of a system of inequalities of the form

$$A_{ij}\boldsymbol{x}_j \leq \boldsymbol{x}_i \quad \text{for all } (i,j) \in [d]^2,$$

with $A \in \mathbb{R}^{d\times d}_{\max}$. We will use the notation $\mathcal{Z} = \mathsf{linzone}(A)$.

[8] Pay attention to the fact that we distinguish the term *zone*, which refers to such sets, from the abstract elements of the domain Zone. The latter are mapped to the former by the concretization operator γ_{Zone}.

Definition 7.4. Let $\mathcal{Z} \subset \mathbb{R}^d_{\max}$. The set $\mathcal{Z}$ is said to be an *affine zone* if it is equal to the set of the solutions of a system of inequality of the form

$$\begin{cases} A_{ij}\boldsymbol{x}_j \leq \boldsymbol{x}_i & \text{for all } (i,j) \in [d]^2 \\ \boldsymbol{b}_i \leq \boldsymbol{x}_i & \text{for all } i \in [d] \\ \boldsymbol{c}_i\boldsymbol{x}_i \leq \mathbb{1} & \text{for all } i \in [d] \end{cases}$$

with $A \in \mathbb{R}^{d\times d}_{\max}$ and $\boldsymbol{b}, \boldsymbol{c} \in \mathbb{R}^d_{\max}$. We will use the notation $\mathcal{Z} = \mathsf{affzone}(A, \boldsymbol{b}, \boldsymbol{c})$.

Proposition 7.21. *Linear and affine zones are respectively tropical polyhedral cones and tropical polyhedra.*

Proof. It can be easily verified that each inequality corresponds to a tropical (affine) halfspace. □

The following lemma expresses homogenization of affine zones:

Lemma 7.22. *Let $\mathcal{Z} = \mathsf{affzone}(A, \boldsymbol{b}, \boldsymbol{c})$ be a non-empty affine zone ($A \in \mathbb{R}^{d\times d}_{\max}$, $\boldsymbol{b}, \boldsymbol{c} \in \mathbb{R}^d_{\max}$). Then $\widehat{\mathcal{Z}}$ is a linear zone, and $\widehat{\mathcal{Z}} = \mathsf{linzone}(A')$ where $A \in \mathbb{R}^{(d+1)\times(d+1)}_{\max}$ is defined by:*

$$A' = \begin{pmatrix} A & \boldsymbol{b} \\ {}^t\boldsymbol{c} & \mathbb{1} \end{pmatrix}.$$

Proof. It is a direct consequence of Proposition 2.8. □

Given a matrix $A \in \mathbb{R}^{n\times p}_{\max}$, we define the *residuate matrix* $A/A \in \overline{\mathbb{R}}^{n\times n}_{\max}$ by

$$(A/A)_{ij} \stackrel{def}{=} \min_{1\leq k\leq p} A_{ik} - A_{jk},$$

with the convention $-\infty + \infty = -\infty$.

Proposition 7.23. *Let $\mathcal{C} \subset \mathbb{R}^d_{\max}$ be a tropical polyhedral cone. Let G be the matrix whose columns are the scaled extreme elements of $\mathcal{C}$. Suppose that G has no row consisting of null entries.*

Then the set $\mathsf{linzone}(G/G)$ is the smallest linear zone which contains $\mathcal{C}$.

Proof. First observe that all coefficients of G/G are elements of $\mathbb{R}_{\max}$.

For the sake of simplicity, we assimilate G to the set formed by the scaled extreme elements of $\mathcal{C}$. Clearly, for all $\boldsymbol{h} \in G$ and $(i,j) \in [d]^2$, we have:

$$\boldsymbol{h}_i - \boldsymbol{h}_j \geq \min_{\boldsymbol{g}\in G}(\boldsymbol{g}_i - \boldsymbol{g}_j)$$

so that

$$\boldsymbol{h}_i \geq \min_{\boldsymbol{g}\in G}(\boldsymbol{g}_i - \boldsymbol{g}_j)\boldsymbol{h}_j$$

since $\min_{\boldsymbol{g}\in G}(\boldsymbol{g}_i - \boldsymbol{g}_j) \in \mathbb{R}_{\max}$. This shows that $G \subset \mathsf{linzone}(G/G)$, hence $\mathcal{C} \subset \mathsf{linzone}(G/G)$.

Now, let $\mathcal{Z} = \mathsf{linzone}(A)$ be a linear zone such that $\mathcal{C} \subset \mathcal{Z}$. For all $\boldsymbol{g} \in G$, we have:

$$A_{ij}\boldsymbol{g}_j \leq \boldsymbol{g}_i$$

thus

$$A_{ij} \leq \boldsymbol{g}_i - \boldsymbol{g}_j$$

which shows that $A_{ij} \leq (G/G)_{ij}$ for all $(i,j) \in [d]^2$. This implies that $\mathsf{linzone}(G/G) \subset \mathsf{linzone}(A)$. □

Proposition 7.24. *Let $\mathcal{P} \subset \mathbb{R}_{\max}^d$ be a tropical polyhedron such that $\mathcal{P} \cap \mathbb{R}^d$ is not empty. Let (P, R) be the scaled minimal representation of $\mathcal{P}$. Let $A \in \mathbb{R}_{\max}^{d \times d}$ and $\boldsymbol{b}, \boldsymbol{c} \in \mathbb{R}_{\max}^d$ be defined by the relation:*

$$\begin{pmatrix} A & \boldsymbol{b} \\ {}^t\boldsymbol{c} & \mathbb{1} \end{pmatrix} = \iota(P,R)/\iota(P,R),$$

where $\iota(P, R)$ is assimilated the matrix whose columns are the elements of $\iota(P, R)$.

Then the set $\mathsf{affzone}(A, \boldsymbol{b}, \boldsymbol{c})$ is the smallest affine zone which contains $\mathcal{P}$.

Proof. Let G be the matrix whose columns are the elements of $\iota(P, R)$. The fact that $\mathcal{P} \cap \mathbb{R}^d \neq \emptyset$ ensures that G has no identically null row.

Thanks to Corollary 2.10, we know that $\widehat{\mathcal{P}}$ is the tropical cone generated by the columns of G.

Using Proposition 7.23, we know that $\widehat{\mathcal{P}} \subset \mathsf{linzone}(G/G)$ so that $\mathcal{P} = \{\, \boldsymbol{x} \in \mathbb{R}_{\max}^d \mid (\boldsymbol{x}, \mathbb{1}) \in \widehat{\mathcal{P}} \,\} \subset \{\, \boldsymbol{x} \in \mathbb{R}_{\max}^d \mid (\boldsymbol{x}, \mathbb{1}) \in \mathsf{linzone}(G/G) \,\}$. It can be verified that since G contains at least one element $\boldsymbol{g}$ such that $\boldsymbol{g}_{d+1} \neq \mathbb{0}$ (since $\mathcal{P} \neq \emptyset$), then the set $\{\, \boldsymbol{x} \in \mathbb{R}_{\max}^d \mid (\boldsymbol{x}, \mathbb{1}) \in \mathsf{linzone}(G/G) \,\}$ is precisely the affine zone $\mathsf{affzone}(A, \boldsymbol{b}, \boldsymbol{c})$.

Now, consider $\mathcal{Z} = \mathsf{affzone}(A', \boldsymbol{b}', \boldsymbol{c}')$ such that $\mathcal{P} \subset \mathcal{Z}$. Then $\widehat{\mathcal{P}} \subset \widehat{\mathcal{Z}}$, so that by Proposition 7.23, $\mathsf{linzone}(G/G) \subset \widehat{\mathcal{Z}}$. By Lemma 7.22, we have:

$$\begin{pmatrix} A & \boldsymbol{b} \\ {}^t\boldsymbol{c} & \mathbb{1} \end{pmatrix} \geq \begin{pmatrix} A' & \boldsymbol{b}' \\ {}^t\boldsymbol{c}' & \mathbb{1} \end{pmatrix},$$

which ensures that $\mathsf{affzone}(A, \boldsymbol{b}, \boldsymbol{c})$ is contained into $\mathcal{Z}$. □

Using the notations of Section 6.3.2.b, Zone denotes the abstract domain of (affine) zones over the variables Vars. Recall that its elements are either represented by $\bot_{\mathsf{Zone}}$ or by matrices of $\mathbb{R}_{\max}^{(d+1)\times(d+1)}$, and that its concretization operator γ_{Zone} maps them to affine zones of $\mathbb{R}_{\max}^d$.

We can now define the primitive $\mathsf{toZone} : \mathsf{MaxPoly} \to \mathsf{Zone}$ by:

$$\mathsf{toZone}(\mathcal{X}) \stackrel{def}{=} \begin{cases} \bot_{\mathsf{Zone}} & \text{if } \mathcal{X} = \bot, \\ \iota(P,R)/\iota(P,R) & \text{if } \mathcal{X} = ((P,R), \cdot), \end{cases}$$

According to Proposition 7.24, the following proposition holds:

Corollary 7.25. *Let $\mathcal{X} \in \mathsf{MaxPoly}$. Then $\mathsf{toZone}(\mathcal{X})$ is the smallest element M of Zone such that $\gamma(\mathcal{X}) \subset \gamma_{\mathsf{Zone}}(M)$.*

Proposition 7.26. *Let $\mathcal{X} = ((P, R), \cdot) \in \mathsf{MaxPoly}$. The time complexity of the call to $\mathsf{toZone}(\mathcal{X})$ is equal to $O(d^2(|P| + |R|))$.*

Example 7.13. Figure 7.10 provides an illustration of the primitive toZone on an element of MaxPoly.

Remark 7.14. Observe that the zones returned by toZone are necessarily under closed form.

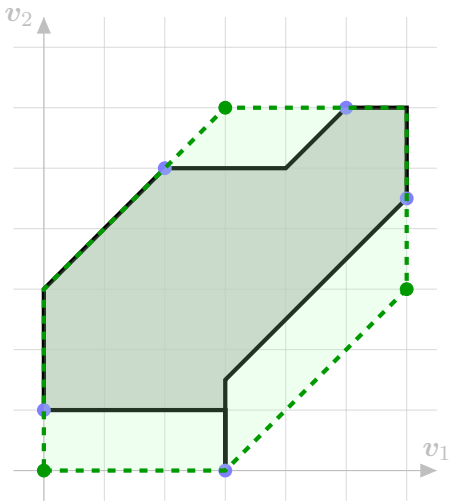


Figure 7.10: Smallest zone containing an element of MaxPoly

7.1.7.b Reduction of abstract primitives with zones. Let us consider a concrete primitive $F : (\mathbb{R}^{\mathsf{Vars}})^p \to \mathbb{R}^{\mathsf{Vars}}$ of arity p ($p \geq 0$). Suppose that $\mathcal{F}$ is a sound abstract counterpart defined on our abstract domain MaxPoly, *i.e.* $\mathcal{F} : \mathsf{MaxPoly}^p \to \mathsf{MaxPoly}$ satisfies: for all $\mathcal{X}_1, \ldots, \mathcal{X}_p \in \mathsf{MaxPoly}$,

$$F(\gamma(\mathcal{X}_1), \ldots, \gamma(\mathcal{X}_p)) \subset \gamma(\mathcal{F}(\mathcal{X}_1, \ldots, \mathcal{X}_p)).$$

Suppose that the abstract domain of zones is provided with a sound abstraction $\mathcal{F}_{\mathsf{Zone}} : \mathsf{Zone}^p \to \mathsf{Zone}$ of F. Then the precision of the function $\mathcal{F}$ can be refined thanks to the function $\mathcal{F}_{\mathsf{Zone}}$. Indeed, let us define $\mathcal{F}' : \mathsf{MaxPoly}^p \to \mathsf{MaxPoly}$ by:

$$\mathcal{F}'(\mathcal{X}_1, \ldots, \mathcal{X}_p) \stackrel{def}{=} (\!|\mathcal{F}_{\mathsf{Zone}}(\mathsf{toZone}(\mathcal{X}_1), \ldots, \mathsf{toZone}(\mathcal{X}_p))|\!) \circ \mathcal{F})(\mathcal{X}_1, \ldots, \mathcal{X}_p).$$

Note that given an abstract element $M \in \mathsf{Zone}$, we denote by $(\!|M|\!)$ the abstract operator on MaxPoly which applies on the elements of MaxPoly the conditions given by the affine inequalities representing the zone M: if M is represented by the matrix $\left(\begin{smallmatrix} A & \boldsymbol{b} \\ \boldsymbol{c} & d \end{smallmatrix}\right)$, $(\!|M|\!)$ is defined as

$$(\!|A_{ij}\boldsymbol{v}_j \leq \boldsymbol{v}_i \text{ for all } i,j,\ \boldsymbol{b}_i \leq \boldsymbol{v}_i \text{ and } \boldsymbol{c}_i\boldsymbol{v}_i \leq \mathbb{1} \text{ for all } i, \text{ and } d \leq \mathbb{1}|\!).$$

The following statement holds:

Proposition 7.27. *Let $\mathcal{X}_1, \ldots, \mathcal{X}_p \in \mathsf{MaxPoly}$. Then $\mathcal{F}'$ is a sound abstraction of F, and is more precise than $\mathcal{F}$:*

$$F(\gamma(\mathcal{X}_1), \ldots, \gamma(\mathcal{X}_p)) \subset \gamma(\mathcal{F}'(\mathcal{X}_1, \ldots, \mathcal{X}_p)) \subset \gamma(\mathcal{F}(\mathcal{X}_1, \ldots, \mathcal{X}_p)).$$

Besides, $\mathcal{F}'$ is more precise than $\mathcal{F}_{\mathsf{Zone}}$: if M_i is the smallest zone containing $\gamma(\mathcal{X}_i)$ for all $i \in [p]$, then:

$$\gamma(\mathcal{F}'(\mathcal{X}_1, \ldots, \mathcal{X}_p)) \subset \gamma_{\mathsf{Zone}}(\mathcal{F}_{\mathsf{Zone}}(M_1, \ldots, M_p)).$$

Proof. It is a straightforward consequence of the soundness and exactness of the operator $(\!|cond|\!)$ where *cond* is a system of tropically affine inequalities (Proposition 7.8), and of Corollary 7.25. □

As a consequence, all abstract primitives defined on MaxPoly can be systematically refined so as to be more precise than their analogues on the abstract domain Zone. Note however that this refinement is useless for abstract primitives which are already the best possible abstraction (in particular, those which are exact).

7.1.8 Non-tropically affine abstract primitives

Tropically affine conditions and assignments are handled by abstract primitives. This includes conditions of the form $\boldsymbol{v}_i \diamond \alpha$ or $\boldsymbol{v}_i + \beta \diamond \boldsymbol{v}_j$ where $\diamond \in \{ \leq, =, \geq \}$, and assignments $\boldsymbol{v}_i \leftarrow \alpha$ or $\boldsymbol{v}_j \leftarrow \boldsymbol{v}_i + \beta$ $(\alpha, \beta \in \mathbb{R})$.

However, other forms of conditions and assignments, such as classical affine ones (and which are not tropically affine), are not handled in general. Some techniques have been developed in [AGG08] to tropically linearize assignments of the form $\boldsymbol{v}_k \leftarrow \boldsymbol{v}_i + \boldsymbol{v}_j$ or $\boldsymbol{v}_j \leftarrow \alpha \times \boldsymbol{v}_i$.

A systematic way to handle general conditions and assignments is to treat them as non-deterministic, *i.e.* conditions are safely ignored (which is sound), and assignments are replaced by non-deterministic ones. The resulting abstract primitives can be then refined using the corresponding primitives on zones, and the method of Section 7.1.7. This ensures that the abstract primitive defined on MaxPoly is at least more precise than its counterpart defined on Zone (Proposition 7.27).

7.1.9 Summary of abstract primitives behavior

Albeit we did not mention it for the moment, it can be verified that the abstract primitives of union, intersection, condition, assignment, and reduction to zones, are all monotone. Therefore, the abstract domain MaxPoly meets the requirements of the analysis developped in Section 6.5.

Figure 7.11 recapitulates which components of the abstract double descriptions are used by the abstract primitives that we have previously defined.

7.2 Inferring min-invariants: the abstract domain MinPoly

In this section, we explain how tropical polyhedra can be used to infer min-invariants over the set of variables $\mathsf{Vars} = \{ \boldsymbol{v}_1, \ldots, \boldsymbol{v}_d \}$. Let Vars^- be a disjoint set of variables $\{ \boldsymbol{w}_1, \ldots, \boldsymbol{w}_d \}$. We define $\mathsf{MinPoly}(\mathsf{Vars})$ as the abstract domain whose elements belong to $\mathsf{MaxPoly}(\mathsf{Vars}^-)$, and the concretization operator is defined by:

$$\gamma_{\mathsf{MinPoly}}(\mathcal{X}) \stackrel{def}{=} \{ \nu \in \mathbb{R}^{\mathsf{Vars}} \mid \nu^- \in \gamma_{\mathsf{MaxPoly}}(\mathcal{X}), \nu^-(\boldsymbol{w}_i) = -\nu(\boldsymbol{v}_i) \text{ for all } i \}.$$

Intuitively, each $\boldsymbol{w}_i$ represent the opposite of $\boldsymbol{v}_i$, and elements of MaxPoly are used to abstract max-invariants over the $-\boldsymbol{v}_i$.

Naturally, zone invariants over the variables Vars^- are still zone invariants over Vars, so that the abstract elements of MinPoly express disjunctions of zone invariants, like the elements of MaxPoly (see Section 7.1.1.c).

abtract primitive	first operand	second operand	result
$\sqsubseteq$	generators	generators or constraints	NA
$\top$	NA	NA	generators and constraints
$\bot$	NA	NA	NA
$\sqcup$	generators	generators	generators
$\sqcap$	constraints	constraints	constaints
tropical conditions	constraints	NA	constraints
	generators	NA	generators
tropical assignments	generators	NA	generators
non-deterministic assignments	generators	NA	generators
∇	constraints	generators	constraints
∇_{gen}	generators	generators	generators
toZone	generators	NA	constraints

Figure 7.11: Summary of the kinds of components involved in abstract primitives

Observe that the operator $\gamma_{\mathsf{MinPoly}}$ is obtained as the composition of $\gamma_{\mathsf{MaxPoly}}$ with the one-to-one correspondence $\mathbb{R}^{\mathsf{Vars}^-} \to \mathbb{R}^{\mathsf{Vars}}$ which maps $\nu \in \mathbb{R}^{\mathsf{Vars}^-}$ to ν' defined by $\nu'(\boldsymbol{v}_i) = -\nu(\boldsymbol{w}_i)$ for all $i \in [d]$. This allows to easily reuse all the abstract primitives defined on MaxPoly. If necessary, we distinguish the abstract primitives of MinPoly from those of MaxPoly by adding the domain in subscript (for instance $(\!|\cdot \leftarrow \cdot|\!)_{\mathsf{MaxPoly}}$).

7.2.1 Order-theoretic abstract primitives

It can be shown that the primitives $\bot$, $\top$, $\sqsubseteq$, $\sqcup$, $\sqcap$, ∇, and ∇_{gen} on MaxPoly can be used as such on MinPoly, and that they preserve their properties:

Proposition 7.28. *The abstract primitives $\bot$, $\top$, $\sqsubseteq$, $\sqcup$, and $\sqcap$ are all sound. Besides, $\sqsubseteq$ and $\sqcap$ are exact, and $\sqcup$ is the best possible abstraction of the union. Finally, ∇ and ∇_{gen} are widening operators.*

7.2.2 Conditions and assignments

Recall that the min-plus semiring refers to set $\mathbb{R}_{\min} = \mathbb{R} \cup \{+\infty\}$, endowed with the laws $\oplus'$ and $\otimes'$ given by $x \oplus' y \stackrel{def}{=} \min(x, y)$ and $x \otimes' y \stackrel{def}{=} x + y$.

Tropically affine conditions and assignments now have to be interpreted in the min-plus semiring. They are respectively of the form:

$$A \otimes' (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d) \oplus' \boldsymbol{c} \leq B \otimes' (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d) \oplus' \boldsymbol{d} \qquad \text{with } A, B \in \mathbb{R}_{\min}^{p \times d}, \boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}_{\min}^{p}$$

$$\boldsymbol{v}_k \leftarrow \lambda_0 \oplus' \bigoplus_{l=1}^{d}{}' \lambda_l \boldsymbol{v}_l \qquad \text{with } \lambda_0, \ldots, \lambda_d \in \mathbb{R}_{\min}$$

Given $x \in \mathbb{R}_{\min}$, we denote by $-x$ the opposite element in $\mathbb{R}_{\max}$, with the convention $-(+\infty) = -\infty$. This notation can be also used for the reciprocal association, and is extended to matrices. Then we define:

$$\begin{aligned}(\!|A \otimes' (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d) \oplus' \boldsymbol{c} \leq B \otimes' (\boldsymbol{v}_1), \ldots, \boldsymbol{v}_d)) \oplus' \boldsymbol{d}|\!)_{\mathsf{MinPoly}} \\ \stackrel{def}{=} (\!|(-A) \otimes (\boldsymbol{w}_1), \ldots, \boldsymbol{w}_d) \oplus (-\boldsymbol{c}) \leq (-B) \otimes (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_d) \oplus (-\boldsymbol{d})|\!)_{\mathsf{MaxPoly}}\end{aligned}$$

and

$$(\!|\boldsymbol{v}_k \leftarrow \lambda_0 \oplus' \bigoplus_{l=1}^{d}{}' \lambda_l \boldsymbol{v}_l|\!)_{\mathsf{MinPoly}} \stackrel{def}{=} (\!|\boldsymbol{w}_k \leftarrow (-\lambda_0) \oplus \bigoplus_{l=1}^{d} (-\lambda_l)\boldsymbol{w}_l|\!)_{\mathsf{MaxPoly}}.$$

Non-deterministic assignments still have the same form, and the corresponding abstract primitive is defined by:

$$(\!|\boldsymbol{v}_k \leftarrow ?\,|\!)_{\mathsf{MinPoly}} \stackrel{def}{=} (\!|\boldsymbol{w}_k \leftarrow ?\,|\!)_{\mathsf{MaxPoly}}.$$

The abstract primitive of this class of conditions and assignments are both sound and exact:

Proposition 7.29. *Let $\mathcal{X} \in \mathsf{MinPoly}$. Then we have:*

$$\begin{aligned}\{\, \boldsymbol{v} \in \gamma_{\mathsf{MinPoly}}(\mathcal{X}) \mid A \otimes' (\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) \oplus' \boldsymbol{c} \leq B \otimes' (\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d)) \oplus' \boldsymbol{d}\,\} \\ = \gamma((\!|A \otimes' (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d) \oplus' \boldsymbol{c} \leq B \otimes' (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d) \oplus' \boldsymbol{d}|\!)_{\mathsf{MinPoly}}(\mathcal{X}))\end{aligned}$$

and

$$\left\{ \nu[\boldsymbol{v}_k \mapsto \lambda_0 \oplus' \bigoplus_{l=1}^{d}{}' \lambda_l \nu(\boldsymbol{v}_l)] \mid \nu \in \gamma_{\mathsf{MinPoly}}(\mathcal{X}) \right\} = \gamma_{\mathsf{MinPoly}}((\!|\boldsymbol{v}_k \leftarrow \lambda_0 \oplus' \bigoplus_{l=1}^{d}{}' \lambda_l \boldsymbol{v}_l|\!)_{\mathsf{MinPoly}}(\mathcal{X}))$$

$$\{\, \nu[\boldsymbol{v}_k \mapsto x] \mid \nu \in \gamma_{\mathsf{MinPoly}}(\mathcal{X}), x \in \mathbb{R}\,\} = \gamma((\!|\boldsymbol{v}_k \leftarrow ?\,|\!)(\mathcal{X})).$$

Proof. This is a direct consequence of Propositions 7.8 and 7.9. □

7.2.3 Reduction with zones

Observe that the image of an affine zone $\mathsf{affzone}(A, \boldsymbol{b}, \boldsymbol{c})$ by the map $\boldsymbol{x} \mapsto -\boldsymbol{x}$ is precisely the affine zone $\mathsf{affzone}({}^tA, \boldsymbol{c}, \boldsymbol{b})$. It follows that the reduction with zones on MinPoly can be defined as follows: for all $\mathcal{X} \in \mathsf{MinPoly}$,

$$\mathsf{toZone}_{\mathsf{MinPoly}}(\mathcal{X}) \stackrel{def}{=} \begin{cases} \bot_{\mathsf{Zone}} & \text{if } \mathcal{X} = \bot, \\ {}^t\mathsf{toZone}_{\mathsf{MaxPoly}}(\mathcal{X}) & \text{if } \mathcal{X} \neq \bot. \end{cases}$$

Proposition 7.30. *Let $\mathcal{X} \in \mathsf{MinPoly}$. Then $\mathsf{toZone}_{\mathsf{MinPoly}}(\mathcal{X})$ is the smallest element M of* Zone *such that $\gamma_{\mathsf{MinPoly}}(\mathcal{X}) \subset \gamma_{\mathsf{MinPoly}}(M)$.*

Any abstract primitive on MinPoly can be then refined using its counterpart on Zone. This refinement is analogous to the method discussed in Section 7.1.7.

7.3 Inferring min- and max-invariants: the domain MinMaxPoly

We now discuss how to infer both min- and max-invariants on a set of variables $\mathsf{Vars} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d\}$.

Let us define $\mathsf{Vars}^{\pm} = \mathsf{Vars} \cup \mathsf{Vars}^{-}$. Then the abstract domain $\mathsf{MinMaxPoly}(\mathsf{Vars})$ is formed by elements of $\mathsf{MaxPoly}(\mathsf{Vars}^{\pm})$. The concretization operator $\gamma_{\mathsf{MinMaxPoly}}$ is defined as follows:

$$\gamma_{\mathsf{MinMaxPoly}}(\mathcal{X}) \stackrel{def}{=} \left\{ \nu \in \mathbb{R}^{\mathsf{Vars}} \;\middle|\; \nu^{\pm} \in \gamma_{\mathsf{MaxPoly}}(\mathcal{X}),\ \text{and for all } i, \left\{ \begin{array}{l} \nu^{\pm}(\boldsymbol{v}_i) = \nu(\boldsymbol{v}_i) \\ \nu^{\pm}(\boldsymbol{w}_i) = -\nu(\boldsymbol{v}_i) \end{array} \right\} \right.,$$

so that each element of $\mathsf{MaxPoly}(\mathsf{Vars}^{\pm})$ abstract max-properties on the variable $\boldsymbol{v}_i$ and their opposite. In particular they are able to infer both min- and max-invariants over Vars at the same time. Similar, they can express disjunctions of zone invariants over $\mathsf{Vars}^{\pm}$, *i.e.* octagonal invariants over Vars.

7.3.1 Order-theoretic abstract primitives

Like in Section 7.2, the abstract primitives $\bot$, $\top$, $\sqsubseteq$, $\sqcup$, $\sqcap$, ∇, and ∇_{gen} on MaxPoly can be used as on MinMaxPoly.

Proposition 7.31. *The abstract primitives $\bot$, $\top$, $\sqsubseteq$, $\sqcup$, and $\sqcap$ are all sound. Besides, $\sqcap$ is exact. Finally, ∇ and ∇_{gen} are widening operators.*

However, $\sqsubseteq$ is not exact anymore, and $\sqcup$ is not the best possible abstraction of the union. This loss of precision can be intuitively explained by the fact that the primitives do not perform any "communication" between the variables $\boldsymbol{v}_i$ and $\boldsymbol{w}_i$, while in the concrete world, the $\boldsymbol{w}_i$ are necessarily equal to the opposite of the $\boldsymbol{v}_i$.

Proof. We only detail the proof of the exactness of $\sqcap$. Let $\mathcal{X}, \mathcal{X}' \in \mathsf{MinMaxPoly}$. Consider $\nu \in \gamma_{\mathsf{MinMaxPoly}}(\mathcal{X} \sqcap \mathcal{X}')$, and let $\nu^{\pm} \in \gamma_{\mathsf{MaxPoly}}(\mathcal{X} \sqcap \mathcal{X}')$ be the associated element of $\mathbb{R}^{\mathsf{Vars}^{\pm}}$. Then $\nu^{\pm} \in \gamma_{\mathsf{MaxPoly}}(\mathcal{X})$ using Proposition 7.6, so that clearly $\nu \in \gamma_{\mathsf{MinMaxPoly}}(\mathcal{X})$. Similarly, it can be shown that $\nu \in \gamma_{\mathsf{MinMaxPoly}}(\mathcal{X}')$, which completes the proof. □

7.3.2 Conditions and assignments

Using the corresponding abstract primitives of MaxPoly, the abstract domain MinMaxPoly is able to handle conditions and assignment of the form:

$$\begin{array}{ll} A(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d, -\boldsymbol{v}_1, \ldots, -\boldsymbol{v}_d) \oplus \boldsymbol{c} & \\ \qquad \leq B(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d, -\boldsymbol{v}_1, \ldots, -\boldsymbol{v}_d) \oplus \boldsymbol{d} & \text{with } A, B \in \mathbb{R}_{\max}^{p \times 2d}, \boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}_{\max}^{p} \\ \pm \boldsymbol{v}_k \leftarrow \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l \boldsymbol{v}_l \oplus \bigoplus_{l=1}^{d} \lambda_{l+d}(-\boldsymbol{v}_l) & \text{with } \lambda_0, \ldots, \lambda_{2d} \in \mathbb{R}_{\max} \end{array}$$

Observe that this includes the class of conditions and assignments handled by both abstract domains MaxPoly and MinPoly. The abstract primitives on MinMaxPoly are defined by:

$$\begin{array}{l} (\!| A(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d, -\boldsymbol{v}_1, \ldots, -\boldsymbol{v}_d) \oplus \boldsymbol{c} \leq B(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d, -\boldsymbol{v}_1, \ldots, -\boldsymbol{v}_d) \oplus \boldsymbol{d} |\!)_{\mathsf{MinMaxPoly}} \\ \qquad \stackrel{def}{=} (\!| A(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d, \boldsymbol{w}_1, \ldots, \boldsymbol{w}_d) \oplus \boldsymbol{c} \leq B(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d, \boldsymbol{w}_1, \ldots, \boldsymbol{w}_d) \oplus \boldsymbol{d} |\!)_{\mathsf{MaxPoly}} \end{array}$$

and

$$\begin{aligned}
&(\!|\boldsymbol{v}_k \leftarrow \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l \boldsymbol{v}_l \oplus \bigoplus_{l=1}^{d} \lambda_{l+d}(-\boldsymbol{v}_l)|\!)_{\mathsf{MinMaxPoly}} \\
&\qquad \stackrel{def}{=} (\!|\boldsymbol{v}_k \leftarrow \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l \boldsymbol{v}_l \oplus \bigoplus_{l=1}^{d} \lambda_{l+d} \boldsymbol{w}_l \text{ and } \boldsymbol{w}_k \leftarrow ?\,|\!)_{\mathsf{MaxPoly}}, \\
&(\!|-\boldsymbol{v}_k \leftarrow \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l \boldsymbol{v}_l \oplus \bigoplus_{l=1}^{d} \lambda_{l+d}(-\boldsymbol{v}_l)|\!)_{\mathsf{MinMaxPoly}} \\
&\qquad \stackrel{def}{=} (\!|\boldsymbol{w}_k \leftarrow \lambda_0 \oplus \bigoplus_{l=1}^{d} \lambda_l \boldsymbol{v}_l \oplus \bigoplus_{l=1}^{d} \lambda_{l+d} \boldsymbol{w}_l \text{ and } \boldsymbol{v}_k \leftarrow ?\,|\!)_{\mathsf{MaxPoly}}.
\end{aligned}$$

Remark 7.15. Note that the variable $\mp\boldsymbol{v}_k$ opposed to $\pm\boldsymbol{v}_k$ is necessarily assigned to a non-deterministic value. Indeed, the opposite assignement cannot be expressed as a tropical assignment in general.

However, this operation can be omitted when the assignment is of the form $\pm\boldsymbol{v}_k \leftarrow \lambda_0$ or $\pm\boldsymbol{v}_k \leftarrow \lambda_j(\pm\boldsymbol{v}_j)$, since in that case, the opposite assignment is given by $\mp\boldsymbol{v}_k \leftarrow -\lambda_0$ and $\mp\boldsymbol{v}_k \leftarrow (-\lambda_j)(\mp\boldsymbol{v}_j)$ respectively.

Non-deterministic assignments are also handled, but both $\boldsymbol{v}_k$ and its opposite $\boldsymbol{w}_k$ have to be updated:

$$(\!|\boldsymbol{v}_k \leftarrow ?\,|\!)_{\mathsf{MinMaxPoly}} \stackrel{def}{=} (\!|\boldsymbol{w}_k \leftarrow ?\,|\!)_{\mathsf{MaxPoly}} \circ (\!|\boldsymbol{v}_k \leftarrow ?\,|\!)_{\mathsf{MaxPoly}}$$

Proposition 7.32. *The condition and assignment primitives defined above are sound. Condition primitives are exact.*

Proof. The proof of the exactness of the condition primitives is similar to the proof of the exactness of $\sqcap$ on MinMaxPoly (Proposition 7.31). □

7.3.3 Reduction with octagons

As discussed in Section 6.3.2.b, octagonal invariants are given by inequalities of the form $\pm\boldsymbol{v}_i \pm \boldsymbol{v}_j \geq \alpha$, and can be seen as zone invariants over the set of variables $\mathsf{Vars}^{\pm}$. Such invariants are precisely expressed by the abstract domain Oct of octagons. Its elements can be encoded by elements of the abstract domain $\mathsf{Zone}(\mathsf{Vars}^{\pm})$, and its concretization operator can be defined as:

$$\gamma_{\mathsf{Oct}}(M) \stackrel{def}{=} \{\, \nu \in \mathbb{R}^{\mathsf{Vars}} \mid (\nu(\boldsymbol{v}_1), \ldots, \nu(\boldsymbol{v}_d), -\nu(\boldsymbol{v}_1), \ldots, -\nu(\boldsymbol{v}_d)) \in \gamma_{\mathsf{Zone}}(M) \,\}.$$

The abstract domain Oct is provided with a reduction operator $\eta_{\mathsf{Oct}} : \mathsf{Oct} \to \mathsf{Oct}$ which, in particular, makes communicate information between the variables $\boldsymbol{v}_i$ and their opposite (here $\boldsymbol{w}_i$ following our convention). Formally, given $M \in \mathsf{Oct}$, $\eta_{\mathsf{Oct}}(M)$ is the smallest element of Oct which contains $\gamma_{\mathsf{Oct}}(M)$. This reduction operator can be used to allow the same information sharing in elements of MinMaxPoly.

More precisely, we define the primitive $\mathsf{toOct} : \mathsf{MinMaxPoly} \to \mathsf{Oct}$ which extracts an abstract element of Oct from an element of MinMaxPoly:

$$\mathsf{toOct} \stackrel{def}{=} \eta_{\mathsf{Oct}} \circ \mathsf{toZone}$$

This primitive can be shown to be sound:

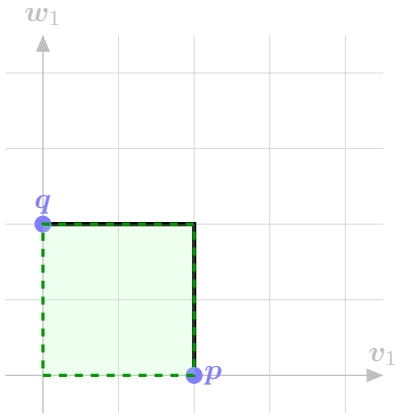
Proposition 7.33. *Let $\mathcal{X} \in \mathsf{MinMaxPoly}$. Then we have:*

$$\gamma_{\mathsf{MinMaxPoly}}(\mathcal{X}) \subset \gamma_{\mathsf{Oct}}(\mathsf{toOct}(\mathcal{X})).$$

Proof. Let $M = \mathsf{toZone}_{\mathsf{MaxPoly}}(\mathcal{X})$. According to Corollary 7.25, $\gamma_{\mathsf{MaxPoly}}(\mathcal{X}) \subset \gamma_{\mathsf{Zone}}(M)$. Now, let us consider $\nu \in \gamma_{\mathsf{MinMaxPoly}}(\mathcal{X})$, and let $\nu^{\pm} \in \gamma_{\mathsf{MaxPoly}}(\mathcal{X})$ be associated to ν. Then $\nu^{\pm} \in \gamma_{\mathsf{Zone}}(M)$, so that $\nu \in \gamma_{\mathsf{Oct}}(M)$. Since $\gamma_{\mathsf{Oct}}(M) = \gamma_{\mathsf{Oct}}(\eta_{\mathsf{Oct}}(M))$, we have $\nu \in \gamma_{\mathsf{Oct}}(\mathsf{toOct}(\mathcal{X}))$. □

Nevertheless, unlike its analogue toZone on MaxPoly or MinPoly, the primitive toOct does not necessarily yield the smallest octagon which contains $\gamma_{\mathsf{MinMaxPoly}}(\mathcal{X})$, as shown in the following example.

Example 7.16. Consider the abstract element $\mathcal{X}$ of MinMaxPoly over the variable $\boldsymbol{v}_1$, which can also be seen as an element of $\mathsf{MaxPoly}(\{\boldsymbol{v}_1, \boldsymbol{w}_1\})$, represented by the tropical polyhedron depicted in black:



Its generator component is formed by the vertices $\boldsymbol{p} = (2, 0)$ and $\boldsymbol{q} = (0, 2)$. Seen as an element of MinMaxPoly, its concretization $\gamma_{\mathsf{MinMaxPoly}}(\mathcal{X})$ is empty. Indeed, for any $\nu^{\pm} \in \gamma_{\mathsf{MaxPoly}}(\mathcal{X})$, $\nu^{\pm}(\boldsymbol{v}_1)$ and $\nu^{\pm}(\boldsymbol{w}_1)$ are never opposed.
It can be verified that the zone $\mathsf{toZone}(\mathcal{X})$ over the variables $\boldsymbol{v}_1$ and $\boldsymbol{w}_1$ corresponds to the square depicted in green. It represents the invariants $0 \leq \boldsymbol{v}_1 \leq 2$ and $0 \leq \boldsymbol{w}_1 \leq 2$, and is encoded by the matrix:

$$\begin{pmatrix} 0 & -2 & 0 \\ -2 & 0 & 0 \\ -2 & -2 & 0 \end{pmatrix}.$$

This matrix also represents the element of Oct characterized by the invariants $0 \leq \boldsymbol{v}_1 \leq 2$ and $0 \leq -\boldsymbol{v}_1 \leq 2$. It is reduced by the operator η_{Oct} to the octagon representing the invariant $\boldsymbol{v}_1 = 0$. As a non-empty octagon, it is not the smallest element of Oct containing $\gamma_{\mathsf{MinMaxPoly}}(\mathcal{X}) = \emptyset$.

Even if the lack of precision of the abstract primitive toOct is a little disappointing, it should be stressed that the abstract elements of MinMaxPoly precisely interact with octagons. Indeed, thanks to the exactness of the intersection with octagonal invariants (Proposition 7.32), for all $\mathcal{X} \in \mathsf{MinMaxPoly}$ and $M \in \mathsf{Oct}$, we have:

$$\gamma_{\mathsf{MinMaxPoly}}(\llparenthesis M \rrparenthesis(\mathcal{X})) \subset \gamma_{\mathsf{Oct}}(M) \cap \gamma_{\mathsf{MinMaxPoly}}(\mathcal{X}),$$

where $\llparenthesis M \rrparenthesis$ is the intersection abstract primitive of MinMaxPoly which applies the conditions

represented by the octagon M. The domain MinMaxPoly can be therefore successfully reduced with the abstract domain of octagons (for instance using reduced product [CC79]).

Following the principle of the refinement on abstract primitives developed in Section 7.1.7, any abstract primitive $\mathcal{F} : \mathsf{MinMaxPoly}^p \times \mathsf{MinMaxPoly}$ can be refined into $\mathcal{F}' : \mathsf{MinMaxPoly}^p \times \mathsf{MinMaxPoly}$, defined as:

$$\mathcal{F}'(\mathcal{X}_1, \ldots, \mathcal{X}_p) \stackrel{def}{=} (\!|\mathcal{F}_{\mathsf{Oct}}(\mathsf{toOct}(\mathcal{X}_1), \ldots, \mathsf{toOct}(\mathcal{X}_p))|\!)_{\mathsf{MinMaxPoly}} \circ \mathcal{F})(\mathcal{X}_1, \ldots, \mathcal{X}_p).$$

This technique can be used to improve the precision on primitives such as the abstract union or assignments, because it allows to gain precision on the $\pm\boldsymbol{v}_i$ using the information on their opposite. Naturally, since the abstract primitive toOct is not as precise as possible, the primitive $\mathcal{F}'$ is not ensured to be more precise than $\mathcal{F}_{\mathsf{Oct}}$.

7.4 Experiments

The three numerical abstract domains MaxPoly, MinPoly, and MinMaxPoly have been implemented in the library TPLib [All09], atop the module providing the algorithms ComputeExtRays and ComputeExtRaysPolar on tropical polyhedra. The implementation of the domains with the abstract semantics presented in Section 6.5 is not publicly available.

In this section, we discuss some experiments on programs of various kinds. We first present the principles of the implementation of the static analysis tool (Section 7.4.1). We then focus on the analysis of memory manipulation programs (Section 7.4.2), and show that our tropical polyhedra based domains can also be useful for array predicate abstractions (Section 7.4.3). Sections 7.4.4 and 7.4.5 are devoted to scalability benchmarks, showing that our domains are able to represent highly disjunctive invariants. Section 7.4.6 finally provides general remarks on the performance of the analyzer.

At the end of the section, Table 7.1 summarizes the data relative to the analysis experiments on the programs discussed in this section. For each program, the number of lines and of variables are provided. The execution time of the analyzer using the standard widening ∇ and the widening on the generator component ∇_{gen} are also given. It is compared to our first implementation of the analyzer presented in [AGG08]. Finally, the number of generators in the final abstract element is provided.

7.4.1 Principles of the implementation

The static analysis tool is fully parametric, since it can use one of the three numerical abstract domains, and one of the two widening operators ∇ or ∇_{gen}, depending on the options passed by the user.

As mentioned in Section 7.1.1, the analyzer does not manipulate systematically double descriptions of tropical polyhedra. Generator or constraint components are computed lazily from the other. The tool returns the min-/max-invariants inferred by the final abstract element. Thus, if necessary, the constraint component of this element is computed from its generator component.

An interesting feature is that the tool manipulates only integers, and no floating-point numbers. Indeed, the kernel language defined in Chapter 6 manipulate exclusively integers. Therefore, integer-based representations by constraints or by generators are preserved by all abstract primitives. In practice, we use arbitrary precision integers provided by the library GMP [GMP].

7.4.2 Analysis of memory manipulating programs

7.4.2.a The function `memcpy`. This section deals with experiments on memory manipulating programs. We first begin with the analysis of the function `memcpy`. In our kernel language, it can be written as follows:

```
1 :  assume ((p ≥ 1) ∧ (q ≥ 1));
2 :  src := malloc(p);
3 :  dst := malloc(q);
4 :  assume ((n ≤ p) ∧ (n ≤ q));
5 :  i := 0;
6 :  while i ≤ n − 1 do
7 :     dst[i] := src[i];
8 :     i := i + 1;
9 :  done;
10 :
```

The function itself is implemented from Lines 5 to 9. The part of the program between Lines 1 and 4 allows to create a general memory context, in which *src* and *dst* are initialized to arrays with an arbitrary content, and of size p and q respectively. The condition $n \leq p$ and $n \leq q$ allows to avoid that the call to `memcpy` leads to a heap overflow.

This example is analyzed using the abstract domain MinPoly. Then the tool returns the following invariant:

$$\begin{cases} 1 \leq sz_{src} = p,\ 1 \leq sz_{dst} = q \\ i = n,\ n \leq p,\ n \leq q \\ 0 \leq len_{src} \leq sz_{src},\ 0 \leq len_{dst} \leq sz_{dst}, \\ \min(len_{src}, n) = \min(len_{dst}, n) \end{cases} \tag{7.7}$$

In particular, the tool has successfully inferred the invariant $\min(len_{src}, n) = \min(len_{dst}, n)$, which exactly encodes the disjunction of the cases (i) and (ii) presented in Chapter 1.

7.4.2.b The function `strncpy`. The function `strncpy` is another well-known string manipulating function in C. It takes as input three arguments, a destination array *dst*, a source array *src*, and a integer parameter n. Paraphrasing its manual page:

> *The* `strncpy` *function copies not more than n characters (characters that follow a null character are not copied) from the array src to the array dst.*
>
> ...
>
> *If the array src stores a string that is shorter than n characters, null characters are appended to the copy in the array dst, until n characters in all are written.*

In our kernel language, it can be implemented as follows:

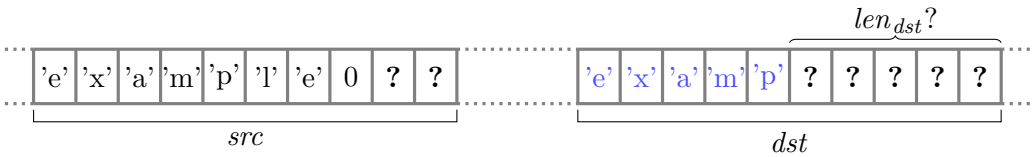
```
1 :  assume ((p ≥ 1) ∧ (q ≥ 1));
2 :  src := malloc(p);
3 :  dst := malloc(q);
4 :  assume ((n ≤ p) ∧ (n ≤ q));
5 :  i := 0;
6 :  while (i ≤ n − 1) ∧ (src[i] ≠ 0) do
7 :     dst[i] := src[i];
8 :     i := i + 1;
9 :  done;
10 : while i ≤ n − 1 do
11 :    dst[i] := 0;
12 :    i := i + 1;
13 : done;
14 :
```

As for the function `memcpy`, the final invariant between len_{dst}, len_{src}, and n is a disjunction of two cases:

- either $n > len_{src}$, in which case the null terminal character of src is copied into dst. Some other null characters may be copied after, but in any case, $len_{dst} = len_{src}$.
  On the string "example" and $n = 9$, we obtain:

  Observe that the null characters represented in red have been inserted by the second loop (Lines 10 to 13).

- or n is smaller than the source length len_{src}, in which case, $len_{dst} \geq n$. On the same example, but with $n = 5$, we have:



As a consequence, the final invariant is identical to (7.7). It is also successfully inferred by our tool. Note however that the intermediate invariants computed in the analysis of `strncpy` and `memcpy` are naturally not the same.

7.4.2.c Application to our running example. We can now experiment our static analyzer on the example given in the conclusion of Chapter 6. We use the instantiation by the numerical domain MinPoly. The abstract collecting semantics is represented in Figure 7.12. Note that we do not include the first part of the abstract semantics (*i.e.* from control points 1 to 9), since the corresponding invariants are precisely the same as those inferred by convex polyhedra, and given in Figure 6.11.[9]

[9]These invariants are indeed zone invariants, both contained in the abstract domains MinPoly and of convex polyhedra. Note that in Figure 6.11, we should append to each state of the abstract collecting semantics, the

Contrary to its instantiation by convex polyhedra, our static analyzer is now able to precisely compute the loop invariant of control point 11 (in particular, the min-invariant $\min(len_s, i) = \min(len_{upp}, i)$). This allows to show that in control point 15, len_{upp} is equal to len_s. Since we know that the latter is strictly less than $n = sz_s = sz_{upp}$, this ensures that $len_{upp} \leq sz_s - 1$. In particular, the loop from Lines 16 to 20 stops strictly before reaching the end of the array *upp*. Thus, the analyzer proves that the program does not cause any heap overflow.

Also note that our static analyzer is now able to prove that at control point 24, the length result of the upper case character string *upp* is precisely equal to the length of the original string s, as expected.

7.4.3 Application to array predicate abstractions

In this section, we illustrate that our numerical abstract domains can also be useful for other kinds of memory analyses, such as array predicate abstractions [FQ02, Cou03, GRS05, JM07, BHMR07, GMT08, All08, HP08]. The latter aim at automatically determining properties over consecutive array elements. The following program is a typical target of such abstractions:

```
1 :  assume (n ≥ 1);
2 :  t := malloc(n);
3 :  assume (p ≤ n) ∧ (q ≤ n);
4 :  i := p;
5 :  while i ≤ q − 1 do
6 :     t[i] := c;
7 :     i := i + 1;
8 :  done;
9 :
```

It is an incrementing initialization program which fills an array t with a value c, from the indexes p to $q - 1$. Array predicate abstractions are parameterized by a numerical abstract domains. For instance, for the loop invariant at Line 5, they are able to determine the property $\big[\, \boldsymbol{c}(t, u, v),\ u = p,\ v = i - 1 \,\big]$, which means that the array t contains the value c between the indexes $u = p$ and $v = i - 1$. However, the final invariant over p, q, and i at Line 9 relies on a disjunction of the two following cases:

(i) either the loop is entered at least once, in which case the final value of i is equal to q, and $p \leq q - 1$,

(ii) or $p \geq q$, so that the loop is not executed, and $i = p$.

Our analyzer parameterized with the domain MaxPoly is able to successfully infer the exact encoding $i = \max(p, q)$ of this disjunction. Using classical convex polyhedra, we would obtain only the inequalities $i \geq p$ and $i \geq q$, which would be a major loss of precision.

Note that our string analysis described in Section 6.5 is not able to infer a precise invariant on the array t, since the latter is here manipulated as an integer data storage rather than as a string. Integrating our numerical abstract domains in an array predicate abstraction is an interesting objective for future works.[10]

invariants $len_{upp} = sz_{upp} = 0$ corresponding to the fact that the array *upp* is not allocated before control point 10.

[10] Also observe the array predicate abstractions described in [GRS05, HP08] rely on a strong disjunction of array properties. They are consequently able to infer the disjunction of the two cases (i) and (ii), but they

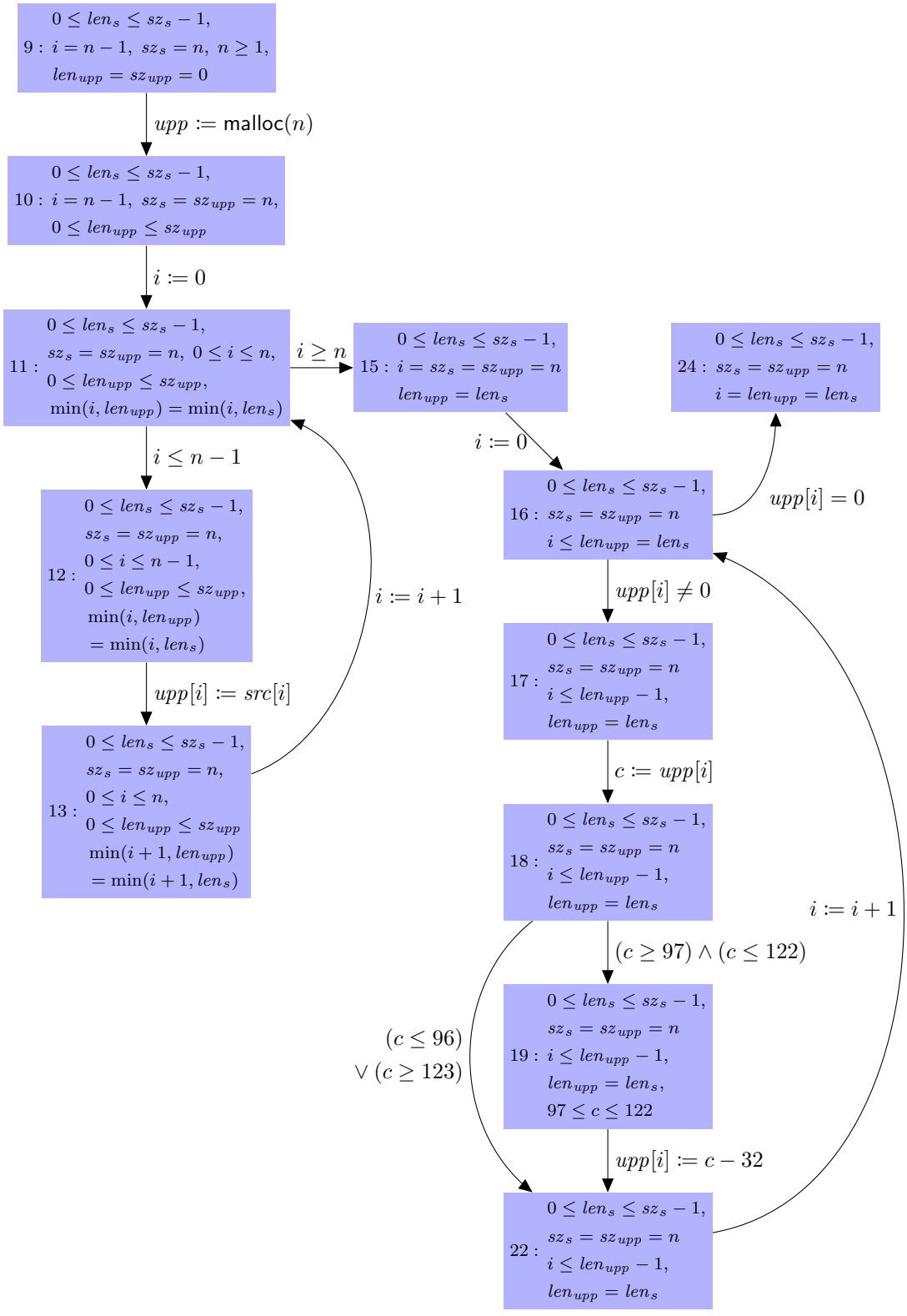


Figure 7.12: The abstract collecting semantics of the second part of our running example

7.4.4 Efficiently handling many disjunctions

The array manipulating program discussed in Section 7.4.3 has inspired us to create a set of benchmarks to test the scalability of our numerical abstract domains, and compare it to the other existing disjunctive techniques. Consider the following program whose length is parameterized by an integer $n \geq 1$:

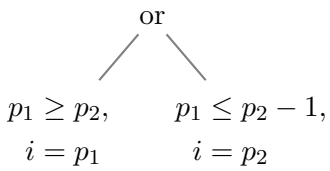
```
1 :   i := p_1;
2 :   while i ≤ p_2 − 1 do
3 :     i := i + 1;
4 :   done;
5 :
6 :   while i ≤ p_3 − 1 do
7 :     i := i + 1;
8 :   done;
9 :
      ⋮
4n − 6 :  while i ≤ p_n − 1 do
4n − 5 :    i := i + 1;
4n − 4 :  done;
4n − 3 :
```

It consists of $(n-1)$ incrementing loops in the same vein as those involved in the program of Section 7.4.3. Our analyzer, equipped with the domain MaxPoly, is able to infer the final max-invariant

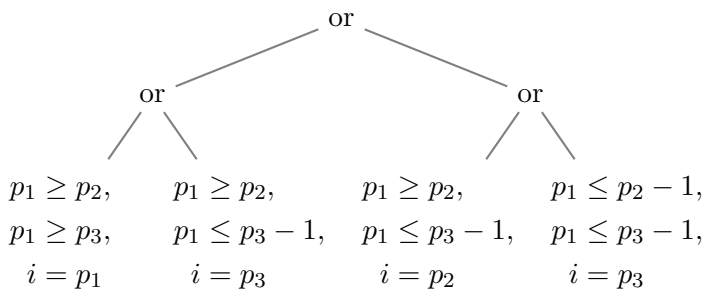
$$i = \max(p_1, \ldots, p_n).$$

The benchmarks are given in Table 7.1, under the name of *incrementing-n*, for several values of n, up to 60. A very interesting feature is that the number of generators in the tropical polyhedra involved in the analysis grows linearly, so that the analysis can scale up to large values of n.

As a comparison, in order to precisely analyze such programs, existing analyzers usually rely on trace partitioning techniques (*e.g.* [HT98, MR05b, RM07]). For instance, at Line 5, such techniques infer a disjunction between the memory states arising directly from Line 1, and those generated by at least one loop iteration. As a consequence, they infer the disjunction:



However, the size of the disjunction clearly grows exponentially during the analysis of the following loops. For instance, after Line 9, we should get a disjunction of size 4:



may also suffer from an explosion of the size of abstract states.

In practice, such techniques cannot be used.[11] Indeed, supposing that only one byte is necessary to store the abstract element in each leaf, the element would take in memory 2^{60} bytes, *i.e.* 100 petabytes (or equivalently, 10^5 terabytes), for $n = 60$.

As a consequence, this family of examples shows the ability of tropical polyhedra based domains to intrinsically handle many disjunctions without any explosion of the size of the representations, which was not possible with the existing techniques.

7.4.5 Sort algorithms

This section deals with the analysis of sort algorithms. As illustration, we have chosen to experiment our static analysis tool on the odd-even sort algorithm [Bat68]. The odd-even sort of n elements is performed by a matrix of $O(n^2)$ elementary sorting blocks, each taking as input two arguments x, y, and outputting $\min(x, y)$ and $\max(x, y)$ respectively. The odd-even sort of 8 variables is depicted in Figure 7.13. The code of an elementary block is given beside. Using the instantiation with the domain MinPoly (resp. MaxPoly), the static analyzer is able to infer that at the end of the algorithm, the leftmost (resp. rightmost) element is the minimum (resp. maximum) of the initial variables. Note that the exact invariants on the intermediary elements cannot be inferred, since they involve expressions composing the operator min and max (for instance of the form $\max(\min(\dots), \dots, \min(\dots))$).

Contrary to the experiments discussed in Section 7.4.4, the growth of the size of the representation by generators is here exponential (see the entries oddeven-n in Table 7.1). Despite that, tropical polyhedra are still a very good alternative to the existing disjunctive techniques, which would not be able to scale up to such values of n. For instance, for $n = 10$, the program oddeven-n contains 45 elementary sorting blocks. At each block, a trace partitioning technique would distinguish abstract states arising from the execution of the "if" branch, from those arising from the "else" branch. At the end of the program, it would provide a disjunction of 2^{45} abstract states, which would be of prohibitive size.

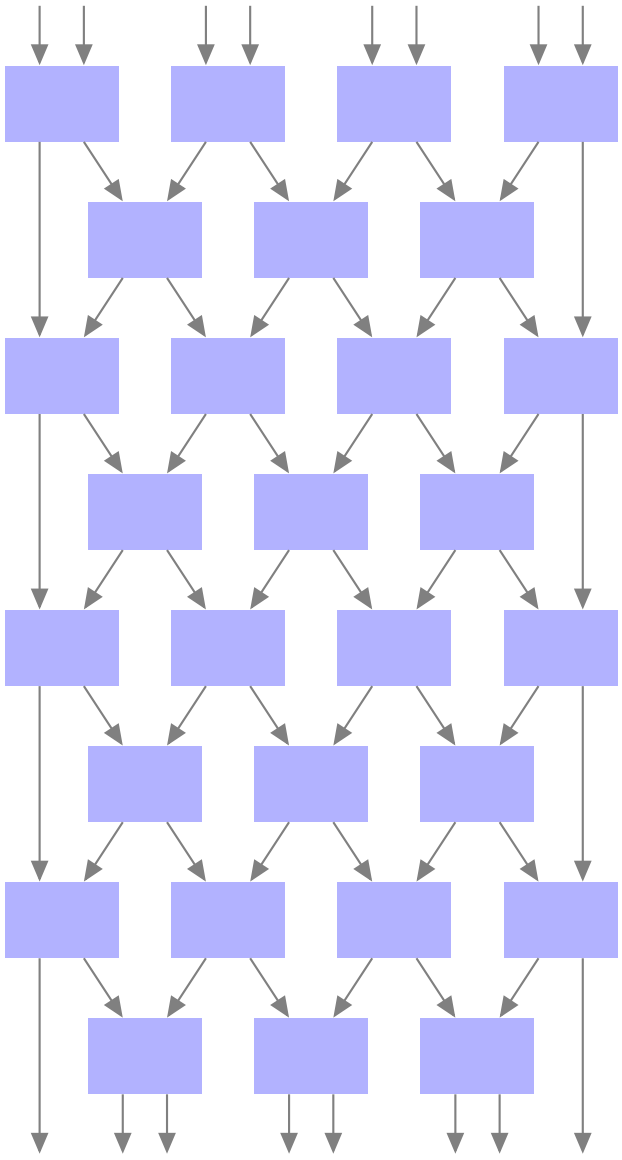
In these experiments, the abstract elements are exclusively computer under their generator component. However, the final abstract element is converted to the constraint form in order to display the final min-/max-invariants to users. This conversion is possibly more costly than the rest of the analysis, while it was negligible in the other experiments. For this reason, the time execution of the rest of the analysis and of this conversion are reported separately in Table 7.1.

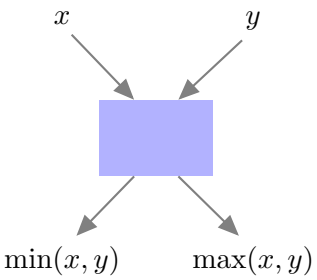
7.4.6 Performance of the analysis

Table 7.1 shows that our current version of the analysis clearly outperforms the initial version implemented in [AGG08]. The former benefits from the improvements provided by ComputeExtRays and ComputeExtRaysPolar, while the latter is based on the previous algorithm OldComputeExtRays. In particular, in the analysis of oddeven-n, the implementation of [AGG08] is not able to compute in reasonable time the min-/max-invariants represented by the final abstract element (for $n = 4$, it takes almost 80 seconds).

In every experiment, both widening operators allow to enforce the convergence of the sequences of invariants after only one iteration. However, the widening ∇_{gen} is always more efficient than the standard widening ∇ on the experimented programs. This is due to the fact that the latter needs the constraint component of its first operand, while the abstract elements

[11] Unless merging some elements of the disjunctions, and subsequently losing precision.





Corresponding code:

```
if x ≥ y + 1 then
   u := x;
   x := y;
   y := u;
else
   skip;
end
```

Figure 7.13: Odd-even sort algorithm of 8 elements

Table 7.1: Analysis benchmarks on a 3 GHz Pentium with 4 Gb RAM

Program	# line	# var.	time (s) with ∇	time (s) with ∇_{gen} + narrowing	time (s) [AGG08]	# gen.
`memcpy`	16	8	0.056	0.024	2.87	7
`strncpy`	20	8	0.044	0.024	2.82	7
incrementing-1	7	3	< 0.001	< 0.001	0.004	3
incrementing-2	10	4	0.004	< 0.001	0.04	4
incrementing-3	14	5	0.012	0.004	0.128	5
incrementing-4	17	6	0.024	0.008	0.336	6
incrementing-5	20	7	0.040	0.016	0.95	7
incrementing-6	22	8	0.060	0.020	2.85	8
incrementing-7	25	9	0.092	0.028	4.61	9
incrementing-8	28	10	0.128	0.036	10.75	10
incrementing-9	31	11	0.184	0.048	16.625	11
incrementing-10	34	12	0.244	0.064	27.3	12
incrementing-11	37	13	0.336	0.088	49.64	13
incrementing-12	40	14	0.44	0.108	77.12	14
incrementing-13	43	15	0.580	0.136	130.65	15
incrementing-14	46	16	0.732	0.158	158.28	16
incrementing-15	49	17	0.94	0.210	245.32	17
incrementing-20	64	22	2.52	0.5	1289.29	22
incrementing-25	79	27	5.63	1.0	5258.55	27
incrementing-30	94	32	11.24	1.7	15692.9	32
incrementing-40	124	42	35.41	4.7	87134.8	42
incrementing-45	139	47	57.5	7.0	188866.0	47
incrementing-60	184	62	190.2	19.0	—	62
			analysis + conversion		analysis + conv.	
oddeven-4	39	9	0.012 + 0.016		0.028 + 79.51	16
oddeven-5	70	11	0.10 + 0.064		0.47 + —	32
oddeven-6	86	13	0.52 + 0.57		3.08 + —	64
oddeven-7	102	15	4.05 + 4.48		59.55 + —	128
oddeven-8	118	17	21.90 + 31.6		437.17 + —	256
oddeven-9	214	19	202.2 + 254.38		8240.65 + —	512
oddeven-10	240	19	1979.7 + 2591.0		81050.27 + —	1024

are most often represented only by their generator component. Therefore, the standard widening causes an additional call to the algorithm ComputeExtRaysPolar. However, note that to obtain the same level of precision, ∇_{gen} has to be combined with a narrowing operator. Here, the narrowing operator Δ maps any decreasing sequence of elements $\mathcal{X}_0 \sqsupseteq \ldots \sqsupseteq \mathcal{X}_n \sqsupseteq \ldots$ to the following sequence:

$$\begin{cases} \mathcal{Y}_0 \stackrel{def}{=} \mathcal{X}_0 \\ \mathcal{Y}_{n+1} \stackrel{def}{=} \mathcal{Y}_n \,\Delta_n\, \mathcal{X}_{n+1} \end{cases}$$

where Δ_1 is defined as the intersection operator $\sqcap$, and $\mathcal{X}\,\Delta_n\,\mathcal{Y} = \mathcal{X}$ for any $n \geq 2$. Intuitively, this narrowing operator applies the intersection on the two first elements $\mathcal{X}_0$ and $\mathcal{X}_1$, and then returns the sequence constant to $\mathcal{X}_0 \sqcap \mathcal{X}_1$. Observe that the sort algorithms oddeven-n does not involve loops, so that their analysis does not use widening or narrowing operators.

7.5 Conclusion of the chapter

In this chapter, we have introduced three new numerical abstract domains, which respectively infer max-invariants, min-invariants, and a superset of min- and max-invariants, and which are based on tropical polyhedra. We have also defined sound and precise abstract primitives on these domains. The two first domains are more precise that the abstract domain of zones, while the third one is able to precisely interact with the abstract domain of octagons.

We have implemented these three abstract domains in the library TPLib, and experimented them on several program analyses. We have shown that they allow to infer precise invariants on memory manipulating programs. We have also seen that our numerical abstract domains are able to scale up to highly disjunctive invariants, which would be practically impossible with other existing disjunctive abstractions.

CHAPTER 8

Conclusion

8.1 Summary of the contributions

Advances in combinatorics and algorithmics of tropical polyhedra. We have defined two algorithms allowing to pass from the external to the internal representations of tropical polyhedra, and vice versa (Chapter 5).

The key ingredient of these two algorithms is a better understanding of the notion of extremality in tropical polyhedra and cones defined as the intersection of tropical halfspaces (Chapter 3). We have indeed reduced the characterization of the extremality of an element $\boldsymbol{x}$ in a tropical polyhedron $\mathcal{P}$ to the existence of a greatest strongly connected component in a given directed hypergraph. This hypergraph corresponds to a combinatorial encoding of the neighborhood of $\boldsymbol{x}$ in the polyhedron $\mathcal{P}$. On top of that, we have derived an even simpler extremality criterion in the polar of cones defined by means of generating sets (Section 5.2.3). This is a singular result, which constrasts with the exact duality between the extremality characterization in cones and their polar in the classical case.

The first algorithm, ComputeExtRays, computes a minimal generating representation of a tropical polyhedron from a system of constraints. It consists in a combination of an incremental method and the extremality criterion previously discussed. The former, *the tropical double description method*, is a generalization of the famous algorithm of Motzkin *et al.* [MRTT53] to the tropical setting. It allows to compute a generating set of a tropical polyhedron by induction on the system of constraints defining it. We have precisely characterized the worst-case complexity of ComputeExtRays. Thanks to our efficient extremality

criterion, it has been proved to be more efficient than the existing methods by several orders of magnitude. This result has been confirmed by a practical comparison of their implementations, which has shown that our method is able to scale to instances inaccessible until now.

The second algorithm, ComputeExtRaysPolar, determines a minimal generating set of the polar of a tropical cone defined by means of generators. It is based on a dual variant of the tropical double description method, and an efficient implementation of the extremality criterion in polar cones. We have shown that, in terms of worst-case complexity, it is more efficient than the naive dualization of the algorithm ComputeExtRays or of the other existing methods (Section 5.2.5).

We have completed this algorithmic study by novel results on the maximal number of extreme elements in tropical polyhedra. We have established that a bound analogue to McMullen's one (which holds in the classical setting) is also valid in the tropical case. We have studied a particular class of tropical polyhedra which have appeared as potential candidates to maximize the number of extreme elements. This has allowed to show that the McMullen-type bound is asymptotically reached when the dimension d tends to infinity, while the number p of inequalities defining the polyhedron is fixed. This has also proved that the number of elements is still exponential in some other configurations of d and p. The problem of finding the exact upper bound and the maximizing class of polyhedron is however still open.

Strongly connected components in directed hypergraphs. Chapter 4 has been devoted to the computation of strongly connected components which are maximal for the order induced by the reachability relation in directed hypergraphs. Very surprisingly, algorithms discovering strongly connected components in directed hypergraphs had not been studied in the literature. The only existing solution consisted in a sequence of calls to an algorithm of Gallo *et al.* [GLPN93], which provides the set of reachable nodes from a given node of a hypergraph. We have shown that this approach is not optimal. For this reason, we have introduced an original algorithm which discovers the maximal components in quasi-linear time in the size of the hypergraph given in input. It is defined as an extension of Tarjan's algorithm on directed graphs. Correctness and complexity have been formally proved. We have also discussed why determining all strongly connected components in directed hypergraphs appears to be a harder problem.

Tropical polyhedra based abstract domains. We have defined a family of disjunctive numerical abstract domains based on tropical polyhedra (Chapter 7).

The two first ones, MaxPoly and MinPoly, respectively allow to infer max- and min-invariants over a given set of variables, of the form $f(\alpha_1+\boldsymbol{v}_1, \ldots, \alpha_d+\boldsymbol{v}_d) \leq f(\beta_1+\boldsymbol{v}_1, \ldots, \beta_d+\boldsymbol{v}_d)$ where f is one of the two operators max and min. Such invariants correspond to a certain class of disjunctions of zone invariants (of the form $\boldsymbol{v}_i - \boldsymbol{v}_j \geq \kappa$). The domain MaxPoly and MinPoly are both provided with the usual abstract primitives. All have been proven to be sound, and many precision results have been established. In particular, the union operator has been shown to be as precise as possible, while intersection and assignment primitives are exact. These primitives rely on the description of tropical polyhedra either by means of generating representations, or by means of tropically affine constraint systems. They are consequently based on the conversion algorithms previously discussed. Their complexity have been precisely characterized. Two kinds of widening operators have been introduced. The

first one can be seen as a generalization of the widening primitive defined by Cousot and Halbwachs in [CH78]. The second one is entirely original, and is based on a projection operator on tropical polyhedra. The two domains MaxPoly and MinPoly have been proved to be more precise than Mine's zone abstract domain. In particular, we have defined a primitive allowing to extract the smallest zone containing the abstract elements of MaxPoly and MinPoly.

The third domain MinMaxPoly is a generalization of the two previous domains. Its abstract elements express max-invariants over the variables $\boldsymbol{v}_i$ and their opposite, which includes both min- and max-invariants over the $\boldsymbol{v}_i$. By extension, they encode some disjunctions of octagonal invariants ($\pm\boldsymbol{v}_i \pm \boldsymbol{v}_j \geq \kappa$). It is also provided with sound abstract primitives. Besides, the domain can be precisely combined with the abstract domain of octagons, which allows to enable a communication of the numerical information between the $\boldsymbol{v}_i$ and their opposite.

The three numerical abstract domains have been integrated in a static analysis on memory manipulations, able to automatically compute sound invariants between integer variables, the size of arrays, and the length of strings (introduced in Chapter 6). The whole contribution has been implemented in a prototype analyzer, and experimented on several programs. We have shown that the resulting analyzer successfully infers precise properties on memory manipulating programs, while non-disjunctive techniques were not sufficient. We have also seen that our tropical polyhedra based domains are a significant contribution as a scalable and entirely automatic method to compute disjunctive invariants. This scalability is mostly provided by the algorithmic improvements on tropical polyhedra brought by ComputeExtRays and ComputeExtRaysPolar. The abstract domains based on tropical polyhedra consequently appear as a real alternative to the existing disjunctive analysis techniques, such that disjunctive completion or trace partitioning.

Software. All the algorithms on tropical polyhedra and on directed hypergraphs defined in this work, and the abstract domains MaxPoly, MinPoly and MinMaxPoly, have been implemented in the library TPLib [All09], written in OCaml and distributed under the LGPL.

8.2 Perspectives

8.2.1 Combinatorics and algorithmics of tropical convex sets

8.2.1.a Faces in the tropical setting. The combinatorial criterion developed in Chapter 3 gives a better insight into the notion of extreme points in tropical polyhedra. We think that it would be very useful to generalize this work to the study of faces.

Nevertheless, faces are not yet precisely defined in the tropical setting. In [Jos05], Joswig proposed a first defininition of facets of a tropical polytope $\mathcal{P}$, as the tropical convex hull of the extreme points contained in a tropical halfspace $\mathcal{H}$ such that $\mathcal{H} \supset \mathcal{P}$. Then, faces are intuitively defined as intersection of facets, thus forming a distributive lattice. However, Develin and Yu discovered in [DY07] that this definition was not satisfactory in dimensions greater than 2. For this reason, they revisited faces, following the initial approach to tropical geometry developed by Richter-Gebert *et al.* [RGST05]. In this latter work, a tropical polytope is defined as the image by the degree map of a *lift*, which is a convex polytope over the d-fold of the Puiseux series field with real exponents $\mathbb{R}[[t^\alpha]]$. Then Develin and Yu defined the faces of a tropical polytope $\mathcal{P}$ by means of the faces of the possible lifts of $\mathcal{P}$. They conjectured that with this formalism, faces are "well-defined" (see [DY07, Conjecture 4.7]).

In any case, it would be of benefit, both from a combinatorial and an algorithmic point of view, to get an "intrisic" definition of faces, *i.e.* which would rely on the description of tropical polyhedra as the convex hull of points or as the intersection of tropical halfspaces. Indeed:

(i) faces would bring a better understanding of the combinatorial complexity of tropical polyhedra. They could certainly lead to establish an upper bound on the number of extreme elements, and provide the maximizing class, by developing a tropical analogue of the theory of f-vectors for instance.

(ii) facets could also help to provide a canonical, and hopefully minimal in some sense, external representation of tropical polyhedra and polyhedral cones. Indeed, even if the extreme elements of the polars can be used as such, some of them necessarily represent redundant inequalities, which can be annoying in some algorithmic applications.

8.2.1.b Tropical linear programming and related problems. An other important algorithmic challenge is to develop algorithms allowing to perform linear programming in the tropical case, *i.e.* to solve problems of the form:

$$\begin{aligned} &\text{Maximize } \boldsymbol{e}\boldsymbol{x} \\ &\text{Subject to } A\boldsymbol{x} \oplus \boldsymbol{b} \leq C\boldsymbol{x} \oplus \boldsymbol{d} \end{aligned}$$

or of the form:

$$\begin{aligned} &\text{Minimize } \boldsymbol{f}\boldsymbol{x} \\ &\text{Subject to } A\boldsymbol{x} \oplus \boldsymbol{c} \leq B\boldsymbol{x} \oplus \boldsymbol{d} \end{aligned}$$

where $A, B \in \mathbb{R}_{\max}^{p\times d}$, $\boldsymbol{c}, \boldsymbol{d} \in \mathbb{R}_{\max}^{p}$, and $\boldsymbol{e}, \boldsymbol{f} \in \mathbb{R}_{\max}^{d}$. They consist in respectively maximizing or minimizing tropical linear forms under polyhedral constraints. Such problems have been studied by Butkovič and Aminu in [BA08]. They have defined pseudo-polynomial time algorithms able to handle instances with integer entries. The question "Can linear programming problems be solved in worst-case polynomial time?" is however still open.

A related problem, "Find a solution of a tropical affine system $A\boldsymbol{x}\oplus\boldsymbol{c} \leq B\boldsymbol{x}\oplus\boldsymbol{d}$", is also of great interest. It can be seen as a particular instance of the two previous problems. Naturally, using the tropical double description method for this problem would be an overkill, since this latter computes the whole set of the solutions, while here, only one solution is expected. The corresponding linear problem "Find a non-identically null solution of the tropical linear system $A\boldsymbol{x} \leq B\boldsymbol{x}$" has already been studied in the literature. The *alternating method*, developed by Cuninghame-Green and Butkovič in [CGB03], allows to find a solution $(\boldsymbol{x}, \boldsymbol{y})$ of an equation of the form $C\boldsymbol{x} = D\boldsymbol{y}$.[1] It runs in pseudo-polynomial time when the entries of the matrices are integers. Butkovič and Zimmerman later proposed an algorithm to find a non-identically null solution of $A\boldsymbol{x} = B\boldsymbol{x}$ [BZ06]. They thought that its complexity was linear, but this statement was disproved by Bezem *et al.* [BNRC08a]. The latter proved that this problem belongs to the complexity class $\mathsf{NP} \cap \mathsf{coNP}$ [BNRC08b]. This result has been generalized by Akian *et al.* [AGG09a], who have shown that the linear and affine problems, and some other problems related to tropical linear convexity, reduce to mean payoff games. They similarly

[1] Solving $A\boldsymbol{x} \leq B\boldsymbol{x}$ and $C\boldsymbol{x} = D\boldsymbol{y}$ are two equivalent problems. The latter can be used to solve the former defining $C = \left(\begin{smallmatrix} A \\ B \end{smallmatrix}\right)$ and $D = \left(\begin{smallmatrix} I_d \\ I_d \end{smallmatrix}\right)$, where I_d is the tropical identity $d \times d$-matrix. Conversely, the latter can be seen as an instance of $A(\boldsymbol{x}, \boldsymbol{y}) \leq B(\boldsymbol{x}, \boldsymbol{y})$ by setting $A = \left(\begin{smallmatrix} C & \mathbb{0}_{p\times d} \\ \mathbb{0}_{p\times d} & D \end{smallmatrix}\right)$ and $B = \left(\begin{smallmatrix} \mathbb{0}_{p\times d} & D \\ C & \mathbb{0}_{p\times d} \end{smallmatrix}\right)$.

derived that they are all in $\mathsf{NP} \cap \mathsf{coNP}$. Finding polynomial time algorithms is still an open problem, which consequently also have important applications to game theory.

8.2.1.c Further work on the tropical convex hull problem. As discussed in the conclusion of Chapter 5, a very interesting perspective is to develop further algorithms allowing to pass from the external form of tropical polyhedra to their internal form, and conversely. The algorithms ComputeExtRays and ComputeExtRaysPolar are both defined incrementally. In the classical case, some other well-known algorithms, such as the optimal solution of Chazelle [Cha93], or the Beneath-and-Beyond method of Seidel [Sei81] use a similar principle. It would be natural to ask whether they can be generalized to the tropical setting.

However, the major drawback of the previous algorithms is that none are output-sensitive (*i.e.* their time complexity can not be bounded in terms of the size of the result), since their complexity highly depends on the size of the intermediate results. For this reason, we think that an important challenge is to define another class of algorithms, such as *pivoting algorithms*, for tropical polyhedra. In the classical case, pivoting algorithms usually refer to variants of the simplex algorithm. The initial purpose of the simplex method is linear programming [Dan63]. It consists in iterating over the vertices of a polyhedron given by a system $A\boldsymbol{x} \leq \boldsymbol{b}$, by enumerating the feasible bases. A *basis* corresponds to a subsystem $A'\boldsymbol{x} \leq \boldsymbol{b}'$ of rank d. It is said to be *feasible* when the unique solution of $A'\boldsymbol{x} = \boldsymbol{b}'$ also satisfies the initial system $A\boldsymbol{x} \leq \boldsymbol{b}$. Each vertex of the polyhedron can naturally be associated to at least one feasible basis. It can be shown that starting from a vertex, and then iterating over the feasible bases by "pivoting", *i.e.* by exchanging at each step one of the inequalities of the basis by another constraint of the system $A\boldsymbol{x} \leq \boldsymbol{b}$, allows to discover the whole set of vertices. One of the most famous convex hull pivoting algorithms is due to Avis and Fukuda [AF92]. It consists in iterating over all the feasible bases using a particular traversal algorithm, the *reverse-search method*, which has been later generalized to other combinatorial enumeration problems [AF96]. Pivoting algorithms are particularly efficient for *simple* polytopes, in which each vertex admits a unique feasible basis.[2] For instance, the time complexity of the reverse-search method can be shown to be polynomial in the size of the input and the output. However, in practice, pivoting algorithms usually have poor performance on "degenerate" inputs, which admit much more feasible bases than vertices. In constrast, incremental methods behave much better on such instances.

Observe that the classical pivoting algorithms all start from a particular vertex. As previously discussed, finding such a point in the tropical setting may not be that easy (in particular, we do not have yet polynomial time methods in the general case). However, the conversion problems, from external to internal forms, and inversely from internal to external forms, may not be computationally equivalent. In particular, finding an extreme element of the polar of a cone is straightforward, since any tautology $\boldsymbol{x}_i \geq \mathbb{0}$ is a valid candidate. That is why we think that in a first step, it could be easier to define pivoting algorithms to compute the extreme elements of polar cones. Also note that a pivoting algorithm in the tropical setting would also be interesting to solve tropical linear programming problems discussed in Section 8.2.1.b.

Finally, we know that in general, computing the whole set of the vertices of a polyhedron given by a system of inequalities is a NP-hard problem [KBB+06]. Is it still true in the tropical case?

[2] Formally, a polytope is said to be simple when every vertex is in precisely d facets.

8.2.1.d Studying other operations on tropical polyhedral sets. While the conversion algorithms that we have defined in Chapter 5 are of critical importance, algorithms performing other kinds of operations are also of interest. For instance, it would be interesting to study how to efficiently compute the intersection of two tropical polyhedra $\mathcal{P}$ and $\mathcal{Q}$ when both are provided by generating representations. If the corresponding homogenized cones are given by two generating sets G and H, then the homogenized cone of $\mathcal{P} \cap \mathcal{Q}$ is formed by the vectors $\boldsymbol{x}$ such that there exist $\boldsymbol{\lambda} \in \mathbb{R}_{\max}^{|G|}$, $\boldsymbol{\mu} \in \mathbb{R}_{\max}^{|H|}$ satisfying the system:

$$\boldsymbol{x} = G\boldsymbol{\lambda} = H\boldsymbol{\mu}, \tag{8.1}$$

where the sets G and H are assimilated to matrices whose columns are formed by their elements. A generating family of the cone $\widehat{\mathcal{P} \cap \mathcal{Q}}$ can be computed by solving the system (8.1) over the triples $(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$, and then projecting the result on the first d coordinates. However, this projection may yield redundant elements. That is why we may ask whether there exists a method to directly compute a minimal generating representation of $\mathcal{P} \cap \mathcal{Q}$. Similarly, computing the inverse image by a tropical linear map $f : \boldsymbol{x} \mapsto M\boldsymbol{x}$ ($M \in \mathbb{R}_{\max}^{p \times d}$) of a tropical polyhedra cone $\mathcal{C} = \mathsf{cone}(G)$ amounts to solve the system

$$M\boldsymbol{x} = G\boldsymbol{\lambda}$$

in the unknown $(\boldsymbol{x}, \boldsymbol{\lambda})$, and then projecting the provided generating set on the first d coordinates. Can we compute a minimal generating set of $f^{-1}(\mathcal{C})$ directly, *i.e.* without having to generate redundant candidates? Generalizing the extremality criterion of Chapter 3 to such configurations could certainly help to answer these questions.

8.2.2 Numerical abstract domains.

8.2.2.a Improving precision: mixing tropical and classical linear invariants. The numerical domains developed in Chapter 7 are able to infer precise invariants involving the operators min and max, which is an important advantage over the existing convex domains. However, they do not precisely handle classical linear invariants, except zone and octagonal properties. That is why we may wonder how to combine them with some classical convex domains.

In the abstract interpretation framework, domains are usually combined using *reduced product* [CC79]. The latter intuitively consists in a cartesian product of domains, equipped with a reduction operator enabling communication between the different domains. Nevertheless, a more powerful combination could be performed following a technique due to Nelson and Oppen [NO79], and generalized to abstract domains by Gulwani and Tiwari in [GT06]. Such a combination of the abstract domain MaxPoly with the domain of classical convex polyhedra would allow for instance to express the following invariant:

$$\max(\boldsymbol{v}_1, \boldsymbol{v}_2 + \boldsymbol{v}_3) \leq \max(0, \boldsymbol{v}_2 + 2\boldsymbol{v}_4) - 2\max(1, \boldsymbol{v}_1 + 1).$$

This relation is actually encoded by a couple of abstract elements of each domain. Each abstract element expresses invariants over the variables $\boldsymbol{v}_i$ and some *shared* variables, denoted here $\boldsymbol{s}_i$, which are the key ingredient of the combination. In our example:

$$\underbrace{\begin{aligned} \max(\boldsymbol{v}_1, \boldsymbol{s}_1) &= \boldsymbol{s}_2 \\ \max(0, \boldsymbol{s}_3) &= \boldsymbol{s}_4 \\ \max(1, \boldsymbol{v}_1 + 1) &= \boldsymbol{s}_5 \end{aligned}}_{\mathsf{MaxPoly}} \qquad \underbrace{\begin{aligned} \boldsymbol{s}_1 &= \boldsymbol{v}_2 + \boldsymbol{v}_3 \\ \boldsymbol{s}_3 &= \boldsymbol{v}_2 + 2\boldsymbol{v}_4 \\ \boldsymbol{s}_2 &\leq \boldsymbol{s}_4 - 2\boldsymbol{s}_5 \end{aligned}}_{\text{convex polyhedra}}$$

Unfortunately, because of the non-convexity of the domain MaxPoly, the additional time cost of the Nelson-Oppen combination may be exponential in the number of shared variables. However, this combination is very expressive, and applies to a very general setting. Hopefully we could develop an ad-hoc combination which would be maybe less precise but more efficient. We could for instance restrict the expressiveness to invariants of the form:

$$\max(\alpha_0, \alpha_1 + f_1, \ldots, \alpha_p + f_p) \leq \max(\beta_0, \beta_1 + f'_1, \ldots, \beta_p + f'_q) \tag{8.2}$$

where the f_i and f'_j are linear forms over the variables $\boldsymbol{v}_k$.[3] This would allow to be more precise than the abstract domain developped by Gulavani and Gulwani in [GG08], and applied to timing analysis.

8.2.2.b Improving scalability: towards subpolyhedral domains. An other challenge is the definition of more scalable tropically convex domains. In particular, it would be particularly interesting to find subclasses of invariants which could be inferred in polynomial time.

Naturally, zones are an illustration of such domains. We think that it would be certainly instructive to rediscover this abstract domain as a tropical subpolyhedral domain, and to redefine its abstract primitives, in particular widening operators, so as to use the generator form.[4]

Similarly, we could develop a tropical analogue of the abstract domain of linear templates [SSM05]. This domain infers lower and upper bounds on a pre-fixed set of linear forms $f_1,\ldots,f_p$ over the set of the variables $\boldsymbol{v}_1,\ldots,\boldsymbol{v}_d$. Its abstract primitives are based on linear programming algorithms, and their time complexity are therefore polynomial. Generalizing this domain to the tropical setting would naturally require to have a better insight into the complexity of tropical linear programming. However, the existing algorithms discussed in Section 8.2.1.b would be probably a good starting point.

8.2.2.c Application to further static analyses. Finally, we believe that other static analyses could benefit from the precision of the abstract domains based on tropical polyhedra.

As discussed in Section 7.4.3, these domains could be used in array predicate abstraction. More generally, they could be also involved in static analyses inferring "pattern" invariants on string buffers, such as

$$\texttt{/[\^{}/]}^{\boldsymbol{i}_1}\texttt{/[\^{}/]}^{\boldsymbol{i}_2}\texttt{/[\^{}/]}^{\boldsymbol{i}_3}$$

which represents a path of depth 3 in a UNIX filesystem (using a POSIX-like regular expression notation). To get a precise information, these patterns would be decorated with numerical data (here the $\boldsymbol{i}_1$, $\boldsymbol{i}_2$, $\boldsymbol{i}_3$ represent the number of characters located between the delimiters /). The whole invariants on strings would include numerical properties on these variables. Such invariants are omnipresent in string processing software.[5] In particular, precisely analyzing

[3] Also observe that if the coefficients of the linear forms are positive integers, the invariant (8.2) is actually a non-affine polynomial relation in the tropical sense. For instance, a term of the form $\max(\boldsymbol{v}_1, \boldsymbol{v}_2 + 2\boldsymbol{v}_3)$ can be indeed written as $\boldsymbol{v}_1 \oplus (\boldsymbol{v}_2 \otimes \boldsymbol{v}_3^2)$. Consequently, another approach could be to define an abstract domain able to infer tropical polynomial invariants. Nevertheless, the complexity of such a domain would be certainly much worse that the affine domains introduced in this manuscript.

[4] Until now, no generator form for zones could be used in practice, since zones were seen as convex sets in the classical case. They could therefore have an exponential number of extreme elements in the worst case (consider for instance a hypercube).

[5] This includes a very large class of software, such as web, email, database servers and clients, spam filters, versioning tools, *etc.*

such pattern manipulations certainly requires to express min- or max-relations between the program variables and the numerical data decorating the pattern invariants.

Final words

Finally, we really hope that the free distribution of the algorithms on tropical polyhedra and the corresponding abstract domains in the open source library TPLib will help to overcome the challenges previously discussed. The integration of the abstract domains of TPLib into the APRON library [JM] is one of the goals of the project "ASOPT" of the French National Agency of Research (ANR).

Bibliography

[ABS97] David Avis, David Bremner, and Raimund Seidel. How good are convex hull algorithms? *Comput. Geom. Theory Appl.*, 7(5-6):265–301, 1997.

[AF92] David Avis and Komei Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete Comput. Geom.*, 8(3):295–313, 1992.

[AF96] David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Appl. Math.*, 65(1-3):21–46, 1996.

[AFF01] Giorgio Ausiello, Paolo Giulio Franciosa, and Daniele Frigioni. Directed hypergraphs: Problems, algorithmic results, and a novel decremental approach. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *Theoretical Computer Science, 7th Italian Conference, ICTCS 2001, Proceedings*, volume 2202 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2001.

[AFFG97] Giorgio Ausiello, Paolo Giulio Franciosa, Daniele Frigioni, and Roberto Giaccio. Decremental maintenance of reachability in hypergraphs and minimum models of horn formulae. In Hon Wai Leong, Hiroshi Imai, and Sanjay Jain, editors, *Algorithms and Computation, 8th International Symposium, ISAAC '97, Singapore, December 17-19, 1997, Proceedings*, volume 1350 of *Lecture Notes in Computer Science*, pages 122–131. Springer, 1997.

[AGG08] X. Allamigeon, S. Gaubert, and É. Goubault. Inferring min and max invariants using max-plus polyhedra. In *SAS'08*, volume 5079 of *LNCS*, pages 189–204. Springer, Valencia, Spain, 2008.

[AGG09a] M. Akian, S. Gaubert, and A. Guterman. Tropical polyhedra are equivalent to mean payoff games. preprint, 2009.

[AGG09b] Xavier Allamigeon, Stéphane Gaubert, and Eric Goubault. Computing the Extreme Points of Tropical Polyhedra. arXiv:math/0904.3436, 2009.

[AGG10] Xavier Allamigeon, Stéphane Gaubert, and Eric Goubault. The Tropical Double Description Method. In *Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science (STACS'10)*, Nancy, France, March 2010. To appear.

[AGH06] Xavier Allamigeon, Wenceslas Godard, and Charles Hymans. Static Analysis of String Manipulations in Critical Embedded C Programs. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 35–51, Seoul, Korea, August 2006. Springer Verlag.

[AGK05] M. Akian, S. Gaubert, and V. N. Kolokoltsov. Set coverings and invertibility of functional galois connections. In G. L. Litvinov and V. P. Maslov, editors, *Idempotent Mathematics and Mathematical Physics*, Contemporary Mathematics, pages 19–51. American Mathematical Society, 2005.

[AGK10] Xavier Allamigeon, Stéphane Gaubert, and Ricardo D. Katz. The number of extreme points of tropical polyhedra. *Journal of Combinatorial Theory, series A*, 2010. To appear, Eprint arXiv:math/0906.3492.

[AH07] Xavier Allamigeon and Charles Hymans. Analyse Statique par Interprétation Abstraite : Application à la Détection de Dépassement de Tampon. In Eric Filiol, editor, *5ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'07)*, pages 347–384, Rennes, France, June 2007. ESAT.

[AH08] Xavier Allamigeon and Charles Hymans. Static Analysis by Abstract Interpretation: Application to the Detection of Heap Overflows. *Journal in Computer Virology*, 4(1):5–23, 2008.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[AI91] Giorgio Ausiello and Giuseppe F. Italiano. On-line algorithms for polynomially solvable satisfiability problems. *J. Log. Program.*, 10(1/2/3&4):69–90, 1991.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, 1978.

[All08] Xavier Allamigeon. Non-disjunctive Numerical Domain for Array Predicate Abstraction. In Sophia Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 163–177, Budapest, Hungary, April 2008. Springer Verlag.

[All09] Xavier Allamigeon. TPLib: Tropical polyhedra library, 2009. Available at `http://penjili.org/tplib.html`.

[Ast] Astree. `http://www.astree.ens.fr`.

[BA08] P. Butkovič and A. Aminu. Introduction to max-linear programming. *IMA Journal of Management Mathematics*, 2008.

[Bat68] K.E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference 32*, pages 307–314, 1968.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN PLDI'03*, volume 548030, pages 196–207. ACM Press, June 2003.

[BCC+07a] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification*, April 2007. To appear.

[BCC+07b] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification*, April 2007. To appear.

[BCOQ92] F. Baccelli, G. Cohen, G.J. Olsder, and J.P. Quadrat. *Synchronization and Linearity*. Wiley, 1992.

[BH84] P. Butkovič and G. Hegedüs. An elimination method for finding all solutions of the system of linear equations over an extremal algebra. *Ekonomicko-matematicky Obzor*, 20, 1984.

[BH04] W. Briec and C. Horvath. $\mathbb{B}$-convexity. *Optimization*, 53:103–127, 2004.

[BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 300–309, New York, NY, USA, 2007. ACM Press.

[BHR05] W. Briec, C. Horvath, and A. Rubinov. Separation in $\mathbb{B}$-convexity. *Pacific Journal of Optimization*, 1:13–30, 2005.

[BHRZ03] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 337–354, San Diego, California, USA, 2003. Springer-Verlag, Berlin.

[BHT06] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy shape analysis. In T. Ball and R.B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006, Seattle, WA, August 16-20)*, LNCS 4144, pages 532–546. Springer-Verlag, Berlin, 2006.

[BHZ06] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. *Software Tools for Technology Transfer*, 8(4/5):449–466, 2006.

[BHZ08] 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):3–21, 2008.

[Bla] Blast. `http://mtc.epfl.ch/software-tools/blast/`.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. *SIGPLAN Not.*, 36(5):203–213, 2001.

[BNRC08a] Marc Bezem, Robert Nieuwenhuis, and Enric Rodríguez-Carbonell. Exponential behaviour of the butkovič-zimmermann algorithm for solving two-sided linear systems in max-algebra. *Discrete Appl. Math.*, 156(18):3506–3509, 2008.

[BNRC08b] Marc Bezem, Robert Nieuwenhuis, and Enric Rodríguez-Carbonell. The max-atom problem and its relevance. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008*, volume 5330 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2008.

[BSS07] P. Butkovič, H. Schneider, and S. Sergeev. Generators, extremals and bases of max cones. *Linear Algebra Appl.*, 421(2-3):394–406, 2007.

[BZ06] Peter Butkovic and Karel Zimmermann. A strongly polynomial algorithm for solving two-sided linear systems in max-algebra. *Discrete Applied Mathematics*, 154(3):437–446, 2006.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CC92b] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CC04] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. In Roberto Giacobazzi, editor, *Static Analysis Symposium (SAS)*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2004.

[CCF+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In M. Sagiv, editor, *Proceedings of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, Scotland, April 2–10 2005. © Springer.

[CDNB08] Christopher L. Conway, Dennis Dams, Kedar S. Namjoshi, and Clark Barrett. Points-to analysis, conditional soundness, and proving the absence of errors. In *Static Analysis Symposium (SAS)*, pages 62–77, Valencia, Spain, July 2008.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CG76] R. A. Cuninghame-Green. Projections in minimax algebra. *Mathematical Programming*, 10(1):111–123, December 1976.

[CG79] R. A. Cuninghame-Green. *Minimax algebra*, volume 166 of *Lecture Notes in Economics and Mathematical Systems*. Springer-Verlag, Berlin, 1979.

[CG95] R.A Cuninghame-Green. Minimax algebra and applications. *Advances in Imaging and Electron Physics*, 90, 1995.

[CGB03] R. A. Cuninghame-Green and P. Butkovic. The equation ax = by over (max, +). *Theor. Comput. Sci.*, 293(1):3–12, 2003.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[CGMQ] G. Cohen, S. Gaubert, M. McGettrick, and J.-P. Quadrat. Maxplus toolbox of SCILAB. Available at `http://minimal.inria.fr/gaubert/maxplustoolbox`; now integrated in SCICOSLAB.

[CGQ99] G. Cohen, S. Gaubert, and J.P. Quadrat. Max-plus algebra and system theory: where we are and where to go now. *Annual Reviews in Control*, 23:207–219, 1999.

[CGQ01] G. Cohen, S. Gaubert, and J.P. Quadrat. Hahn-Banach separation theorem for max-plus semimodules. In J.L. Menaldi, E. Rofman, and A. Sulem, editors, *Optimal Control and Partial Differential Equations*, pages 325–334. IOS Press, 2001.

[CGQ04] G. Cohen, S. Gaubert, and J. P. Quadrat. Duality and separation theorem in idempotent semimodules. *Linear Algebra and Appl.*, 379:395–422, 2004.

[CGQS05] G. Cohen, S. Gaubert, J. P. Quadrat, and I. Singer. Max-plus convex sets and functions. In G. L. Litvinov and V. P. Maslov, editors, *Idempotent Mathematics and Mathematical Physics*, Contemporary Mathematics, pages 105–129. American Mathematical Society, 2005.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[Cha93] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete and Compututational Geometry*, 10:377–409, 1993.

[Che68] N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[CL] T. Christof and A. Löbel. The porta library. Available at `http://www.zib.de/Optimization/Software/Porta/`.

[CM96] Joseph Cheriyan and Kurt Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.

[Coq] Coq. `http://inria.coq.fr`.

[Cou03] P. Cousot. Verification by abstract interpretation. In N. Dershowitz, editor, *Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, pages 243–268, Taormina, Italy, June 29 – July 4 2003. © Springer-Verlag, Berlin, Germany.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[CWE] Cwe-119: Failure to constrain operations within the bounds of a memory buffer. `http://cwe.mitre.org/top25/index.html#CWE-119`.

[Dan63] G.B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, 1963.

[Deu92] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In James R. Cordy and Mario Barbacci, editors, *ICCL'92, Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13, 1992.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 230–241, New York, NY, USA, 1994. ACM.

[DLGKL09] Michael Di Loreto, Stephane Gaubert, Ricardo D. Katz, and Jean-Jacques Loiseau. Duality between invariant spaces for max-plus linear discrete event systems. Eprint arXiv:0901.2915., 2009.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International*

Conference, TACAS 2006, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, March 2006.

[DRS03] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167, New York, NY, USA, 2003. ACM Press.

[DS04] M. Develin and B. Sturmfels. Tropical convexity. *Doc. Math.*, 9:1–27 (electronic), 2004.

[DY07] M. Develin and J. Yu. Tropical polytopes and cellular resolutions. *Experimental Mathematics*, 16(3):277–292, 2007.

[EMCGP99] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[ER89] Mark W. Eichin and John A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. *Security and Privacy, IEEE Symposium on*, 0:326, 1989.

[Fil03] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[Fil08] Jean-Christophe Filliâtre. Implementation of hash sets for OCaml, 2008. Available at `http://www.lri.fr/\~filliatr/software.fr.html`.

[Fla] Flawfinder. `http://www.dwheeler.com/flawfinder/`.

[FP96] Komei Fukuda and Alain Prodon. Double description method revisited. In *Selected papers from the 8th Franco-Japanese and 4th Franco-Chinese Conference on Combinatorics and Computer Science*, pages 91–111, London, UK, 1996. Springer-Verlag.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202, New York, NY, USA, 2002. ACM Press.

[fS99] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, December 1999.

[Fuk] K. Fukuda. The cdd library. `http://www.ifor.math.ethz.ch/~fukuda/cdd_home/`.

[Gab00] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000.

[Gan05] Jack Ganssle. Big Code. Available at `http://www.embedded.com/columns/embeddedpulse/171203287?_requestid=1130518`, May 2005.

[Gau92] S. Gaubert. *Théorie des systèmes linéaires dans les dioïdes*. Thèse, École des Mines de Paris, July 1992.

[GG08] Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference*, volume 5123 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2008.

[GGPR98] Giorgio Gallo, Claudio Gentile, Daniele Pretolani, and Gabriella Rago. Max horn sat and the minimum cut problem in directed hypergraphs. *Math. Program.*, 80:213–237, 1998.

[GK06] S. Gaubert and R. Katz. Max-plus convex geometry. In R. A. Schmidt, editor, *RelMiCS/AKA 2006*, volume 4136 of *Lecture Notes in Comput. Sci.*, pages 192–206. Springer, 2006.

[GK07] S. Gaubert and R. Katz. The Minkowski theorem for max-plus convex sets. *Linear Algebra and Appl.*, 421:356–369, 2007.

[GK09a] S. Gaubert and R. Katz. The tropical analogue of polar cones. *Linear Algebra and Appl.*, 431:608–625, 2009.

[GK09b] Stéphane Gaubert and Ricardo D. Katz. Minimal Half-Spaces and External Representation of Tropical Polyhedra. arXiv:math/0908.1586, 2009.

[GLPN93] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete Appl. Math.*, 42(2-3):177–201, 1993.

[GM08] S. Gaubert and F. Meunier. Carathéodory, Helly and the others in the max-plus world. To appear in Discrete Computational Geometry, also arXiv:0804.1361v1, 2008.

[GMP] Gnu multiple precision library. `http://www.gmplib.org`.

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL'08: Proceedings of the 35th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008. To Appear.

[GP97] S. Gaubert and M. Plus. Methods and applications of (max,+) linear algebra. In R. Reischuk and M. Morvan, editors, *STACS'97*, number 1200 in LNCS, Lübeck, March 1997. Springer.

[GP06] Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 18–5134, Seoul, Korea, August 2006. Springer Verlag.

[GR98] R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Science of Computer Programming*, 32(1–3):177–210, 1998.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In Samson Abramsky and T. S. E. Maibaum, editors, *International Joint Conference on Theory and Practice of Software Development*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192, Brighton, UK, 1991. Springer.

[Gra97] Philippe Granger. Static analyses of congruence properties on rational numbers (extended abstract). In Pascal Van Hentenryck, editor, *Static Analysis Symposium (SAS)*, volume 1302 of *Lecture Notes in Computer Science*, pages 278–292, Paris, France, 1997. Springer.

[GRS05] Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. *SIGPLAN Not.*, 40(1):338–350, 2005.

[GT06] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 376–386, New York, NY, USA, 2006. ACM.

[Hal79] N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. Thèse de 3ème cycle d'informatique, Université scientifique et médicale de Grenoble, Grenoble, France, March 1979.

[HL08] Charles Hymans and Olivier Levillain. Newspeak, Doubleplussimple Minilang for Goodthinkful Static Analysis of C. Technical Note 2008-IW-SE-00010-1, EADS IW/SE, 2008.

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 339–348. ACM, 2008.

[HT98] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In Giorgio Levi, editor, *Static Analysis, 5th International Symposium, SAS '98*, volume 1503 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 1998.

[Isa] Isabelle. `http://isabelle.in.tum.de/`.

[JKSY05] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In Igor Siveroni Chris Hankin, editor, *Static Analysis: 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005. Proceedings*, Lecture Notes in Computer Science, pages 203–217. Springer-Verlag, September 2005.

[JM] Bertrand Jeannet and Antoin Miné. APRON numerical abstract domain library. `http://apron.cri.ensmp.fr/library/`.

[JM07] Ranjit Jhala and Kenneth L. McMillan. Array abstractions from proofs. In *CAV*, pages 193–206, 2007.

[Jos05] M. Joswig. Tropical halfspaces. In *Combinatorial and computational geometry*, volume 52 of *Math. Sci. Res. Inst. Publ.*, pages 409–431. Cambridge Univ. Press, Cambridge, 2005.

[Jos08] Michael Joswig. Tropical convex hull computations, November 2008. Eprint arXiv:0809.4694, to appear in Contemporary Mathematics.

[JRGJF06] Jason R. Ghidella and Jon Friedman. Streamlined development of body electronics systems using model-based design. `http://www.mathworks.com/company/pressroom/newsletter/sept06/body_electronics.html`, September 2006.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[Kat07] R. D. Katz. Max-plus (A, B)-invariant spaces and control of timed discrete event systems. *IEEE Trans. Aut. Control*, 52(2):229–241, 2007.

[KBB+06] Leonid Khachiyan, Endre Boros, Konrad Borys, Khaled Elbassioni, and Vladimir Gurvich. Generating all vertices of a polyhedron is hard. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 758–765, New York, NY, USA, 2006. ACM.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206, 1973.

[LLF+96] J.L. Lions, L. Lübeck, J.L. Fauquembergue, G.Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. Ariane 5, flight 501 failure, report by the inquiry board. `http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`, July 1996.

[LS98] Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 53–66. Springer, 1998.

[LV92] H. Le Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, February 1992.

[Mar05] Vincent Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison-Wesley Professional, 2005.

[McC04] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[McM70] P. McMullen. The maximum numbers of faces of a convex polytope. *Mathematika*, 17:179–184, 1970.

[Min01a] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053 of *LNCS*, pages 155–172. Springer-Verlag, May 2001. `http://www.di.ens.fr/~mine/publi/article-mine-padoII.pdf`.

[Min01b] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. `http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf`.

[Min04] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004. `http://www.di.ens.fr/~mine/these/these-color.pdf`.

[Min06] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *ACM SIGPLAN LCTES'06*, pages 54–63. ACM Press, June 2006.

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on programming languages and systems*, 30(3):12, May 2008.

[MOS04] Markus Müller-Olm and Helmut Seidl. A Note on Karr's Algorithm. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 1016–1028. Springer, 2004.

[MR05a] L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *ESOP'05*, 2005.

[MR05b] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.

[MRTT53] T.S. Motzkin, H. Raiffa, G.L. Thompson, and R.M. Thrall. The double description method. In H.W. Kuhn and A.W. Tucker, editors, *Contributions to the Theory of Games*, volume II, pages 51–73, 1953.

[Nil71] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill Pub. Co., 1971.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[NP89] S. Nguyen and S. Pallottino. Hyperpaths and shortest hyperpaths. In *COMO '86: Lectures given at the third session of the Centro Internazionale Matematico Estivo (C.I.M.E.) on Combinatorial optimization*, Lectures Notes in Mathematics, pages 258–271, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[NPG98] Sang Nguyen, Stefano Pallottino, and Michel Gendreau. Implicit enumeration of hyperpaths in a logit model for transit networks. *Transportation Science*, 32(1):54–64, 1998.

[NS07] V. Nitica and I. Singer. Max-plus convex sets and max-plus semispaces. I. *Optimization*, 56(1–2):171–205, 2007.

[NuS] Nusmv. `http://nusmv.irst.itc.it/`.

[Off92] United States General Accounting Office. Patriot missile defense, software problem led to system failure at dhahran, saudi arabia. `http://archive.gao.gov/t2pbat6/145960.pdf`, February 1992.

[Özt08] Can C. Özturan. On finding hypercycles in chemical reaction networks. *Appl. Math. Lett.*, 21(9):881–884, 2008.

[Pen] Penjili. `http://www.penjili.org`.

[Pin98] J.-E. Pin. Tropical semirings. In *Idempotency (Bristol, 1994)*, volume 11 of *Publ. Newton Inst.*, pages 50–69. Cambridge Univ. Press, Cambridge, 1998.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Pol] Polyspace. `http://www.polyspace.com`.

[Pre00] Daniele Pretolani. A directed hypergraph model for random time dependent shortest paths. *European Journal of Operational Research*, 123(2):315–324, June 2000.

[Pre03] Daniele Pretolani. Hypergraph reductions and satisfiability problems. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2003.

[PVS] PVS specification and verification system. `http://pvs.csl.sri.com/`.

[RGST05] J. Richter-Gebert, B. Sturmfels, and T. Theobald. First steps in tropical geometry. In *Idempotent mathematics and mathematical physics*, volume 377 of *Contemp. Math.*, pages 289–317. Amer. Math. Soc., Providence, RI, 2005.

[Ric56] H. Gordon Rice. On completely recursively enumerable classes and their key arrays. *J. Symb. Log.*, 21(3):304–308, 1956.

[RJP96] Ron J. Pehrson. Software development for the Boeing 777. January 1996.

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5), 2007.

[SAN07] SANS. Top 25 most dangerous programming errors. `http://cwe.mitre.org/top25/index.html#CWE-119`, 2007.

[Scia] ScicosLab. `http://www.scicoslab.org`.

[Scib] Scilab. `http://www.scilab.org`.

[Sei81] Raimund Seidel. A convex hull algorithm optimal for point sets in even dimensions. Technical report, Vancouver, BC, Canada, Canada, 1981.

[Sin97] I. Singer. *Abstract convex analysis*. Wiley, 1997.

[SK02] A. Simon and A. King. Analyzing String Buffers in C. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology*, volume 2422 of *LNCS*, pages 365–379, Reunion Island, France, September 2002. Springer.

[SKH03] A. Simon, A. King, and J. M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In M. Leuschel, editor, *Logic-Based Program Synthesis and Transformation*, volume 2664 of *LNCS*, pages 71–89, Madrid, Spain, September 2003. Springer.

[Spi] Spin. `http://spinroot.com/`.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[SRW99] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3–valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[SSM05] S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proceedings of VMCAI*, volume 3385. LNCS, 2005.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[TVL] Tvla. `http://www.math.tau.ac.il/~tvla/`.

[Ull82] Jeffrey D. Ullman. *Principle of database systems*. Computer Science Press, 1982.

[VB04] Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded C programs. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 231–242, New York, NY, USA, 2004. ACM Press.

[Ven02] Arnaud Venet. Nonuniform alias analysis of recursive data structures and arrays. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 36–51, London, UK, 2002. Springer-Verlag.

[Ven04] Arnaud Venet. A scalable nonuniform pointer analysis for embedded programs. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004*, volume 3148 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2004.

[Vor67] N.N. Vorobyev. Extremal algebra of positive matrices. *Elektron. Informationsverarbeitung und Kybernetik*, 3:39–71, 1967. in Russian.

[Why] Why. `http://why.lri.fr`.

[Yen92] Hsu-Chun Yen. A unified approach for deciding the existence of certain petri net paths. *Inf. Comput.*, 96(1):119–137, 1992.

[Zie98] Gunter M. Ziegler. *Lectures on Polytopes*. Springer-Verlag New York Inc., second edition, 1998.

[Zie00] Gunter M. Ziegler. Lectures on 0/1-polytopes. In G. Kalai and G.M. Ziegler, editors, *Polytopes — combinatorics and computation (Oberwolfach, 1997)*, volume 29 of *DMV Seminar*, pages 1–41. Birkhäuser-Verlag, 2000.

[Zim77] K. Zimmermann. A general separation theorem in extremal algebras. *Ekonom.-Mat. Obzor*, 13(2):179–201, 1977.

APPENDIX A

Non-disjunctive numerical domain for array predicate abstraction

We present a numerical abstract domain to infer invariants on (a possibly unbounded number of) consecutive array elements using array predicates. It is able to represent and compute affine equality relations over the predicate parameters and the program variables, without using disjunctions or heuristics. It is the cornerstone of a sound static analysis of one- and two-dimensional array manipulation algorithms. The implementation shows very good performance on representative benchmarks. Our approach is sufficiently robust to handle programs traversing arrays and matrices in various ways.

A.1 Introduction

Program analysis now involves a large variety of methods able to infer complex program invariants, by using specific computer-representable structures, such as intervals [CC77], octagons [Min01b], linear (more exactly affine) equality constraints [Kar76], or affine inequality constraints [CH78]. Each abstract domain induces an equivalence relation: two abstract elements are equivalent if and only if they represent the same concrete elements. In this context, an *equivalence class* corresponds to a set of equivalent abstract elements, called *representatives*. Although all representatives are equivalent, they may not be identically treated by abstract operators or transfer functions, which implies that the choice of a "bad" represen-

```
1:  int i, n, p; bool t[n];
2:  assert 0 <= p <= n;
3:  i := 0;
4:  while i < n do
5:    t[i] := 0;
6:    i := i+1;
7:  done;
8:  while i > p do
9:    t[i-1] := 1;
10:   i := i-1;
11: done;
12:
```

Figure A.1: Incrementing then decrementing array manipulations

```
int i, n; bool t[n];
i := 0;
while i < n do
  t[i] := 0;
  i := i+1;
done;
while ... do
  if ... then
    write_one();
  else
    write_zero();
  end;
done;
```

```
write_one() {
  if i > 0 then
    t[i-1] := 1;
    i := i-1;
  end;
}

write_zero() {
  if i < n then
    t[i] := 0;
    i := i+1;
  end;
}
```

Figure A.2: Both incrementing and decrementing array manipulations. The notation ... stands for a non-deterministic condition.

tative may cause a loss of precision. Most numerical domains (for instance, reduced product [CC92a]) are provided with a reduction operator which associates each abstract element to a "good" equivalent element, which will allow gaining precision.

Unfortunately, in some abstract domains, it may not be possible to define a precise reduction operator, because for some equivalence classes, the notion of "good" representatives may depend on further analysis steps, or on parts of the program not yet analyzed. This difficulty appears in abstract domains based on universally quantified predicates ranging over (a possibly unbounded number of) consecutive array elements (first introduced in [Cou03]). The abstract elements of these domains consist of a predicate $\mathbf{p}$ and two parameters u and v: $\mathbf{p}(u, v)$ means that all the elements whose index is between u and v (both included) contain values for which the statement $\mathbf{p}$ holds. These predicates are then combined with classic numerical abstractions to bind their parameters to the values of the program variables.

Overview of the problem. As an example, let us try to analyze the first loop of the program given in Figure A.1, which initializes the array `t` with the boolean 0. For that purpose, we introduce the predicate **zero** (which means that the associated array elements contain the value 0), combined with the affine inequality domain. Informally, the loop invariant consists in joining the abstract representations Σ_k of the concrete memory states arising after exactly k loop iterations. For example, after one loop iteration ($k = 1$), the instruction `t[i] := 0` has assigned a zero to the array element of index 0, so that $\mathbf{zero}(u, v)$, with $u = v = 0$, $i = 1$ and $n \geq 1$. Similarly, after ten loop iterations, the ten first array elements have been initialized, thus $\mathbf{zero}(u, v)$, with $u = 0$, $v = 9$, $i = 10$ and $n \geq 10$. It can be shown that joining all the abstract states Σ_k with $k \geq 1$, *i.e.* which have entered the loop at least once, yields the invariant $\mathbf{zero}(u, v)$, with $u = 0$, $v = i - 1$, and $1 \leq i \leq n$. We now have to join this invariant with Σ_0 to obtain the whole loop invariant. The abstract state Σ_0 represents the concrete memory states which have not entered the loop. Since the array t is not initialized, Σ_0 is necessarily represented by a *degenerate* predicate, *i.e.* a predicate $\mathbf{zero}(l, m)$ such that $l > m$, which ranges over an empty set of array elements. Degenerate predicates naturally form an equivalence class, containing an infinite number of representatives, while

non-degenerate predicates form classes containing a unique representative. Now, choosing the degenerate predicate $\mathbf{zero}(u, v)$ with $u = 0$, $v = -1$, $i = 0$, and $n \geq 0$, to represent Σ_0, yields the expected loop invariant $u = 0$, $v = i$, and $0 \leq i \leq n$. On the contrary, if we choose $\mathbf{zero}(u, v)$ with $u = 10$, $v = 9$, $i = 0$, and $n \geq 0$, we obtain an invariant $\mathbf{zero}(u, v)$ with much less precise affine inequality relations, in which, in particular, the value of u is not known exactly anymore (it ranges between 0 and 10). Therefore, the representative $\mathbf{zero}(0, -1)$ is a judicious choice in the first loop analysis. But choosing the same representative for the second loop analysis will lead to a major loss of precision. The second loop partly initializes the array with the boolean 1 between from the index $n - 1$ to the index p. Using a predicate **one** to represent array elements containing the value 1, the analysis yields the expected invariant only if the representative $\mathbf{one}(t, s)$ with $t = n$ and $s = n - 1$ is chosen to represent the class of degenerate predicates **one**.

This example illustrates that the choice of right representatives for the degenerate classes to avoid loss of precision, is not an obvious operation, even for simple one-dimensional array manipulations. In [Cou03, GRS05], some solutions are proposed to overcome the problem: (i) use heuristics to introduce the right degenerate predicates. This solution is clearly well-suited for the analysis of programs involving very few different natures of loops, such as incrementing loops always starting from the index 0 of the arrays, but is not adapted for more complex array manipulations. In particular, we will see in Section A.4 that even classic matrix manipulation algorithms involve various different configurations for degenerate predicates. (ii) partition degenerate and non-degenerate predicates, instead of merging them in a single (and convex) representation. However such a disjunction may lead to an algorithmic explosion, since at least one disjunction has to be preserved for each predicate, including at control points located after loops: for example, the expected invariant at the control point 12 in Figure A.1 is $\mathbf{zero}(u, v) \wedge \mathbf{one}(s, t)$ with $u = 0$, $v = p-1$, $s = p$, and $t = n-1$. Without further information on n and p, this invariant contains non-degenerate and degenerate configurations of both predicates $\mathbf{zero}(u, v)$ and $\mathbf{one}(s, t)$. Partitioning these configurations yields the disjunction $(n = p = 0) \vee (n > p = 0) \vee (p = n > 0) \vee (0 < p < n)$. And, if the program contains instructions after control point 12, the disjunction must be propagated through the rest of the program analysis. Therefore, this approach may not scale up to programs manipulating many arrays.[1] (iii) partition traces [MR05a], for instance unroll loops, in order to distinguish traces in which non-degenerate predicates are inferred, from others. This solution is adapted to simple loops: as an example, for the loop located at control point 4 in Figure A.1, degenerate predicates occur only in the trace which does not enter the loop. But, in general, it may be difficult to automatically discover well-suited trace partitions: for example, in Figure A.2, traces in which the functions `write_one` and `write_zero` are called the same number of times, or equivalently, $i = n$, should be distinguished from others, since they contain a degenerate form of the predicate **one**. Besides, if traces are not ultimately merged, trace partitioning may lead to an algorithmic explosion for the same reasons as state partitions, while merging traces amounts to the problem of merging non-degenerate and degenerate predicates in a non-disjunctive way.

As we aim at building an efficient and automatic static analysis, we do not consider any existing solution as fully satisfactory.

[1] However, some techniques could allow merging disjunctions in certain cases. We will see at the end of Section A.3 that these techniques coincide with the join operation that we develop in this paper.

Contributions. We present a numerical abstract domain to be combined with array predicates. It represents sets of equivalence classes of predicates, by inferring affine invariants on some representatives of each class. In particular, the right representatives are automatically discovered, without any heuristics. As it is built as an extension of the affine equality constraint domain [Kar76, MOS04], it does not use any disjunctive representations. Several abstract transfer functions are defined, all are proven to be sound. This domain allows the construction and the implementation of a sound static analysis of array manipulations. It is adapted to array predicates ranging over the elements of one-dimensional or two-dimensional arrays. Our work does not focus on handling a very large and expressive family of predicates relative to the content of the array itself, but rather on the complexity due to the automatic discovery of affine relations among program variables and predicate parameters, hence of right representatives for degenerate predicates. Therefore, the analysis has been experimented on programs traversing arrays and matrices in various ways. In all cases, the most precise invariants are discovered, which proves the robustness of our approach.

Section A.2 presents the principles of the representation of equivalence classes of array predicates. Section A.3 introduces the domain of *formal affine spaces* to abstract sets of equivalence classes of array predicates by affine invariants on some of their representatives. In Section A.4, the construction of the array analysis and experiments are discussed. Finally, related work is presented in Section A.5.

A.2 Principles of the representation

As explained in Section A.1, array predicates are related by an equivalence relation, depending on their nature (degenerate or non-degenerate): for an one-dimensional array predicate $\mathbf{p}$, two representations $\mathbf{p}(u, v)$ and $\mathbf{p}(u', v')$ are equivalent if and only if both are degenerate, *i.e.* $u > v \wedge u' > v'$, or they are equal ($u = u' \wedge v = v'$). More generally, given predicates with p parameters, we assume that there exists an equivalence relation $\sim$ over $\mathbb{R}^{\mathsf{p}}$, defining the equivalence of two numerical p-tuples of predicate parameters.

Given a program with n scalar variables, a memory state can be represented by an element of $\mathbb{R}^{\mathsf{n}+\mathsf{p}}$, where each scalar variable is associated to one of the n first dimensions, and array predicate parameters are mapped to the p last ones. Then, the equivalence relation $\sim$ can be extended to $\mathbb{R}^{\mathsf{n}+\mathsf{p}}$ to characterize memory states which are provided with equivalent predicates: two memory states M, N in $\mathbb{R}^{\mathsf{n}+\mathsf{p}}$ are equivalent, which is denoted by $M \simeq N$, if and only if M and N coincide on their n first dimensions, and if the p-tuples formed by the p last dimensions are equivalent w.r.t. $\sim$. We adopt the notation $[M]$ to represent the equivalence class of M, *i.e.* the set of elements equivalent to M.

We have seen in Section A.1 that the representation of equivalence classes by arbitrarily-chosen representative elements may lead to a very complex invariant, possibly not precisely representable in classic numerical domains. Our solution consists in representing an equivalence class by a *formal representative* instead: it consists in a $(\mathsf{n} + \mathsf{p})$-tuple, whose n first coordinates contain values in $\mathbb{R}$, while the p last ones (related to predicate parameters) contain *formal variables*, taken in a given set $\mathcal{X}$. A formal representative R is provided with a set of valuations over $\mathcal{X}$: each valuation ν maps R to a point $R\nu$ of $\mathbb{R}^{\mathsf{n}+\mathsf{p}}$, by replacing each formal variable x in R by the value $\nu(x) \in \mathbb{R}$. Then, an equivalence class C can be represented by a formal representative R and a set of valuations V such that for any $\nu \in V$, the element $R\nu$ is in the class C. In other words, a formal representative can represent several elements

of a same equivalence class.

Let us illustrate the principle of formal representative with the program in Figure A.1, with $\mathsf{n} = 3$ scalar variables i, n, and p. Consider the equivalence class of a memory state at control point 4 which has not yet entered the loop, thus in which the predicate $\mathbf{zero}(u, v)$ is degenerate, and in which, for instance, $i = 0$, $n = 10$, and $p = 5$. It can be represented by the formal representative $R = (0, 10, 5, x, y)$ (written as a row vector for reason of space) and the set of valuations $V = \{\nu \mid \nu(x) > \nu(y)\}$: indeed, each representative $R\nu$ corresponds to a predicate $\mathbf{zero}(u, v)$ such that $u > v$. In that case, all the equivalent numerical configurations for the degenerate predicate $\mathbf{zero}(u, v)$ are represented in the formal representative.

Therefore, formal representatives allow keeping several representatives for a given class C instead on focusing on only one of them. In the following sections, we define *formal affine spaces*, which extend the affine equality domain to range over formal representatives. These formal affine spaces are combined with sets of valuations represented by affine inequality constraints over $\mathcal{X}$, giving the right values for the representatives. Besides, we describe how to compute the formal affine spaces, so as to automatically discover affine invariants on some representatives of distinct equivalence classes.

A.3 Formal affine spaces

We now formally introduce the abstract domain to represent sets of equivalence classes of array predicates. We follow the abstract interpretation methodology [CC77], by defining a concretization operator, and then abstract operators such as union.

Let Δ be the set of equivalence classes w.r.t the equivalence relation $\simeq$, and $\Delta(\mathcal{X})$ be the set of formal representatives. Formally, $\Delta(\mathcal{X})$ is isomorphic to the cartesian product of $\mathbb{R}^{\mathsf{n}}$, representing the set of memory states over scalar variables, with $\mathcal{X}^{\mathsf{p}}$. Given a formal representative M, $\pi_1(M)$ represent the n-tuple consisting in the n first coordinates. This element of $\mathbb{R}^{\mathsf{n}}$ is called the *real component* of M. Besides, the p last coordinates of M forms $\pi_2(M)$, called *formal component* of M. Similarly, the ith coordinate of M is said to be *real* (respectively *formal*) if $\mathsf{i} \leq \mathsf{n}$ (resp. $\mathsf{i} > \mathsf{n}$).

While the affine equality domain was initially introduced using conjunctions of equality constraints [Kar76], affine spaces can be represented by means of generators as well [MOS04]. An *affine generator system* $E + \Omega$ is given by a family $E = (e_i)_{1 \leq i \leq s}$ of linearly independent vectors of $\mathbb{R}^{\mathsf{n}}$, and a point $\Omega \in \mathbb{R}^{\mathsf{n}}$. It is associated to the affine space defined by:

$$\mathsf{Span}(E + \Omega) = \left\{ \Omega + \sum_{i=1}^{s} \lambda_i e_i \mid \lambda_1, \ldots, \lambda_s \in \mathbb{R} \right\} , \tag{A.1}$$

corresponding to the set of the points generated by the addition of linear combinations of the vectors e_i to the point Ω. Affine generator systems are equivalent to sets of affine constraints. Indeed, the elimination of the λ_i in the combinations given in (A.1) yields an equivalent set of affine constraints over the coordinates of the points.

Formal affine spaces are defined by extending affine generator systems of $\mathbb{R}^{\mathsf{n}}$ with p formal coordinates: generators are now elements of $\Delta(\mathcal{X})$, provided with a set of valuations.

Definition A.1. A *formal affine space* $E + \Omega \natural V$ is given by a family $E = (e_1, \ldots, e_s)$ of vectors of $\Delta(\mathcal{X})$, a point Ω of $\Delta(\mathcal{X})$ verifying:

- the $(\pi_1(e_i))_{1 \leq i \leq s}$ are linearly independent,

$$\begin{array}{l}\pi_1\left\{\\ \pi_2\left\{\end{array}\right.\left[\left(\begin{array}{c}0\\1\\0\\x_1\\x_2\end{array}\right),\left(\begin{array}{c}0\\0\\1\\y_1\\y_2\end{array}\right)\right]+\left(\begin{array}{c}0\\0\\0\\z_1\\z_2\end{array}\right)\ \vdots\ V$$

where $V = \{x_1 = x_2 \wedge y_1 = y_2 \wedge z_1 > z_2\}$

$$\begin{array}{l}\pi_1\left\{\\ \pi_2\left\{\end{array}\right.\left[\left(\begin{array}{c}0\\1\\0\\x'_1\\x'_2\end{array}\right),\left(\begin{array}{c}0\\0\\1\\y'_1\\y'_2\end{array}\right)\right]+\left(\begin{array}{c}1\\0\\0\\z'_1\\z'_2\end{array}\right)\ \vdots\ V'$$

where $V' = \{x'_1 = x'_2 = 0 \wedge y'_1 = y'_2 = 0 \wedge z'_1 = z'_2 = 0\}$

Figure A.3: Two formal affine spaces for $\mathsf{n} = 3$ and $\mathsf{p} = 2$

- any two formal variables occurring in $(\pi_2(e_i))_i$ and $\pi_2(\Omega)$ are distinct,

and an affine inequality constraint system V over the formal variables occurring in $(\pi_2(e_i))_i$ and $\pi_2(\Omega)$.

Figure A.3 gives an example of formal affine spaces. We abusively denote by $\nu \in V$ the fact that the valuation ν satisfies the constraint system V. Similarly to "classic" affine generator systems, a formal affine space $E + \Omega \ \vdots\ V$ generates a set of formal representatives, written as combinations $\Omega + \sum_i \lambda_i e_i$. As explained in Section A.2, each formal representative R, provided with the set of valuations satisfying V, represents a set of several representatives which belong to a same equivalence class C: for any $\nu \in V$, $C = [R\nu]$. Following these principles, the concretization operator γ maps any formal space $E + \Omega \ \vdots\ V$ to the set of the equivalence classes represented by the generated formal representatives:

$$\gamma(E + \Omega \ \vdots\ V) \stackrel{def}{=} \{C \mid R \in \mathsf{Span}(E + \Omega) \wedge \forall \nu \in V.\ C = [R\nu]\} \quad , \tag{A.2}$$

where $\mathsf{Span}(E + \Omega)$ consists of the combinations $\Omega + \sum_{i=1}^{s} \lambda_i e_i$, for $\lambda_i \in \mathbb{R}$.

Example A.1. Consider the formal affine space $E + \Omega \ \vdots\ V$ on the left-hand side of Figure A.3. Any combination in $\mathsf{Span}(E + \Omega)$ is a formal representative R of the form $(0, \lambda, \mu, \lambda x_1 + \mu y_1 + z_1, \lambda x_2 + \mu y_2 + z_2)$ (written as a row vector for reason of space) where $\lambda, \mu \in \mathbb{R}$. Suppose that the dimensions respectively represent the scalar variables i, n, p, and the parameters u and v of a predicate $\mathbf{zero}(u, v)$. Then R represents the equivalence classes of memory states in which $i = 0$, n and p have independent values, and for any valuation $\nu \in V$,

$$u = \lambda\nu(x_1) + \mu\nu(y_1) + \nu(z_1) > \lambda\nu(x_2) + \mu\nu(y_2) + \nu(z_2) = v \quad , \tag{A.3}$$

or equivalently, the predicate $\mathbf{zero}(u, v)$ is degenerate. In particular, $E + \Omega \ \vdots\ V$ allows abstracting the memory states at control point 4 in Figure A.1 which have not yet entered the loop. Besides, it represents several representatives for the degenerate predicate $\mathbf{zero}(u, v)$, while a "classic" affine invariant would select only one of them. Similarly, the formal affine space $F + \Omega' \ \vdots\ V'$ on the right-hand side of Figure A.3 yields formal representatives R' corresponding to classes of memory states such that $i = 1$, n and p are arbitrary, and $u = v = 0$, since for $i \in \{1, 2\}$, $\lambda\nu'(x'_i) + \mu\nu'(y'_i) + \nu'(z'_i) = 0$ for any valuation $\nu' \in V'$. Then, it is an abstraction of the memory states after the first iteration of the first body loop in Figure A.1: the first element of the array t (index 0) contains the value 0. □

A.3.1 Joining two formal spaces

We wish to define a union operator $\sqcup$ which provides an over-approximation of two formal affine spaces $E + \Omega \restriction V$ and $F + \Omega' \restriction V'$. Let us illustrate the intuition behind the definition of $\sqcup$ by sufficient conditions.

Suppose that $G + O \restriction W$ is the resulting formal space. A good start is to require $\sqcup$ to be sound w.r.t. the underlying real affine generator systems: if $\pi_1(G + O)$ denotes the real affine generator system obtained by applying π_1 on each vector g_i of G and on the origin, then $\pi_1(G+O)$ has to represent a larger affine space than those generated by $\pi_1(E+\Omega)$ and $\pi_1(F+\Omega')$. To ensure this condition, let us build $G+O \restriction W$ by extending the *sum system* of the two real systems $\pi_1(E+\Omega)$ and $\pi_1(F+\Omega')$.[2] More precisely, if $G_r + O_r$ denotes the sum system, we add p fresh formal variables to each vector of G_r and to O_r, which yields $G + O$.

Then, to ensure $\gamma(E+\Omega \restriction V) \subset \gamma(G+O \restriction W)$, we require $\mathsf{Span}(E+\Omega)$ to be "included" in $\mathsf{Span}(G + O)$. Although the inclusion already holds for their real components ($\mathsf{Span}(\pi_1(E + \Omega)) \subset \mathsf{Span}(\pi_1(G + O))$), $\mathsf{Span}(E + \Omega)$ and $\mathsf{Span}(G + O)$ can not be directly compared since they may contain different formal variables. Therefore, we build a substitution σ_P over the formal variables occurring in $\pi_2(E + \Omega)$, such that for any $R \in \mathsf{Span}(E + \Omega)$, we have $R\sigma_P \in \mathsf{Span}(G+O)$. This substitution is induced by the *change-of-basis matrix* P from $\pi_1(E+\Omega)$ to $\pi_1(G+O)$, which verifies $\mathsf{mat}(\pi_1(E+\Omega)) = \mathsf{mat}(\pi_1(G+O)) \times P$ ($\mathsf{mat}(\pi_1(E+\Omega))$ is the matrix whose columns are formed by the vectors $(\pi_1(e_i))_i$ and $\pi_1(\Omega)$). The matrix P expresses the coefficients of the (unique) decomposition of each $\pi_1(e_i)$ and $\pi_1(\Omega)$ in terms of the $\pi_1(O)$ and $(\pi_1(g_k))_k$. It allows to express the $\pi_2(e_i)$ and $\pi_2(\Omega)$ in terms of the $\pi_2(O)$ and $(\pi_2(g_k))_k$ as well, by defining σ_P by $\sigma_P(\mathsf{mat}(\pi_2(E + \Omega))) \stackrel{def}{=} \mathsf{mat}(\pi_2(G + O)) \times P$.

Now, it suffices that W be a stronger system of constraints than $V\sigma_P$, the system obtained by applying the substitution σ_P on V. Indeed, for any class $C \in \gamma(E + \Omega \restriction V)$, there exists $R \in \mathsf{Span}(E + \Omega)$ such that for any $\nu \in V$, $C = [R\nu]$. Then, for any $\nu' \in W$, we have $\nu' \in V\sigma_P$, so that there exists a valuation $\nu \in V$ such that $\forall x.(\sigma_P(x))\nu' = \nu(x)$. This implies $(R\sigma_P)\nu' = R\nu$, hence $C = [(R\sigma_P)\nu']$. A similar reasoning can be performed for $F + \Omega' \restriction V'$, which leads to the following definition of $\sqcup$:

Definition A.2. The union $(E+\Omega \restriction V) \sqcup (F+\Omega' \restriction V')$ is defined as the formal space $G+O \restriction W$ where $\pi_1(G+O)$ is the sum of $\pi_1(E+\Omega)$ and $\pi_1(F+\Omega')$, yielding two change-of-basis matrices P and Q respectively, and W is the conjunction of the two systems of constraints $V\sigma_P$ and $V'\sigma_Q$.

The following proposition states that the union operator is sound.

Proposition A.1. *The union $(E + \Omega \restriction V) \sqcup (F + \Omega' \restriction V')$ over-approximates the union of the sets of classes represented by $E + \Omega \restriction V$ and $F + \Omega' \restriction V'$.*

Example A.2. Consider the formal spaces $E + \Omega \restriction V$ and $F + \Omega' \restriction V'$ introduced in Ex. A.1. The sum of the two real affine generator systems $\pi_1(E + \Omega)$ and $\pi_1(F + \Omega')$ is a system in

[2] The *sum system* is obtained by extracting a free family G_r from the vectors $(\pi_1(e_i))_i$, $(\pi_1(f_i))_j$, and $\pi_1(\Omega') - \pi_1(\Omega)$, and choosing $O_r = \pi_1(\Omega)$. Then, $G_r + O_r$ generates the smallest affine space greatest than the affine spaces represented by both $\pi_1(E + \Omega)$ and $\pi_1(F + \Omega')$.

$$P = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad Q = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \forall i \in \{1,2\}. \begin{cases} \sigma_P(x_i) \mapsto \mathbf{y}_i \\ \sigma_P(y_i) \mapsto \mathbf{z}_i \\ \sigma_P(z_i) \mapsto \mathbf{t}_i \end{cases} \quad \begin{cases} \sigma_Q(x'_i) \mapsto \mathbf{y}_i \\ \sigma_Q(y'_i) \mapsto \mathbf{z}_i \\ \sigma_Q(z'_i) \mapsto \mathbf{t}_i + \mathbf{x}_i \end{cases}$$

Figure A.4: Change-of-basis matrices and their associated substitutions

which i, n, and p are all independent, so that:

$$G + O \overset{def}{=} \begin{matrix} i \\ n \\ p \\ u \\ v \end{matrix} \left[\begin{pmatrix} 1 \\ 0 \\ 0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ \mathbf{z}_1 \\ \mathbf{z}_2 \end{pmatrix} \right] + \begin{pmatrix} 0 \\ 0 \\ 0 \\ \mathbf{t}_1 \\ \mathbf{t}_2 \end{pmatrix} \quad . \tag{A.4}$$

The corresponding change-of-basis matrices P and Q are given in Figure A.4. In particular, these matrices represent the relation $\pi_1(\Omega') = \pi_1(O) + \pi_1(g_1)$, which generates the substitutions $z'_1 \mapsto \mathbf{t}_1 + \mathbf{x}_1$ and $z'_2 \mapsto \mathbf{t}_2 + \mathbf{x}_2$. The associated substitutions σ_P and σ_Q are then defined in Figure A.4. Applying them on the constraint systems V and V' yields: $V\sigma_P = \{\mathbf{y}_1 = \mathbf{y}_2 \wedge \mathbf{z}_1 = \mathbf{z}_2 \wedge \mathbf{t}_1 > \mathbf{t}_2\}$ and $V'\sigma_Q = \{\mathbf{y}_1 = \mathbf{y}_2 = 0 \wedge \mathbf{z}_1 = \mathbf{z}_2 = 0 \wedge \mathbf{t}_1 + \mathbf{x}_1 = \mathbf{t}_2 + \mathbf{x}_2 = 0\}$, so that:

$$W = \{\mathbf{x}_1 = -\mathbf{t}_1 \wedge \mathbf{x}_2 = -\mathbf{t}_2 \wedge \mathbf{y}_1 = \mathbf{y}_2 = 0 \wedge \mathbf{z}_1 = \mathbf{z}_2 = 0 \wedge \mathbf{t}_1 > \mathbf{t}_2\} \quad . \tag{A.5}$$

It can be intuitively verified that $G + O \restriction W$ contains the formal spaces $E + \Omega \restriction V$ and $F + \Omega' \restriction V'$:

- when $i = 0$, we have $u = \mathbf{t}_1 + \lambda\mathbf{y}_1 + \mu\mathbf{z}_1$ and $v = \mathbf{t}_2 + \lambda\mathbf{y}_2 + \mu\mathbf{z}_2$ for some $\lambda, \mu \in \mathbb{R}$, so that for any $\nu \in W$, $u\nu = \nu(\mathbf{t}_1) > \nu(\mathbf{t}_2) = v\nu$. Then the predicate $\mathbf{zero}(u, v)$ is degenerate.
- when $i = 1$, we have $u = \mathbf{t}_1 + \mathbf{x}_1 + \lambda\mathbf{y}_1 + \mu\mathbf{z}_1$ and $v = \mathbf{t}_2 + \mathbf{x}_2 + \lambda\mathbf{y}_2 + \mu\mathbf{z}_2$, hence $u\nu = v\nu = 0$ for any valuation $\nu \in W$. In that case, the predicate $\mathbf{zero}(u, v)$ ranges over the first element of the array.

The resulting formal space $G + O \restriction W$ is an over-approximation of the memory states arising at control point 4 in Figure A.1, after at most one loop iteration.

We could show that joining $E + \Omega \restriction V$ with the formal space resulting from the loop body execution on $G + O \restriction W$, yields the affine space $G + O \restriction W'$, where $W' = \{\mathbf{x}'_1 = 0 \wedge \mathbf{x}'_2 = 1 \wedge \mathbf{y}'_1 = \mathbf{y}'_2 = 0 \wedge \mathbf{z}'_1 = \mathbf{z}'_2 = 0 \wedge \mathbf{t}'_1 = 0 \wedge \mathbf{t}'_2 = -1\}$. It could be also verified that this affine space is a fixpoint of the loop transfer function. It represents the expected invariant $u = 0$ and $v = i - 1$. In particular, the computation automatically discovers the right representative $\mathbf{zero}(0, -1)$ (obtained with $i = 0$) among all the representatives $\mathbf{zero}(u, v)$ such that $u > v$ contained in $E + \Omega \restriction V$. □

Definition A.2 and Ex. A.2 raise some remarks. Firstly, when considering increasing formal affine spaces, the underlying real affine generators are logically growing, while the sets of valuations become smaller (the constraint system becomes stronger). Intuitively, this corresponds to an increasing determinism in the choice of the representatives in the equivalence

classes abstracted by the formal space. In particular, when considering formal spaces obtained by iterating an increasing transfer function to compute a global invariant, two cases (among possibly more) are singular: when the set of valuations is reduced to a singleton, and when this set is empty. In the former, the formal affine space coincide with an affine generator system over $\mathbb{R}^{\mathsf{n}+\mathsf{p}}$: in other words, some representatives in the over-approximated equivalence classes are bound with program variables by an affine invariant. This situation happens at the end of Ex. A.2, in which $u = 0$ and $v = i - 1$ in the affine space over-approximating the loop invariant. In the latter case, the discovery of an affine invariant failed: by definition of γ, the concretization of the formal space is the entire set Δ.

Secondly, consider the two abstract memory states that we tried to join in Section A.1 to compute an invariant of the first loop in Figure A.1: on the one hand, a degenerate predicate $\mathbf{zero}(u, v)$ with $i = 0$, and on the other hand, a non-degenerate one $\mathbf{zero}(u, v)$ with $u = 0$, $v = i - 1$, and $1 \leq i \leq n$. We could verify that joining the two representations by means of formal spaces, and in particular, computing the conjunction of the two corresponding constraint systems $V\sigma_P$ and $V'\sigma_Q$, exactly amounts to check whether the affine relations $u = 0$ and $v = i - 1$ match the degenerate condition $u > v$ when $i = 0$. More generally, when it succeeds, the approach based on matching degenerate condition coincides with the operations performed when joining two formal spaces. The major advantage of formal affine spaces is that it is adapted to any program or coding style, while matching degenerate conditions may fail. For example, let us consider the piece of program `i := n-1; if ... then t[i] := 0; i := i-1; fi;`. The matching approach would check if the non-degenerate invariant $\mathbf{zero}(u, v) \wedge u = v = i + 1 = n - 1$ match the degenerate condition when $i = n - 1$, which is obviously false.

A.3.2 Precision and further abstract operators

All usual abstract operators can be defined on formal affine spaces. For reason of space, we only give an enumeration. First, a partial order $\sqsubseteq$, defined in a similar way to the union operator, can be introduced. Then, the concretization γ can be shown to be monotonic, and the union $\sqcup$ is the best possible join operator w.r.t. the order $\sqsubseteq$. Furthermore, the definition of guard, constraint satisfiability, and assignment operations closely follows the definition of the same primitives on real affine generator systems [Kar76, MOS04], thus their design is simple. The main difference is that guards, satisfiability and assignments over predicate parameters involve operations on both the family of generators and the system of constraints representing the sets of valuations. For the latter, only usual operators, such as assignments or extracting a valuation satisfying the set of constraints, are necessary. All the operators on formal affine spaces are proven to be conservative. Moreover, exactness holds for guards, satisfiability, and invertible assignments, when they are applied to a formal affine space whose system of constraints representing the valuations is satisfiable.

A.4 Application to the analysis of array manipulations

Formal affine spaces has been implemented to analyze array manipulation programs. The analysis computes abstract memory states consisting in a finite sequence of predicates, and a formal affine space over the program variables and the predicate parameters. Note that a reduced product of formal affine spaces with convex polyhedra [CH78] over scalar variables

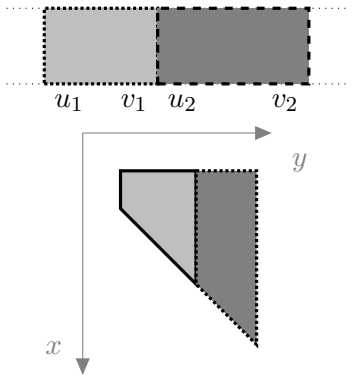


Figure A.5: Merging two contiguous predicates

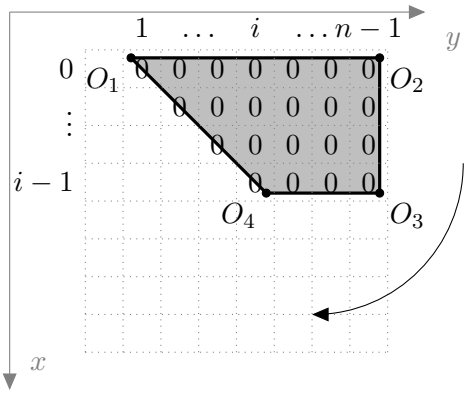


Figure A.6: Example of a two-dimensional predicate

is used to increase precision, since affine generator systems do not precisely handle inequality guards.

Array assignments (*i.e.* assignments of the form `t[i] := e`) introduce new predicates in the abtract state (intuitively, non-degenerate predicates of the form $\mathbf{p}(u, v)$ with $u = v = i$). Then, some predicates may represent contiguous memory areas of a same array, and thus can be merged in a single predicate. The situations in which two predicates $\mathbf{p}$ and $\mathbf{q}$ can be merged correspond to simple geometric configurations. Two of these configurations for one- and two-dimensional are depicted respectively at the top and the bottom of Figure A.5. All these situations can be expressed as conjunctions of affine equality constraints over the parameters of the two predicates. When these constraints are satisfied, a new predicate $\mathbf{p} \curlyvee \mathbf{q}$ is introduced in the abstract state. The statement $\mathbf{p} \curlyvee \mathbf{q}$ itself over-approximates $\mathbf{p}$ and $\mathbf{q}$: it expresses a property on the values of the array element which is weaker than those expressed by $\mathbf{p}$ and $\mathbf{q}$. And its parameters are initialized to fit the whole area obtained by concatenating the memory areas corresponding to $\mathbf{p}$ and $\mathbf{q}$. Finally, the predicates $\mathbf{p}$ and $\mathbf{q}$ and their parameters are removed from the abstract state.

One-dimensional Predicates. Two kinds of predicates are used to analyze array manipulations, depending on the type of arrays.

For arrays whose elements take their values in a finite set of cardinal K (such as booleans or C enumerations), we consider one predicate $\mathbf{c}$ per possible value. Then $\mathbf{c}(u, v)$ states that the array contains the value c between the indices u and v. We allow at most K pairwise distinct predicates $\mathbf{c}_1, \ldots, \mathbf{c}_K$ per array. The merging operations are applied only to predicates representing the same constant. Besides, if two predicates $\mathbf{c}(u_1, v_1)$ and $\mathbf{c}(u_2, v_2)$ ranging over the same array can not be merged, they are simply removed from the abstract state. Although this choice is very strict, it offers a tractable analysis, which is precise enough to handle the examples given in Figs. A.1 and A.2, as reported in Table A.1.

For integer arrays, conjunctions of interval and bounded difference constraints (*i.e.* of the form $c_1 \leq x \leq c_2$ or $c_1 \leq x - y \leq c_2$) between the array content and the scalar variables are used. For instance, the predicate $\langle 0 \leq t \leq n-1 \rangle\,(u, v)$ represents the fact that the elements of the array t located between the indices u and v all contain values between 0 and $n-1$ (n being

Table A.1: Analysis benchmarks

Programs	Invariants (by default, at the end of the program)	Time
Figure A.1	$\mathbf{zero}(0, p-1) \wedge \mathbf{one}(p, n-1)$	$\sim 0.6\,\mathrm{s}$
Figure A.2	outer loop invariant: $\mathbf{zero}(0, i-1) \wedge \mathbf{one}(i, n-1)$	$\sim 0.7\,\mathrm{s}$
`full_init`	i. and d. $\langle 0 \leq t \leq n-1 \rangle\,(0, n-1)$	$< 0.2\,\mathrm{s}$
`range_init`	i. and d. $\langle p \leq t \leq q-1 \rangle\,(p, q-1)$	$< 0.2\,\mathrm{s}$
`partial_init`	i. $\langle 0 \leq t \leq n-1 \rangle\,(0, j-1)$ and d. $\langle 0 \leq t \leq n-1 \rangle\,(j, n-1)$	$\sim 0.2\,\mathrm{s}$
`partition`	i. $\langle ge \geq 0 \rangle\,(0, gelen-1) \wedge \langle lt \leq -1 \rangle\,(0, ltlen-1)$	$\sim 0.4\,\mathrm{s}$
	d. $\langle ge \geq 0 \rangle\,(gelen, n-1) \wedge \langle lt \leq -1 \rangle\,(ltlen, n-1)$	$\sim 0.5\,\mathrm{s}$
`full_matrix`	r. $\langle m = 0 \rangle\,((0,0), (0, n-1), (n-1, n-1), (n-1, 0))$	12.9 s
	c.. $\langle m = 0 \rangle\,((n-1, 0), (0,0), (0, n-1), (n-1, n-1))$	13.4 s
`lower_triang`	r. $\langle m = 0 \rangle\,((0,1), (0, n-1), (i-1, n-1), (i-1, i))$	12.6 s
(outer loop	c. $\langle m = 0 \rangle\,((0,1), (0,1), (0, j-1), (j-2, j-1))$	14.7 s
invariants)	dg. $\langle m = 0 \rangle\,((0,1), (0, k-1), (n-k, n-1), (n-2, n-1))$	11.3 s
`upper_triang`	r. $\langle m = 0 \rangle\,((1,0), (1,0), (i-1, i-2), (i-1, 0))$	14.6 s
(outer loop	c. $\langle m = 0 \rangle\,((n-1, 0), (1,0), (j, j-1), (n-1, j-1))$	13.1 s
invariants)	dg. $\langle m = 0 \rangle\,((n-1, 0), (n-k+1, 0), (n-1, k-2), (n-1, 0))$	15.0 s

a program scalar variable). Such predicates are implemented under the form of $\mathsf{n}+1$ intervals: one to bound the array values in an interval, n to bound the differences with the n scalar variables. Then, the analysis allows at most one predicate per array. If a predicate associated to an array is introduced during the computation while this array already has a predicate, both are merged if possible, or simply removed if not. Moreover, to ensure termination, the statement $\mathbf{p} \curlyvee \mathbf{q}$ is obtained by pointwise widening the intervals contained in $\mathbf{p}$ and $\mathbf{q}$.

Two-dimensional Predicates. We use two-dimensional predicates which range over convex quadrilateral areas of two-dimensional arrays. Predicates are of the form $\mathbf{p}(O_1, O_2, O_3, O_4)$, and have now eight parameters, corresponding the x- and y-coordinates of the associated vertices O_1, O_2, O_3, and O_4. Degenerate and non-degenerate predicates are distinguished by the rotation direction of the points O_1, O_2, O_3, and O_4. We use the convention that the interior of the polygon $O_1O_2O_3O_4$ is not empty if and only if O_1, O_2, O_3, and O_4 are ordered clockwise, as in Figure A.6. The shape of the polygons $O_1O_2O_3O_4$ is restricted by requirements, not fully detailed here, but implying in particular that the coordinates of the O_i are integer, and the lines (O_iO_{i+1}) are either horizontal, vertical, or diagonal. These requirements are weak enough to express the invariants used in the targeted algorithms. Moreover, they allow characterizing degenerate polygons by a condition consisting of several affine inequalities over the predicate parameters.

The analysis allows for each matrix at most two predicates: one is one-dimensional, while the other is two-dimensional. Indeed, the matrix algorithms we wish to analyze performs intermediate manipulations on rows, columns, or diagonals. Thus, the former predicate is used to represent the invariant on the current one-dimensional structure, while the latter collects the information on the older structures, which form a two-dimensional shape. The predicates propagate bounded difference constraints relative to the matrix content.

Benchmarks. Table A.1 reports the invariants discovered by our analyzer, implemented in Objective Caml (5000 lines of code), and the time taken for each analysis on a 1 Gb RAM laptop using one core of a 2 GHz Intel Pentium Core Duo processor. The first six programs involve only one-dimensional arrays. The two first programs are successfully analyzed us-

ing constant predicates, and the right array shape is discovered. The third one, `full_init`, initializes each element `t[i]` of the array `t` of size `n` with the value `i`. It results in a fully initialized array with values ranging between 0 and $n-1$. The program `range_init` has a similar behavior, except that it performs the initialization between the indices `p` and `q` only. The programs `partial_init` and `partition` are taken from [GRS05] and [BHMR07] respectively. The former copies the value `i` in `t[j]` when the values of two other arrays `a[i]` and `b[i]` are equal, and then increments `j`. The latter partitions the positive or null and strictly negative values of a source array `a` into the destination arrays `ge` and `lt` respectively. The three last programs involve matrices. The first one, `full_matrix`, fully initializes a matrix `m` of size $n \times n$. The two last ones only fill the upper- and lower-triangular part of the matrix with the value 0. Each program contains two nested loops. As an illustration, the invariant of the outer loop of the column-after-column version of `lower_triang` discovered by the analysis is given in Figure A.6. The reader can verify that the final invariant obtained for $i = n-1$ corresponds to a lower-triangular matrix. Several versions of each program are analyzed: for one-dimensional array manipulation algorithms, incrementing (i.) or decrementing (d.) loops (except for the programs in Figs. A.1 and A.2 which already use both versions of loops), and for matrix manipulation loops, row-after-row (r.), column-after-column (c.), or diagonal-after-diagonal (dg.) matrix traversal.[3] All the examples involving one-dimensional arrays only are successfully analyzed in less than a second. Analysis time does not exceed 15 s on programs manipulating matrices, which is a good result, considering the complexity of the merge conditions for two-dimensional predicates, and the fact that these programs contain nested loops. These benchmarks show that the analysis is sufficiently robust to discover the excepted invariant for several stategies of array or matrix manipulations programs. In particular, the right representatives for degenerate predicates are automatically found out in various and complex situations. As an example, the degenerate predicates discovered for the programs `lower_triang` (obtained with $i = 0$, $j = 1$, and $k = 1$) and `upper_triang` (obtained with $i = 1$, $j = 0$, and $k = 1$) all represent different configurations of interior-empty quadrilateral shapes. Furthermore, although not reported in Table A.1, the analysis handles simple transformations (such as loop unrolling) on the experimented programs, without any loss of precision. Finally, for one-dimensional predicates, we have experimented, with formal affine spaces, the manual substitution of the general degenerate condition $u > v$ by the right degenerate configurations for each program. In that case, operations on formal affine spaces roughly coincide with operations in a usual equality constraint domain. We have found that the additional cost in time due to formal affine spaces is small (between 8% and 30%), which suggests that this numerical abstract domain has good performance, while it automatically discovers the right representatives.

A.5 Related work

Several static analyses use predicates to represent memory shape properties: among others, [SRW99, BMMR01, DOY06, BCC+07b, BHT06] infer elaborate invariants on dynamic memory structures, such as lists and trees. Most of these works do not involve a precise treatment of arrays. Some abstract interpretation based analyzers [VB04, CCF+05, AGH06] precisely handle manipulations of arrays whose size is exactly known. Besides, [CCF+05] can represent

[3] The source code of each program is available at `http://www.lix.polytechnique.fr/Labo/Xavier.Allamigeon`.

all the array elements by a single abstract element (*array smashing*). Albeit not very precise, it could also represent an unbounded number of array elements.

To our knowledge, only [FQ02, Cou03, GRS05, JM07, BHMR07, GMT08] handle precise properties ranging over an unbounded number of one-dimensional array elements. Most of them involve the predicates presented in this paper, and some other expressing more properties on the values of the array elements, such as equality, sorting or pointer aliasing properties. The approach of [FQ02, JM07, BHMR07] differs ours in the use of a theorem prover in order to abstract reachable states in [FQ02], and of counterexample-guided abstraction refinement in [JM07, BHMR07]. They share with our analysis common benchmarks: for example, [JM07, BHMR07] analyzes the program `full_init` in respectively 1.190 s and 0.27 s, and `partition` in 7.960 s and 3.6 s.[4] The returned invariants are the same as those given in Table A.1. The other works [Cou03, GRS05, GMT08] use the abstract interpretation framework. The analysis developed in [GMT08] involves predicates on arrays and lists, and allows expressing invariants of the form $E \wedge \bigwedge_j \forall U_j(F_j \Rightarrow e_j)$, where E, F_j and e_j are quantifier-free facts. This approach is more general than ours, since it automatically discovers universally quantified predicates, while we explicitely define the family of predicates (uni- or two-dimensional) in our analysis. The drawback is that it requires under-approximation abstract domains and associated operators because of the universal quantification. In constrast, our concretization operator (defined in (A.2)) involves a universal quantifier over valuations $\nu \in V$, which can be shown to commute with the existential quantifier $\exists R \in \mathsf{Span}(E + \Omega)$. Then, *exact* operations on the inequality constraint systems representing the valuations, such as intersections or assignments, yield sound and precise results (see Section A.3.2). In [GMT08], `full_init` and `partition` are respectively analyzed in 3.2 s and 73.0 s on a 3 GHz machine, yielding the same invariants than with our analysis. In [Cou03], semantic loop unrolling and introduction by heuristics of well-chosen degenerate predicates (called *tautologies*) are combined. It handles array initialization algorithm (the exact nature of the algorithm, partial, incrementing, decrementing, *etc*, is not mentionned), and bubble sort and QuickSort algorithms. In [GRS05], array elements are distinguished according to their position w.r.t. to the current loop index (strictly before, equal to, or strictly after). This yields a partition of the memory configurations into distinct categories, which are characterized by the presence or the absence of array elements having a certain position w.r.t. to a loop index. The program `partial_init` is analyzed in 40 s on a 2.4 GHz machine, and yields a partition of four memory configurations corresponding to the invariant given in Table A.1. Finally, as far as we know, no existing work reports any experiments on two-dimensional array manipulation programs.

A.6 Conclusion

We have introduced a numerical abstract domain which allows to represent sets of equivalence classes of predicates, by inferring affine invariants on some representatives of each class, without any heuristics. Combined with array predicates, it has been experimented in a sound static analysis of array and matrix manipulation programs. Experimental results are very good, and the approach is sufficiently robust to handle several array traversing stategies. Future work will focus on the extension of the abstraction to other systems of generators, such as convex polyhedra, in order to incorporate the reduced product implemented in the analysis into the abstraction of equivalence classes.

[4] A 1.7 GHz machine was used in both works.

APPENDIX $\mathcal{B}$

Additional proofs

Proof of Proposition 4.4. Let $\mathcal{H}' = f(\mathcal{H})$. Suppose that $s \rightsquigarrow_{\mathcal{H}} t$. Observe that if X, Y are subsets of N, $f(X) \subset f(Y)$ as soon as $X \subset Y$, and $f(X \cup Y) \subset f(X) \cup f(Y)$. Therefore, if $e_1, \ldots, e_p$ is a hyperpath from s to t, then:

$$\begin{aligned} T(e_i) &\subset \{\, s \,\} \cup H(e_1) \cup \cdots \cup H(e_{i-1}) && \text{for all } 1 \leq i \leq p \\ t &\in H(e_p) \end{aligned}$$

so that:

$$\begin{aligned} f(T(e_i)) &\subset \{\, f(s) \,\} \cup f(H(e_1)) \cup \cdots \cup f(H(e_{i-1})) && \text{for all } 1 \leq i \leq p \\ f(t) &\in f(H(e_p)) \end{aligned}$$

It follows that $f(s) \rightsquigarrow_{f(\mathcal{H})} f(t)$.

Conversely, suppose that $f(t)$ is reachable from $f(s)$ in $\mathcal{H}'$, and that $f(t) \neq f(s)$ (the case $f(t) = f(s)$ is trivial). Let $H_0 = \{\, s \,\}$ and $T_{p+1} = \{\, t \,\}$.

By definition, there exist $e_1 = (T_1, H_1), \ldots, e_p = (T_p, H_p)$ in E such that for each $i \in \{\, 1, \ldots, p+1 \,\}$, $f(T_i) \subset f(H_0) \cup \cdots \cup f(H_{i-1})$.

Also note that for any $s \in \wp(N)$, $f(s) = s$ in $s \cap \{\, x, y \,\} = \emptyset$ and $f(s) = s \cup \{\, z \,\} \setminus \{\, x, y \,\}$ otherwise. In particular, as soon as $z \notin f(s)$, $f(s)$ coincides with s. Besides, $f(s) \setminus \{\, z \,\} \subset s \subset f(s) \setminus \{\, z \,\} \cup \{\, x, y \,\}$.

Two cases can be distinguished:

(a) suppose that z does not belong to any $f(H_j)$, so that $f(H_j) = H_j$. Similarly, for each

$i \geq 1$, $f(T_i)$ does not contain z, hence $f(T_i) = T_i$. Besides, $T_i \subset H_0 \cup \cdots \cup H_{i-1}$ for each i, so that is is straightforward that $f(s) \rightsquigarrow_{\mathcal{H}'} f(t)$.

(b) now, if z in one of the $f(H_j)$, let k be the smallest integer such that $z \in f(H_k)$. Say for instance that $x \in H_k$. Let $(T'_1, H'_1), \ldots, (T'_q, H'_q)$ be taken from a hyperpath from x to y in $\mathcal{H}$.

When $i \leq k$, $f(T_i)$ does not contain z, hence $f(T_i) = T_i$ and $T_i \subset f(H_0) \cup \cdots \cup f(H_{i-1}) = H_0 \cup \ldots \cup H_{i-1}$.

Besides, $T'_1 = \{ x \} \subset H_0 \cup \ldots \cup H_k$, and for each $i \in \{ 2, \ldots, q \}$, $T'_i \subset H_0 \cup \cdots \cup H_k \cup H'_1 \cup \cdots \cup H'_{i-1}$ since $x \in H_k$.

Finally, let us prove for $i \geq k+1$ that $T_i \subset H_0 \cup \cdots \cup H_k \cup H'_1 \cup \cdots \cup H'_q \cup H_{k+1} \cup \ldots H_{i-1}$. Clearly, $f(T_i) \setminus \{ z \} \subset \bigcup_{j=0}^{i-1} (f(H_j) \setminus \{ z \})$. Besides, $x \in H_k$ and $y \in H'_q$, and since T_i is included into $f(T_i) \setminus \{ z \} \cup \{x, y\}$, then T_i is also contained in $H_0 \cup \cdots \cup H_k \cup H'_1 \cup \cdots \cup H'_q \cup H_{k+1} \cup \cdots \cup H_{i-1}$.

It follows that $(T_i, H_i)_{i=1,\ldots,k}, (T'_i, H'_i)_{i=1,\ldots,q}, (T_i, H_i)_{i=k+1,\ldots,p}$ forms a hyperpath from s to t in $\mathcal{H}$. □

APPENDIX C

List of symbols

$\mathbb{R}_{\max}$ 23
The set $\mathbb{R} \cup \{-\infty\}$.

$\oplus$ 23
Additive law of the tropical semiring, here max.

$\otimes$ 23
Multiplicative law of the tropical semiring, here $+$.

$\mathbb{0}$ 24
Zero element of the tropical semiring, here $-\infty$.

$\mathbb{1}$ 24
Unit element of the tropical semiring, here 0.

α^{-1} 24
When $\alpha > \mathbb{0}$, refers the tropical inverse of α w.r.t. $\otimes$, here the classical opposite $-\alpha$.

$\boldsymbol{x}, \boldsymbol{y}, \ldots$ 24
Elements of $\mathbb{R}_{\max}^d$.

$\boldsymbol{x}_i$ 24
When $\boldsymbol{x} \in \mathbb{R}_{\max}^d$, refers to the i-th entry of $\boldsymbol{x}$. This notation is also used for row-vector $\boldsymbol{x} \in \mathbb{R}_{\max}^{1\times d}$.

$\mathbb{0}$ 24
Element of $\mathbb{R}^d_{\max}$ whose all entries are $\mathbb{0}$.

$\mathbb{1}$ 24
Element of $\mathbb{R}^d_{\max}$ whose all entries are $\mathbb{1}$.

$S \oplus S'$ 25
Minkowski sum of two sets $S, S' \subset \mathbb{R}^d_{\max}$.

$\mathsf{co}(S)$ 26
Tropical convex hull of $S \subset \mathbb{R}^d_{\max}$.

$\mathsf{cone}(S)$ 26
Tropical cone generated by $S \subset \mathbb{R}^d_{\max}$.

$\langle x \rangle$ 26
Tropical ray generated by a non-null $\boldsymbol{x} \in \mathbb{R}^d_{\max}$.

$cl(S)$ 26
Closure of a set $S \subset \mathbb{R}^d_{\max}$ for the usual topology.

$\mathsf{rec}(\mathcal{C})$ 27
Recession cone of a closed tropical convex set $\mathcal{C} \subset \mathbb{R}^d_{\max}$.

$\mathsf{extp}(\mathcal{C})$ 28
Set of the extreme points of a convex set $\mathcal{C} \subset \mathbb{R}^d_{\max}$.

$\mathsf{extg}(\mathcal{C})$ 28
Set of the extreme generators (also called extreme elements) of a tropical cone $\mathcal{C} \subset \mathbb{R}^d_{\max}$.

$\|\cdot\|$ 28
Norm over $\mathbb{R}^d_{\max}$ defined as $\|\boldsymbol{x}\| = \max_{1 \leq i \leq d} e^{\boldsymbol{x}_i}$.

$\sigma(\boldsymbol{x})$ 28
When $\boldsymbol{x} \neq \mathbb{0}$, refers to the scaled element $\|\boldsymbol{x}\|^{-1}\boldsymbol{x}$.

$\widehat{\mathcal{C}}$ 30
Homogenized cone of a closed convex set $\mathcal{C} \subset \mathbb{R}^d_{\max}$.

ι 39
Correspondence between the scaled minimal generating representions of a tropical polyhedron and its homogenized polyhedral cone.

$[d]$ 43
The set of the integers between 1 and d.

$\mathsf{supp}(\boldsymbol{x})$ 43
Given $\boldsymbol{x} \in \mathbb{R}^d_{\max}$, refers to the set of the coordinates i such that $\boldsymbol{x}_i \neq \mathbb{0}$.

$\mathcal{T}(\boldsymbol{x}, \mathcal{C})$ 45
Tangent cone to the tropical cone $\mathcal{C}$ at the element $\boldsymbol{x} \in \mathcal{C}$.

$\sqcap$ 127
Abstract meet operator.

$\bot$ 127
Abstract least element.

$\top$ 127
Abstract greatest element.

∇ 128
Widening operator.

Δ 129
Narrowing operator.

$(\!|\boldsymbol{v}_i \leftarrow \cdot|\!)$ 135
Abstract assignement operator on the variable $\boldsymbol{v}_i$.

$(\!|cond|\!)$ 136
Abstract condition operator.

$\mathcal{C}(P)$ 140
Abstract collecting semantics of a program.