Finding constancy in linear routines

Steven De Oliveira

université PARIS-SACLAY

NNT : 2018SACLS207

Thèse de doctorat

Finding constancy in linear routines

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud 11

École doctorale n°580 : SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION (STIC)

Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Palaiseau, le 28/06/2018, par

Steven de Oliveira

Composition du Jury :

Nicole Bidoit-Tollu Professeur, Paris Saclay (– LRI)	Président
Andreas Podelski Professeur, University of Freiburg	Rapporteur
Ahmed Bouajjani Professeur, Université Paris Diderot (– IRIF)	Rapporteur
Sylvie Putot Professeur, Ecole Polytechnique (– LIX)	Examinateur
Antoine Miné Professeur, Université Paris 6(– LIP6)	Examinateur
Saddek Bensalem Professeur, Université Grenoble Alpes(– VERIMAG)	Directeur de thèse
Virgile Prevosto Ingénieur de recherche, CEA(– CEA, List)	Co-Directeur de thèse

Synthèse

La criticité des programmes dépasse constamment de nouvelles frontières car ils sont de plus en plus utilisés dans la prise de décision (voitures autonomes, robots chirurgiens, etc.). Le besoin de développer des programmes sûrs et de vérif er les programmes existants émerge donc naturellement.

Pour prouver formellement la correction d'un programme, il faut faire face aux déf s de la mise à l'échelle et de la décidabilité. Programmes composés de millions de lignes de code, complexité de l'algorithme, concurrence, et même de simples expressions polynomiales font partis des problèmes que la vérif cation formelle doit savoir gérer. Pour y arriver, les méthodes formelles travaillent sur des abstractions des programmes étudiés af n d'analyser des approximations de leur comportement. L'analyse des boucles est un axe entier de la vérif cation formelle car elles sont encore aujourd'hui peu comprises. Certaines d'entre elles peuvent facilement être traitées, pourtant il existe des exemples apparemment très simples mais dont le comportement n'a encore aujourd'hui pas été résolu (par exemple, on ne sait toujours pas pourquoi la suite de Syracuse, simple boucle linéaire, converge toujours vers 1). L'approche la plus commune pour gérer les boucles est l'utilisation d'invariants de boucle, c'est à dire de relations sur les variables manipulées par une boucle qui sont vraies à chaque fois que la boucle recommence. En général, les invariants utilisent les mêmes expressions que celles utilisées dans la boucle : si elle manipule explicitement la mémoire par exemple, on s'attend à utiliser des invariants portant sur la mémoire. Cependant, il existe des boucles contenant uniquement des affectations linéaires qui n'admettent pas d'invariants linéaires, mais polynomiaux.

Cette thèse présente de nouvelles propriétés sur les boucles linéaires et polynomiales. Il est déjà connu que les boucles linéaires sont polynomialement expressives, au sens ou si plusieurs variables évoluent linéairement dans une boucle, alors n'importe quel monôme de ces variables évolue linéairement. La première contribution de cette thèse est la caractérisation d'une sous classe de boucles polynomiales exactement aussi expressives que des boucles linéaires, au sens où il existe une boucle linéaire avec le même comportement. Ensuite, deux nouvelles méthodes de génération d'invariants sont présentées. La première méthode est basée sur l'interprétation abstraite et s'intéresse aux filtres linéaires convergents. Ces filtres jouent un rôle important dans de nombreux systèmes embarqués (dans l'avionique par exemple) et requièrent l'utilisation de flottants, un type de valeurs qui peut mener à des erreurs d'imprécision s'ils sont mal utilisés. Aussi, la présence d'affectations aléatoires dans ces filtres rend leur analyse encore plus complexe. La seconde méthode traite d'une approche différente basée sur la génération d'invariants pour n'importe quel type de boucles linéaires. Elle part d'un nouveau théorème présenté dans cette thèse qui caractérise les invariants comme étant les vecteurs propres de la transformation linéaire traitée. Cette méthode est généralisée pour prendre en compte les conditions, les boucles imbriquées et le non déterminisme dans les affectations.

La génération d'invariants n'est pas un but en soi, mais un moyen. Cette

thèse s'intéresse au genre de problèmes que peut résoudre les invariants générés par la seconde méthode. Le premier problème traité est problème de l'orbite (Kannan-Lipton Orbit problem), dont il est possible de générer des certif cats de non accessibilité en utilisant les vecteurs propres de la transformation considerée. En outre, les vecteurs propres sont mis à l'épreuve en pratique par leur utilisation dans le model-checker CaFE basé sur la verif cation de propriétés temporelles sur des programmes C.
Les résultats expérimentaux des outils en comparaison de l'état de l'art existant démontrent une efficacité notable. En outre, leur développement dans le langage de programmation OCaml garantit leur fiabilité et leur mise en open-source contribue à l'écosystème des programmes libres.

Abstract

The criticality of programs constantly reaches new boundaries as they are relied on to take life-or-death decisions in place of the user (autonomous cars, robot surgeon, etc.). This raised the need to develop safe programs and to verify the already existing ones. Anyone willing to formally prove the soundness of a program faces the two challenges of scalability and undecidability. Million of lines of code, complexity of the algorithm, concurrency, and even simple polynomial expressions are part of the issues formal verification have to deal with. In order to succeed, formal methods rely on state abstraction to analyze approximations of the behavior of the analyzed program. The analysis of loops is a full axis of formal verification, as this construction is still today not well managed. Though some of them can be easily handled when they perform simple operations, there still exist some seemingly basic loops whose behavior has not been solved yet (the Syracuse sequence for example is suspected to be undecidable [Con13]). The most common approach for the treatment of loops is the use of loop invariants, i.e. relations on variables that are true at the beginning of the loop and after every step. Intuitively, invariants are expected to use the same set of expressions used in the loop: if a loop manipulates the memory on a structure for example, invariants will naturally use expressions involving memory operations. However, there exist loops containing only linear instructions that admit only polynomial invariants (for example, the sum on integers $\sum_{i=0}^{n} i$ can be computed by a linear loop and is a degree 2 polynomial in n), hence using expressions that are syntactically absent of the loop. The intuition stated above is thus a bit naive and we should seek for more relations between invariants and loop instructions. This thesis presents new insights on loops containing linear and polynomial instructions. It is already known that linear loops are polynomially expressive, in the sense that if a variable evolves linearly, then any monomial of this variable evolves linearly. The first contribution of this thesis is the extraction of a class of polynomial loops that is exactly as expressive as linear loops, in the sense that there exists a linear loop with the exact same behavior. Then, two new methods for generating invariants are presented.

- The first method is based on abstract interpretation [CH78] and is focused on a specific kind of linear loops called linear filters. Linear filters play a role in many embedded systems (plane sensors for example) and require the use of floating point operations, that may be imprecise and lead to errors if they are badly handled. Also, the presence of non deterministic assignments makes their analysis even more complex.

- The second method [OBP16] treats of a more generic subject by finding a complete set of linear invariants of linear loops that is easily computable. This technique is based on the linear algebra concept of eigenspace. It is extended to deal with conditions, nested loops and non determinism in assignments [OBP17].

Generating invariants is an interesting topic, but it is not an end in itself, it must serve a purpose. This thesis investigates the expressivity of invariants generated by the second method by generating counter examples for the Kannan-Lipton Orbit problem [KL80]. It also presents the tool PILAT implementing this technique and compares its efficiency technique with other state-of-the-art invariant synthesizers. The effective usefulness of the invariants generated by PILAT is demonstrated by using the tool in concert with CaFE [OPB], a model-checker for C programs based on temporal logics.

Remerciements

Durant ces quatre ans passés au CEA, j'ai fait beaucoup de rencontres. J'en ai tellement fait qu'il est difficile de quantifier la valeur de chacun d'entre eux et d'en faire un classement. Non pas qu'il faille forcément en faire un, mais je n'arrive pas à me faire à l'idée qu'il faille mettre un nom en premier. Dois-je mettre le nom de la personne qui m'a le plus aidé, de la première personne avec qui j'ai travaillé, qui m'a le plus motivé ou qui a le plus contribué à cette thèse ? A vrai dire cette personne est la même, et sans lui cette thèse n'aurait jamais vu le jour. Merci Virgile. Je remercie tout autant mon directeur de thèse, Saddek Bensalem, qui m'a soutenu tout le long de cette expérience. Je tiens également à remercier la présidente du Jury Professeur Nicole Bidoit, les Professeurs Ahmed Bouajjani et Andreas Podelski pour la pertinence de leur rapport, ainsi que le Professeur Antoine Miné et le Professeur Sylvie Putot.

Avant de commencer cette thèse, j'avais certains à-prioris sur le déroulement d'une thèse, notamment sur sa difficulté et la quantité de stress qu'elle génère. J'ai passé trois années pleines de rires et de bons moments qui ont éclipsé la grande majorité des difficultés, notamment grâce au soutien du Laboratoire de Sûreté et Sécurité du Logiciel et du Département tout entier, avec une mention spéciale à Hugo qui a dû supporter son insupportable cobureau. En vrac, je remercie Alexandre, André, Benjamin, Boris, David, Diane, Florent, François, Frank, Jacques-Charles, Jean-Christophe, Jean-Yves, Julien, Lionel, Mathieu, Nikolaï, Quentin, Sébastien, Tristan, Valentin, Vincent, Zak, ainsi que l'ensemble des acteurs en herbe (et Frédérique l'actrice professionnelle), les permanents, doctorants et post-doctorants dont j'oublie le nom mais que je n'oublierai jamais.

Contrairement à ce que l'on peut penser, un thésard à une vie à coté de la thèse. Cette vie a forgée par des bons et des mauvais moments, mais je n'aurais jamais pu arriver là où j'en suis sans l'amour de mes parents, José et Patricia (et de toute ma famille bien evidemment, mais là il y a vraiment beaucoup de monde), sans le soutien d'Adeline, ni sans l'amitié de Guillaume et de Pierrot. Je vous suis éternellement redevable.

Oh, et un conseil aux doctorants et futurs doctorants qui ont eu le courage de lire jusqu'ici : ecrire une page de remerciements, c'est plus difficile qu'il n'y parait. Ne vous y prenez pas comme moi, à la dernière minute.

Contents

Part I

Formal verification

Chapter 1

Introduction

Contents

Everyday, a ship sails in the binary sea.

The captain shouts its instructions to the deck hands, giving to each of them specific instructions.
Clear the deck of every single useless resource.
Reach the maximum speed.
And avoid all the reefs.

Storms are coming, and care is the burden of chiefs.
What if one of the men fails to achieve its deed?
Will the tides crash the boat with tremendous forces?
Will their journey end with Neptune's introductions?

Dangerous is sailing in the binary sea.

1.1 Context

1.1.1 The electronic boat

Ensuring a boat will reach its destination is a difficult task. Every single person on the ship must be of use and know precisely what to do, when to do it and what to expect from the others. The whole system is functionning thanks to the collaboration of all the local actors. In this analogy, computer systems are electronic boats.

Since Turing's formalization of computer systems in 1936, the field of computer science has been and is still exponentially growing. Where mechanical machines require bolts, gears and mechanical inputs, computers require memory, instructions and numerical inputs. Computers have been set up in order to perform simple calculations extremely fast. Modern processors are able to process more than 2 billion operations per second.

1.1.2 Origins of bugs

Many users face bugs at a non critical scale that can be relatively easily solved, often requiring them at most to reboot their machine. In the mean time, computer systems influence our lives at a wider scale as *embedded systems* (computer systems as part of a larger device) are used in traffic controlling, avionics, energy management, economy or autonomous driving to give a few. Such critical fields require an extremely high level of trust as their failure can cause huge damages in terms of human lives, ecological environment and economical resources. The issue of bugs is even harder considering that even the smallest bug can have terrible effects on the short and the long term. One of the bug with that could have had the most terrible consequence has happened during the cold war, in the USSR. In 1983, a nuclear early-warning system detected multiple ballistic missiles launched from bases in the US, even though no missile had been. While the worst has been avoided thanks to the discernment of the person in charge, this bug could literally have wiped out mankind as we know it. More recently the Meltdown and Spectre bugs [Koc+18], discovered in 2018, affected Intel cores so hard the only patch solving the issue induces a loss of performance of 5-30%. Even the said patch contained bugs that forced Intel to ask users to stop downloading it. Computer systems tend nowadays to become autonomous and make their own decisions, based on algorithms. It is vital to certify that these systems are functioning correctly.

Computers are extremely complex machines, sending information as binary signals (sequences of 1s and 0s). Instead of working directly in binary, multiple layers of abstractions have been created to help computer scientists to express their computations. As Figure 1.1 shows, it is necessary to go through multiple steps to get an idea understood by a computer. *Each of these steps can be flawed*, causing different kind of bugs.

- An algorithm is a sequence of understandable instructions that require an input and produces an output[1]. Programmers can make mistakes writing an algorithm by not thinking everything through. For example, let us consider the Euclidean division algorithm for calculating the quotient q of two integers x and y. Starting with $q = 0$, it consists in decreasing x of y and add 1 to q until $x < y$. At the end, q should contain the quotient of x and y.

[1] A cooking recipe is an algorithm for example, its input being ingredients and its output being a cake.

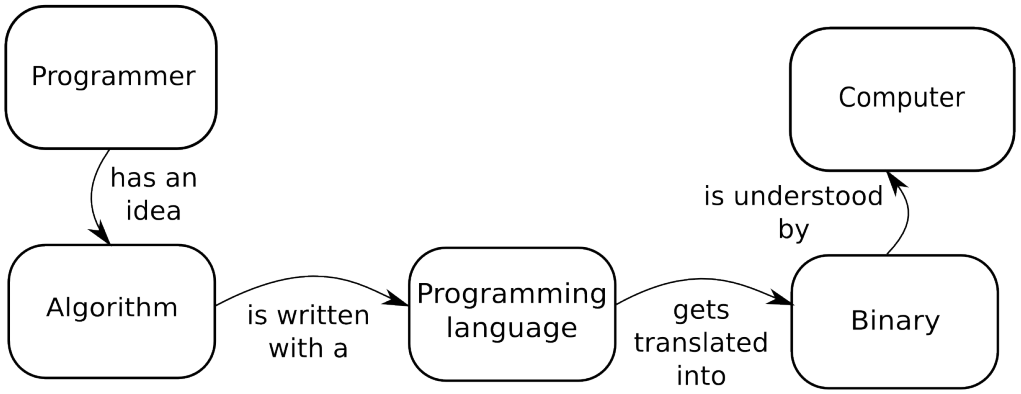


FIGURE 1.1: From the thinking of a programmer to the comprehension of the computer.

This simple algorithm contains a bug. In the case where $y \leqslant 0$ this algorithm will never stop.

- Assume now we have an algorithm that we know contains no mistake. There exist plethora of different languages (C, *OCaml*, C++, Java, Python, ...) that we can choose to write out algorithm with. The representation by a programming language of an algorithm is called its *implementation*. An implementation may require to use data structures that are out of the scope of the algorithm, but that enhances its efficiency. For example, if our algorithm has to save the value of multiple elements, it can use a structure of set to save them and access them. The implementation of sets is irrelevant to the functioning of the algorithm in itself, but its functioning must be understood by the programmer. Otherwise, the programmer may use them wrong which could cause issues. Among others, the Java implementation of sets require a total order[2] on the elements of the set. If the programmer cannot provide such an order, the implementation will not be valid. This shows that bugs can take their roots in the most unexpected parts of a program.

- Assume now we manage to write a program that is correct. We need to translate it into assembly to be understood by the computer. This translation is performed by an independent program, the compiler. As a program, it can contain bugs and therefore insert bugs in the original program.

- Once the program is written in assembler, there is one last step to cross. Computers are able to read assembler instructions as binary signals made out of electricity. Physical interferences can however alter these signals and provoke undesired behaviors on the system. This kind of bugs is particularly hard to solve as the programmer has very few ways to physically protect a system.

This thesis will focus on the correctness of the transition between the *algorithm* state to the *program state*.

[2] A total order $\leq$ on a set of elements S must verify $\forall x, y \in S, x \leq y$ or $y \leq x$

1.1.3 Get rid of bugs

While it is possible for simple algorithms to be proven correct by hand, industry requires safety on millions lines of code (LoC) algorithms. The PDF viewer you may be using right now to read this document has been developed with more than 100000 LoC[3]. It is necessary for provers to rely on computers to automatize the verification task, which is humanely impossible to handle. And yet, the problem of proving the correct behavior of a program is undecidable in general. The most famous undecidable problem in computer science is the halting problem, stating : *does there exist a program A able to decide that a program P with an input I ends in finite time ?* If such a program existed, then it would be possible to create a program B (cf Figure 1.2) such that:

1. if B ends, then B doesn't end;
2. if B does not end, then B ends.

As this is clearly absurd, the halting problem is undecidable. There exist multiple similar problems that can be reduced to the halting problem, in the sense that if they admit a solution, an algorithm solving the halting problem can be built. The Rice Theorem [Ric53] generalizes this principle and states that the verification of any non trivial semantic property (i.e. non syntactical properties that are not true nor false) on a Turing-complete machine/language is undecidable in general. This is applicable on properties like "the program never fails" or "this program is a virus".

1.1.4 Formal methods

As we saw, anyone willing to formally prove a large program faces the two challenges of scalability and undecidability. They can be overcome by giving up completeness (i.e. giving up the ability of disproving a property) or correctness (i.e. giving up the ability of proving a property). To lighten the burden of proving large programs, the field of *formal methods* provide techniques and tools to ease and automatize proofs (or automatize the search of counter examples).

When programmers try to find bugs in its program without formal methods, they can either launch their code and check if the output is consistent with their expectations or read their code to check if there is something missing. Formal methods are basically the automation of these two intuitions, respectively called *dynamic analysis* and *static analysis*.

Dynamic analysis is based on the analysis of execution paths of a program by executing them with different inputs. If a tested execution is not conforming to the program specifications, then dynamic can provide it as a counter example to the verifier. On the other hand, if every tested execution satisfies the specifications, it is impossible to conclude on the correctness of the program as the set of every possible input is too large to be exhaustively tested.

[3]The size of MuPDF, a lightweight PDF viewer, has approximatively 140000LoC

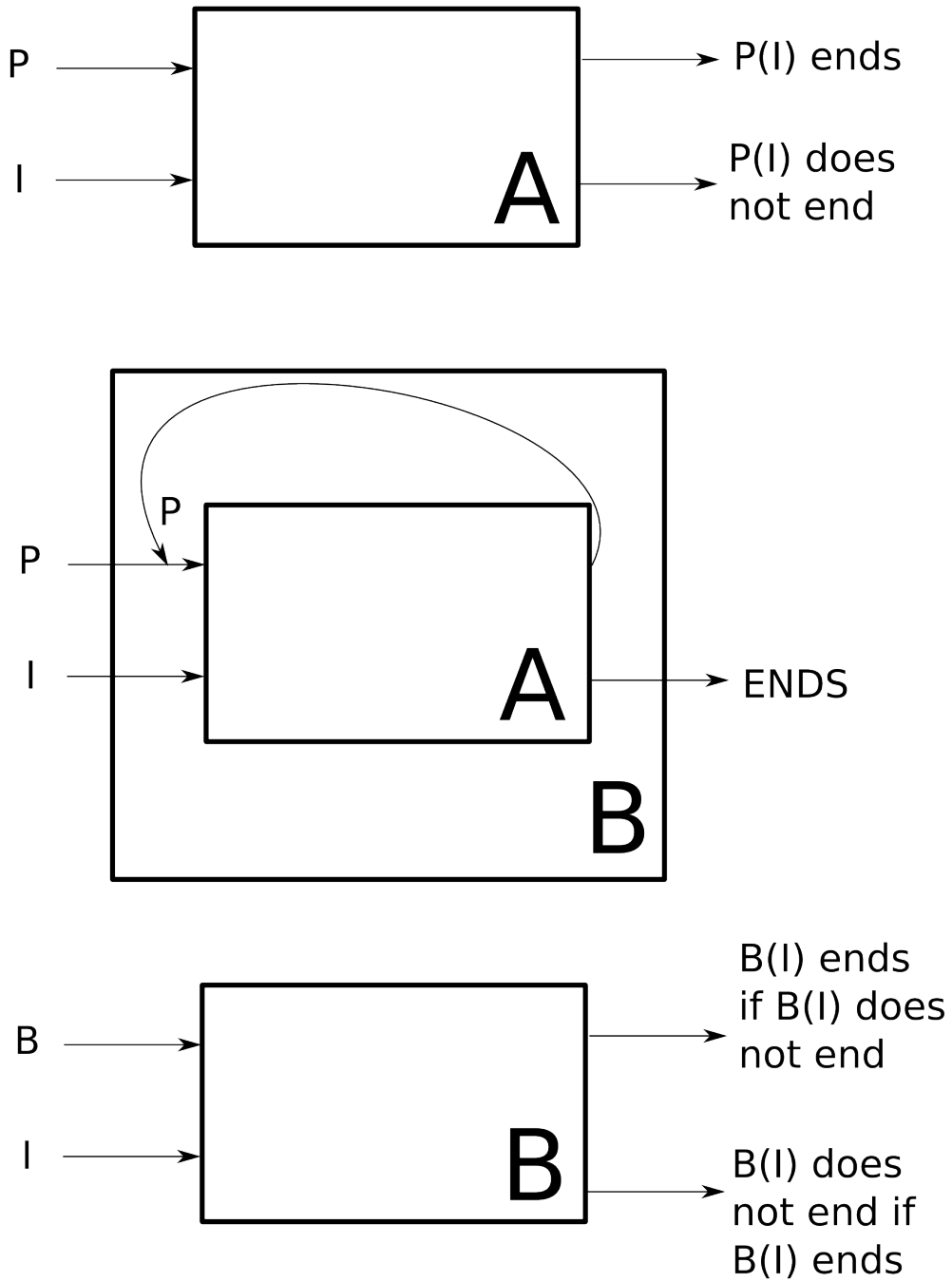


FIGURE 1.2: Construction of an impossible program.

Static analysis is complementary to dynamic analysis. The semantic of each instruction, defined as the most precise mathematical characterization of program behaviors, is abstracted to a simpler one. This simplification allows different kind of techniques to infer information on these simplified programs, that are also correct on the original programs. Static analysis techniques tend to over approximate the program behavior. As a consequence, the main issue of static analysis is to miss some important properties that have not been kept by the program abstraction.

Static analysis itself can be divided in many different fields, whose most used today are *abstract interpretation*, *model checking* and *deductive verification*.

Abstract interpretation aims at inferring logical properties on a program by propagating abstractions of states. For example, values of integer variables can be abstracted by intervals modified by the program instructions. Such a method requires to define a new semantic of instructions and expressions so that they are consistent with abstract values.

Model checking consists in comparing the possible program executions to a model-based specification. The program as well as the negation of the specification must admit an automaton representation[4] that are explored simultaneously and exhaustively to check if there exists an execution of the program model that matches an execution of the specification model.

Deductive verification is the closest approach to mathematical reasoning. The program is specified with *contracts* that it must satisfy. Preconditions give constraints on the input of the program, postconditions give constraints on its output, and assertions are properties that are always verified at a given point of a program. Deductive verification reasons with these constraints to prove the correctness of a program.

1.1.5 Loops

As computer systems are expected to repeat the same task for an indefinite amount of time, loops are at the core of programming. Almost every interesting algorithm contains at least one. Though some of them can be easily handled when they perform simple operations, there still exist some very simple loops whose behavior has not been solved yet. The Syracuse sequence for example $(S_n)_{n\in\mathcal{N}}$ is defined as follows:

1. if S_n is even, then $S_{n+1} = \frac{S_n}{2}$;
2. otherwise, $S_{n+1} = 3.S_n + 1$.

For every tested initial value so far, this sequence eventually reaches 1. As simple as it may seem, proving that for every S_0 the sequence reaches 1 has

[4] An automaton, as defined in the next Chapter, is an oriented graph with extra properties on edges and nodes.

still not been proven and is today suspected to be undecidable [Con13]. Linear loops, like the Syracuse sequence, are of high interest in the field of formal verification as they lie at the border of undecidability. In general, the easiest way (and seemingly the most efficient way) to handle loops in proofs is to *delete them.* With an over-approximation of the number of times the loop is taken, this can be done by unrolling loop. When there is no information about the number of necessary unrollings, it is still possible to *over-approximate the loop behavior*, in the sense of finding relations on variables that are true when the loop ends. Those over approximations are commonly called *invariants.*

This thesis will give new insights on the undecidability border of linear loops (i.e. loops in which expressions are linear combinations), especially by proving that a subclass of polynomial loops (i.e. loops with polynomial expressions) are as expressive as linear loops. It also extracts a complete characterization of linear invariants of linear loops, which can be applied for solving the Kannan-Lipton Orbit problem [KL80] and help formal tools to conclude their proofs.

1.2 Overview and contributions

This first part introduces the context of this thesis along with Chapter 2, presenting the standard notations of static analysis and linear algebra that are used in the next chapters. The fundamental content of this thesis is developed in Part II.

- Chapter 3 presents the concept of *linearization*. This study of polynomial loops (loops with polynomial assignments) formally prove they can be divided into 2 different types: those that can be represented by linear applications and those that admit an exponential behavior.

- The next two Chapters study the problem of generating invariants for linear loops. Chapter 4 presents a method for detecting good candidate invariants found by an *abstract interpretation* analysis with the zonotope domain on linear filters (linear loops with specific hypotheses). Chapter 5 presents a general characterization of linear invariants for linear loops. Combined with the results of the previous chapter, this characterization is generalizable to polynomial invariants for multi-path, nested and non deterministic loops.

The following Chapter 6 is devoted to the use of the previous characterization to synthesize proofs of different instances the Kannan-Lipton Orbit problem [KL80].

The last part of this thesis introduces two tools. The first, *Pilat*, implements in Chapter 7 the algorithm described in Chapter 5 with all its extensions for generating invariants for C programs. The practical use of invariants is shown in Chapter 8 by a presentation of *CaFE*, a model-checker using the informations provided by *Pilat* to prove temporal properties expressed in the temporal logic *CaRet*.

Chapter 2

Mathematical definitions of program verification

Contents

2.1 Linear algebra

Linear algebra is the branch of mathematics studying vectorial spaces and linear transformations. It plays an important role in the next chapters. The principal definitions and notations used in linear algebra are developed in this section. A more in-depth presentation can be found in [WBR13]

2.1.1 Vector spaces

A field $\mathbb{K}$ is a set of elements, called scalars, associated with two operators $+$ and $*$. Both operators are associative ($a+(b+c)=(a+b)+c$ and $a*(b*c)=(a*b)*c$) and commutative ($v+w=w+v$ and $v*w=w*v$). Multiplication is distributive over addition ($a*(b+c)=a*b+a*c$). Both these operators admit a neutral element, respectively $0_{\mathbb{K}}$ (or 0) and $1_{\mathbb{K}}$ (or 1), such that for all $v \in \mathbb{K}$, $v+0_{\mathbb{K}}=v$ and $v*1_{\mathbb{K}}=v$. Every element of $\mathbb{K}$ admits an inverse for the $+$ operator, in the sense that for any v there exists w such that $v+w=0_{\mathbb{K}}$. The inverse of v by the $+$ operator is denoted $-v$. Also, every element except $0_{\mathbb{K}}$ admits an inverse for the $*$ operator, in the sense that for any v, there exists w such that $v*w=1_{\mathbb{K}}$ and w is denoted v^{-1}.

Vectors are n-tuples of elements of $\mathbb{K}$; the set of every vector of size n will be denoted $\mathbb{K}^n$ (Cartesian product). Let $v=(v_1,...,v_n)$ and $w=(w_1,...,w_n)$ two vectors of $\mathbb{K}^n$. The addition operator is extended to vectors by adding each coordinates ($v+w=(v_1+w_1,...,v_n+w_n)$) and vectors can be multiplied by scalars ($k*v=(k*v_1,...,k*v_n)$). Scalar or vectorial expressions involving only those operators are called linear combinations. v and w are collinear if there exists k such that $k*v=w$. Otherwise, v and w are said independent. Independence can be generalized to sets of vectors. A set B of m vectors is said independent if for all non-trivial linear combination (i.e. linear combinations involving at least one non null vector) $f:(\mathbb{K}^n)^m \mapsto \mathbb{K}^n$ we have $f(B) \neq (0,...,0)$

$\mathbb{K}$-vector spaces are subsets of $\mathbb{K}^n$ *stable by addition and scalar multiplication*. Here are some example of vector spaces:

- $\mathbb{K}^n$
- $\{(0,...,0)\}$
- $\{v : \exists k,k'.v=k*(0,...,0,1)+k'*(1,0,...,0)\}$

In the last example, two vectors are used to construct every element of the vector space. These vectors form a *generator family* $\mathcal{F}$ of this vector space, and $Vect_{\mathbb{K}}(\mathcal{F})$ denotes the vector space generated by $\mathcal{F}$ with scalar coefficients in $\mathbb{K}$. A *base* of vector space is an independent generator family, and its size is called the dimension of the vector space. The third example admits $\mathcal{B}=\{(0,...,0,1);(1,0,...,0)\}$ as a minimal base, thus its dimension is 2. The dimension of $\mathbb{K}^n$ is n, and the dimension of $\{(0,...,0)\}$ is 0. For our purposes, we will only study properties on vector spaces of finite dimension, but there exists vector spaces of infinite dimension (for example, polynomials with one variable X are linear combinations of $1, X, X^2, ...$).

A family of vectors $\mathcal{B}$ is associated with a *determinant* $det(\mathcal{B})$, which is defined as follows:

$$\sum_{\sigma \in \Sigma_n} \epsilon(\sigma) \prod_{j=1}^{n} \mathcal{B}_{j,\sigma(j)}$$

where Σ_n represents the set of permutations of n elements, ϵ the signature of a permutation ($\epsilon(\sigma) = (-1)^{N(\sigma)}$ with $N(\sigma)$ the number of inversions of σ) and $\mathcal{B}_{i,j}$ the j^{th} component of the i^{th} vector of $\mathcal{B}$. The determinant is non null if and only if the family $\mathcal{B}$ is independent.

2.1.2 Linear transformations and matrices

Linear transformations are applications mapping a vectorial space to another. They are only allowed to use scalar multiplications of variables and additions. Hence, a linear transformation f follows two canonical properties:

- $f(v) + f(w) = f(v + w)$
- $k * f(v) = f(k * v)$

Vectors themselves describe linear transformations thanks to the *scalar product* operator.

Definition 1 *Let $v = (v_1, ..., v_n)$ and $w = (w_1, ..., w_n)$ two vectors. The scalar product of v and w is denoted $\langle v, w \rangle$ and is defined as follows:*

$$\langle v, w \rangle = \sum_{i=1}^{n} v_i * w_i$$

By fixing v, $f_v(w) = \langle v, w \rangle$ is a linear combination on the coordinates of w, thus f_v is a linear transformation. Similarly, $f_w(v) = \langle v, w \rangle$ is also a linear transformation. On the other hand, let $f(w_1, w_2, w_3) = 2 * w_1 + 3 * w_2 + 5 * w_3$ a linear combination. The application f can be seen as the scalar product of $(2, 3, 5)$ and (w_1, w_2, w_3). Therefore, f can be assimilated to the vector $(2, 3, 5)$. From now on, vectors will be written as a line $(v_1, ..., v_n)$ when they represent linear transformations, while they will be written as a column $\begin{pmatrix} w_1 \\ \dots \\ w_n \end{pmatrix}$, or $(w_1, ..., w_n)^t$, when they denote elements of $\mathbb{K}^n$.

Linear transformations are not restricted to transform a vector into a scalar. For example, $g(x, y, z) = (x+y, y+z, z+x)$ is a valid linear transformation as it verifies the canonical properties of linear transformations with the vector addition and scalar multiplication. To extend the vectorial notation of linear transformations for multi-dimensional images, the concept of *matrices* is necessary. Matrices are vector arrays generalizing the notation of linear transformations defined by scalar product. Let us study the example of g. The line vector associating (x, y, z) to $x+y$ is $(1, 1, 0)$, as $\langle (1, 1, 0), (x, y, z) \rangle = x+y$. Similarly, $(0, 1, 1)$ and $(1, 0, 1)$ represent the two next coordinates of g. Therefore,

the matrix associated to g is

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

Every linear transformation $f : \mathbb{K}^m \mapsto \mathbb{K}^n$ admits a matrix representation $M \in \mathcal{M}_{m,n}(\mathbb{K})$, where $\mathcal{M}_{m,n}(\mathbb{K})$ is the set of matrices of m columns and n lines. The notation of the set $\mathcal{M}_{n,n}$ of square matrices is simplified into $\mathcal{M}_n$. By extension, linear transformations $f : \mathbb{K}^n \mapsto \mathbb{K}^n$ will be refered to as *square* transformations as they admit a square matrix. In general, every notation defined for matrices is valid for linear transformations. For a matrix M, $M_{i,j}$ represents the j^{th} coefficient of M_i the i^{th} line of M. Every matrix $M \in \mathcal{M}_{m,n}(\mathbb{K})$ admits a transpose $M^t \in \mathcal{M}_{n,m}$ defined as a substitution of its coefficients: $M^t_{i,j} = M_{j,i}$. The addition operator $+$ of $\mathbb{K}$ is extended to matrices of same size, as it was extended to vectors, by applying it coordinatewise. In other words for two matrices M and N of same size, $(M+N)_{i,j} = M_{i,j} + N_{i,j}$. Matrix multiplication extends the scalar multiplication $*$ for $M \in \mathcal{M}_{l,m}$ and $N \in \mathcal{M}_{m,n}$, which returns $M*N \in \mathcal{M}_{l,n}$. It is defined using the scalar product of Definition 1 as follows:

$$(M * N)_{i,j} = \left\langle M_i, N^t_j \right\rangle$$

As vectors can be seen as matrices with one line and n columns or one column and n lines, they also can be multiplied by matrices with respectively n lines or n columns. We denote $\circ$ the usual composition operator. The main interest of vector/matrix multiplication is to easily apply input vectors and compose linear applications, as the result of matrix multiplication is the matrix representing the composition of the two linear transformations associated to the initial matrices. In other words, if a linear transformation f is represented by the matrix M_f, g by M_g and v is a vector, then $M_f * v = f(v)$ and $M_f * M_g$ represents $f \circ g$ the composition of f and g.

The kernel of a linear transformation f, denoted $\ker(f)$, is the vector space defined as $\ker(f) = \{x | x \in \mathbb{K}^n, f(x) = 0\}$. The same notation is used for matrices representing a linear transformation.

A square transformation f is said to be invertible if there exists a linear transformation g such that $f \circ g = g \circ f = Id$ the identity transformation, and g will be denoted f^{-1}. The successive application of f n times will be denoted f^n and its associated matrix is M^n_f where M_f is associated to f. If there exists n such that $f^n = 0$, then f will be said *nilpotent*.

As matrices are vector arrays, the concept of determinant is extendable to linear transformations. Particularily, we have that a linear transformation is invertible iff it has a non null determinant. Also, the determinant of f equals the determinant of f^*. It is also equal to the inverse of the determinant of f^{-1} if f is invertible.

2.1.3 Duals and orthogonal space

Let f a linear transformation associated to a matrix M_f. f admits a *dual* transformation $f^* : (\mathbb{K} \to \mathbb{K}) \to (\mathbb{K} \to \mathbb{K})$ defined as:

$$\forall \varphi, f^*(\varphi) = \varphi^t \circ f$$

where φ is a column vector. The dual of the dual of f is f, or in other words $(f^*)^* = f$. The matrix associated to f^* is M_f^t.

Let E be a $\mathbb{K}$ vector space, $F \subset E$ a sub vector space of E and x an element of F. A vector y is *orthogonal* to x if $\langle x, y \rangle = 0$. We denote $F^\perp$ the set of vectors orthogonal to every element of F. The orthogonal of the orthogonal of a vectorial space V is V, or $(V^\perp)^\perp = V$. The concept of dual and orthogonal spaces are deeply connected. In particular, they verify the following lemma:

Lemma 1 *Let $\mathbb{K}$ be a field, E be a $\mathbb{K}$ vectorial space, F a sub-$\mathbb{K}$ vectorial space of E and $f : E \to E$ a linear application.*
$f(F) \subset F \Leftrightarrow f^*(F^\perp) \subset F^\perp$

Proof. By definition we have $\langle f(x), x' \rangle = \langle x, f^*(x') \rangle$. Let $x \in F, x' \in F^\perp$. If $f(F) \subset F$, then $\langle f(x), x' \rangle = 0$, thus $\langle x, f^*(x') \rangle$. As we have $f^*(F^\perp) \subset F^\perp$, we conclude by using the fact that $(F^\perp)^\perp = F$ and $(f^*)^* = f$. □

2.1.4 Eigenvalues and eigenvectors

The ring of polynomials $\mathbb{K}[X]$ is the set of all polynomials with coefficients in $\mathbb{K}$. In other words, $\mathbb{K}[X] = Vect_{\mathbb{K}}(\{1, X, X^2, ...\})$. We note $\overline{\mathbb{K}}$ the algebraic closure of $\mathbb{K}$, $\overline{\mathbb{K}} = \{x : \exists P \in \mathbb{K}[X], P(x) = 0\}$. Every square matrix A is associated to a characteristic polynomial $P \in \mathbb{K}[X]$ such that $P(X) = det(A - X.Id)$. Roots $\lambda \in \overline{\mathbb{K}}$ of this polynomial are called eigenvalues. Their associated eigenspace E_λ, defined as $E_\lambda = \ker(A - \lambda Id)$, where Id is the identity matrix and $E_\lambda \neq \{0\}$. The multiplicity of an eigenvalue is its multiplicity as a root of the characteristic polynomial (i.e. λ has a multiplicity m if $\frac{P(X)}{(X-\lambda)^m} \in \mathbb{K}[X]$ and $\frac{P}{(X-\lambda)^{m+1}} \notin \mathbb{K}[X]$). An eigenvector of f^* is denoted a *left-eigenvector* of f. The *left* adjective is similarly generalized to eigenspaces (but not to eigenvalues, as f and f^* have the same eigenvalues). Generalized (left-)eigenspaces extend the previous concept and are defined as $E_\lambda^n = \ker((A - \lambda Id)^n)$. A vector φ is a generalized eigenvector of order n if $\varphi \in E_\lambda^n$ and $\varphi \notin E_\lambda^{n-1}$.

2.1.5 Properties of the determinant

The determinant has useful properties that will be used through this thesis.

- If all the coefficients of a matrix belong to a ring R, then its determinant belongs to R. This comes from the determinant formula that involves only multiplication and additions of coefficients of the matrix.

- The product of all eigenvalues to their multiplicity equals the determinant. This comes from the fact that eigenvalues are root of the characteristic polynomial. If $\lambda_1, ..., \lambda_n$ are roots of P, then we can write P as $(\lambda_1 - X)...(\lambda_n - X)$. Hence,

$$det(A) = P(0) = \lambda_1 ... \lambda_n$$

Therefore, the product of all eigenvalues of a linear transformation belong to the ring in which its coefficients belong. This is also true when eigenvalues do not belong to this ring: for example the transformation $f(x, y) = (y, -x)$ admits the complex eigenvalues i and $-i$, while its determinant is 1.

2.1.6 Jordan normal form

The matrix of a linear transformation is expressed in a certain base of a vector space. So far, every example was implicitly expressed in the canonical base $\mathcal{B}$ of $\mathbb{K}^n$ defined as $\mathcal{B} = \{e_1 = (1, 0, ..., 0)^t, e_2 = (0, 1, 0, ..., 0)^t, ..., e_n = (0, ..., 0, 1)^t\}$. For example, let $f(x, y, z) = (x + y + z, 2 * y + z, 5 * z)$ a linear transformation. In the canonical base $\mathcal{B} = \{e_1 = (1, 0, 0)^t, e_2 = (0, 1, 0)^t, e_3 = (0, 0, 1)^t\}$, this transformation has the following matrix representation:

$$M = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 2 & 1 \\ 0 & 0 & 5 \end{pmatrix}$$

As we can see, $f(e_1) = e_1$, therefore the first column correspond to *one time* e_1, or $(1, 0, 0)^t$. We also have $f(e_2) = e_1 + 2 * e_2$ and $f(e_3) = e_1 + e_2 + 5 * e_3$. Columns of the matrix are images of the elements of the base in which the linear transformation is expressed.

It is possible to express linear transformations in a different base than the canonical base. For example, in the base $\mathcal{B}' = \{e'_1 = (1, 0, 0), e'_2 = (1, 1, 0), e'_3 = (1, 1, 3)\}$, we have that $f(e'_1) = e'_1$, $f(e'_2) = 2 * e'_2$ and $f(e'_3) = 5 * e'_3$. Hence, in the base $\mathcal{B}'$, f is defined as $f_{\mathcal{B}'}(x, y, z) = (x, 2*y, 5*z)$ and admits the matricial representation

$$M_{\mathcal{B}'} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 5 \end{pmatrix}$$

Changing the base is a linear operation on vectors, therefore it can be performed by an invertible linear transformation p. It is sufficient to apply p^{-1} for returning to the original base.

Definition 2 *Two linear mappings f, g are said* similar *if and only if there exists an invertible linear transformation p such that $f = p^{-1} \circ g \circ p$.*

Similarity between transformations has the interesting property to be preserved when calculating f^n. Indeed, $f^n = p^{-1} \circ g^n \circ p$ is true for any positive n by Definition 2.

In the case of f and $f_{\mathcal{B}'}$, they are similar with $p(x, y, z) = (x, x + y, x + y + 3 * z)$. The careful reader will notice that the vectors of $\mathcal{B}'$ are actually eigenvectors of f, and the values on the diagonal are exactly eigenvalues of f. Expressed in this base, f admits a diagonal matrix representation, in the sense that for all i, j such that $i \neq j$ we have that $M_{\mathcal{B}'i,j} = 0$. More generally, when a linear transformation is similar to a diagonal transformation, we will say it is *diagonalizable*. In any case, every linear transformation is similar to an upper triangular transformation (i.e. admits a matrix where all the coefficients below the diagonal are null).

For any linear transformation f, there always exists a base $\mathcal{J}$ such that $f_{\mathcal{J}}$ is associated to a matrix J defined as follows:

$$J = \begin{pmatrix} J_1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & J_k \end{pmatrix}$$

and

$$J_i = \begin{pmatrix} \lambda_i & 1 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & \dots & 0 & \lambda_i \end{pmatrix}$$

for $1 \leqslant i \leqslant k$ where λ_i is an eigenvalue of f. This base is called the *Jordan base* of the transformation f, and its expression in this base is called the *Jordan normal form* of f.

When a matrix A is diagonalizable, its Jordan normal form J is its diagonal form and the columns of the matrix P of the base changing application ($A = P^{-1}JP$) are exactly eigenvectors. Otherwise, the columns are composed of specific generalized eigenvectors φ of order n that satisfy $J.\varphi = \lambda\varphi + \psi$, where ψ is a generalized eigenvector of order $n - 1$.

2.2 Programming model

2.2.1 State machines

Computer systems are implementations of much simpler machines, called *transition systems* [Wag+06].

Definition 3 *Let $\mathcal{A}$ a set of symbols called an* alphabet. *A word of size n on $\mathcal{A}$ is a sequence (finite or infinite) of n elements of $\mathcal{A}$. A transition system S on $\mathcal{A}$ is a tuple (L, I, T, F) such that:*

- *L is a set of locations, or* states*;*
- *$I \subset L$ is a set of initial states;*
- *$F \subset L$ is a set of final states*

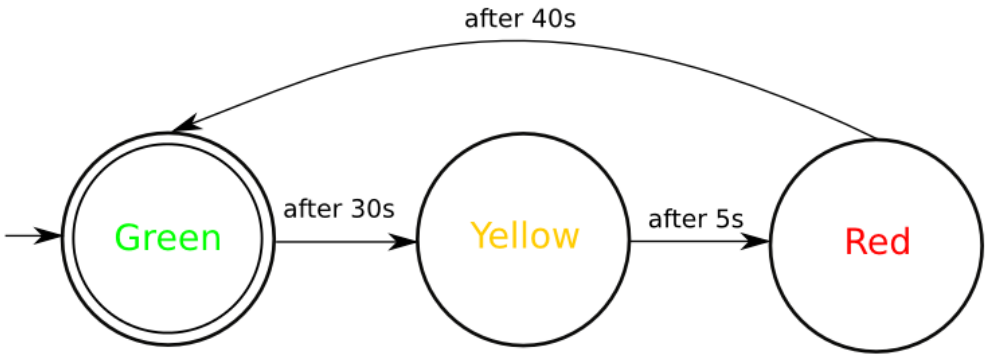


FIGURE 2.1: Finite transition system regulating a traffic light.

- $T \in L \times L$ *is a set of transitions, or edges.*

An execution of a transition system is a sequence (finite or infinite) of n *states where for all* $i < n$, $(l_i, l_{i+1}) \in T$.

Two different structures are derivable from transition systems.

- Automatons are transition systems where edges are labeled with elements of the alphabet. In other words, $T = L \times \mathcal{A} \times L$. A word $\omega = \omega_1\omega_2...\omega_n$ of size n belong to the language of the automaton iff there exists an execution $\pi = \pi_1\pi_2...\pi_n$ of A of same size such that for all i, $(\pi_i\omega_i\pi_{i+1}) \in T$ and $\pi_n \in F$.
- Kripke structures are transition systems where locations are labeled with elements of the alphabet. In other words, they are extended with a labeling function $\eta : L \mapsto \mathcal{P}(A)$ associating to each state a symbol or a set of symbols. A word $\omega = \omega_1\omega_2...\omega_n$ of size n belong to the language of the Kripke structure iff there exists an execution $\pi = \pi_1\pi_2...\pi_n$ of A of same size such that for all i, $\omega_i \in \eta(\pi_i)$ and $\pi_n \in F$.

The language of a transition system S is denoted $\mathcal{L}(S)$. As words and executions are closely related, they can be assimilated to each other. Transition systems are structural representations of a system composed of a given number of states (potentially infinite) and transitions linking those states. For example, Figure 2.1 depicts a transition system regulating a traffic light. It has an initial state, green, from which it starts. After 30 seconds the light turns yellow, 5 seconds later it turns to red, and finally after 40 seconds it goes back to green. Seen as an automaton on the alphabet $\mathcal{A} = \{wait\ 30s - wait\ 5s - wait\ 40s\}$, its language would be concatenations of the word *wait 30s - wait 5s - wait 40s*. Seen as a Kripke structure on the alphabet $\mathcal{A} = \{green, yellow, red\}$, its language would be concatenations of the word *green - yellow - red* When L is finite (respectively infinite), we will say that the transition system is finite (infinite). Multiple transitions going out of a single

state with non-excluding transitions labels (i.e. multiple transitions can be chosen) implies a non deterministic choice between the different reachable states. When such construction occurs in a transition system, we say that it is *non-determinisitc*.

Turing machines are a specific class of transition systems. They are defined as an infinite set of states $L = \{l_i : i \in \mathbb{Z}\}$, called a *tape* where there exists bi-directional transitions between l_i and l_{i+1} for all $i \in \mathbb{Z}$. Each state is associated to a *memory cell*, empty at the beginning, in which it can read and write data. Transitions are provided with conditional rules and rewriting instructions. For example, a Turing machine writing the value $101010...$ can be defined as follows:

1. If the current state is l_n and l_{n-1} is empty or contains 0, then write 1 and go to l_{n+1}.
2. If the current state is l_n and l_{n-1} contains 1, then write 0 and go to l_{n+1}.

Turing machines allows computing any algorithmic computation, which makes them a convenient formalism for studying actual computer systems.

2.2.2 Computer systems

Computer systems are basically machines manipulating a memory composed of different chunks of data. Some chunks can be reserved to receive a specific value and associated to a name. We refer to these associations as *program variables*, or simply variables. A program is based on a set of instructions that alters variables. These instructions compose a programming language. Most of the languages used in today's programs (C, Java, HTML 5 + CSS 3, ...) are Turing-complete, i.e. are as expressive as Turing-machines. The Rice Theorem [Ric53] states that the class of non-trivial properties are undecidable in general for Turing-complete systems. Instead of working on real-life languages, we will mainly work on a toy language that is Turing-complete so that the studies of the Part II on this simple language have a meaning for real-life programs. Part III will then apply these methods on programs written in the C language.

Let Var the set of variables used by a program. Variables take their value in a set defined by their type, but to simplify our analysis, we will use $\mathbb{R}$ for all variables. A program state is then a partial mapping $Var \rightharpoonup \mathbb{R}$. Any given program only uses a finite number n of variables, thus program states can be represented as vectors $X = (x_1, ..., x_n)^t$. Finally, we assume that for all programs, there exists $x_{n+1} = \mathbb{1}$ a constant variable always equal to 1. This allows representing any affine assignment by a matrix.

Conditional instructions (loops and conditional paths) are considered non-deterministic. The expression $non\_det(exp_1, exp_2)$ returns a random value between the valuation of exp_1 and exp_2 when the program reaches this location. Multiple variables assignments occur simultaneously within a single instruction. We say an assignment $X = exp$ is affine when exp is an affine combination of the variables. Also, we say that an instruction is non-deterministic

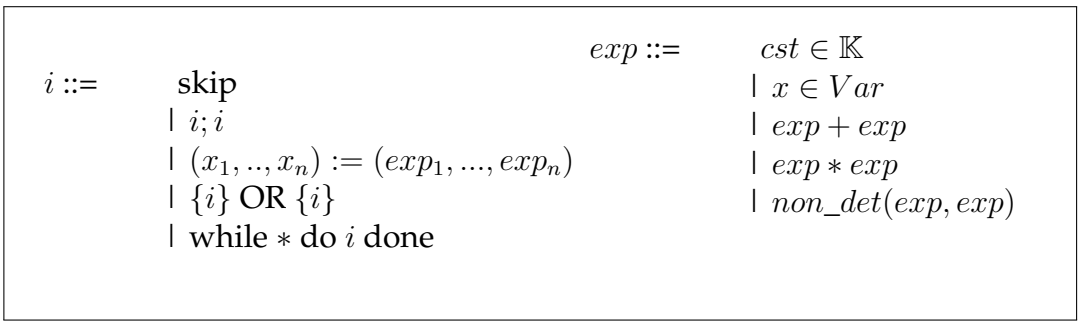


FIGURE 2.2: Code syntax

when it is an assignment in which the right value contains the expression non_det.

For any variable v and any assignment, we denote v' the new value of v after the application of the assignment. This notation is extended to vectors instead of variables and applications instead of instructions in general.

Property 1 *This toy language is Turing-complete.*

Proof. By [Min67], a finite state machine with at least two counters, instructions on transitions manipulating them and one initial state is Turing-complete. Instructions on these counters are of the form $x = \alpha * x + \beta * y + \gamma$, where x and y are variables and α, β and γ are coefficients in $\mathbb{Z}$. y also can be modified by such instructions.

Let F a finite state machine manipulating counters. We will build a program with our toy language that simulates F For each state s of F, we define a variable $v_s \in \{0, 1\}$. The variable v_i associated to the unique initial state i is set to 1, the others to 0. We also add a variable for each counter. Then, we add a loop in which there will be multiple assignments. If there exist a transition from s to s', then we add the instruction $(v_s, v_{s'}) = (0, v_s)$. If there are multiple transitions going out of s, then their corresponding instructions are put is an OR instruction. The counter instruction $x = \alpha * x + \beta * y + \gamma$ follows the previous instruction and is written: $x = (1 - v_{s'}) * x + v_{s'} * (\alpha * x + \beta * y + \gamma)$.

Hence, we built a program that computes F. □

2.3 Model checking

2.3.1 Models

Program representation as a transition system is part of a larger conception of formal verification called *model-based verification*. Let us recall the traffic light example, a transition system with three states. A program that implements such a controller must take care of many parameters, like captors or possible physical breakdowns of the system. The robustness of the system must be guaranteed in every case. An implementation of a traffic light controller doesn't need to match precisely the corresponding automaton, but its states must follow a certain pattern : initially, the program is in a green state,

then in a yellow and a red state, and finally goes back to a green state. In other words, it must accept a specific language (here, the language is the repetition of green, yellow and red). Hence, it is possible to specify the behavior of programs with automata. Model based specification is widely used in the context of verifying programs containing infinite loops. Such programs are indeed hard to specify with contracts as relations between the input and the output are irrelevant for non terminating programs.

In concurrent programming, safety, and liveness constrants are expected to be met, especially in embedded systems. When multiple processes work on a same ressource, locks can be used to enforce the coherence of modifications performed by each process. Then, the following examples of requirements may need to be met:

- during a given event, the lock must not be taken (safety requirement);
- if an event occurs in the function f, then when f has been called, the lock was not taken (contextual requirement);
- every function must free the lock before it returns (liveness requirement).

While these properties can be represented by automata, it is clearer to express them as readable sentences. In fact, they can be expressed via *temporal logics*.

2.3.2 Temporal logics

Instead of using automata, *temporal logics* [Pnu77] can be used.

Definition 4 *Let AP an alphabet. Operators of Temporal Logics are defined as:*

$$\varphi ::= \top \mid p \in AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid X\varphi \mid \varphi U \varphi$$

Given a successor function $succ : \mathbb{N} \to \mathbb{N}$, the satisfaction relation $\vDash$ of a Temporal Logic formula φ by a sequence $\pi = (\pi_i)_{i\in\mathbb{N}}$ where:

$$\begin{array}{lll}
\pi_i \vDash \top & & \\
\pi_i \vDash p & \Leftrightarrow & p \in \pi_i \\
\pi_i \vDash \neg\varphi & \Leftrightarrow & \pi_i \nvDash \varphi \\
\pi_i \vDash \varphi \wedge \psi & \Leftrightarrow & \pi_i \vDash \varphi \text{ and } \pi_i \vDash \psi \\
\pi_i \vDash X\varphi & \Leftrightarrow & \pi_{succ(i)} \vDash \varphi \\
\pi_i \vDash \varphi U \psi & \Leftrightarrow & \pi_i \vDash \psi \vee (\pi_i \vDash \phi \wedge \pi_{succ(i)} \vDash \varphi U \psi)
\end{array}$$

Operators $\vee, \Rightarrow$ and $\Leftrightarrow$ are defined as usually from the operators $\neg$ and $\wedge$.

Temporal logics have been developped as a specification language readable intuitively. If the elements of the alphabet are events, the property $(\neg t)\ U\ t'$ is read as *"event t must not occur until event t' occurs"*. Hence, a temporal property defines a language on an alphabet of events.

One of the very first Temporal Logic is the *Linear Temporal Logic* (or LTL). It is defined as a Temporal Logic of Definition 4 with $succ(i) = i + 1$, and its

language is equivalent to the language accepted by a generic kind of automaton, called *Büchi transition system*.

Definition 5 *A Büchi transition system is a finite automaton* (L, I, δ, F) *where* $F \subset L$ *a set of accepting states. The language of such a structure is composed of words for which there exists a finite accepting execution of the automaton* or *an infinite execution passing infinitely many times through an accepting state.*

2.3.3 Model-checking

Specifying a program with an automaton give a temporality to the property that we want to check. In other words, it is not a static property that we want to prove, but a property that evolves over time. Therefore, when the program performs a step, the automaton must check that this step is consistent with its own behavior. For example, a traffic light controller program must not go directly from green state to red state.

In order to prove the correctness of the program with respect to an automaton, the model-checking algorithm relies on *transition system product*.

Definition 6 *Let* A *and* B *two transition systems.* $C = A \times B$ *is the product of* A *and* B *such that every execution of* C *corresponds to an execution of* A *and* B*. States of* C *are pairs of states of* A *and states of* B*, and initial states (respectively final states) of* C *are pairs* (a, b) *such that* a *is an initial state (resp. a final state) of* A *and* b *an initial state (resp. a final state) of* B*. There exists a transition from* (a, b) *to* (c, d) *if* A *defines a transition from* a *to* c *and* B *defines a transition from* b *to* d*.*

For example, Figure 2.3 depicts the product between the traffic light controller and a morning-afternoon controller, keeping track of when it is the morning and when it is the afternoon. The resulting transition system contains every possible state reachable by a traffic light controller that keeps track of mornings and afternoons.

Instead of checking if the program P verify the specification, a model-checking procedure tries to find an execution of P that doesn't verify it. To do so, two transition systems are necessary:

- a Kripke structure R representing the program, i.e. accepting executions are effective executions of the program;
- an automaton L that accept every word *that doesn't verify the specification*.

The product of A and L will be an automaton accepting every effective execution of the program that doesn't verify $\mathcal{L}$.

2.3.4 Limitations

This intuition of the model-checking algorithm faces multiple limitations. Model-checking is in general undecidable for actual programs as many automata allow specifying the halting problem. Also, a program is often too

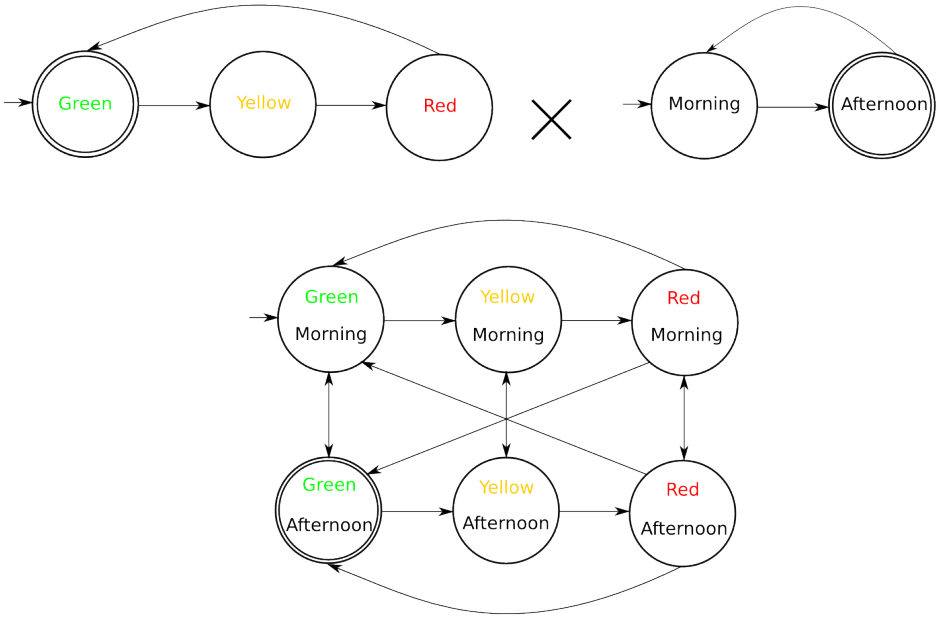


FIGURE 2.3: A traffic light controller coupled with a morning-afternoon controller.

complex to be represented as a finite automaton on a given language. Hence, a computable model-checking procedure requires to strongly abstract the automaton representing a program so that it becomes less expressive than a Turing machine (otherwise it would contradict the Rice theorem) or reduce the expressivity of the specification languages. In general, the choice is made of preserving the soundness of the analysis, i.e. if the procedure does not find a counter-example, then the program is valid (i.e. it verifies the specification). On the other hand, if a counter-example is found, it does not necessarily indicate that the program is invalid: the counter-example might be spurious. In that case, techniques like CEGAR [Cla+00] infers information from spurious counter examples to simplify the product automaton.

Another issue of model-checking of temporal properties comes from the exponential size of the automaton accepting the same language than a temporal formula. A Büchi automaton has an exponential number of states in the size of the automaton it is associated to. The transition systems product and the search of counter examples doesn't scale in practice

2.3.5 The temporal logic CaRet

Recursive state machines.

Recursive state machines, or *RSM*, are Kripke structures equivalent to pushdown automata. Intuitively, they are sets of standard transition systems with multiple initial and final states. Each transition system is granted the right to *call* another system of the set. Their shape fits well the inter-procedural

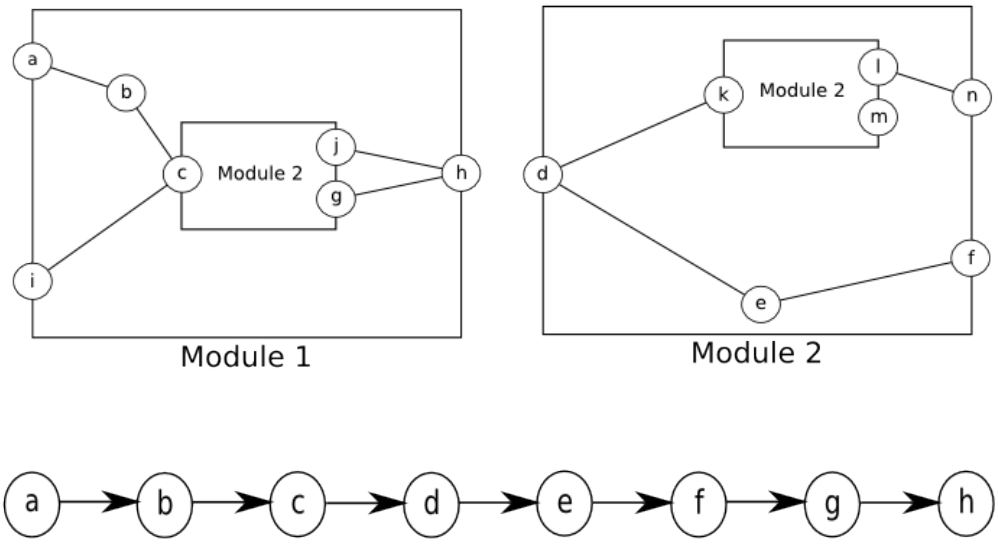


FIGURE 2.4: Example of an *A RSM* and one of its possible execution (or word).

control flow graph of a program. Figure 2.4 represents an example of *RSM*. The nodes c and k are able to call the Module 2 and the nodes j, g, l and m are return sites.

Definition 7 *Let AP a set of atomic propositions and Γ an alphabet. A* recursive state machine *R over AP is a tuple $(M, \{R_m\}_{m \in M}, \eta, init)$, where M is a finite set of labels.*
For each $m \in M$, exist a module

$$R_m = (N_m, B_m, Y_m, En_m, Ex_m, Call_m, Ret_m, \delta_m)$$

such that:

- *N_m is a finite set of nodes, each associated to a letter $\gamma \in \Gamma$.*
- *B_m is a finite set of boxes.*
- *$Y_m : B_m \to M$ associates each box to a module.*
- *En_m and Ex_m are two non-empty subsets of N_m respectively representing entry and exit nodes of a module.*
- *$Call_m = \{(b, e) | b \in B_m, e \in En_{Y_m(b)}\}$ and $Ret_m = \{(b, x) | b \in B_m, x \in Ex_{Y_m(b)}\}$*
- *$\delta_m : N_m \cup Ret_m \to 2^{N_m \cup Call_m}$ defines transitions between nodes.*

  When considering the whole automaton, the same notations are kept without the m index (the set of nodes for the whole RSM is denoted N).

- $\eta : (N \cup Call \cup Ret) \times AP \to \{\top, \bot, \star\}$ *is a labeling application associating to each couple (node,atomic proposition) a truth value* $\top$ *if the property is correct,* $\bot$ *if it is not.*
- $init \subseteq N$ *are the initial states of* R.

Remark.

Nested words.

The execution path of Figure 2.4 is linear, in the sense that it always goes step by step toward the same direction. The presence of call and return sites allows to consider executions not as simple words, but as *nested words* [AM09].

Definition 8 *Let* Γ *an alphabet,* $\Gamma^* = \Gamma \times \{\texttt{call}, \texttt{ret}, \texttt{int}\}$ *the extended alphabet of* Γ. *A nested word is a well parenthesized word with respect to* `call` *and* `ret`.

Executions of an *RSM* (and of an *A RSM*) can be seen as a nested word (Figure 2.5) The whole sequence of events is represented as the general path of

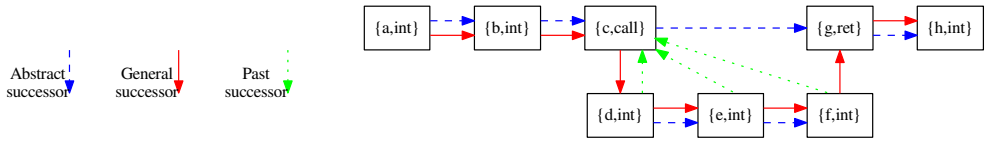


FIGURE 2.5: Execution of the *A RSM* in Figure 2.4 seen as a nested word.

the execution, while the sequence of events *in a single module* is called the abstract path, linking every `call` to the corresponding `ret`). Contextual properties require to have visibility of what happened at the call site of the current module: this is provided by the past path, linking every step to its associated call site.

Definition 9 *The partial successor applications* $succ^g_\gamma$, $succ^a_\gamma$ *and* $succ^-_\gamma$ *are defined such that:*

- $succ^g_\gamma$ *: the linear successor,* $succ^g_\gamma(i) = i + 1$.
- $succ^a_\gamma$ *: the abstract successor, pointing to the next local successor, i.e.* $succ^a_\gamma(i) =$ *the index of the associated* `ret` *if* γ_i *is a* `call`, $i + 1$ *otherwise.*
- $succ^-_\gamma$ *: the past successor, associating* γ_i *to the* `call` *site of the current module.*

When there is no ambiguity on γ, $succ^b_\gamma$ *will be denoted as* $succ^b$ *with* $b \in \{a, g, -\}$.

node	a	b	c	d	e	f	g	h
$succ^g$	b	c	d	e	f	g	h	$\perp$
$succ^a$	b	c	g	e	f	$\perp$	h	$\perp$
$succ^-$	$\perp$	$\perp$	$\perp$	c	c	c	$\perp$	$\perp$

FIGURE 2.6: Results of the application of the successor application on each node of the execution example of Figure 2.4

```
int x=0 ;

void lock(void) { x = 1; }

void unlock(void) { x = 0 ; }

int canAccess(void){ return x == 0; }
```

FIGURE 2.7: Simple C representation of a lock.

For example, let the word

$$\gamma = (a, \texttt{int}), (b, \texttt{int}), (c, \texttt{call}), (d, \texttt{int}), (e, \texttt{int}), (f, \texttt{int}), (g, \texttt{ret}), (h, \texttt{int})$$

depicted in Figure 2.5. The module 2 is called from the module 1 from c, and returns in g. The successors applications $succ^b_\gamma$ are presented in Figure 2.6.

The *CaRet* Temporal Logic

Similarly to defining a Temporal Logic for standard words, nested word can be extended with their own temporal logic using the new successor functions.

Definition 10 *Let AP a set of atomic propositions. The Temporal Logic CaRet is defined by the following operators:*

$$\varphi ::= p \in AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid X^g\varphi \mid X^a\varphi \mid X^-\varphi \mid \varphi U^g \varphi \mid \varphi U^a \varphi \mid \varphi U^- \varphi$$

where X^b and U^b for $b \in \{g, a, -\}$ are defined as Temporal Logic operations of Definition 4 with the successor function $succ_b$.

Specification examples. In Section 2.3.1 were presented three possible requirements that would be useful to express with temporal logics. Figure 2.7 depicts a naive implementation of a lock, represented by x. If x is equal to 0 the resource is free, otherwise the resource has been locked.

1. $G^b\ (p \Rightarrow x == 0)$, or when an event p occurs, then the lock must not be taken. p could be the call of the function $lock$ for example.
2. $G^g\ (p \Rightarrow X^-(x == 0)$, or when p occurs, the lock was not taken when the current function was called.

3. $G^g\ (F^a x == 0)$, or anytime, the lock must have beed freed before termination of the current function.

2.4 Invariance and inductivity

2.4.1 Floyd-Hoare axiomatic semantics

In order to prove that an imperative program satisfies a given specification, the main approach remains the Floyd-Hoare style of axiomatic semantics. The Floyd-Hoare logic allows to express elegantly the relation between predicates Pre and $Post$ respectivelely before and after the application of a statement S as follows:

$$\{Pre\}\ S\ \{Post\}$$

For example, the Floyd-Hoare triplet $\{x + y = 0\}\ x = x + 1\ \{x + y = 1)$ is valid.

2.4.2 Contracts

Symbolic computation is the branch of formal methods that analyses the behavior of a program by symbolizing program states by predicates, and propagates these predicates through the *control flow graph* of the program (i.e. the transition system with program instructions on its transitions). Reasonning about the program requires to perform different steps on the programs and verifying properties along the control flow graph. This is called *deductive verification*. In this approach, programs must be annotated with *contracts* on their behaviors (relations between input and output) and *assertions* giving, once proven, informations on the program at a given point. Even in model-checking, assertions plays a fundamental role in the construction of the Kripke structure and more precisely, the definition of the labelling function η supposed to link a program state to the set of properties that are verified at this state. In particular, loops have to be annotated with *loop invariants*.

Example of deductive verification with loop invariants The use of invariants is introduced by Floyd [Flo67] and Hoare [Hoa69]. Let us consider the Euclid's algorithm of Figure 2.8.

Let us assume this algorithm satisfy the precondition $x > 0$ and $y > 0$. This algorithm must satisfy the post-condition :

$$Result = gcd(x, y)$$

where x, y are natural integers and gcd is the greatest common divisor of a and b. We can notice that this loop admits multiple invariants:

- $a > 0$
- $b > 0$
- $gcd(a, b) = gcd(x, y)$.

```
a = x;
b = y;
while (a != b) do
 if (a > b)
   a = a-b;
 else
   b = b - a;
done
Result = a;
```

FIGURE 2.8: An implementation of the Euclid's algorithm. The *if* and the *while* conditions appear to match the actual algorithm.

Proving the two first invariants is easy, as a and b are compared before getting assigned. The third can be proven by induction. It is true at the first iteration of the loop as $a = x$ and $b = y$. Assume now that for a given iteration, the invariant holds. Two cases are possible, either $a > b$ or $b > a$. The two possibilities leads to the respective Hoare triplets $(gcd(a, b))\ a = a - b\ (gcd(a - b, b))$ and $(gcd(a, b))\ b = b - a\ (gcd(a, b - a))$ It is easy to prove that $gcd(a - b, b) = gcd(a, b)$ when $a > b$, as well as $gcd(a, b - a)$ when $b > a$. Also, $a = b$ is not possible at the beginning of the loop as it would contradict the loop condition. Therefore, the invariant $gcd(a, b) = gcd(x, y)$ holds. When the loop ends, $gcd(a, b) = gcd(a, a) = a$, it naturally comes that $a = gcd(x, y)$ by the last invariant.

2.4.3 Inductivity

Invariants that are preserved by a loop iteration are called *inductive* invariants. This denomination is significant as not every loop invariant is preserved by a loop iteration. Indeed, take for example the loop in Figure 2.9. In Chapter 5, we will prove that this loop admits the inductive invariant $x^2 + y^2 \leqslant 2$. This invariant directly implies that $x \in [-\sqrt{2}, \sqrt{2}]$ and $y \in [-\sqrt{2}, \sqrt{2}]$ are also invariants of the loop. These invariants are however not inductive as if $x = \sqrt{2}$ and $y = \sqrt{2}$, then after one loop step we have $y = 1.32\sqrt{2} \notin [-\sqrt{2}, \sqrt{2}]$.

2.4.4 The field of invariant generation

The large size of industrial programs make the manual writing of invariants burdensome, if not humanely impossible. That is why formal tools often rely on invariant synthesizers to automatically generate loop invariants. Different approaches have been developed in this direction[1].

[1] Abstract interpretation is not mentioned in this list though it plays a major role in the invariant generation field. It is treated in the next section.

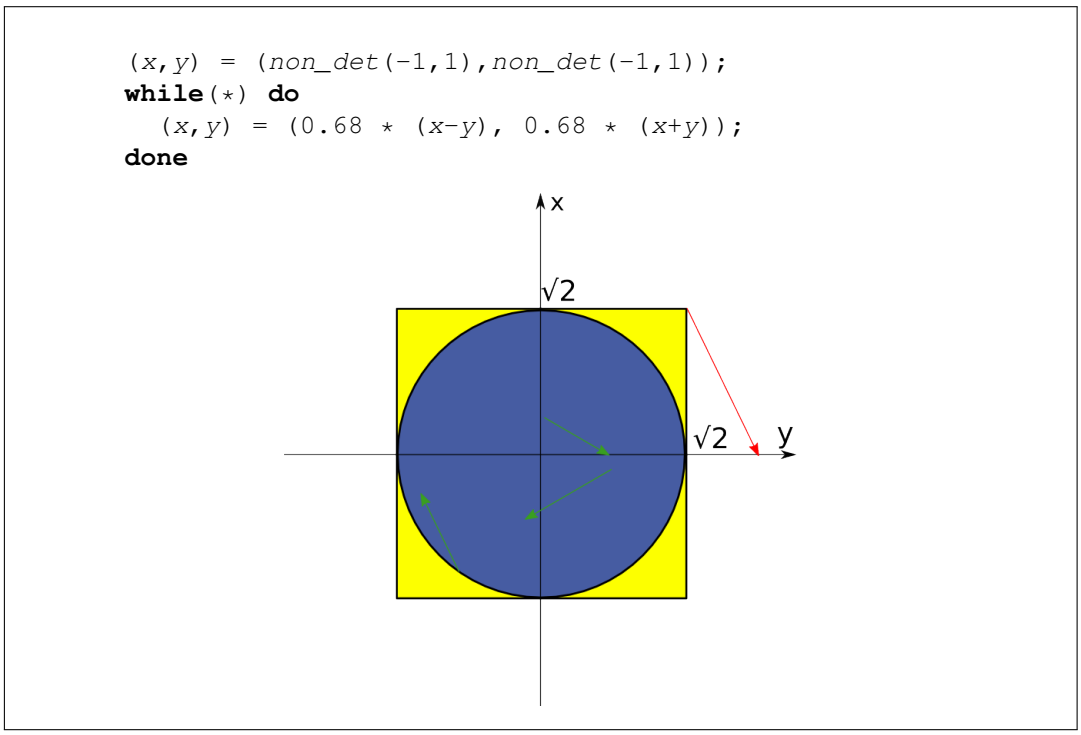


FIGURE 2.9: Affine loop admitting $x^2 + y^2 \leqslant 2$ as an inductive invariant (in blue). The yellow square is also an invariant, but it is not inductive.

Dynamic analysis

The most intuitive manner to check if a program behaves well or not is to manually check the output of a random execution. Dynamic analysis is the automation of the analysis of execution paths. *LLVM* [LA04] (for Low Level Virtual Machine) has its own dynamic tester, AddressSanitizer [Ser+12] that detects memory errors. E-ACSL [SKV17], part of the *Frama-C* [Kir+15] suite, is based on symbolic and dynamic execution of C programs annotated with *ACSL* [Bau+16] predicates. While it guarantees the presence of a bug if the execution doesn't validate its expected behavior, dynamic analysis is unsound as it can miss a undesired behavior or generate a false invariant due to the large size of the input set. Proofs of reliability cannot be inferred from dynamic analysis alone: it has to rely on static analyzers.

Invariant synthesizers also have their dynamic equivalents. The most widely used is *Daikon* [Ern+01] that tries to infer likely invariants, i.e. valid invariants for a large amount of loop iterations and therefore, good invariant candidates. These candidates must be proven valid afterwards as they may be valid for every execution tested, but without a full coverage of all possible executions, there may exists an execution that doesn't verify the invariant.

Acceleration

Dynamic analysis may be prohibitive due to its general imprecision. It generates invariants procedurally, in the sense that it is not the loop in itself that

is analyzed but its behavior with respect to an input. When verification requires a precise study, it may be tempting to guess the exact set of reachable states, or at least give a controlled over-approximation. In the field of linear loop invariant generation, *acceleration* [Bar+05] has shown to provide excellent results in terms of precision. Accelerating techniques are based on linear algebra properties, taking advantage of *finite monoïd transformations* [GS14] simple behavior. Finite monoid transformations are affine transformations on a vector of variables x such that $x = Ax + b$, where b is a vector and A is a matrix such that there exists m and n such that $A^m = A^{m+n}$. The simple form of the matrix A allows to get a linear relation between the initial state and the loop counter. When linear transformations are not in this class, alternatives involving matrix parametrization [JSS14] are used to over-approximate the behavior of the transformation.

Direct techniques

Sometimes, a loop has a particular shape that allows mathematical theorems to be directly applied to find an invariant. For example, *solvable loops* of [RK07] can be handled with Gröbner bases to generate an ideal of polynomial that contains all the invariants of the loop. Other techniques, like Karr's algorithm [Kar76] find invariants in simple linear programs given any form of initial equality relation between variables. This algorithm can be extended with polynomial invariants with the elevation technique of [MS04].

2.5 Abstract interpretation

Floyd-Hoare triplets are nice formalisms for reasoning on programs, as they keep track of every possible behavior. In practice, not every piece of information is relevant, which leads to bad computation time in generating the predicates and proving the property. Abstracting predicates is often a good trade-off between precision and computation time.

2.5.1 Intuition of abstract interpretation

As we saw in the Introduction of this thesis, keeping track of every possible behavior of a program is necessary to prove its correctness. In static analysis, behaviors are represented as contracts and hints provided by the user to direct the proof of preconditions and postconditions. Very often, these contracts manipulates relations between variables (in our example, inputs are positive integers and the output is the gcd of the input). Keeping track of every relation between variables can be difficult, especially when conditions occurs as it is then necessary to keep track of each possibility. This would result in an exponential number of relations in the number of conditions to keep track of.

It is often sufficient to focus on specific properties of variables. For example, arithmetic overflows[2] can be detected by analyzing only the intervals in which the variables evolve.

Abstract interpretation [CC77] is a framework based on the analysis of specific kind of properties. Instead of computing every possible information, an abstract interpeter transfers *abstract values* through the program. These abstract values represent an abstraction of the concrete program state, or in other words, a predicate on the state that is verified at a given program point. An *abstract semantic* is defined for each instruction so that they also can alter abstract states (as the concrete semantics of an instruction alters the memory).

2.5.2 Abstract domains

Let P a set of atomic predicates associated to a partial ordering $\preceq$ such that:

- $\top \in P$ represents a predicate that is always true, and for all $v \in P$ we have that $v \preceq \top$;
- $\bot \in P$ represents a predicate that is never true, and for all $v \in P$ we have that $\bot \preceq v$.

We add to P a *join* operator, i.e. an operator $\sqcup$ associating to two values v and w a value z such that z is the smallest value such that $v \preceq z$ and $w \preceq z$. $(\preceq, \sqcup)$ defines a *lattice*. Abstract interpretation is based on two lattices: the concrete set and the abstract set. These two sets are related by two total applications that maps one element of a set to an element of the other. The *concretization application* γ maps an element $v^\sharp$ of the abstract set to an element v of the concrete set, that is $v = \gamma(v^\sharp)$ is a concretization of $v^\sharp$. Similarily, the *abstraction application* α maps v to $v^\sharp$, and $v^\sharp$ is an abstraction of v. The couple (α, γ) forms a *Galois connection* of the two lattices, i.e. the following relation is satisfied:

$$\alpha(v) \preceq v' \Leftrightarrow v \preceq \gamma(v')$$

2.5.3 A semantics on abstract values

The concrete semantics is a function f_i mapping an instruction i and a concrete value v to the image of v after the application of the instruction i. A semantics f_i' is a *valid abstraction* of f_i if for every abstract value v, the concretization of $f_i(v)$ is lower or equal (with respect to $\preceq$) than the concretization of $f_i'(v)$. In other words:

$$(f_i \circ \gamma)(v) \preceq (\gamma \circ f_i')(v)$$

Example. Let us consider the program in Figure 2.11 manipulating three integers x, y and z. Our goal will be to prove that $z \in [0, 1]$ at the end. The

[2] Values of variables are encoded in a finite number of bytes depending on its type. Overflows occur when the result of an operation gets higher than the maximal encodable value for the type.

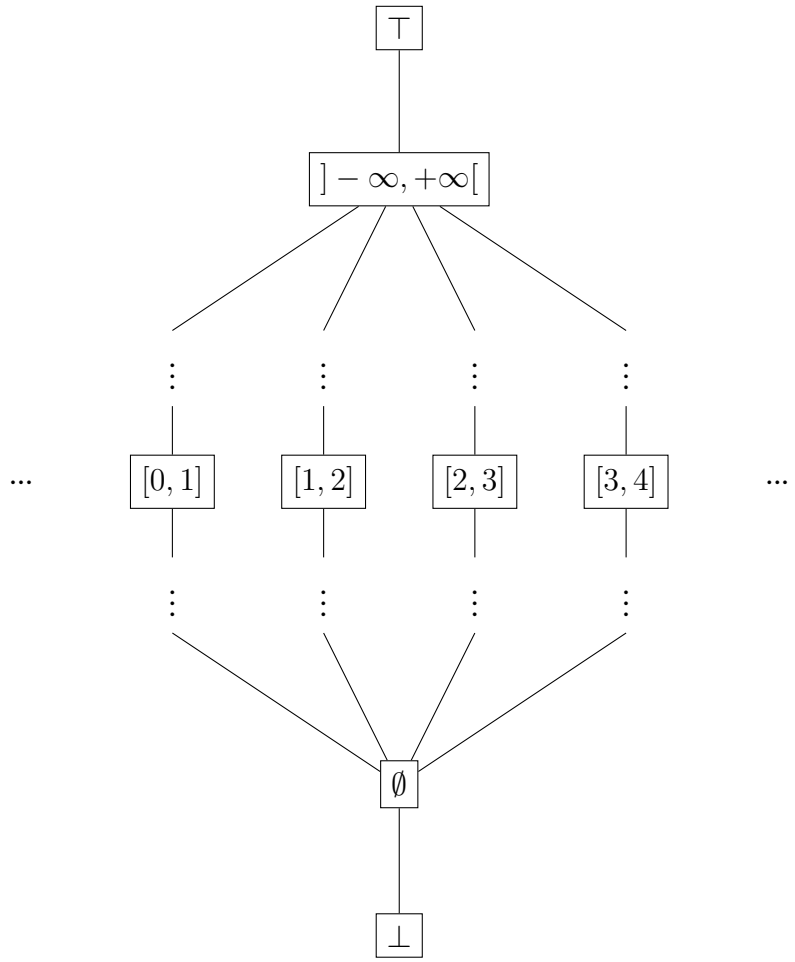


FIGURE 2.10: A lattice for the inverval abstract domain. The partial order can be defined with the inclusion: $p \preceq q \Leftrightarrow p \subseteq q$. We find in this lattice intervals and union of disjoint intervals.

```
   y = [-1,1];
1. x = y
2. OR
3. x = y + 1;
4. z = x - y;
```

FIGURE 2.11: Example of a very simple program starting with the initial state $y \in [-1, 1]$.

concrete set V is defined as $\mathcal{P}(\mathbb{Z})$, the concrete semantics is the application of an assignment on the left term. The semantics for the condition instruction is defined as the union of the two sets (this is the join operator).

First, let us analyze this program with the interval abstract domain I as defined in Figure 2.10. The abstract semantics of intervals is defined as the concrete semantics applied to the bounds of the interval. The classical interval operations are defined as follows:

- $[a, b] + [c, d] = [a + c, b + d]$
- $k * [a, b] = [k * a, k * b]$ if $k > 0$, $[k * b, k * a]$ otherwise
- $[a, b] * [c, d] = [min(a * c, a * d, b * c, b * d), max(a * c, a * d, b * c, b * d)]$.

The abstract semantics on the intervals is easier to compute than the concrete semantics on sets, as the concrete semantics requires to apply an instruction on every element of the set where the abstract semantics is only interested in maximums and minimums.

The program starts with $y \in \{-1, 0, 1\}$ as a concrete state, which is abstracted by $y^\sharp = \alpha(\{-1, 0, 1\}) = [-1, 1]$. The first instruction is a condition, which respectively adds the information that $x^\sharp = [-1, 1]$ and $x^\sharp = [-1, 1] + [1, 1] = [0, 2]$. Therefore at the end of the condition, the union of the two intervals is performed, which returns $x^\sharp = [-1, 2]$. By the last instruction, we have that $z^\sharp = [-1, 2] - [-1, 1] = [-2, 3]$. The concretization $\gamma(z^\sharp)$ of $z^\sharp$ returns that $z \in \{-2, -1, 0, 1, 2, 3\}$. This result is not very precise considering that z can only be equal to 0 or 1. Note however that $[0, 1] \subset [-2, 3]$, hence the analysis remains sound.

The interval abstract domain is a *non-relational* domain as it doesn't keep track of relations between variables, which was the precision issue on the example of Figure 2.11. Relational abstract domains like the octagon abstract domain [Min06] allow to keep track of this information, but is generally slower and harder to implement. In general, properties inferred by abstract interpretations are over approximations of the set of possible states reachable at a given program point.

2.5.4 Loops and widening operators

When an abstract interpreter reaches a loop, it tries to guess an overapproximation of the reachable set of states at the beginning of the loop. In other words, it will try to compute an invariant of the loop. Loops can be seen as an undefinite number of conditions. Hence, their treatment require an undefinite number of applications of the join operator, which is not possible in general. For example, let us consider the loop starting at $x = 0$ and applying $x = x + 1$ until $x \neq N$, with N an integer. The interval abstract domain as we defined it will start with $x_0^\sharp = [0, 0]$, then after one loop iteration $x_1^\sharp = [0, 1]$, then $x_2^\sharp = [0, 2]$, etc.. If the value of N is positive, then it is possible to find after some applications of the join operator to find an *inductive abstract value* overapproximating as well as possible the program state. This abstract value is called the *smallest fixed point*. However, if N is negative, the smallest fixed

point would here be $x^\sharp_\infty = [0, +\infty[$. There exists an infinite number of abstract value between $[0, 1]$ and $x^\sharp_\infty$, therefore the analysis will not stop. One possibility to solve this issue is to use a *widening operator*.

Widening operators ∇ are similar to the join operator $\sqcup$ for a lattice L, except that they work on a different lattice L' with the order $\preceq_{L'}$. There must be in this new lattice *no strictly growing infinite chain* with respect to $\preceq_{L'}$. In other words, the widening operator will always be called a finite number of time as there exists a finite number of elements before reaching $\top$. For the interval abstract domain, $\preceq_{L'}$ can be defined as follows:

- $\bot \preceq_{L'} \emptyset$
- $]-\infty, +\infty[\preceq_{L'} \top$
- $\forall a, b : [a, b] \preceq_{L'} [a, +\infty[\preceq_{L'}]-\infty, +\infty[$
- $\forall a, b : [a, b] \preceq_{L'}]-\infty, b] \preceq_{L'}]-\infty, +\infty[$

With this new lattice, increasing an upper bound automatically sets it to $+\infty$ and decreasing a lower bound sets it to $-\infty$. Applying the widening operator on $x^\sharp_1 = [0, 1]$ and $x^\sharp_2 = [0, 2]$ results in $x^\sharp_\nabla = x^\sharp_1 \nabla x^\sharp_2 = [0, +\infty[$, which is here the smallest fixed point.

The widening operator doesn't always[3] return the smallest fixed point. If N had been calculated by the program in Figure 2.11, as z, the abstract interpretor would also conclude with $x^\sharp_\nabla = [0, +\infty[$ as it would consider N to be possibly negative. The smallest fixed point would be in this case $[0, 1]$, which is included in $x^\sharp_\nabla$.

2.5.5 A widely used framework

Since [CC77], abstract interpretation became a more and more influent actor of the formal verification field. Its genericity allows to define one's own abstract domain dedicated to the proof of very specific properties.

Abstract domains. The following list is a non-exhaustive list of different abstract domains that are used in modern abstract interpreters.

- *The octagon abstract domain* [Min06] keeps track of relations of the form $\pm x \pm y \leqslant k$, where x and y are variables of the program. This abstract domain have shown to be faster than the polyhedra abstract domain [CH78] that expresses general linear inequalities over the variables of a program.
- *The ellipsoid abstract domain* [Rou+12] propagates constraints as polynomials of degree 2. It has shown to efficiently approximate the behavior of convergent linear filters[4]. However, they sometimes fail to catch precise properties while domains *approximating ellipsoids* tend to be faster and more precise.

[3] In practice, almost never.
[4] We will study convergent lienar filters in details in Chapter 4.

- *The zonotope abstract domain* [GGP09] is based on a quantified representation of the possible values of variables. Each variable is associated to a sum of the form $\alpha_0 + \sum_{i=1}^{n} \alpha_i \varepsilon_i$ where α_i are coefficients defining the zonotope and ε_i are parameters that belong to $[-1, 1]$. Zonotopes are convex symetric polyheadra, hence they approximate elliposids quite well.

- *The gauge abstract domain* [Ven12] is based on the discovering of linear inequalities on linear programs. The representation of variables is similar to the zonotope representation, except the parameters do not belong to a fixed interval but are counters evolving in $\mathbb{N}$.

- *The relational shape abstract domain* [ILR17] is based on the discovery of relational properties over the memory manipulation of a program.

- Some approaches are directly inspired from abstract interpretation. For example in [MBR16], the initial state is divided in multiple subsets abstracted in a given domain. The inductivity of each subset is tested: if it is inductive, it is conserved in the final invariant; otherwise it is divided again, etc.

Abstract interpreters. The Frama-C framework [Kir+15] implements a plug-in, EVA [BBY17], that uses different domains provided by APRON abstract domain library [JM09] for the analysis of C programs. Polyspace [Deu03] is an abstract interpretation tool that have been initially developped as a prototype automatically detecting the Ariane 5 bug [Lan97] and which is now designed to analyze C and C++ source code. Fluctuat [Gou13] uses the zonotope abstract domain [GPV12] that handles floating point operations and approximations. Java also has its own abstract interpreter, the Julia Static Analyzer for Java [Spo82].

Part II

Polynomial invariants for polynomial loops

Chapter 3

Polynomial loops don't exist

Contents

Requirements: Linear algebra (Section 2.1)

The role of a loop statement is to compute a certain amount of time, possibly infinite, the same transformation. Expressing the exact transformation performed by the whole loop is as hard as determining the number of iterations n, in the sense that knowing n allows to unfold the loop and determining the exact operation described by the loop statement. When n is infinite, verification problems are to guessing whether if a given state is reachable in a finite but arbitrary number of steps [KL80] or not. Determining either n is undecidable in general, that is why the study of loops is a challenge in program analysis.

Consider a polynomial transformation f of degree d. As such, f belong to one of the following categories:

- f^n is exponential in n, for example if $f(x) = x^2$ we have $f^n(x) = x^{2n}$.
- f^n is polynomial in n, for example if $f(x, y) = (x + y^2, y + 1)$ we have $f^n(x, y) = (x + \frac{(y+n-1)(y+n)(2(y+n)-1)}{6}, y + n)$.

```
while(...){
  x = x + y * y;

  y = y + 1;
}
```

```
one = 1;
y2 = y * y;
while(...){
  x = x + y2;
  y2 = y2 + 2*y + one;
  y = y + one;
}
```

FIGURE 3.1: Example of polynomial loop and its linear equivalent

Let's focus on the second example $f(x, y) = (x + y^2, y + 1)$ in Figure 3.1. This transformation contains a polynomial expression, y^2. The value of y^2 after one iteration of f is $(y + 1)^2 = y^2 + 2y + 1$, which is a linear combination of y^2, y and 1. If we add variables y_2 and $\mathbb{1}$ respectively initialized to y^2 and 1, then $g(x, y, y_2, \mathbb{1}) = (x + y_2, y + \mathbb{1}, y_2 + 2y + \mathbb{1}, \mathbb{1})$ is a linear transformation that computes the same result than f.

The purpose of this chapter is to correlate the property of f^n to be a polynomial in n and the existence of a linear transformation g computing the exact same image given some preconditions over the initialization on the variables used by g. Section 3.1 formalizes *elevation*, a technique used for representing the evolution of monomials of variables transformed by an affine transformation without using polynomial expressions. Elevation can be applied to a certain class of polynomial transformations, known as *solvable* transformations. Section 3.2 defines the concept of *linearization*, allowing the representation of solvable transformations as linear transformations, like in Figure 3.1. It also proves the equivalence between the solvable transformation and the linearizable transformation classes. Section 3.3 introduces two algorithms. The first presents a quadratic test of the linearizability of a given polynomial transformation f. The second computes the linear transformation g that has the same image as f. Finally, Section 3.4 studies the eigenvector and eigenvector decomposition of an elevated transformation. This chapter is partially based on work that has been presented in [OBP16].

3.1 Elevation of linear transformations

3.1.1 Principle of the linearization

Linearization relies on the following observation: if some variables evolves affinely, any monomial composed of those variables also evolves affinely. For example, let $g(x, y) = (x+2, y+x)$. The value of x^2, denoted $(x^2)'$, after one application of g is $(x+2)^2 = x^2+4x+4$, which is an affine combination of x^2 and x Also, $(y^2)' = (y+x)^2 = y^2+2xy+x^2$ and $(xy)' = xy+x^2+2y+2x$ are affine combinations of x^2, y^2, xy, x and y. Finally, it is possible to replace the affine constants by a variable $\mathbb{1}$, which remains constant. As a result, the *linear* mapping $f(x, y, x_2, xy, y_2, \mathbb{1}) = (x+2.\mathbb{1}, y+x, x_2+4x+4xy+x_2+2y+2x, y_2+2xy+x_2, \mathbb{1})$

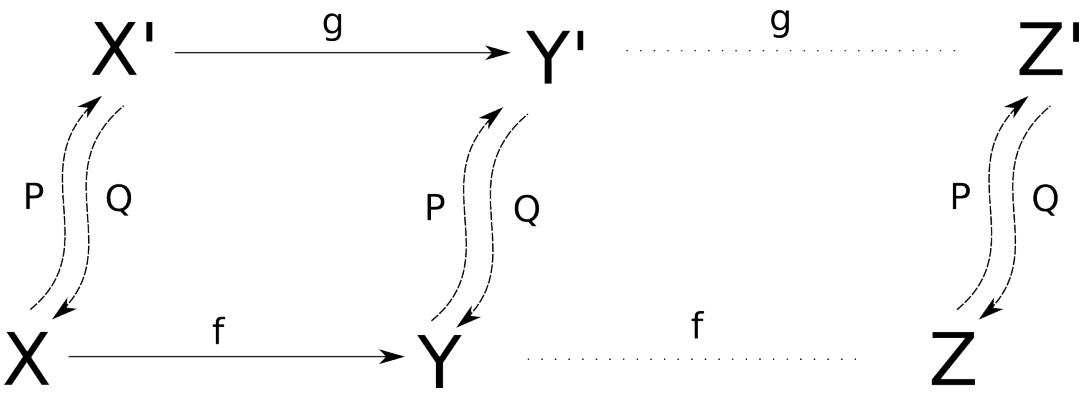


FIGURE 3.2: The elevation principle is closely related to the linear algebra notion of *similarity*. Changing the base X to the base Y is reversible before and after any number of application of the *similar* linear transformations f and g, i.e. for which there exists a linear invertible transformation P such that $g = P^{-1} \circ f \circ P$.

computes the same image as g, extended by the initial monomials value (Figure 3.2) where x_2, y_2 and xy respectively encode the monomials x^2, y^2 and xy. This technique is commonly used to generate polynomial invariants with a linear invariant inference algorithm, as for example in [MS04] using Karr's invariant inference algorithm [Kar76] for affine programs. We claim that every linear transformation g manipulating a vector of variables X can be *elevated* to express the behavior of any monomial of variables of X. In other words:

Property 2 *For every linear transformation $g : \mathbb{Q}^n \to \mathbb{Q}^n$ and every polynomial $P \in (\mathbb{Q}[X]^n)$, there exists a polynomial Q and a linear transformation f such that*

$$(f \circ Q)(X) = (P \circ g)(X)$$

Proof. Let $X = (x_1, ..., x_n)$ a vector of variables and $m = \prod_{i=1}^{n} x_i^{p_i}$ any monomial of degree d of those variables. The value of m', according to g, is

$$\prod_{i=1}^{n} (g(X)_{|_{x_i}})^{p_i}$$

which is a polynomial of degree d. It can then be expressed as a linear combination of monomials of degree d or lower. As this is true for any monomial of any degree, they all are linear combinations of monomials of degree d or lower. As such, the image of a monomials sum, i.e. a polynomial, is a polynomial of lower or equal degree. By setting Q the polynomial computing the required monomials to express m', $P \circ g$ can be expressed as a linear combination of monomials, whose new values are linear combinations of monomials.

3.1.2 Linearization

The elevation principle can be extended to polynomial transformations. Assume a monomial appears in a transformation, and that every variable of this monomial evolves affinely. By Property 2, it is possible to replace this monomial by a new variable evolving linearly. By extension, it is then possible to *replace a polynomial mapping by a linear mapping*.

Definition 11 *g is linearizable if there exists two polynomials P and Q such that $Q \circ P = Id$ and a linear mapping f such that for all X*

$$g(X) = (Q \circ f \circ P)(X) \tag{3.1}$$

Linearizability, just like elevation, is similar to changing the base of a given linear transformation, with the difference that the new base extends the old one by polynomial expressions on new dimensions (Figure 3.2). Note that we also have that $g^n(X) = (Q \circ f^n \circ P)(X)$ for any $n \in \mathbb{N}$ as $Q \circ P = Id$. In order to have this identity, it is possible to define P as a transformation performing, among other, the identity on each variables, and as Q the projection on the initial variables. For example, $P(x, y) = (x, y, x^2, x.y, y^2)$ and $Q(x, y, x_2, xy, y_2) = (x, y)$ verify $Q \circ P = Id$.

Example 1. Let $g_1(x, y, z) = (x + 2, y + x^2, z + y^2)$ a polynomial mapping. The only monomials appearing in g_1 are x^2 and y^2. As x evolves linearly, it is possible to express the value of $(x^2)' = x^2 + 4x + 4$. Given the right initial value of x^2, the mapping $f(x, y, x_2) = (x + 2, y + x_2, x_2 + 4x + 4)$ is an affine transformation that computes the value of y' with respect to g_1. In this case it is also possible to linearize $(y^2)' = y^2 + 2x^2y + x^4$, as x^4 and x^2y also are linearizable by expressing x^3 and xy. To precisely match Definition 11, $P(x, y) = (x, y, x^2, x^3, x^4, xy, x^2y, y^2)$ and $Q(x, y, x_2, x_3, x_4, xy, x_2y, y_2) = (x, y)$.

Example 2. Let $g_2(x) = x^2$. There is only one monomial, x^2, whose next value according to g_2 is $(x^2)^2 = x^4$. This new monomial has a higher degree and is directly dependent on x. This transformation is actually not linearizable, which could have been guessed from the beginning. Indeed, for any initial value of $x > 1$, its evolution through multiple executions of g_2 is exponential in the variable x $(2, 4, 16, 256, ...)$. There exists no affine transformation that can perform such a calculation.

3.1.3 Linearizable and exponential

There exist two different types of polynomial transformations. First, linearizable transformations like g_1 admit a polynomial behavior that can be simulated by an affine mapping. Next, non-linearizable transformations like g_2 allow an exponential growth that linear algebra cannot handle. The first class admits multiple types of transformations as we saw in Example 1:

- affine transformations (in the Example 1, x);

- polynomial transformations depending on affine variables (y depends on x);
- polynomial transformations depending on linearizable variables (z depends on y).

There exists a class of polynomial transformations introduced in [RK07] that are known to be linearizable, namely *solvable* mappings.

Definition 12 *Let $g \in \mathbb{K}[X]^m$ be a polynomial mapping. g is solvable if there exists a partition of X into sub-vectors of variables $w_1, ..., w_k$ such that $\forall j,\ 1 \leqslant j \leqslant k$ we have*

$$g_{w_j}(X) = M_j w_j^t + P_j(w_1, ..., w_{j-1})$$

with $(M_i)_{1 \leqslant i \leqslant k}$ a matrix family and $(P_i)_{1 \leqslant i \leqslant k}$ a family of polynomial mappings.

Section 3.2 will detail the link between solvable mappings and linearizability.

3.2 Linearization

3.2.1 Intuition

Linearization is inspired from Carleman linearization [KS91] that is used in the field of differential equations. Linearizing transformations requires to linearize successively each subset of the partition. w_1 is already affine, each variable of w_2 have in their expressions linear combinations of variables of w_2 and monomials of w_1. Thus they can be linearized, and so does w_3 for the same reason, etc.

3.2.2 Linearization theorem

This section is dedicated to the proof of the following theorem:

Theorem 1 *Let g be a polynomial transformation. g is solvable $\Leftrightarrow$ g is linearizable*

Solvable mappings are linearizable

Let $g \in \mathbb{Q}[X]^m$ be a solvable polynomial mapping. There exists a partition of variables $X = w_1 \cup ... \cup w_k$ such that

$$g_{w_j}(X) = M_j w_j^T + P_j(w_1, ..., w_{j-1})$$

We will prove that g is linearizable, i.e. there exists f, P and Q such that $Q \circ f \circ P = g$ with $Q \circ P = Id$. Let us take P the polynomial computing every monomial of variables of X and Q the projection on the initial variables (i.e. the monomials of degree 1). The idea is to prove that the image of every monomial described by P evolves linearly. We proceed by induction on the size k of the partition of the variables of g. We can state that :

- If $k = 1,\ X = w_1$, then $g_{w_1}(X) = M_1 w_1^T + P_1$, where P_1 is a constant. Then it is clear that g_{w_1} is an affine transformation.

- Assume we can compute a linear application f from g such that $g(X) = Q \circ f \circ P(X)$ if there exists a partition of k sets of variables satisfying the solvable hypothesis. Let h be a solvable polynomial mapping for which there exists a partition of X into $k+1$ subvectors of variables $X = w_1 \uplus ... \uplus w_{k+1}, w_i \cap w_j = \emptyset$ if $i \neq j$. By induction hypothesis, we can linearize h_{w_i} for $1 \leqslant i \leqslant k$. Now, the key point is to find a way to linearize
$$h_{w_{k+1}}(x) = M_{k+1} w_{k+1}^T + P_{k+1}(w_1, ..., w_k)$$
First, let's note that no variable of w_{k+1} have been used in any other h_i. Let $v = \prod_{i=0}^{n} v_i^{\lambda_i}$ a product of variables in $(w_1 \cup ... \cup w_k)$. It can appear in P as v^d, where d in an integer. We know, by induction hypothesis, that the evolution of v_i following the h transformation can be expressed as a linear application f with the help of extra variables.

  We can then use Property 2, stating that there exists Q and f such that $(f \circ Q)(X) = g(X)$ (here, g is a polynomial). Let Q the polynomial computing all the monomials of variables of w_1 up to the maximal degree of a monomial appearing in the expression of w_2. Expressions of variables of w_2 are now linear as all monomials are now replaced by new variables introduced by Q. By induction, let g can similarly be linearized up to w_{n-1}. The same argument as for w_2 can be used by using the partitionning $w_1' = w_1 \uplus ... \uplus w_{n-1}$ (*linearized* variables) and $w_2' = w_n$ (polynomials of newly linear variables).

  □

Non-solvable mappings are not linearizable.

Example 2 presented the non-linearizable transformation $g_2(x) = x^2$. Let us take a similar example $g_3(x, y) = (y, x^2)$. g_3 is not solvable, but no variable directly polynomially depend on itself. However the next value of the monomial xy is $x.xy$, a polynomial of xy.

There exists a link between the polynomial self dependency of variables (or monomials) and the non-linearizability, which was not the case in Example 1. Let us formally define *dependency* as follows:

Definition 13 *Let $g \in (\mathbb{K}[X])^m$ m polynomial mappings. Let x, y two variables (possibly equal). We define the dependency operators $\lhd$ and $\lhd\!\!\lhd$ such that:*

- $x \lhd y \quad \Leftrightarrow \quad y$ appears in the expression of x' (depends on)
- $x \lhd\!\!\lhd y \quad \Leftrightarrow \quad y$ is multiplied by at least one variable in the expression of x' (polynomially depends on)

By extension, a monomial of variables $m = \prod v^{p_v}$ linearly or polynomially depend on variables or monomial of variables, by considering $m' = \prod g_{|_v}(X)^{p_v}$
Note that $x \lhd\!\!\lhd y \Rightarrow x \lhd y$

Remark. Consider that $x \lhd y$ when y appears in the expression of $f(X)_{|_x}$ and $x \lhd\!\lhd y$ when y appears in a monomial of degree $d > 1$ in the expression of $f(X)_{|_x}$. In the case of the Example 1, the dependencies of y can be expressed as $(y \lhd y)$ and $(y \lhd\!\lhd x)$, while dependencies of x is $x \lhd x$ and dependencies of z are $(z \lhd z)$ and $(z \lhd\!\lhd y)$.

For the Example 2, it is clear that $x \lhd\!\lhd x$.

Property 3 *Let g a polynomial transformation. If there exists a monomial of variables m such that $m \lhd\!\lhd m$, then g is not linearizable.*

Proof. Let m a monomial of variables. If $m \lhd\!\lhd m$, then m appears in a monomial $m * n$ with n a monomial of degree at least 1 in the expression of m' of the new value of m after one application of g. The new value of $m.n$ will depend on $m.n^2$, that will depend on $m.n^3$, etc. Assume that g is linearizable, i.e. there exist two polynomials P, Q and a linear application f such that $g(X) = (Q \circ f \circ P)(X)$. P is a finite polynomial. By definition, m depend on $m.n$, thus f is necessarily able to compute the image of $m.n$ ($m' = m.n + ... \Rightarrow m.n = m' - ...$). Similarly, it necessarily must be able to compute the image of $m.n^k$ for any $k \in \mathbb{N}$. With the finite information given by P, can the finite linear application f express every monomial $m.n^k$? If so, there would exist then an expression of $c * m.n^k$ for all $c \in \mathbb{K}$ and $k \in \mathbb{N}$. Then, we would be able to express $x.(\sum_{i \in \mathbb{N}} \frac{m^i}{i!}) = x.e^m$ as a linear transformation. As e^m is not linear in m, this is clearly absurd. Thus, there exist no linear application capable of representing g if $m \lhd\!\lhd m$.

This property immediately allows concluding on the non-linearizability of g_2 in Example 2. The key of the proof is to extract a monomial that necessarily will depend on itself

Definition 14 *A set of variables $v_1, ..., v_n$ is a* dependency chain *if $\forall 1 \leqslant i < n, v_i \lhd v_{i+1}$. It is said* polynomial *if there exists a polynomial dependency in the chain, otherwise it is said* linear*. In any case, we say that v_1* eventually depends *on v_n. A* dependency cycle *is a chain verifying $v_n \lhd v_1$. If the cycle contains a polynomial dependency, we speak of a* polynomial dependency cycle*. Note that in that case, if we have $v_i \lhd\!\lhd v_{i+1}$, then $v_{i+1}, ..., v_n, v_1, v_i$ is also a polynomial dependency cycle, with a polynomial dependency between the last and first element: by convention, we will assume in the following that all such cycles are written that way.*

A difference between g_1 in Example 1 and g_2 in Example 2 is the existence of a dependency cycle with one polynomial dependency in the second case, while the first one only has linear dependency cycles. This observation can be generalized for solvable transformations:

Property 4 *g is solvable $\Leftrightarrow$ there exists no polynomial dependency cycle*

Proof.

- If g is solvable, then by definition there exists a partition $w_1, ..., w_n$ of variables such that

$$g_{w_j}(x) = M_j w_j^t + P_j(w_1, ..., w_{j-1})$$

Assume there exists a polynomial dependency cycle $v_1, ..., v_m$ with $v_m \in w_{m'}$. As $v_m \lhd v_1$, we have $v_1 \in w_k$ with $k < m'$ by definition of solvable transformations. Also, as $\forall i, v_{i-1} \lhd v_i$, we have that $v_{m-1} \in w_{k'}$ with $k' \leqslant k < m'$. Variables of $w_{k'}$ cannot depend on variables of $w_{m'}$, so v_{m-1} cannot depend on v_m, which is a contradiction. Thus, there exists no polynomial dependency cycle.

- If there exists no polynomial dependency cycles for a given transformation g, we will build by recurrence a partition of variables showing that g is solvable. First, we need the following lemma to simplify the proof.

Lemma 2 *Let V a finite variable set. If for any variable $v \in V$ there exists a chain $\mathcal{C} = v_1, ..., v_c$ with $v_1 = v$ containing a polynomial dependency, then there exists a polynomial dependency cycle in V.*

Proof. Because there is a finite number of variables, it is possible to show the existence of a cycle. Starting with $v = v_1$, we build the chain $\mathcal{C}_0$ up to v_i such that $v_{i-1} \lhd v_i$. Then, we build $\mathcal{C}_1$ from v_i which eventually polynomially depends on another variable v_j. If by building $\mathcal{C} = \mathcal{C}_0 \rightarrow \mathcal{C}_1 \rightarrow ....$ we get to a variable $z \in \mathcal{C}$, then there would exist a polynomial dependency cycle. As there is a finite number of variables and $\mathcal{C}$ grows, $\mathcal{C}$ eventually reaches a visited variable.

The initial set w_1 is characterized by two properties: variables evolve linearly and only depend on themselves.

1. First, there necessarily exist variables that evolve linearly. Otherwise, every variable v would polynomially depend on another variable. In other words, any variable v would start a polynomial dependency chain, which by lemma 2 implies the existence of a polynomial dependency cycle.
2. Let L the set of variables evolving linearly. To build w_1, we also need to show that there exists a subset of L that only depends on itself, i.e. any variable $\ell \in L$ linearly depends on variables of L. Assume that any variable $x \in L$ belongs to an affine dependency chain C ending with y and such that $y \notin L$. In other words, x eventually depends on y, which polynomially depends on another variable. Thus elements of L all start a polynomial dependency chain, inducing by lemma 2 the existence of a polynomial dependency cycle which is absurd. There exists thus a set of variables $w_1 \subseteq L$ depending only on themselves.

Assume now g is solvable from w_1 to w_j (i.e. satisfies the properties of a solvable mapping up to j) and that there are still variables not in w_i with $1 \leqslant i \leqslant j$. We will show that the set of variables polynomially depending on $w_1, ..., w_j$ and that linearly depends on themselves is not empty. Let $W = (w_1 \cup ... \cup w_j)$, $\bar{W} = X \backslash W$ the variables not in W ($\bar{W} \neq \emptyset$) and let $v \in \bar{W}$.

1. Assume now any variable $v \in \bar{W}$ polynomially depend on a variable of $\bar{W}$ or start a polynomial dependency chain in $\bar{W}$. Then by lemma 2 there would exist a cycle, which is absurd. Thus, there exists variables of $\bar{W}$ that that do not polynomially depend on variables of $\bar{W}$.
2. Let $P \subset \bar{W}$ the set of variables that do not polynomially depends on variables of $\bar{W}$. We have to show that there exists a non empty $P' \subseteq P$ such that for any $p \in P'$, p linearly depend only on variables of $P' \cup W$ and polynomially on variables of W. Assume there exists no cycles and that for any variable $p \in P$, p belongs to a linear dependency chain C ending with $q \notin P \cup W$. By construction of P, q polynomially depends on at least one variable of $\bar{W}$ (otherwise q would be in P). Any variable v of $\bar{W} \backslash P$ polynomially depends on variables of $\bar{W}$ by definition. Thus, for any variable of $\bar{W}$ there exists a polynomial dependency chain of variables, which by lemma 2 implies the existence of a polynomial dependency cycle. This is absurd, thus there exists at least one variable p that admit no dependency chain ending with a variable out of $P \cup W$. Let P' the set of such variables. Every variable v of this set polynomially depends on variables of W, and linearly depends on variables of P that admits no dependency chain out of $P \cup W$. In other words, v depends on variables of $P' \cup W$ only. We conclude this proof by setting $w_{j+1} = P'$.

As there exist only a finite number of variables, eventually we will have $\bar{W} = \emptyset$

□

Completeness theorem. We can now prove Theorem 1, stating that any non-solvable transformation cannot be linearized. Let g be a non solvable transformation. By Property 4, there exists then a polynomial dependency cycle $\mathcal{C} = (v_1, ..., v_n)$. Let $m = v_1 * v_2 * ... * v_n$ the monomial of all the variables of $\mathcal{C}$. With respect to g, the new value of $m = v_1 * v_2 * ... * v_n$ would then polynomially depend on $v_2 * ... * v_n * v_1 = m$. By Property 3, g is then not linearizable.

3.3 Algorithm

There are two aspects of the problem : detecting whether a transformation f is solvable or not and linearize f if possible.

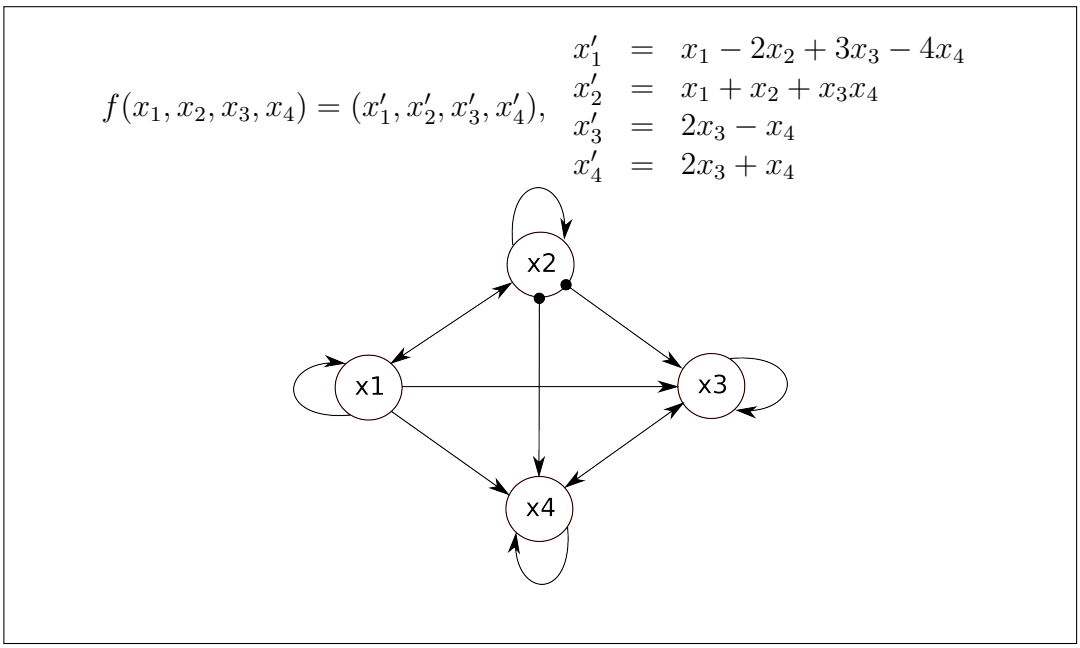


FIGURE 3.3: Dependency graph of the transformation

3.3.1 Solvability test

This first aspect can be reduced to finding cycles in a directed graph, with the slight variation of constraining the type of transition that must be taken along the way. This graph will contain a node for each variable of the transformation and transitions representing the dependency type between variables. There exist thus two types of transitions : linear dependency transitions and polynomial dependency transitions.

Definition 15 *Let g be a polynomial transformation. Its dependency graph $G = (V, E)$ is a directed graph defined as follows.*

- *V is the set of vertices labeled by each variable of g*
- *$E = E_l \cup E_p$ the set of edges such that*
  - $(x, y) \in E_l \Leftrightarrow x \lhd y$
  - $(x, y) \in E_p \Leftrightarrow x \lhd\!\!\lhd y$

Figure 6.1 illustrates such a graph for the transformation f. Transitions (x_2, x_3) and (x_2, x_4) represent polynomial dependency transitions. There is no cycle in this graph going through a polynomial dependency transition, thus this mapping is solvable.

Detecting such cycles can be checked by a linear time algorithm (cf. Figure 3.4) and in the same time generate the variable partitioning. The function *color_dfs* searches for non trivial cycles in the graph. If the cycle contains polynomial edges, it returns $\perp$. If there is no cycle, it changes the color of the visited nodes and returns $\emptyset$. Later, if dfs enters a colored node it can guess whether it is a cycle or an already visited branch, making each node visited only once. The function *merge* merges nodes of a linear dependency cycle in a

Require:g: a polynomial mapping
Ensure:a partitioning of variables if g is solvable or $\perp$ if it is not solvable

(V, E) = dep_graph(g)
for all $v \in V$ **do**
  $w =$ color_dfs($v, (V, E)$)
  if $w \neq \perp$ **then**
    merge(w)
  else
    return $\perp$
  end if
end for

FIGURE 3.4: Solvability checking and partitioning.

Require:v a node, (V, E) a graph, $path$ the current path
Ensure:a linear cycle of variables w or $\perp$ if there exists a polynomial cycle
$v' = v$
$path = \emptyset$
if v' is colored **then**
  $cycle$ = get_cycle($path$)
  if $path$ has a polynomial edge **then**
    return $\perp$
  else
    return $cycle$
  end if
else
  color(v')
  for all $n \in$ next(v') **do**
    $cycle =$ color_dfs($n, (V, E), path \cup \{v'\}$)
    if $cycle = \perp$ **then**
      return $\perp$
    else
      if $cycle \neq \emptyset$ **then**
        return $cycle$
      end if
    end if
  end for
  return $\emptyset$
end if

FIGURE 3.5: Recursive version of the color_dfs algorithm.

single node. Transitions going in and out of any of the node merged are given to the new node. Finally, it deletes nodes of the cycle. When the loop ends,

Require:w a node set
Ensure:a merged node n_w
n_w = new_node()
for all $v \in w$ **do**
  add_succs_and_preds(v, n_w)
  delete(v)
end for

FIGURE 3.6: Algorithm of *merge*.

there is no more cycle on the loop unless the mapping is not solvable. In the first case, variables of the leaves evolve linearly and only depend on themselves : they form the first partitioning set. Their fathers may polynomially depend on them, but linearly depend on themselves and on no other variables, they form the second partitioning set. The function *partitions* applies this principle by going through the tree and registering each node merged .

Example. On the previous example, applying the algorithm returns the graph in Figure 3.7. By starting on x_1, the dfs detects the linear cycle (x_1, x_2, x_1). It merges the two nodes and continues from the new node x_1, x_2. From there, it detects the other linear cycle (x_3, x_4, x_3), and merges the two nodes. Partitionning is now complete, and it is easy to check this new graph has no polynomial cycles.

Complexity. Each node and each transition of the dependency graph is reached at most once during the deep-first search if, when a *merge* is performed, the dfs starts from the merged node. The complexity of the solvability test is $O(n + |D|)$, where n is the number of variables and $|D|$ the number of dependencies of the tested transformation. At most, every variable is in relation with every other variable, thus $|D| = n^2$.

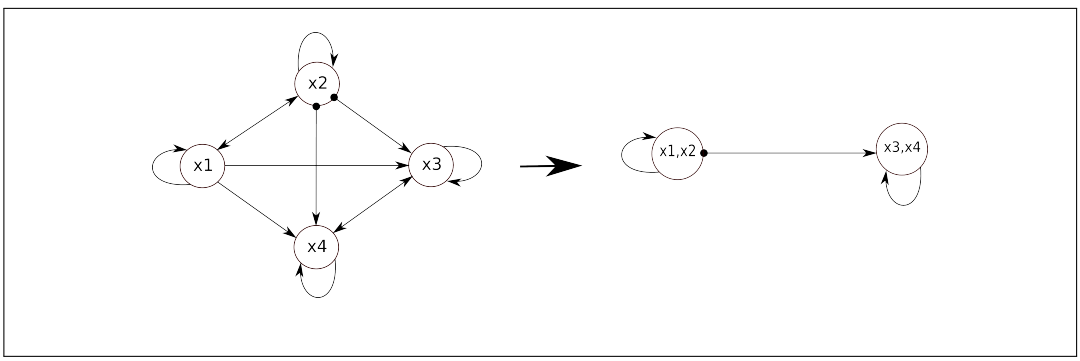


FIGURE 3.7: Result of the partitioning algorithm.

Property 5 *The solvability test has a complexity $O(n^2)$, where n is the number of variables.*

3.3.2 Linearization

For each monomial m in the transformation, compute the next value of m as a linear transformation of monomials M, which must be recursively linearized too. If the mapping is solvable, the set of variables composing monomials

Require: g a solvable mapping
Ensure: f the linearized mapping
$f = g$
for all monomial $m \in f$ **do**
  $m' = 1$
  for all monomial $v^d \in m$ **do**
    $m' = m' * f(v)^d$
  end for
  f.add($m \mapsto m'$)
end for
for all monomial $m \in f$ **do**
  $v =$ new_var(m)
  substitute(m,v)
end for

FIGURE 3.8: Linearization algorithm.

of M eventually decreases as polynomial expressions occurring in the transformation uses variables of a smaller set. Thus, linearization always ends. Once the first loop is over, it is necessary to replace monomials by the new variables representing monomials. This is the purpose of the second loop, creating fresh variables for each monomials and substituting them in f to their corresponding monomials.

Complexity. Only linearizing monomials appearing in the initial transformation is not sufficient. During the nested loop of the algorithm in Figure 3.8, new monomials can appear during the computation of m'. For example, when linearizing $f(x, y, z) = (x + y^3, y + z^3, z + 1)$, z^9 needs to be linearized though it does not appear in f. If we express all monomials of maximal degree d, we need $\binom{d+n}{n}$ new variables. For solvable mappings, this degree is in the worst case a $O(d')$ where d' is the product of all degrees. In other words, the worst case complexity of linearization is $O(\binom{d'+n}{n})$

Note that this is a strict over-approximation that is never reached in practice. For the previous example, not all monomials of degree 9 (or less) need to be linearized (for example, x^2 will not appear in the linearized transformation of f). In the worst case, all variables of w_i must be elevated up to d_i, where $d_i = \prod_{j=i}^{k-1} d_{w_j}$, and d_{w_j} the degree of $g(X)_{|_{w_j}}$.

Remark about elevation. Elevation is very similar to linearization. It starts with a linear transformation g and a degree d, then generates the linear transformation f transforming of all monomials of degree d. It is sufficient to express the monomials new value with respect to g as polynomial transformations (i.e. $P_d \circ g$), then replacing each monomial by a brand new variable. This elevated transformation f requires $\binom{n+d}{n}$ variables.

3.4 Properties of elevated matrices

After linearization, the matrix representation of a solvable transformation has specific properties.

3.4.1 Elevation matrix

Let us define the *elevation matrix*, i.e. the linear transformation expressing the new monomials values.

Definition 16 *Let X be a vector of variables. We denote $\Psi_d(X)$ the vector of monomials of variables of X of degree d or less. By extension, we define $\Psi_d(A)$ the elevated linear transformation of A such that $\Psi_d(A).\Psi_d(X) = \Psi_d(A.X)$*

For example, if we have $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ as a transformation for $X = (x, y)$, we have $\Psi_2(X) = (x^2, xy, y^2, x, y)$ and

$$\Psi_2(A) = \begin{pmatrix} a^2 & 2ab & b^2 & 0 & 0 \\ ac & ad+bc & bd & 0 & 0 \\ c^2 & 2cd & d^2 & 0 & 0 \\ 0 & 0 & 0 & a & b \\ 0 & 0 & 0 & c & d \end{pmatrix}$$

3.4.2 Eigenvector decomposition of $\Psi_d(A)$

Ψ_d conserves a lot of very interesting properties of matrices:

Lemma 3

1. $\Psi_k(A.B) = \Psi_k(A).\Psi_k(B)$
2. $\Psi_k(A^{-1}) = \Psi_k(A)^{-1}$

Proof.

1. $\Psi_d(A).\Psi_d(B)\Psi_d(X) = \Psi_d(A).\Psi_d(B.X) = \Psi_d(A.B.X) = \Psi_d(A.B)\Psi_d(X)$
2. $\Psi_d(A^{-1}).\Psi_d(A).\Psi_d(X) = \Psi_d(A.A^{-1}X) = \Psi_d(X)$ so $\Psi_d(A^{-1}).\Psi_d(A) = Id.$

□

The multiplication and inversion of the elevation matrix allows to add the following property:

Property 6 *Let A, B two similar matrices. For any d, $\Psi_d(A)$ and $\Psi_d(B)$ are similar.*

Proof. If A and B are similar, there exists a matrix P such that $A = P^{-1}BP$.

By Lemma 3, $\Psi_k(A) = \Psi_k(P)^{-1}.\Psi_k(B)\Psi_k(P)$, therefore $\Psi_k(A)$ is similar to $\Psi_k(B)$. □

Also, elevation preserves the triangularity of the matrix:

Property 7 *If J is upper (or lower) triangular, then $\Psi_d(J)$ is upper (or lower) triangular for any d.*

Proof. To understand the proof, let's see what happens for $d = 2$ and $n = 2$.

If λ and λ' are eigenvalues of A, then $J = \begin{pmatrix} \lambda & b \\ 0 & \lambda' \end{pmatrix}$ with $b = 0$ or 1 . then:

$$\Psi_2(J) = \begin{pmatrix} \lambda^2 & 2\lambda b & b^2 & 0 & 0 \\ 0 & \lambda\lambda' & b\lambda' & 0 & 0 \\ 0 & 0 & \lambda'^2 & 0 & 0 \\ 0 & 0 & 0 & \lambda & b \\ 0 & 0 & 0 & 0 & \lambda' \end{pmatrix}$$

As the second variable only depends on itself, its monomials also only depend on themselves. This is the key of the general proof.

Definition 17 *Let f a linear application. We define a* dependency order $\prec_f$ *on Var a total order such that for all $x \in Var$, $f(X)$ restricted to x depends only on a linear combination of variables V for which $\forall y \in V, y \preceq_f x$. We also say that f respects $\prec_f$.*

The idea behind this is that an upper-triangular matrix J induces such an order: the last element x_n only depend on himself, the previous element x_{n-1} depends on himself and x_n, so $x_n \prec_A x_{n-1}$, etc..

We define $\prec_J$ as an order that J respects. Let us show that $\Psi_k(J)$ is upper-triangular by choosing the lexicographic order $\prec_J^g$ with respect to $\prec_J$, defined as:

- If $x \prec_J y$ two variables, then $x \prec_J^g y$.
- If $m_1 \prec_J^g m_2$, then for any monomial m_3, $m_1.m_3 \prec_J^g m_2.m_3$.

If $x \prec_J^g y$ then by definition $x \prec_J y$. Moreover, $m_1 \prec_J^g m_2 \Rightarrow (m_1$ does not appear in the expression of $m_2)$, then let m_3 any monomial. As $m_1.m_3 \prec_J^g m_2.m_3$, we can clearly see that $m_1.m_3$ does not appear in the expression of $m_2.m_3$. Therefore :

Lemma 4 *Let J an upper triangular matrix, $\prec_J$ be a dependency order respected by J. Then for any $k \in \mathbb{N}$, $\prec_J^g$ can be a dependency order for $\Psi_k(J)$.*

By definition of the lexicographic order, we can state that for all $x, y, z \in Var$, if $x \prec_J y \preceq_J z$, then $y^i z^j \prec_J^g x^{i+j}$ is impossible for all i, j as x does not depend on y or z, but the contrary is false. Thus :

Lemma 5 *Let f a linear application, $\prec_f$ a dependency order on f, $x, y \in Var$. Let $Mon_d(x, y)$ the set of variables representing monomials of degree lower or equal to d depending on x and y. Then $z \prec_f x \prec_f y \Rightarrow \forall v \in Mon_d(x, y), z^d \preceq_f^g v$.*

In other words, in a triangular matrix M representing x, y, z and its monomials, if x and y are over z and M respects $\prec_M^g$, then $x^i.y^j$ will always be over z^k, for every i, j, k with $k \leqslant i + j$.

Let x, y two variables and φ_x, φ_y the vector of coefficients of J on the line of respectively x and y, except the coefficient of x and y. In other words, for X the vector of variables, we have: $\begin{array}{rcl} x' & = & \lambda_x.x + \varphi_x.X \\ y' & = & \lambda_y.y + \varphi_y.X \end{array}$

When one develop $x^i.y^j$, there is :

- 0 for *monomial variables* of strictly higher degree ;
- for any variable $t \prec x \prec y$, 0 for *monomial variables* containing t by lemma 5 ;
- 0 for monomials $x^{i'}.y^{j'}$ with $i' + j' = d$ and $i' > i$;
- a coefficient $\lambda_x^i.\lambda_y^j$ for the variable $x^i.y^j$, which will be on the diagonal of $\Psi_k(J)$.

The third point is true because if $x \prec_J y$, then $x^i y^j \prec_J^g x^{i-1}y^{j+1}$ and $\prec_J^g$ is a dependency order for $\Psi_k(J)$ by Lemma 4.

$\Psi_k(J)$ will be upper-triangular itself by respecting $\prec_J^g$.

□

Eigenvectors behave simply by the transformation Ψ_d:

Property 8 *Let $A \in \mathcal{M}_d(\mathbb{Q}), \Lambda(M)$ the eigenvalue set of a matrix M and d an integer. Then for any product p of d or less elements of $\Lambda(A), p \in \Lambda(\Psi_d(A))$.*

Proof. Let us consider J the Jordan normal form of A. As we are working with $\mathbb{C}$, which is an algebraically closed field, A is similar to J (ie. $\exists P.A = P^{-1}JP$), with

$$J = \begin{pmatrix} J_1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & J_k \end{pmatrix}, \text{ and } J_i = \begin{pmatrix} \lambda_i & 1 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & \dots & 0 & \lambda_i \end{pmatrix} \text{ for } 1 \leqslant i \leqslant k.$$

By Property 6, $\Psi_d(A)$ and $\Psi_d(J)$ are similar, hence have the same set of eigenvalues. The transformation of a variable x by J can be either of the form $\lambda.x$, or $\lambda.x + y$, with λ an eigenvalue of J and y another variable.

Let x_1 and x_2 two variables in the base of J. As $x_1' = \lambda_1 x_1 + y_1$ and $x_2' = \lambda_2 x_2 + y_2$, then $(x_1 x_2)' = \lambda_1\lambda_2 x_1 x_2 + \dots$ As $\Psi_d(J)$ is upper triangular by Property 7, then the coefficient of x_1x_2 in the expression of $(x_1x_2)'$ is an eigenvalue. The generalization for more than 2 variables is straightforward by induction.

□

There is also an interesting property about eigenspaces of $\Psi_d(A)$:

Property 9 *If A admits a generalized eigenvector φ associated to λ of order n (i.e. $(A - \lambda.Id)^{n-1}.\varphi \neq 0$ and $(A - \lambda.Id)^n.\varphi = 0$) and an eigenvector associated to λ', then $\Psi_d(A)$ admits a generalized eigenvector associated to $\lambda.\lambda'$ of order n.*

Proof. Let us first prove this theorem for $d = 2$. Let λ, λ' two eigenvalues. If A admit a generalized eigenvector v_λ of order n for the eigenvalue λ, then there exist $v_1, v_2, ..., v_n \in \mathbb{Q}^n$ such that

$$\begin{aligned} v'_1 &= \lambda.v_1 \\ v'_2 &= \lambda.v_2 + v_1 \\ &... \end{aligned}$$

Also we have that $v_{\lambda'} = \lambda'.v_{\lambda'}$

For all i, $v_i(X).v_{\lambda'}(X)$ is a polynomial of degree 2, or in other words a linear combination of monomials of degree 2. Let us prove that w_i the vector such that $w_i(\Psi_2(X)) = v_i(X).v_{\lambda'}(X)$ is a generalized eigenvector associated to $\lambda.\lambda'$ of order n by recurrence. If $n = 1$, then it is clear that $(\Psi_2(A).w_1)(\Psi_2(X)) = \lambda\lambda'.v_1(X).v_{\lambda'}(X) = \lambda\lambda'.w_1(\Psi_2(X))$ for all X. Therefore, w_1 is an eigenvector of $\Psi_2(A)$ associated to $\lambda.\lambda'$.

Assume now w_{i-1} for $i < n$ is a generalized eigenvector of order i associated to $\lambda.\lambda'$. Therefore, we have $(\Psi_2(A) - \lambda.\lambda'.Id)^{i-1}.w_{i-1} = 0$ For all X, the following equalities hold:

$$\begin{aligned} (\Psi_2(A).w_i)(\Psi_2(X)) &= \lambda'.v_{\lambda'}(X).(\lambda.v_i(X) + v_{i-1}(X)) \\ (\Psi_2(A).w_i)(\Psi_2(X)) &= \lambda\lambda'.v_{\lambda'}(X).v_i(X) \\ &\quad +\lambda'.v_{\lambda'}(X).v_{i-1}(X)) \\ (\Psi_2(A) - \lambda\lambda'.Id).w_i &= \lambda'.v_{\lambda'}(X).v_{i-1}(X)) \\ (\Psi_2(A) - \lambda\lambda'.Id).w_i &= \lambda'.w_{i-1} \\ (\Psi_2(A) - \lambda\lambda'.Id)^i w_i &= (\Psi_2(A) - \lambda\lambda'.Id)^{i-1}.\lambda'.w_{i-1} \\ (\Psi_2(A) - \lambda\lambda'.Id)^i w_i &= 0 \end{aligned}$$

Thus, w_i is an eigenvector of order i. This proof is also valid for $d > 2$ as Ψ_d represents monomials of degree d *and lower*. □.

3.5 Application to formal verification

When a polynomial expression occurs in an arithmetic expression, static analyzers have multiple ways to deal with it:

1. consider it belongs to an undecidable class of program and fail, as it is out of the decidable Presburger arithmetic (for example, acceleration techniques are restricted to linear transformations [GS14; Bar+05]);

2. precisely analyse it, with no guarantee to eventually end (for example, SMT-solvers [MB08; Bar+11] cannot fully handle polynomial expressions);

3. approximate it and try to maintain the smallest gap possible between reality and abstraction (for example, in abstract interpretation, affine arithmetic abstract domains such as the zonotope domain [GGP09] and the octagon domain [Min06] perform approximations when treating polynomial expressions).

In the two first cases, linearization seems to be a straightforward manner to enhance their set of applications (though the loop initialization still require

polynomial expressions, which may limit the enhancement of linearization for SMT solvers).

Abstract interpretation [CC77] aims at inferring invariant properties on different program point by propagating an *abstract value* representing an abstraction of the set of possible states through each instruction of a given program. Each possible expression is endowed with transformation rules that are used to transform the abstract value along the analysis. Usually when a loop is encountered, the abstract interpreter performs approximations that guarantees to converge to an abstract value in a finite number of steps: this is called *widening*. For example, the octagon abstract domain [Min06] uses linear inequalities as abstract values. Starting from an initial state $x \leqslant a$, the abstract execution of $x = x + 1$ returns $x \leqslant a + 1$. When a polynomial expression is encountered, the problem becomes harder. If $x \leqslant 5$ and $y \geqslant -3$, we cannot conclude anything about $x * y$ because this expression can be arbitrarily large. Intuitively, polynomial expressions will have a very negative impact on the the precision of the computed abstract values. This is particularly true when a polynomial expression occurs in a loop. If the loop can be linearized, there will still be some polynomial expressions before the loop, to compute the initial state of the additional variables, but they will be computed only once and will not be subject to widening.

Experimentations. Consider the example of Figure 3.1 with the initial state $x \in [-5, 5]$ and $y \in [-5, 5]$. This loop is supposed to have very few iterations as x increases quadratically and y affinely. Using Frama-C's abstract interpreter EVA [Kir+15; BBY17] (Phosphorus version) that propagates different domains simultaneously, we compared the use of linearization by analyzing both loops with the interval domain and the octagon domain. By setting EVA to unroll 11 times each loop, it concludes in the polynomial example to the state $x \in [-5, 239] \wedge y \in [-5, 16]$, while in the linearized example we get $x \in [-5, 146] \wedge y \in [-5, 18]$. Though the second example admits a small loss of precision over y [1], we gain a lot of precision over x.

[1] The loss of precision is suspected to come from a precision error from the polynomial expression that is somehow propagated to y and to the halting condition of the loop.

Chapter 4

A widening operator for the zonotope abstract domain

Contents

Requirements: Linear algebra (Section 2.1), Programming model (Section 2.2), Abstract interpretation (Section 2.5)

The analysis of loop invariants is usually performed through a propagation analysis (the analysis of the behavior of the analyzed code, performed by Abstract Interpretation [CC77]. This framework has the advantage to be customizable enough to adapt its analysis to the analyzed problem by the free choice of abstract domains. Thanks to the previous chapter, we also know that polynomials and linear loops are strongly linked. From this, we will be interested in generating linear and polynomial relations on variables of linear loops.

This Chapter focuses convergent linear filters, that are linear loops containing non-determinism. It presents an abstract interpretation widening operator for the zonotope abstract domain [GPV12] that is based on the parametrization of the abstract values.

4.1 Approximation of convergent linear filters

In a abstract interpreter, the widening operator plays a crucial role in the precision of the analysis. As defined in Section 2.5, a widening operator generalizes the abstract memory state by extrapolating its behavior through one or multiple iterations. In practice, it chooses one of the highest common values inductively preserved by an iteration. In simple domains like intervals, a common value between two iterations can easily be found, but finding an inductive value for a whole loop require the use of a widening operator.

In some cases the abstract interpreter may rely on loop acceleration (cf Section 2.4) returning exactly its set of reachable states. Though it then returns a more precise representation of the loop, it is only applicable under strong hypotheses like finite monoid [GS14]. To illustrate such an approxi-

LISTING 4.1: geometric series

```
x = 0;
while(*)
  x = 0.8*x + [-1,1];
```

LISTING 4.2: simple linear filter

```
S = S0 = S1 = 0.0;
while(*)
  S1 = S0;
  S0 = S;
  S = [-1,1] + 1.4*S0 - 0.7*S1;
```

FIGURE 4.1: Simple non deterministic linear loops

mation, we perform abstract iterations with the interval domain on the geometric series of Figure 4.1. The analysis starts with $x \in [-1, 1]$, then finds $x \in [-1.8, 1.8]$, and $x \in [-2.44, 2.44]$. This process could loop forever if we didn't use a widening operator. One of the simplest widening operators (but one of the most efficient in term of computation time and used in abstract interpetors) consists in generalizing an upper bound increase by $+\infty$ and a lower bound decrease one by $-\infty$. In other words, it return in this example $[-\infty, +\infty]$ as both bounds increase and decrease. Any widening operator finds stable iterations with bounds greater or equal than the minimal bound, which is 5 for this simple example. But how to find a solution near the minimal bound within a limited number of iterations? Moreover, on the second example, bounds for each variable independently grow (independently because intervals do not catch relations between variables), and the simple widening operator for intervals would return $S \in [-\infty, +\infty]$, which is clearly not precise enough considering that an invariant of this loop is $S \in [-7.589424, +7.589424]$. In general, inductive invariants of such numerical program are difficult to approximate efficiently. Synthesis of strong inductive invariants is however critical in order to perform a successful verification.

Convergent linear filters. Convergent linear filters are linear transformations receiving multiple inputs over time. From a computer system point of

view, those inputs are generally randomly picked in a known interval. The purpose of such filters is to keep track of all the information received through its execution while reducing the impact of the past inputs. The examples in Figure 4.1 are convergent linear filters. At first, on the Listing 4.1 filter, the value of x belongs to $[-1, 1]$. It can be written as $\exists \varepsilon_0 \in [-1, 1]$ such that $x = \varepsilon_0$. Then, x admits the following expressions:

$$\begin{array}{llll} 2^{\text{nd}}\text{iteration} & x & = & \exists \varepsilon_0, \varepsilon_1 \in [-1,1] : \quad \varepsilon_1 + 0.8 * \varepsilon_0 \\ 3^{\text{rd}}\text{iteration} & x & = & \exists \varepsilon_0, \varepsilon_1, \varepsilon_2 \in [-1,1] : \quad \varepsilon_2 + 0.8 * \varepsilon_1 + 0.64 * \varepsilon_2 \\ 4^{\text{th}}\text{iteration} & x & = & \exists \varepsilon_0, \varepsilon_1, \varepsilon_2, \varepsilon_3 \in [-1,1] : \quad \varepsilon_3 + 0.8 * \varepsilon_2 + 0.64 * \varepsilon_1 \\ & & & \qquad\qquad\qquad\qquad\qquad\qquad +0.512\varepsilon_0 \end{array}$$

...

with $\varepsilon_i \in [-1, 1]$. As we can see, the coefficient associated to the most recent ε_i is higher, hence it has more impact on the loop state than the older ones. On the previous examples, while the bounds were growing, the difference between two successive abstract values gets smaller after every iteration because of the same reason.

Linear filters also have the advantage to reduce occasional precision errors over time, as they would slowly be erased as time goes by.

Zonotopes and linear filters. In [Rou+12], authors study numerical filters in an abstract domain expressing polynomial inequalities on its variables. Ellipsoids have shown to be very efficient to approximate invariants of linear filters, but they are often not inductive and therefore, hard to generate. This issue can be avoided by approximating ellipsoids. For example, an approach like [MBR16] is able to prove invariants by overapproximating it by constructing an invariant by chunks. Another possibility, which will be studied in this Chapter, is the use of *zonotopes* [GGP09; GPV12; GP15].

Zonotopes are convex symmetric polyedra where every face has a symmetric point. They can be described in two equivalent manners:

- by a collection of edges and nodes, as a geometric figure;
- by a vector of sums of the form $\sum_{i=0}^{n} \alpha_i \varepsilon_i$ where $\alpha_i \in \mathbb{R}$ define the zonotope and $\varepsilon_i \in [-1, 1]$ are parameters.

Figure 4.2 presents the link between the two representations. The property "(x, y) is inside the parallelogram $ABCD$" is equivalent to $\exists \varepsilon_1, \varepsilon_2 \in [-1, 1].(x, y) = (\varepsilon_1, \varepsilon_1 + \varepsilon_2)$

The zonotope abstract domain [GGP09; GP15] is based on this second representation and is particularily well suited for the analysis of linear filters given the similarity between the zonotope expression and the variables reachable state.

This Chapter adresses the problem of generating invariants for convergent linear filters in the context of abstract interpretation on the zonotope abstract domain. Instead of applying the usual widening methodology (i.e. finding a common higher value on a different lattice, cf Section 2.5), this chapter exploits parametrization techniques to find a higher common value.

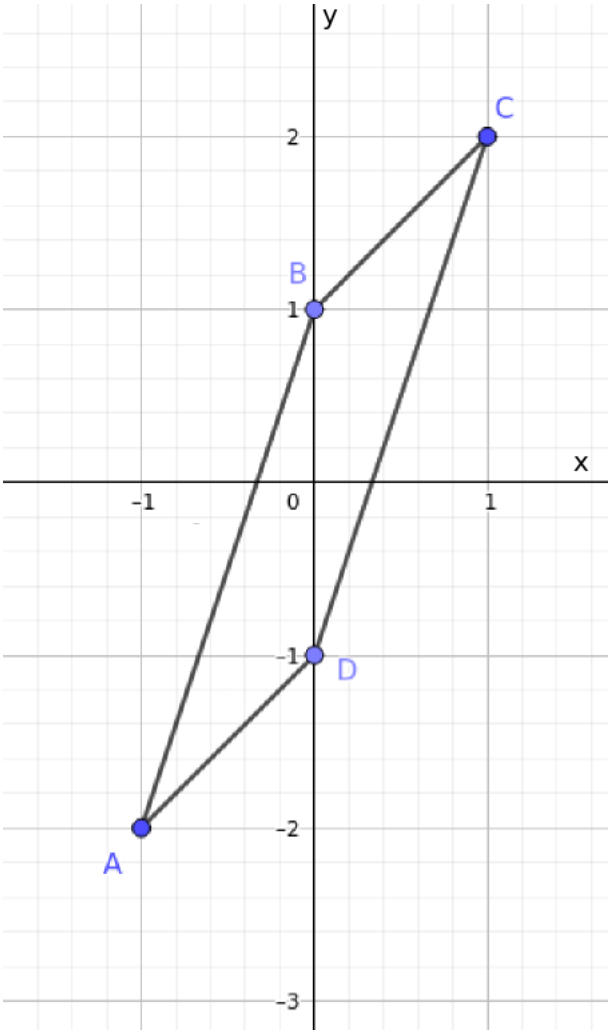


FIGURE 4.2: A zonotope can be defined as a geometric figure with different nodes and edges, or as a vector of Minkowski sums.

Though this has the drawback of not converging in the general case, we will prove that it always converges when applied on convergent linear filters. This heuristic allows a greater precision than abstract interpretation at a cost of genericity, and is comparable to acceleration techniques (cf Table 4.1).

	Widening	Acceleration	Testing	This technique
Termination	✓	✓	×	✓
Genericity	✓	×	✓	×
Precision	×	✓	✓	✓
Soundness	✓	✓	×	✓

TABLE 4.1: Comparison of this technique with other state of the art methods

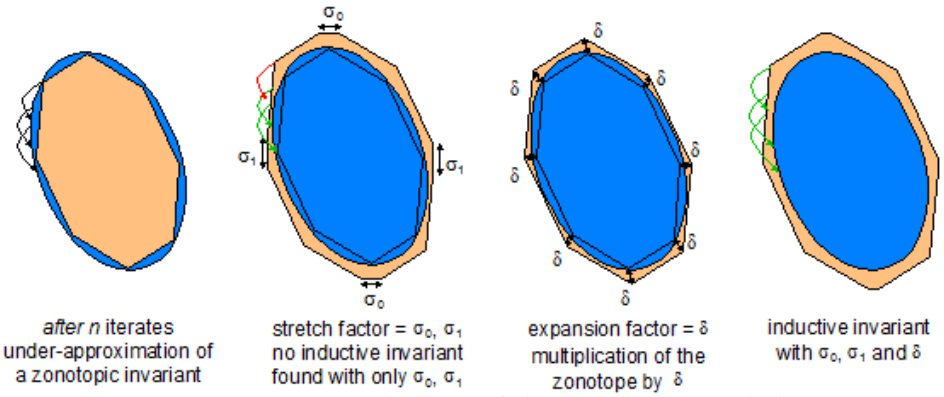


FIGURE 4.3: Perturbation of the zonotope candidate invariant (in beige) in comparison to the actual ellipsoid invariant (in blue).

Outline of the algorithm. The heuristic proceed in two phases. It starts after an abstract iteration over the zonotope abstract domain. As we saw, zonotopes can be defined as vectors of sums parametrized by *noise symbols* ε that are free to evolve in $[-1, +1]$. The value of every program variable v is expressed as a sequence $v_k = \alpha_0 + \sum_{i=1}^{n} \alpha_i * \varepsilon_i$ where v_k is the value of v at the k^{th} iteration, $(\alpha_i)_{1 \leq i \leq n}$ are real constants and $(\varepsilon_i)_{1 \leq i \leq n}$ are *noises*, i.e. non deterministic values.

Let X be the variables of the loop and $\varepsilon \in [-1, +1]$ a non-deterministic value. Let's denote $tf(X, \varepsilon)$ the forward transfer function of the loop body. The abstract interpreter iterates the loop and infers a zonotope for each variable. In parallel of this iteration, the method will try to guess a good candidate invariant by *confronting* two similar successive iterations. When two zonotopes z_1 and z_2 are close enough, z_1 will serve as a base for finding an inductive invariant. Our key heuristic consists in *stretching* and *expanding* z_1.

This process is described in Figure 4.3. The zonotope on the first figure represents an under approximation of the actual ellipsoid invariant of a loop manipulating two variables x and y. Hence, it is in two dimensions: horizontal for x and vertical for y. Intuitively, a very small weakening of the zonotope would be a good candidate invariant as it is already close to the actual invariant. A first method to weaken it is to add to each variable a small value σ so that it encompasses the invariant. Geometrically speaking, with two variables, it would *stretch* the zonotope vertically and horizontally as in the second part of Figure 4.3. By choosing large enough parameters, the resulting candidate is weaker than the searched invariant but it might still not be inductive. A second way to weaken it is to multiply the value of x and y by $(1 + \delta)$ where $\delta > 0$ is a small real [1]. The zonotope would then be *expanded* proportionally in every direction as presented in the third picture. Finally, the candidate invariant in the last picture verify $(1 + \delta)z + \sigma$, where z was the initial zonotope of the first picture. With those two parameters, we can find an inductive zonotope approximating the aimed ellipsoid.

[1] In practice, we will see that a different δ can be choosen for each variable.

The inductiveness of this parameterized candidate is expressed as constraints, then sent to a solver which will search for a valuation of the parameters. If such a valuation is found, the inductiveness of the relation is proven and the method outputs a correct inductive invariant. Otherwise, the abstract interpreter iterates once more and tries again to find an invariant by the same technique over different abstract values.

Let us apply the previous method for the geometric sequence of Figure 4.1, the transfer function is $tf(x, \varepsilon) = 0.8 * x + \varepsilon$ and x_0 starts at 0. First, by applying abstract iterations on the zonotope domain, we find:

$$\begin{array}{lllllll} 1^{\text{st}}\ \textit{iteration:} & \exists \varepsilon_1 \in [-1,+1]. & x_1 & = & \varepsilon_1 & & \\ 2^{\text{nd}}\ \textit{iteration:} & \exists \varepsilon_1, \varepsilon_2 \in [-1,+1]^2. & x_2 & = & \varepsilon_2 & & +0.8\varepsilon_1 \\ & & & & \text{common} & & \text{residue} \end{array}$$

The zonotope of x_2 can be decomposed into two parts: the common part in both equations (ε_1 in x_1 and ε_2 in x_2) that we denote ε_δ and the difference ($0.8\varepsilon_1$), which we will call the residue and denote $0.8\varepsilon_\sigma$. The common part has, by definition, a good chance to appear in the real inductive invariant. The technique first *expands* the common part ε_δ by multiplying it by a parameter $(1 + \delta)$ Then, it *stretches* the zonotope, i.e. it adds to the expanded component a parametrization of the residue for each variable (which widens the zonotope horizontally and vertically). We end up with the following candidate zonotope invariant $z_{\delta,\sigma}(\varepsilon_\delta, \varepsilon_\sigma) = (1 + \delta)\varepsilon_\delta + 0.8\sigma\varepsilon_\sigma$.

We need to find δ, σ such that $\exists \varepsilon_\delta, \varepsilon_\sigma \in [-1, 1]^2. x = z_{\delta,\sigma}(\varepsilon_\delta, \varepsilon_\sigma)$ is inductive, i.e. that this relation is preserved by a loop iteration. If it is inductive and as it is true at the first iteration, then it is an invariant. By performing one iteration over $z_{\delta,\sigma}$, we end up with $z'_{\delta,\sigma}(\varepsilon, \varepsilon_\sigma, \varepsilon_\delta) = tf(z_{\delta,\sigma}(\varepsilon_\delta, \varepsilon_\sigma)) = \varepsilon + 0.8(1 + \delta)\varepsilon_\delta + 0.64\sigma\varepsilon_\sigma$. Proving the inductiveness of the relation is equivalent to proving the inclusion of the two zonotopes $z'_{\delta,\sigma}(\varepsilon, \varepsilon_\sigma, \varepsilon_\delta)$ and $z_{\delta,\sigma}(\varepsilon'_\delta, \varepsilon'_\sigma)$, as it would mean that after one step the program state still belong to the initial zonotope (which is the definition of inductivity). This chapter proposes a simplified version of the inclusion of two zonotopes based on the following mapping between the ε symbols: $\varepsilon'_\delta \overset{\text{def}}{=} \frac{1}{1+\delta}\varepsilon$ and $\varepsilon'_\sigma \overset{\text{def}}{=} \frac{0.8(1+\delta)}{0.8\sigma}\varepsilon + \frac{0.64\sigma}{0.8\sigma}\varepsilon_\sigma$. This inclusion is valid with $\delta \geq 0$ and $0.8\sigma \geq 0.8(1 + \delta) + 0.64\sigma$, satisfied with $\delta = 0$ and $\sigma = 5$. Hence,

$$(1 + 0)\varepsilon + (0.8 * 5)\varepsilon_\sigma = \varepsilon + 4\varepsilon_\sigma$$

is a real inductive invariant for $\delta = 0$ and $\sigma = 5$. Its projection on the interval domain is $[-1, 1] + 4 * [-1, 1] = [-5, 5]$ which is the minimal invariant of this loop on the intervals.

4.2 Context

4.2.1 The family of the numerical linear filters

In this programming model, we consider the family of the numerical linear filters.

Definition 18 *A linear filter is a sequence* $(S_n)_{n\in\mathbb{N}}$ *such that :*

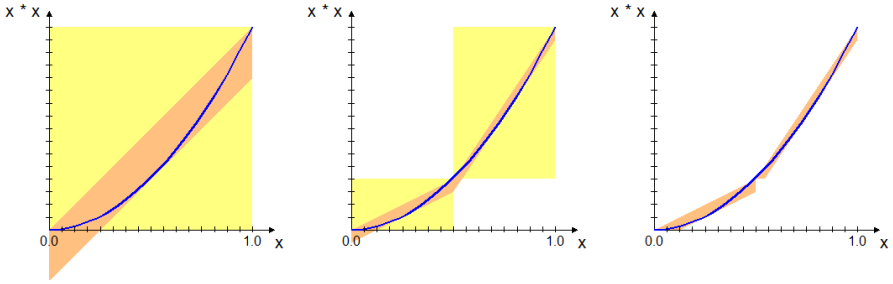
- $(S_0) = \begin{pmatrix} [a_0, b_0] \\ \dots \\ [a_n, b_n] \end{pmatrix}$, $(V_n) = \begin{pmatrix} [u_0, v_0] \\ \dots \\ [u_n, v_n] \end{pmatrix}$ *are non deterministic vectors with constant bounds*
- $(M) = \begin{pmatrix} m_{1,1} & \dots & m_{1,k} \\ \dots & \dots & \dots \\ m_{k,1} & \dots & m_{k,k} \end{pmatrix}$ *is the transformation matrix*
- $(S_{n+1}) = (M).(S_n) + (V_n)$

A linear filter is *convergent* if and only if all the eigenvalues (the real and the complex ones) of the matrix (M) have a norm strictly lower than 1. In this case, all the values of (S_n) remain bounded, i.e. there exists a closed interval I independent of n such that for any vector (V_n) and $\forall n, S_n \in I$. The program syntax defined in Figure 2.2 includes the necessary instruction for describing convergent linear filters, as linear filters don't use conditions nor nested loops.

4.2.2 The zonotope abstract domain

Every variable vector $X = (x_1, ..., x_n)$ is abstracted by a zonotope z^ε of dimension n, i.e. a vector of *Minkowski sums* [Min10] $x_j^\sharp = \alpha_0^j + \sum_{i=1}^{n'} \alpha_i^j * \varepsilon_i$ where the α_i^j are real constants and $\varepsilon = (\varepsilon_1, ...\varepsilon_{n'})$ a vector of non deterministic expressions taking value in $[-1, 1]$. Every coefficient of the vector representing the zonotope share the same ε_i. When ε is clear in the context, we simply refer to z. Let us define *Aff* as the set of such affine forms or linear equations, and their norm is defined as $\|x^\sharp\| = |\alpha_0| + \sum_{i=1}^{n} |\alpha_i|$ the maximal valuation of $|x^\sharp|$. An affine form $x_1^\sharp = \alpha_0 + \sum_{i=1}^{n} \alpha_i \times \varepsilon_i$ is included in $x_2^\sharp = \beta_0 + \sum_{i=1}^{m} \beta_i \times \varepsilon_i'$ when $\forall \varepsilon_1, ..., \varepsilon_n, \exists \varepsilon_1', ..., \varepsilon_m'$ such that $\alpha_0 + \sum_{i=1}^{n} \alpha_i \times \varepsilon_i = \beta_0 + \sum_{i=1}^{m} \beta_i \times \varepsilon_i'$. The inclusion between two zonotopes z_1^ε and $z_2^{\varepsilon'}$ is defined equivalently, i.e. $\forall \varepsilon, \exists \varepsilon'. z_1^\varepsilon = z_2^{\varepsilon'}$. It is stronger than the inclusion of each individual component as ε_i are shared between each component. For example, let $z_1 = (\varepsilon_1, \varepsilon_2)$ and $z_2 = (\varepsilon_1', \frac{1}{2}\varepsilon_1' + \frac{1}{2}\varepsilon_2')$ While each component of z_1 is included in z_2, the point $(1, -1)$ belong to z_1 while it doesn't belong to z_2.

We denote *Meta-Aff* the set of meta-affine equations, i.e. parametrized affine equations : $\mathbb{R}^k \to$ *Aff*. Meta-zonotopes are defined as vectors of meta-affine equations. Zonotopes and meta-zonotopes can also be denoted z^ε, where ε represent the non deterministic elements of the equation. Zonotopes and intervals catch different properties. The abstract interpreter *Fluctuat* [Gou13] infers the intersection of intervals and zonotopes as depicted on Figure 4.4. Zonotopes can also express floating point approximations by assimilating them to new variables ε. As an example, a variable x in the interval $[0, \lambda]$ is abstracted by the meta-equation $x^\sharp = \lambda(0.5 + 0.5\varepsilon_0)$. For $\lambda = 1$, $x^\sharp = 0.5 + 0.5\varepsilon_0$ defines an affine equation. A non-linear computation introducing $c \times \varepsilon_i \times \varepsilon_i$ is linearized by $\frac{c}{2} + \frac{c}{2}\varepsilon_k$, where ε_k is a fresh non deterministic variable. For example, $x^\sharp - x^{\sharp 2}$ is treated as follows:



interval & zonotope | partition in two parts | reduced product & partition

FIGURE 4.4: Abstraction of $f(x) = x^2$ with intervals, zonotopes, and their intersection. The first figure depicts the independent analysis of f on $I = [0, 1]$ with intervals (in yellow) and zonotopes (in orange). Partitioning I into two intervals on the second figure makes the analysis more precise, and intersecting both abstract values in the last figure returs an even more precise approximation.

$$\begin{aligned} x^{\sharp^2} &= (0.5 + 0.5\varepsilon_0) \times (0.5 + 0.5\varepsilon_0) = 0.25 + 0.5\varepsilon_0 + 0.125 + 0.125\varepsilon_1 \\ &= 0.375 + 0.5\varepsilon_0 + 0.125\varepsilon_1 \\ x^\sharp - x^{\sharp^2} &= (0.5 + 0.5\varepsilon_0) - (0.375 + 0.5\varepsilon_0 + 0.125\varepsilon_1) = 0.125 - 0.125\varepsilon_1 \end{aligned}$$

Hence zonotopes prove that $x - x^2 \in [0, 0.25]$, while intervals are only able to prove that $x - x^2 \in [-1, +1]$, or in $[0, +1]$ if $x - x^2$ is rewritten into $x(1 - x)$. Nevertheless, the linearization step (Taylor approximation around the center of the zonotope [GGP09]) might miss some important information: here zonotopes guarantee that $x^2 \in [-0.25, +1]$ whereas intervals guarantee a better interval result: $x^2 \in [0, +1]$ (see Figure 4.4).

Applying a transformation to a zonotope correspond to the application of the given transformation on the Minkowski sums vector representing the zonotope. In other words, a linear application φ mapping a vector to a scalar also maps a zonotope to a Minkowski sum. Hence, the matrix-vector multiplication is also defined for matrices and zonotopes.

Geometric representation of zonotopes. Zonotopes also admit a geometric representation defined by a set of vertices V and sides $S = V \times V$. For a zonotope z^ε, $v \in V$ iff for an extreme valuation of ε (i.e. ε_i are only 1 and -1), $z^\varepsilon = v$. For two valuations ε and ε', there exists a side between $v_1 = z^\varepsilon$ and $v_2 = z^{\varepsilon'}$ for the zonotope z if and only if there is only one coefficient of difference between ε and ε'.

The geometric representation of a zonotope with n noise symbols is defined with 2^n different vertices, which makes this representation hardly usable in practice. That is why we will only focus on Minkowski sums when performing calculations.

4.3 Synthesis by parametrized variation

4.3.1 Description of the method

The algorithm of Figure 4.5 presents the different steps to generate the invariant candidate. When the interpreter encounters a loop, the algorithm starts.

Data: tf : stat; z : zonotope; τ : float;
Result: A zonotope z_{inv} invariant of tf;

$z_{inv} = \bot$;
while $z_{inv} = \bot$ **do**
  $z' := tf(z)$;
  if $z \bowtie^{\tau} z'$ **then**
    z_{cand} := generalize(z,z');
    $z_{inv} := tf(z_{cand}) \bowtie z_{cand}$;
    $z := z'$;
  else
    $z := z'$;
  end
end

FIGURE 4.5: Abstract iterator shortcut algorithm. The *generalize* function represents the parametrization of the zonotope z by the stretch factor and the expansion factor.

Its role is simply to keep iterating the loop to keep track of two successive iterations z and z' and confront them ($z \bowtie^{\tau} z'$), i.e. checks if the two iterations are relatively similar. If the confrontation fails, the algorithm starts on the next abstract iteration of the loop. Otherwise, it builds a candidate invariant z_c by generalizing the shape of z, i.e. adding parameters to its expression. If there exists a valuation of those parameters such that z_c is inductive, then an invariant has been found. Each abstract interpretation step is illustrated by the simple-filter of Figure 4.2 whose loop body is:

```
S1 ← S0; S0 ← S; S ← [-1,1] + 1.4*S0 - 0.7*S1;
```

First step: loop iteration. As long as a candidate is not generated, the algorithm iterates on the loop. This is the standard abstract iteration on zonotopes, computing the abstract value of $S1, S0$ and S after each step. For example, here are the equations stored for each variable after 1, 2 and 3 steps:

$\mathtt{S1} \mapsto 0$	$\mathtt{S0} \mapsto 0$	$\mathtt{S} \mapsto \varepsilon_0$	after 1 iter.
$\mathtt{S1} \mapsto 0$	$\mathtt{S0} \mapsto \varepsilon_0$	$\mathtt{S} \mapsto \varepsilon_1 + 1.4\varepsilon_0$	after 2 iter.
$\mathtt{S1} \mapsto \varepsilon_0$	$\mathtt{S0} \mapsto \varepsilon_1 + 1.4\varepsilon_0$	$\mathtt{S} \mapsto \varepsilon_2 + 1.4\varepsilon_1 + 1.26\varepsilon_0 + 2^{-122}\mu_0$	after 3 iter.

During the abstract iteration, we will take into account rounding errors performed by our internal floating point calculations. The last iteration shows

that even if the analysis follows real semantics, the abstract interpreter cannot exactly compute `1.4 * 1.4 - 0.7 = 1.26`. It can only conclude that this value is in the interval $[\texttt{1.26} - 2 * 2^{-123}, \texttt{1.26} + 2 * 2^{-123}]$ which is represented in the zonotopic domain by $1.26 + 2^{-122}\mu'_0$. [2] Then μ_0 is an abstraction with a new fresh variable for $\varepsilon_0 \times \mu'_0$ that lays in the interval $[-1, 1]$. As it is an internal rounding error, it is assigned to a different type of fresh variable with different properties to not interfere with the ε variables. After 4 iterations, the memory abstraction would have the following content:

$$\texttt{S1} \mapsto \varepsilon_1 + 1.4\varepsilon_0 \quad \texttt{S0} \mapsto \varepsilon_2 + 1.4\varepsilon_1 + 1.26\varepsilon_0 + 2^{-122}\mu_0$$
$$\texttt{S} \mapsto \varepsilon_3 + 1.4\varepsilon_2 + 1.26\varepsilon_1 + 0.784\varepsilon_0 + 3.9 \times 2^{-122}\mu_1$$

In order to simplify the running example in this Chapter, we will stop considering internal accuracy errors.

Second step: confronting iterations and building a candidate. When linearly transforming a zonotope, many similitudes are observable in their expression. The 3rd and the 4th iteration of the example are very similar. Let us compare the value of S at the third and fourth iteration:

$$\begin{array}{lcll} \text{Third iteration} & \texttt{S} & \mapsto & \varepsilon_2 + 1.4\varepsilon_1 + 1.26\varepsilon_0 \qquad +2^{-122}\mu_0 \\ \text{Fourth iteration} & \texttt{S} & \mapsto & \varepsilon_3 + 1.4\varepsilon_2 + 1.26\varepsilon_1 \quad +0.784\varepsilon_0 + 3.9 \times 2^{-122}\mu_1 \end{array}$$

By changing ε_i in the 3rd by ε_{i+1}, the only difference is $0.784\varepsilon_0$ plus the precision error. The objective of the *confrontation* is to capture this difference to find a good candidate invariant. During this step, the abstract interpreter extends the affine forms $equ \in Aff$ with meta affine forms $m\text{-}equ \in Meta\text{-}Aff$. Let $equ_{pre} = \sum_{i=1}^{n} \alpha_i\varepsilon_i$ and $equ_{curr} = \sum_{i=1}^{m} \beta_i\varepsilon'_i$ the equation respectively associated to a given variable at the previous iteration and at the current iteration. The confrontation operator $\bowtie^{\tau}$ will first apply a *global loop renaming* `ren` on equ_{pre} so that the ε introduced in equ_{curr} matches the one in equ_{pre}. In practice, it performs a shift of s on the indexes of equ_{pre} where s is the number of new noise symbols added by the current step of the loop. After the renaming, the equation's projection on intervals remains the same.

In order to check if two zonotopes are *close*, we need to define a metric on zonotopes and a bound τ.

Definition 19 *The* confrontation *constraint* $\bowtie^{\tau}$ *is defined as*

$$equ_{pre} \bowtie^{\tau} equ_{curr} = (||equ_{curr} - \texttt{ren}(equ_{pre})|| \leqslant \tau||equ_{pre}||) \tag{4.1}$$

[2] Our internal representation of reals uses 123 bits for the mantissa, catched by the extra non deterministic noise μ'_0.

Third step: confronting iterations and building a candidate. The success of the confrontation is a good clue that the current zonotope is close to an inductive invariant as the two iterations are close. More precisely, transforming each equ_{pre} by tf does not change it a lot. That is why a small *perturbation* of equ_{pre} is a good invariant candidate. A candidate invariant is built from equ_{pre} by following two heuristics.

- Adding a *generalization* of the previous equation equ_{pre} as a meta equation $m\text{-}equ \in Meta\text{-}Aff$. By confronting equ_{pre} with the next abstract value, we know that the difference is small (τ is expected to be small for the method to find a candidate invariant). To catch an upper bound of this difference in the general case, the generalization will be

  $$\sigma_{pre}.\tau.||equ_{pre}||.\varepsilon_\sigma$$

  with σ_{pre} the *stretch factor* a parameter left to find and ε_σ a fresh noise symbol.

- Adding a perturbation of the initial equation $\texttt{ren}(equ_{pre})$ represented by the meta equation

  $$\delta_{pre}.equ_{pre}$$

  where δ_{pre} is called the *expansion factor*.

The final candidate invariant is $m\text{-}equ_{inv}$ defined by

$$generalize(equ_{pre}) \stackrel{\text{def}}{=} (1+\delta_{pre})equ_{pre} + \sigma_{pre}.\tau.||equ_{pre}||.\varepsilon_\sigma \tag{4.2}$$

Both paramerers are greater than 0. This generalization is performed for every component of the vector describing the zonotope with different parameters δ and σ.
On the simple filter example of Figure 4.1, we choosed $\tau = \frac{1}{4}$. The confrontation succeds for all affine forms and the following candidates are generated:

$$\begin{array}{lcl} \texttt{S1} & := & (1+\delta_0)(\varepsilon_3 + 1.4\varepsilon_2 + 1.26\varepsilon_1) + 3.66\sigma_0\varepsilon_{\sigma_0} \\ \texttt{S0} & := & (1+\delta_1)(\varepsilon_4 + 1.4\varepsilon_3 + 1.26\varepsilon_2 + 0.784\varepsilon_1) + 4.444\sigma_1\varepsilon_{\sigma_1} \\ \texttt{S} & := & (1+\delta_2)(\varepsilon_5 + 1.4\varepsilon_4 + 1.26\varepsilon_3 + 0.784\varepsilon_2 + 0.2156\varepsilon_1) + 4.6596\sigma_2\varepsilon_{\sigma_2} \end{array}$$

Fourth step : finding a valuation of the parameters If all confrontations have been accepted by the previous step, then the invariant synthesizer provide an invariant candidate that involves all the variables modified by the loop. The job of this step is to take the invariant candidate and infer a valuation of the parameters, making it inductive. It first applies the loop transformation to the candidate invariant $m\text{-}z_{inv}$, which returns a new meta zonotope $m\text{-}z_{next}$.

Definition 20 *The* confrontation for inclusion *operator* $\bowtie$
returns a valuation $Val = (\delta_i, \sigma_i)_{0\leq i<n}$ *of the* $2n$ *parameters such that for two meta zonotopes* $m\text{-}z_1, m\text{-}z_2$*, we have :*

- *Val* $\stackrel{\text{def}}{=}$ *m-z*$_1$ $\bowtie$ *m-z*$_2$
- *m-z*$_1$[*Val*] $\subseteq$ *m-z*$_2$[*Val*] *for the zonotope inclusion.*

If no such Val is found, it returns $\bot$.

When this confrontation is applied to *m-z*$_{next}$ and *m-z*$_{pre}$ (*m-z*$_{next}$ $\bowtie$ *m-z*$_{pre}$) verifies that the inferred conclusion can match the induction hypotheses. If *Val* exists and is found, then *m-z*$_{inv}$[*Val*] is inductive.

Quantification of the error rate. The success of the $\bowtie$ operator guarantees to return an inductive invariant w.r.t. the analyzed transfer function. Still without the constraints enforced by the $\bowtie^\tau$ operator between two successive iterations and a small bound on parameters δ and σ, an inductive invariant may be found. However, there would consequently be no guarantee on how overapproximated it would be. Those bounds allow having a precise estimation of the invariant preciseness.

Theorem 2 *Let* z_n *the zonotope at the* n^{th} *iteration of the transfer function* tf. *If* $z_n \bowtie^\tau z_{n+1}$ *is satisfied,* $(\delta, \sigma) = tf(z_c) \bowtie z_c$ *and the resulting candidate invariant* z_c *verifies* $z_c(tf(z_c) \bowtie z_c) = z_{inv} \neq \bot$, *then* z_{inv} *is an inductive invariant of the loop and*

$$z_n \subseteq inv_{opt} \subseteq z_{inv} \subseteq (1 + \delta + \sigma.\tau)z_n$$

where inv_{opt} *is the optimal inductive invariant of the loop verifying* $z_n \subseteq inv_{opt}$.

Proof. The candidate invariant $z_c = (z_1, z_2, ..., z_d)$ is defined in equation (4.2):

$$z_i(\sigma, \delta) = (1 + \delta_i)\texttt{ren}(z_{k,i}) + \sigma_i.\tau.||z_{k,i}||.\varepsilon_\sigma$$

The norm of an affine form is the sum of the absolute value of its coefficients. As $z_{k,i}$ and $\sigma_i.\tau.||z_{k,i}||.\varepsilon_\sigma$ have disjoint noise symbols, the sum of their norm is the norm of their sum. Thus, $||z_i(\sigma, \delta)|| = ||(1 + \delta_i)z_{k,i}|| + \sigma_i.\tau.||z_{k,i}||$. As $||z_{k,i}|| = ||\texttt{ren}(z_{k,i})||$, it is clear that $||z_{k,i}|| \leqslant ||z_i(\sigma, \delta)|| \leqslant (1+\delta_i+\sigma_i\tau)||z_{k,i}||$ □

This theorem guarantees that if the method successfully generates an invariant, the interval projection of the zonotope on the intervals for each variable is strongly bounded by τ, δ_i and σ_i.

4.3.2 Inclusion of meta-zonotopes

Let z_k a zonotope and $z_{k+1} = tf(z_k)$, such that $z_k \bowtie^\tau z_{k+1}$. The candidate invariant z_1^ε is defined by $0 \leq i < n$ a vector of meta affine forms $(1 + \delta)equ_k + \sigma.\tau.||equ_k||.\varepsilon_\sigma$. Finding suitable parameters (δ_i, σ_i) for two meta zonotopes $z_1^{\varepsilon'}$ and $z_2^\varepsilon = tf(z_1^{\varepsilon'})$ to satisfy $z_2^\varepsilon \bowtie z_1^{\varepsilon'}$ requires to solve quantified constraints on ε and ε'. More precisely, the $\bowtie$ operator attempts to find (δ, σ) a vector of parameters such that $\forall\varepsilon, \exists\varepsilon'.z_2^\varepsilon(\delta_0, \sigma_0, ...) = z_1^{\varepsilon'}(\delta, \sigma)$. The presence

of quantifiers makes it impossible to send the constraints generated by $\bowtie$ to an implementation of the simplex algorithm for resolution. Also, noise symbols are shared between meta-equations of the zonotope, which complexify the search of parameters. That is why we propose some heuristics to reduce the number of ε that are quantified in the hope to ease the task of the solver.

1. renaming between ε' and ε;
2. heuristical and temporary definitions of ε' for simplification issues;
3. application of the Fourier-Motzkin algorithm [Mon10] for relation elimination issues.

The first technique is based on the renaming between the non deterministic ε' symbols of z_1 to match those ε of z_2 such that $z_2^{\varepsilon}(\delta,\sigma) = z_1^{\varepsilon'}(\delta,\sigma)$ gets easier to solve. Let us denote Λ_i (resp. Λ'_i) the coefficient of ε_i (resp. ε'_i) in $\textit{m-equ}_{next}$ (resp. $\textit{m-equ}_{pre}$ and $\textit{m-equ}_{inv}$). They are linear combinations of δ_i, σ_i. Let us assume that the transfer function tf adds s new non deterministic symbols to the zonotope. To check the confrontation in the previous section, we applied a shift on the indexes of the noise symbols of $tf(z_{pre})$ of s so that their coefficients match. We will apply the same idea here, in other words, $\textit{m-equ}_{pre} = \Lambda'_0 + \sum_{i=1}^{n} \Lambda'_i\varepsilon'_i$ and $\textit{m-equ}_{next} = \Lambda_0 + \sum_{i=1}^{n} \Lambda_{i+s}\varepsilon_{i+s} + \sum_{i=1}^{s} \Lambda_i\varepsilon_i$ where $(\varepsilon_i)_{s+1\leq i\leq n+s}$ are fresh non deterministic symbols. Three different cases are possible:
If $|\Lambda_{i+s}| \leqslant |\Lambda'_i|$ **for all valuations of** δ, σ, then ε'_i is redefined as ε_{i+s}.
If $|\Lambda_{i+s}| > |\Lambda'_i|$ **for all valuations of** δ, σ, then ε'_i is redefined as $\varepsilon'_i = \frac{\Lambda_{i+s}}{\Lambda'_i} lin\varepsilon_{i+s} + (1 - \frac{\Lambda_{i+s}}{\Lambda'_i} lin)\varepsilon''_i$ where ε''_i is a new fresh non deterministic noise and the linearized divisions $(\frac{a}{b})_{lin}$ are the first order Taylor approximations of $\frac{a}{b}$. The linearization is important because the ε'_i intervene in other affine forms and at the end we send linear constraints in δ, σ to a linear solver. This renaming eases the search of δ, σ satisfying the induction criterion. Indeed, we first notice that $\varepsilon'_i \equiv \frac{\Lambda_{i+s}}{\Lambda'_i} lin\varepsilon_{i+s} + (1 - \frac{\Lambda_{i+s}}{\Lambda'_i} lin)\varepsilon''_i$ for any valuation of Λ_{i+s} and Λ'_i, so we do not lose information through this renaming. Then, we get rid of ε_{i+s} as the following equalities hold:

$$\begin{aligned}\textit{m-equ}_{next} - \texttt{ren}(\textit{m-equ}_{pre}) \;=\; & \Lambda_0 - \Lambda'_0 + \sum_{i=1}^{s} \Lambda_i\varepsilon_i + \sum_{i=1}^{n} (\Lambda'_i - \Lambda'_i \tfrac{\Lambda_{i+s}}{\Lambda'_i} lin)\varepsilon''_i \\ & + \sum_{i=1}^{n} \left(\Lambda_{i+s} - \Lambda'_i \left(\tfrac{\Lambda_{i+s}}{\Lambda'_i} lin\right)\right) \varepsilon_{i+s}\end{aligned}$$

Parameters δ, σ such that $\forall\varepsilon, \exists\varepsilon'.\ \texttt{ren}(\textit{m-equ}_{pre}) = \textit{m-equ}_{next}$ can be found by finding δ, σ satisfying $\exists\varepsilon''$.
$|\Lambda_0 - \Lambda'_0| + \sum_{i=1}^{s} |\Lambda_i| + \sum_{i=1}^{n} \left|\Lambda_{i+s} - \Lambda'_i \left(\frac{\Lambda_{i+s}}{\Lambda'_i} lin\right)\right| = \sum_{i=1}^{n} (\Lambda'_i - \Lambda'_i \frac{\Lambda_{i+s}}{\Lambda'_i} lin)\varepsilon''_i$ which is much simpler as we removed a universal quantifier.
If $|\Lambda_{i+s}| \leqslant |\Lambda'_i|$ **is not decided** , it is still possible to generate two constraint systems. One with the additional constraint $|\Lambda_i| \geqslant |\Lambda'_{i+s}|$ and one with the additional constraint $|\Lambda_i| \leqslant |\Lambda'_{i+s}|$.

Remaining shared symbols. One difficulty of inclusion comes from the shared noise symbols. Particularily, the ε_i'' introduced by the second strategy can be common to every affine form of the zonotope. When the solver is still unable to solve these constraints, we can try to remove them from the equation. A possibility to reduce the number of shared symbols is the Fourier-Motzkin algorithm used in [Mon10].

Incomplete. These heuristics are used in the current implementation of the algorithm as they have shown to increase the efficiency of the analysis. However, they are incomplete in the general case, in the sense that it is possible the renaming does deletes some possible solutions.

4.4 Completness on linear filters

The method described in the previous section relies on the automatic discovery of a precise pattern in the values describing the variables domain. In addition, it must find suitable parameters for the candidate invariant to be inductive, but there is no guarantee such values exists. If the domain diverges, i.e. it keeps growing, the method will not converge as the abstract interpreter will keep iterating. Besides, if we can guarantee that the variables domain converges then both the expansion factor and the stretch factor would actually converge to 0.

Theorem 3 *Let $(S_n)_{n\in\mathbb{N}}$ a linear filter. If S is convergent, the algorithm in Figure 4.5 will find an inductive invariant for S.*

Proof. We will first see in this proof that two successive iterations are getting closer and closer after each iteration. Let S_n a linear filter defined by a matrix M and a sequence of non determinsitic noises V_n. Let $Z_n = (Z_n^1, ..., Z_n^k)$ the sequence of zonotopes defined as $Z_0 = V_0$ and $Z_{n+1} = S.Z_n + V_{n+1}$. Let us denote $tf(Z_n) = S.Z_n + V_{n+1}$. Each component Z_n^i of Z_n is a Minkowski sum $\sum_{j=0}^{n^2} \alpha_j^i \varepsilon_j$ where α_j^i are real constants, possibly null. This sum has indeed n^2 terms at the n^{th} element of the sequence as each step adds at most n new noise symbols (one by variable).

Let $ren(Z_n^k) = \sum_{j=0}^{n^2} \alpha_j^k \varepsilon_{j+n}$ the shift of n on the indexes of ε_j. The shift function is linear, in the sense that $ren(z+z') = ren(z)+ren(z')$. Let us extend the ren notation to Z_n by applying it to each component of the vector. As V_n adds n new noise symbols, we will say that $ren(V_n) = V_{n+1}$. Also, as ren only changes the indexes of ε_j, we have that for any zonotope Z, $ren(Z) \subseteq Z$ and $Z \subseteq ren(Z)$. Let us first prove the following lemma.

Lemma 6 *There exist s such that $(Z_{n+1}^i - ren(Z_n)^i = o(e^{-n}n^s)$.*

Proof. We first notice that

$$Z_n = \sum_{i=0}^{n} M^i V_{n-i}$$

There comes that

$$Z_{n+1} = \sum_{i=0}^{n} M^{i+1} V_{n-i} + V_{n+1}$$

This expression is equivalent to

$$Z_{n+1} = \sum_{i=0}^{n} M^i V_{n-i+1} + M^{n+1} V_0$$

hence we have that

$$Z_{n+1} = ren(Z_n) + M^{n+1} V_0$$

As S is a convergent lienar filter, all the eigenvalues of M are lower than 1. Its Jordan Normal form J (cf Section 2.1.6) is a upper triangular matrix with eigenvalues on its diagonal. As there exist P such that $M = P^{-1}.J.P$, we have that $M^n = P^{-1}.J^n.P$. J is composed of blocks J_i of size s such that

$$J_i = \begin{pmatrix} \lambda_i & 1 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & \dots & 0 & \lambda_i \end{pmatrix}$$

We know then that

$$J_i^n = \begin{pmatrix} \lambda_i^n & \lambda^{n-1} * n & \dots & \lambda^{n-s} * P(n) \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & \dots & 0 & \lambda_i^n \end{pmatrix}$$

where $P(n)$ is a polynomial in n of degree $s-1$. As every λ_i is defined such that $|\lambda_i| \leqslant 1$, the modulus of each coefficients of this block is asymptotically equivalent to $e^{-n}n^{s-1}$ near infinity. We recall that there exist P such that $M^n = P^{-1}.J^n.P$. Finite linear combinations of elements that are asymptotically equivalent to $e^{-n}n^{s-1}$ are also asymptotically equivalent to $e^{-n}n^{s-1}$.

□

This lemma has two interesting corolaries:

- $\lim_{n \to +\infty} (Z_{n+1} - ren(Z_n)) = 0$;
- Z_n converges.

Let us denote $Z^l = \lim_{n \to +\infty} Z_n$. For any δ, there exist n such that $Z_n \subset Z_l \subseteq (1+\delta)Z_n$.

Program	Var	Steps	Time (in ms)
Example 1.1	2	14(2)	7
Example 1.2	3	18(5)	37
Simple filter	3	22(2)	9
[MBR16] filter	3	22(2)	20

TABLE 4.2: Performance results with the method's implementation in *Fluctuat* [Gou13]. The first two columns are the input program and the corresponding number of variables. The number of iterations needed to reach a solvable candidate invariant is in the third column, with the number of solving attempts by $\bowtie$ before finding parameters making the invariant inductive. The 4th column represents the total time taken for inferring the invariant.

Also, tf is continue as it is an affine application on zonotopes (by considering V_n as a sequence of zonotope). As Z_l is a limit of a serie described by tf, then Z_l is a fix point of tf and therefore, an invariant for tf.
□

4.5 Experiments and conclusion

A prototype of the technique described in this Chapter has been implemeted in *Fluctuat* [Gou13], an abstract intepreter based on the zonotope abstract domain. The first implementation only uses the stretch factor on every equation of the linear filter, which was not sufficient to find an inductive invariant. On the other hand, the expansion factor is sufficient for the algorithm to converge (which echoes the argument in the proof of Theorem 3 of choosing $\sigma = 0$) though the composition of the two makes the algorithm converge faster on the tested examples. We have applied the method on several linear filters (see Table 4.2). Choosen parameters for the $\bowtie^{\tau}$ operation $\tau = \frac{1}{64}$, for the stretch factor μ the interval $[-\frac{\|equ_{pre}\|}{64}, \frac{\|equ_{pre}\|}{64}]$ and for δ the interval$[-\frac{1}{64}, \frac{1}{64}]$. In other words, we will study the candidate invariant

$$\frac{64+\lambda_0}{64}\left(\beta_0 + \sum_{i=k+1}^{n} \beta_i \varepsilon_i\right) + \frac{\lambda_1}{64}\left(\sum_{i=0}^{n} |\alpha_i|\right)\varepsilon_p$$

if and only if $\sum_{i=1}^{n} |\beta_i - \alpha_i| \leqslant \frac{1}{64}\sum_{i=1}^{n} |\alpha_i|$. After 23 iterations, the method successfully generates an inductive invariant that is by construction very close to the optimal invariant (by Theorem 2, at most $\frac{1}{64} + \frac{\|equ_{pre}\|}{64^2}$ bigger than the optimal zonotope solution). The low number of tested programs is due to

the difficulty of the inplace solver to find convenient parameters for candidate invariants. While this method generates many candidates passing the confrontation, the constraint solver finds parameters only for a few of them. The zonotope abstract domain is able to catch very precise relations for every variable on a non-deterministic loops. One of the main issue for finding such relations is to make abstract iterations converge. This widening operator allows getting rid of this problem by bypassing all the difficulty thanks to a dedicated solver. Though this solver theoretically solves any confrontation constraint, the optimization choice of reducing as much as possible the value of the meta-zonotopes parameters sometimes lead to bad computation time. The recent development of SMT-solving over non linear constraints over the reals [GKC13] is a promising solution to this issue.

Chapter 5

Eigenvectors as linear invariants of linear loops

Contents

Requirements: Linear algebra (Section 2.1), Programming model (Section 2.2)

The previous Chapter presented a method for determining zonotope invariants for a specific kind of linear loops (convergent linear filters). Abstract iteration is the most commonly used technique when generating invariants for such loops [Mau04; Del+09; Gou13; RG13], as its genericity allows giving a formal semantic to non-determinism (the interval domain, the octagon domain, the zonotope domain, etc.). The issue of such an approach is the lack of control of the analysis. Once the user launches an abstract interpreter, it

has very few control on the operations performed and the computation in general (though semantic rules are consistently defined).

Instead of abstract intepretation, multiple works attempted to generate invariants [Kov08; RK07] by analyzing a loop without value propagation nor predicate abstraction, but as a whole transformation with inherent properties. For example, [RK07] generates polynomial invariants by working on the *Gröbner base* of the polynomial ring and using the ideal properties of invariants. The loop is analyzed once and invariants are directly synthesized. The principal drawback of such techniques compared to abstract intepretation is their over-specialization: they can only work on very specific kind of loops. However, this issue already arise in the previous Chapter as the algorithm only converges on convergent linear filters, a sub class of the linear transformations (cf Theorem 3).

This Chapter will treat of a straightforward generation of invariants for linear loops in general, including linear filters and their non determinism. More precisely, it syntactically extracts a *complete characterization of linear invariants of linear loops*.

This chapter is based on work that has been presented in [OBP16; OBP17]

5.1 Overview

Informally, an invariant for a loop is a formula that

1. is valid at the beginning of the loop (*initialized invariants*);
2. stays valid after every loop step (*semi-invariants*).

We are interested in finding *polynomial numerical relations* on variables complying only with the second criterion such that they can be expressed as a linear equation over X, a vector containing the assignment's original variables and the *monomial variables* generated by the linearization procedure described in Chapter 3. In this setting, a formula satisfying the second criterion can be represented as a vector of coefficients φ such that for a loop transformation f we have

$$\langle \varphi, X \rangle = 0 \Rightarrow \langle \varphi, f(X) \rangle = 0 \tag{5.1}$$

By linear algebra, the following is always true

$$\langle \varphi, f(X) \rangle = \langle f^*(\varphi), X \rangle \tag{5.2}$$

where f^* is the dual of f. If φ happens to be an eigenvector of f^* (i.e. there exists λ such that $f^*(\varphi) = \lambda\varphi$), the equation (5.1) becomes

$$\begin{aligned}
\langle \varphi, X \rangle = 0 &\Rightarrow \langle f^*(\varphi), X \rangle = 0 \text{ by (5.2)} \\
\langle \varphi, X \rangle = 0 &\Rightarrow \langle \lambda.\varphi, X \rangle = 0 \\
\langle \varphi, X \rangle = 0 &\Rightarrow \lambda.\langle \varphi, X \rangle = 0
\end{aligned}$$

which is always true. Therefore, the relation $\langle \varphi, X \rangle = 0$ is inductive for every valuation of X.

Example. The deterministic polynomial loop in Figure 3.1, implements successive applications of the transformation $f(x, y) = (x + y^2, y + 1)$. In Chapter 3, we have proven that this polynomial loop can be replaced by a linear one by replacing its body transformation f by $g(x, y, y_2, \mathbb{1}) = (x+y_2, y+\mathbb{1}, y_2+2y+\mathbb{1}, \mathbb{1})$. The problem of finding polynomial invariants is reduced to finding linear invariants. However g does not admit any useful linear invariant, while f admits $-6.x+y-3.y^2+2.y^3 = 0$ as an invariant relation between variables (if x and y starts at 0). To solve this issue, the elevation principle introduced in Section 3.1 allows expressing the value of y^3 after multiple iterations of f by a new variable y_3, as we did for y^2 with y_2. The loop transformation f can be replaced by $h(x, y, y_2, y_3, \mathbb{1}) = (x+y_2, y+\mathbb{1}, y_2+2y+\mathbb{1}, y_3+3y_2+3y+\mathbb{1}, \mathbb{1})$

We just need to *transpose* the matrix representing h to compute h^* . It returns $h^*(x, y, y_2, y_3, \mathbb{1}) = (x, y + y_2 + y_3, x + y_2 + 3.y_3, y_3, y + y_2 + y_3 + \mathbb{1}, y + y_2 + y_3 + \mathbb{1})$. h^* only admits the eigenvalue 1. The eigenspace of h^* associated to 1 is generated by two independants vectors, $e_1 = (-6, 1, -3, 2, 0)^t$ and $e_2 = (0, 0, 0, 0, 1)^t$. Eventually, we get the formula $F_{k1,k2} = (k_1.(-6.x + y - 3.y_2 + 2.y_3) + k_2.\mathbb{1} = 0)$ as invariant, with $k_1, k_2 \in \mathbb{Q}$. By writing $k = -\frac{k_2}{k_1}$ and replacing $\mathbb{1}$ with 1, we can rewrite it with only one parameter, $F_k = (-6.x + y - 3.y_2 + 2y_3 = k)$. In this case, information on the initial state of the loop allows to fix the value of the parameter k. For example if the loop starts with $(x = 0, y = 0)$, then $-6.x + y - 3.y^2 + 2.y^3 = 0$, and F_0 is an invariant.

Contribution of this Chapter. This overview presented a shade of the link between the eigenvector decomposition of a linear transformation and its family of inductive invariants. Chapter 5 goes further by:

- proving that left-eigenvectors of a transformation f (i.e. eigenvectors of the dual f^*) are *exactly* the set of semi-invariants of a loop;
- studying the effects of the eigenvalues on those invariants;
- extending this characterization to conditional loops, nested loops and, at last, non deterministic loops.

5.2 Simple loops

5.2.1 Semi-invariants

As said before, loop invariants can be characterized by two criteria : they have to hold at the beginning of the loop (initialization criterion) and if they hold at one step, then they hold at the next step (inductivity criterion). The technique described here is based on the discovery of linear combinations of variables that are equal to 0 and satisfying the inductivity criterion. For example, the loop of section 3.1 admits the formula $-6.x + y - 3.y^2 + 2y^3 = k$

as a good invariant candidate. Indeed, if we set k in accordance with the values of the variables at the beginning of the loop, then this formula will be true for any step of the loop. We call such formulas *semi-invariants*.

Definition 21 *Let $\varphi : \mathbb{K}^n \mapsto \mathbb{K}$ and $f : \mathbb{K}^n \mapsto \mathbb{K}^n$ two linear mappings. φ is a semi-invariant for f iff $\forall X, \langle \varphi, X \rangle = 0 \Rightarrow \langle \varphi, f(X) \rangle = 0$.*

Definition 22 *Let $\varphi : \mathbb{K}^n \mapsto \mathbb{K}$, $f : \mathbb{K}^n \mapsto \mathbb{K}^n$ and $X \in \mathbb{K}^n$. φ is an invariant for f with initial state X iff $\langle \varphi, X \rangle = 0$ and φ is a semi-invariant for f.*

5.2.2 Eigenvectors are invariants

The key point of this technique relies on the fact that if there exists $\lambda, f^*(\varphi) = \lambda\varphi$, then we know that φ is a semi-invariant. Indeed, we can rewrite definition 21 by $\langle \varphi, x \rangle = 0 \Rightarrow \langle \varphi, f(x) \rangle = 0$. By linear algebra, we have $\langle \varphi, f(x) \rangle = \langle f^*(\varphi), x \rangle$, with f^* the dual of f. If $\exists \lambda, f^*(\varphi) = \lambda\varphi$, then we can deduce that $\langle \varphi, x \rangle = 0 \Rightarrow \lambda \langle \varphi, x \rangle = 0$. This formula is always true, thus we know that φ is a semi-invariant. Such φ are commonly called *eigenvectors* of f^*. We will not adress the problem of computing the eigenvectors of an application as this problem has been widely studied (in [PC99] for example).

Recall the running example $g(x, y) = (x{+}y^2, y{+}1)$, linearized by the application $f(x, y, y_2, xy, y_3, \mathbb{1}) = (x{+}y_2, y{+}\mathbb{1}, y_2{+}2y{+}\mathbb{1}, xy{+}x{+}y_2{+}y_3, y_3{+}3y_2{+}3y{+}\mathbb{1}, \mathbb{1})$. f^* admits $e_1 = (-6, 1, -3, 0, 2, 0)^t$ and $e_2 = (0, 0, 0, 0, 0, 1)^t$ as eigenvectors associated to the eigenvalue $\lambda = 1$. It means that if $\langle k_1.e_1 + k_2e_2, x \rangle = 0$, then

$$\begin{aligned} \langle k_1.e_1 + k_2e_2, f(X) \rangle &= \langle f^*(k_1.e_1 + k_2e_2), X \rangle \\ &= \langle \lambda(k_1.e_1 + k_2e_2), X \rangle \\ &= 0 \end{aligned}$$

In other words, $\langle k_1.e_1 + k_2e_2, X \rangle = 0$ is a semi-invariant. Then, by expanding it, we can find that $-6.x + y - 3.y_2 + 2y_3 = k$, with $k = -\frac{k_2}{k_1}$ is a semi-invariant. In terms of the original variables, we have thus $-6.x + y - 3.y^2 + 2y^3 = k$.

Being an eigenvector of f^* does not just guarantee a formula to be a semi-invariant of a loop transformed by f. This is also a necessary condition.

Theorem 4 *φ is a semi-invariant of f if and only if φ is a left-eigenvector of f.*

Proof. Let φ a semi-invariant. By definition, $(\langle \varphi, x \rangle = 0 \Rightarrow \langle \varphi, f(x) \rangle = 0)$. This means that $Vect(\varphi)^\perp$ is stable by f, so by Lemma 1, $(Vect(\varphi)^\perp)^\perp = Vect(\varphi)$ is stable by f^*. As $\varphi \in Vect(\varphi)$, we have $f^*(\varphi) = k.\varphi$.

□

Remark. Vectors that do not belong to an eigenspace are not semi-invariants. This is especially true for vectors such as $\varphi = \psi + \sigma$ where ψ and σ are not colinear to φ and belong to two different eigenspaces. In this case, we have that:

```
while ⋆ do
   x = e1;

   {x = e21} OR {x = e22};

   x = e3

done
```

```
while ⋆ do
  {x = e1;
   x = e21;
   x = e3}

   OR

  {x = e1;
   x = e22;
   x = e3}

done
```

FIGURE 5.1: The non deterministic choice in the middle of the first loop can be made at the beginning of the loop.

$$\begin{aligned} f^*(\varphi) &= \lambda_\psi.\psi + \lambda_\sigma.\sigma \\ &= \lambda_\psi.(\psi + \sigma) + \tfrac{\lambda_\sigma}{\lambda_\psi}.\sigma \\ &= \lambda_\psi.\varphi + \tfrac{\lambda_\sigma}{\lambda_\psi}.\sigma \end{aligned}$$

φ is then clearly not an eigenvector, thus by Theorem 4 not a semi-invariant. Semi-invariants of f don't belong to, the direct sum of the eigenspaces, but to *their union*.

An element φ of E_λ of basis $\{e_1, ...e_n\}$ is a linear combination of $e_1, ..., e_n$:

$$\varphi = \sum_{k=1}^{n} k_i e_i$$

The parameters k_i can be chosen with respect to the initial state of the loop.

Algorithm. As we are restricting this study to solvable loops, that we know can be replaced without loss of generality by linear loops, we assume the input of this algorithm is a finite sequence of linear mappings. Their composition A is also linear, and computed by multiplying each matrix. Computing the dual of A is computing the matrix A^T. Then, eigenvectors of A^T can be computed by many algorithms in the linear algebra literature [PC99]. As the eigenvalue problem is known to be polynomial, this invariant generation algorithm is also polynomial.

5.3 Conditions

The construction i OR i of the semantics in Definition 2.2 defines a non deterministic choice between two possible instructions. If such a construction occurs in the middle of a loop, this choice can be lifted to the beginning of the loop [1] (cf Figure 5.1).

[1] This could also be done if the condition was not deterministic. Indeed, testing if $x > 0$ after the assignment $x = x + 1$ is equivalent to test is $x + 1 > 0$ before the assignment.

Definition 23 *Let $F = \{A_i\}_{1 \leqslant i \leqslant n}$ a family of matrices and $Inv(F)$ the set of invariants of a loop whose different bodies can be encoded by elements of F.*

$$Inv(F) = \{\varphi | \forall X, \varphi.X = 0 \Rightarrow \bigwedge_{i=1}^{n} \varphi.A_i.X = 0\}$$

Definition 23 define semi-invariants of conditional loops as relations preserved by each body loops. This definition is consistent with the definition of semi-invariants in general as, if there were a semi-invariant φ of such a loop, φ has to be an invariant for each loop body. The problem to compute $Inv(F)$ can then be reduced to the computation of the intersection of the invariants sets of every loop body.

Property 10 *Let $F = \{A_i\}_{1 \leqslant i \leqslant n}$ a family of matrices.*

$$Inv(F) = \bigcap_{i=1}^{n} Inv(A_i)$$

Proof.

Lemma 7
Let $F = \{A_i\}_{1 \leqslant i \leqslant m}, G = \{B_i\}_{1 \leqslant i \leqslant n}$ two matrix families. Then $Inv(F \cup G) = Inv(F) \cap Inv(G)$

Proof. As we have $((p \Rightarrow q) \wedge (p \Rightarrow r)) \Leftrightarrow (p \Rightarrow (q \wedge r))$,

$$\begin{aligned} Inv(F) \cap Inv(G) &= \left\{ \varphi | \forall X, \begin{array}{rcl} & (\varphi.X = 0 & \Rightarrow \bigwedge_{i=1}^{m} \varphi.A_i.X = 0) \\ \wedge & (\varphi.X = 0 & \Rightarrow \bigwedge_{i=1}^{n} \varphi.B_i.X = 0) \end{array} \right\} \\ &= \{\varphi | \forall X, \varphi.X = 0 \Rightarrow (\bigwedge_{i=1}^{m} \varphi.A_i.X = 0 \wedge \bigwedge_{i=1}^{n} \varphi.B_i.X = 0)\} \\ &= Inv(F \cup G) \end{aligned}$$

□

We can now prove Property 5.3 by induction over the size n of a family F. If n equals 1, it is clearly true. If it is true for a certain n, then $Inv(F \cup \{A\}) = Inv(F) \cap Inv(\{A\})$ by the previous lemma.

□

As the set of invariants of a single-body loop are a vector spaces union, its intersection with another set of invariants is also a vector space union. Although we do not consider the condition used by the program to choose the correct body, we still can discover useful invariants. Let us consider the following example, taken from [RK07], that computes the product of x and y in a variable z :

```
while (*) do
    (x,y,z) := (2x, (y-1)/2, x + z)
    OR
    (x,y,z) := (2x, y/2, z)
done
```

We have to deal with two applications : $f_1(x, y, z) = (2x, (y - 1)/2, x + z)$ and $f_2(x, y, z) = (2x, y/2, z)$. The elevation to the degree 2 of f_1 and f_2 returns applications having both 10 eigenvectors. For simplicity, we focus on invariants associated to the eigenvalue 1.

f_1^* has 4 eigenvectors $\{e_i\}_{i\in[1,4]}$ associated to 1 such that

- $\langle e_1, X\rangle = -x + xy$
- $\langle e_2, X\rangle = x + z$
- $\langle e_3, X\rangle = xz + x^2 + z^2$
- $\langle e_4, X\rangle = \mathbb{1}$

f_2^* also has 4 eigenvectors $\{e'_i\}_{i\in[1,4]}$ associated to 1 such that

- $\langle e'_1, X\rangle = xy$
- $\langle e'_2, X\rangle = z$
- $\langle e'_3, X\rangle = z^2$
- $\langle e'_4, X\rangle = \mathbb{1}$

First, we notice that $e_4 = e'_4$. Then, we can see that $\langle e_1 + e_2, X\rangle = xy + z = \langle e'_1 + e'_2, X\rangle$. Thus, $e_1 + e_2 = e'_1 + e'_2$. Eventually, we find that $e_1 + e_2 + k.e_4 \in (Vect(\{e_i\}_{i\in[1,4]}) \cap Vect(\{e'_i\}_{i\in[1,4]}))$. That's why $(\langle e_1 + e_2 + k.e_4, X\rangle = 0)$ is a semi-invariant for both f_1 and f_2, hence for the whole loop. Replacing $\langle k.e_4, X\rangle$ by $k = -k'$ and $\langle e_1 + e_2, X\rangle$ by $xy + z$ gives us $xy + z = k'$.

Algorithm. The intersection of two vector spaces corresponds to the vectors that both vector spaces have in common. It means that such elements can be expressed by elements of the base of each vector space. Let B_1 and B_2 bases of the two vector spaces. If $e \in \text{Vect}\{B_1\}$ and $e \in \text{Vect}\{B_2\}$, then $e \in \ker\{(B_1B_2)\}$, where B_1B_2 is the concatenation of both bases in a single matrix. To compute the intersection of a vector space union, we just have to compute the kernels of each combination of vector space in the union.

5.4 Nested loops

When a loop is a linear application, it can be represented by the matrix product of each instruction which composes every instruction into a single one. Finding the matrix of a loop is thus only possible when there is no nested loops. When a condition occurs, we simply enumerate all the possible paths, to which we associate their matrices. For example, with the first program of 5.2, we can see that the matrix $C.A$ represents the loop, as applying it to X at the beginning computes one whole step of the loop. Moreover there is only one matrix as there is only one possible path.

In Section 5.3, we have shown that every invariant of this loop is exactly the intersection of the eigenspaces union of the transposed matrices of each

```
while ⋆ do
  X = A.X;
  X = C.X
done
```

```
while ⋆ do
  X = A.X;
  while ⋆ do
    X = B.X
  done;
  X = C.X
done
```

FIGURE 5.2: Invariants of the first loops are left-eigenvectors of $C.A$, but the second loop cannot be represented as a single matrix.

path, which is here the eigenspaces union of $((C.A)^T)$. However, the second program of Figure 5.2 cannot be treated as easily as the first one as, considering non-deterministic conditions, we cannot determine how many times B will be applied. The sequence of matrices associated to this loop is $(\{C.B^n.A\})_{n\in\mathbb{N}}$, i.e. each element of this sequence is a possible path the loop can take. Thus the set of invariants for such loop is :

$$Inv(Prog) = \bigcap_{i\in\mathbb{N}} Inv(C.B^i.A)$$

This infinite intersection is naively impossible to compute, but the following theorema allows us to simplify the computation.

Theorem 5 *Let d the degree of the minimal polynomial of B. Then*

$$Inv(Prog) = \bigcap_{i=0}^{d-1} Inv(C.B^i.A)$$

Proof.

- The idea is to consider the minimal polynomial to stop iterating as soon as we can. Let $P(X) = \sum_{i=0}^{d} p_i X^i$ the minimal polynomial of B. Without losing generality, we will assume that $p_d = 1$

$$\langle \varphi, X\rangle = 0 \quad \Rightarrow \quad \langle \varphi, CA.X\rangle = 0 \tag{5.3}$$

  is the definition of the invariant of CA. As another path possible is CBA, we also need to check for

$$\langle \varphi, X\rangle = 0 \quad \Rightarrow \quad \langle \varphi, CBA.X\rangle = 0$$

  and so on. Let $\mathcal{P}(n) = \mathcal{P}(n-1) \wedge (\langle \varphi, X\rangle = 0 \Rightarrow \langle \varphi, CB^nA.X\rangle = 0)$. The case $\mathcal{P}(0)$ is directly solved by Equation (5.3).

$$\mathcal{P}(d-1) = (\langle \varphi, X \rangle = 0 \;\Rightarrow\; \bigwedge_{i=0}^{d-1} \langle \varphi, CB^iA.X \rangle = 0)$$

The goal is to prove that $\mathcal{P}(d-1) \Rightarrow \mathcal{P}(d)$. Let $\varphi \in \{v : \mathcal{P}(d-1)\}$. Then

$$\begin{array}{rcl} \langle \varphi, X \rangle = 0 & \Rightarrow & \langle \varphi, CA.X \rangle = 0 \\ \langle \varphi, X \rangle = 0 & \Rightarrow & \langle \varphi, CBA.X \rangle = 0 \\ & \cdots & \\ \langle \varphi, X \rangle = 0 & \Rightarrow & \langle \varphi, CB^{d-1}A.X \rangle = 0 \end{array}$$

$P(B) = 0$ by definition, so $B^d = \sum_{i=0}^{d-1} -p_i B^i$. Thus, $\langle \varphi, CB^dA.X \rangle = \langle \varphi, CQ(B)A.X \rangle$ because $\langle ., . \rangle$ is bilinear ($\langle a, b+c \rangle = \langle a, b \rangle + \langle a, c \rangle$) By hypothesis we know that if $\langle \varphi, X \rangle = 0$ then for every $i < d$ $\langle \varphi, CB^iA.X \rangle = 0$, thus by bilinearity we also have

$$\begin{array}{rcl} \langle \varphi, CQ(B)A.X \rangle & = & \sum_{i=0}^{d-1} -p_i \langle \varphi, CB^iA.X \rangle \\ \langle \varphi, CQ(B)A.X \rangle & = & 0 \end{array}$$

□

This theorem allows us to minimize the number of needed iterations of the nested loop that are necessary for generating all invariants of the main one. It relies on the knowledge of the minimal polynomial degree associated to a matrix $M \in \mathcal{M}_n(\mathbb{K})$, which always divides the characteristic polynomial of M whose degree is exactly n. In term of complexity, we know the intersection of the union of k vectorial spaces of dimension n costs $O(k.n^3)$. Thus, the worst-case complexity of dealing with nested loops is $O(k.n^4)$.

5.5 The case $\lambda = 1$

5.5.1 The variable $\mathbb{1}$

We recall from Chapter 3 that affine transformations are linearizable. Every affine constant α is replaced by the expression $\alpha * \mathbb{1}$, where $\mathbb{1}$ is a new variable always equal to 1. More than a syntactic sugar, the variable $\mathbb{1}$ brings interesting properties over the kind of invariants we generate for an application f. The vector $e_{\mathbb{1}}$ such that $\langle e_{\mathbb{1}}, X \rangle = \mathbb{1}$ is always an eigenvector associated to the eigenvalue 1. Indeed, by definition $f(\mathbb{1}) = \mathbb{1}$, hence $f^*(e_{\mathbb{1}}) = e_{\mathbb{1}}$. For example, let's take the mapping $f(x, y, xy, \mathbb{1}) = (2x, \frac{1}{2}y + 1, xy + 2x, \mathbb{1})$. This

mapping admits 3 eigenvalues : 2, $\frac{1}{2}$ and 1. There exist two eigenvectors for the eigenvalue 1 : $(-2,0,1,0)$ and $(0,0,0,1) = e_{\mathbb{1}}$. We have then the semi-invariant $k_1.(-2x + xy) + k_2 = 0$, or $-2x + xy = \frac{-k_2}{k_1}$. This implies that the two parameters k_1 and k_2 can be reduced to only one parameter $k = \frac{-k_2}{k_1}$, which simplifies a lot the equation by providing a way to compute the parameter at the initial state if we know it. For our example, $\frac{-k_2}{k_1}$ would be $-2x_{init} + x_{init}.y_{init}$, where x_{init} and y_{init} are the initial values of x and y. More generally, each eigenvector associated to 1 gives us an invariant φ that can be rewritten as $\varphi(X) = k$, where k is inferred from the initial value of the loop variables.

5.5.2 Quantified expression of invariants as eigenvectors.

We can generalize this observation to eigenvectors associated to any eigenvalue. To illustrate this category, let us take as example $f(x,y,z) = (2x, 2y, 2z)$. Eigenvectors associated to 2 are $e_1 = (1,0,0)$, $e_2 = (0,1,0)$ and $e_3 = (0,0,1)$, thus $k_1x + k_2y + k_3z = 0$ is a semi invariant, for any k_1, k_2 and k_3 satisfying the formula for the initial condition of the loop. However, if we try to set e.g. $k_1 = k_2 = 1$, using $x + y + kz = 0$ as semi invariant, we won't be able to find a proper invariant when y_{init} or $x_{init} \neq 0$ and $z_{init} = 0$. Thus, in order to keep the genericity of our formulas, we cannot afford to simplify the invariant as easily as we can do for invariants associated to the eigenvalue 1. Namely for every e_i, we have to test whether $\langle e_i, X_{init}\rangle = 0$. For each e_i for which this is the case, $\langle e_i, X\rangle = 0$ is itself an invariant if $\langle e_i, X_{init}\rangle = 0$. However, if there exists an i such that $\langle e_i, X_{init}\rangle \neq 0$, then we can simplify the problem. For example, we assume that $z_{init} \neq 0$. Then $k_1x_{init} + k_2y_{init} + k_3z_{init} = 0 \Leftrightarrow \frac{k_1x_{init}+k_2y_{init}}{z_{init}} = -k_3$. We know then that $k_1x + k_2y = \frac{k_1x_{init}+k_2y_{init}}{z_{init}}z$ is a semi-invariant. By writing $g(k_1,k_2) = \frac{k_1x_{init}+k_2y_{init}}{z_{init}}$, we have

$$\begin{cases} x &= g(1,0)z \\ y &= g(0,1)z \end{cases}$$

As g is a linear application, these two invariants implies that $\forall k_1, k_2, k_1x + k_2y = g(k_1,k_2)z$ is a semi-invariant.

Property 11 *Let $\mathcal{F}$ a semi-invariant expressed as $\mathcal{F} = \sum_{i=0}^{n} k_ie_i$.*

If $\langle e_0, X_{init}\rangle \neq 0$, then we have that

$$\bigwedge_{i=1}^{n}(\langle e_i, X\rangle = -\frac{\langle e_i, X_{init}\rangle}{\langle e_0, X_{init}\rangle}\langle e_0, X\rangle) \text{ is an invariant} \Leftrightarrow \langle \mathcal{F}, X_{init}\rangle = 0$$

Proof.

$$\begin{aligned} \langle \mathcal{F}, X_{init}\rangle = 0 &\Leftrightarrow \left\langle \sum_{i=1}^{n} k_ie_i, X_{init} \right\rangle = -k_0 \langle e_0, X_{init}\rangle \\ &\Leftrightarrow \sum_{i=1}^{n} k_i \langle e_i, X_{init}\rangle = -k_0 \langle e_0, X_{init}\rangle \\ &\Leftrightarrow \frac{\sum_{i=1}^{n} k_i\langle e_i, X_{init}\rangle}{\langle e_0, X_{init}\rangle} = -k_0 \end{aligned}$$

Let $g(c_1, ..., c_n) = -\frac{\sum_{i=1}^{n} c_i.\langle e_i, X_{init}\rangle}{\langle e_0, X_{init}\rangle}$. We have $g(u_i) = -\frac{\langle e_i, X_{init}\rangle}{\langle e_0, X_{init}\rangle}$ We note that g is linear, thus :

$$\begin{aligned}\langle \mathcal{F}, X_{init}\rangle = 0 \quad &\Leftrightarrow \quad g(k_1, ..., k_n)\,\langle e_0, X_{init}\rangle = \sum_{i=1}^{n} k_i\,\langle e_i, X_{init}\rangle \\ &\Rightarrow \quad \bigwedge_{i=1}^{n}(\langle e_i, X\rangle = -\tfrac{\langle e_i, X_{init}\rangle}{\langle e_0, X_{init}\rangle}\,\langle e_0, X\rangle)\end{aligned}$$

by setting $k_i = 1$ and $k_j = 0$ for all $j \neq i$.

Now to prove that this transformation does not make us lose precision, we will construct $\mathcal{F}$ with the n equations.

If $\bigwedge_{i=1}^{n}(\langle e_i, X\rangle = g(u_i)\,\langle e_0, X\rangle)$, then as g is a linear application we have that

$$\sum_{i=1}^{n} k_i\,\langle e_i, X\rangle = g(k_1, ..., k_n)\,\langle e_0, X\rangle$$

We conclude by setting k_0 to $-g(k_1, ..., k_n)$

□

We are now able to use pairs of eigenvectors to express invariants by knowing the initial condition.

5.5.3 Elevation degree.

To complete the technique, we will find a minimal degree for which we are sure to find invariants when there exist at least 2 variables associated to the eigenvalue 1 (not counting $\mathbb{1}$). Let A be a linear transformation matrix. First, let us simplify the context by changing the base of the transformation:

Property 12 *Let A and B two similar transformations, Φ_A and Φ_B the sets of their linear invariants. Φ_A is isomorphic to Φ_B.*

Proof. By property 4, we know that the union of any base of the eigenspaces is sufficient to express all linear invariants of A. We will prove that each eigenspace is isomorphic. If A and B are similar, there exists a P such that $A = P^{-1}BP$. Let λ an eigenvalue of A.

$$\begin{aligned}A - \lambda Id &= P^{-1}.B.P - \lambda.Id \\ &= P^{-1}.B.P - P^{-1}.(P.\lambda Id.P^{-1}).P \\ &= P^{-1}.B.P - P^{-1}.(\lambda Id).P \\ A - \lambda Id &= P^{-1}.(B - \lambda Id).P\end{aligned}$$

Let $\varphi \in ker(A)$ a non null vector such that $(A - \lambda Id)\varphi = 0$. In other words, φ is an eigenvector of A associated to λ. As P is invertible, its kernel is reduced to the null vector (as well as the kernel of P^{-1}). By hypothesis, we have that $(P^{-1}.(B - \lambda Id).P)\varphi = 0$, therefore $P.\varphi \in \ker(B - \lambda Id)$.

Elements of Φ_A are in bijection with elements of Φ_B through P. □

Corollary. Working on the original base or on any base is equivalent when searching for invariants. More specifically, let us focus on the Jordan normal base of A. A is similar to J (ie. $\exists P.A = P^{-1}JP$), with

$$J = \begin{pmatrix} J_1 & 0 & ... & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & ... & 0 & J_k \end{pmatrix}, \text{ and } J_k = \begin{pmatrix} \lambda_k & 1 & ... & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & ... & 0 & \lambda_k \end{pmatrix}$$

Assume there exists i such that $\lambda_i = 1$. The associated Jordan block, working on variables $x^{J_i} = (x_1^{J_i}, ..., x_j^{J_i})$ is then

$$J_i = \begin{pmatrix} 1 & 1 & ... & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & ... & 0 & 1 \end{pmatrix}$$

It is easy to compute the n^{th} power of $J_i = Id + N$, where N is nilpotent.

$$J_i^n = \begin{pmatrix} 1 & n & ... & P_j(n) \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1 & n \\ 0 & ... & 0 & 1 \end{pmatrix} \tag{5.4}$$

where $P_j(n)$ is a polynomial of degree j. Let's assume without loss of generality that $x_j^{J_i} = \mathbb{1}$. Then, $x_{j-1}^{J_i}$ acts like a loop counter : it is incremented by 1 after every step of the loop. Hence for every step, $n = x_{j-1}^{J_i} - x_{init}$, where x_{init} is the initial value of $x_{j-1}^{J_i}$. We can thus replace it in J_i^n : every variable is directly in relation with $x_{j-1}^{J_i}$, which implies that every variable of x^{J_i} are polynomials of other variables, thus admit equivalent invariants.

To sum up, every variable associated to the eigenvalue 1 in a Jordan block is directly in relation with the loop counter (cf equation (5.4)). This counter appears in the loop when there exists a Jordan block of size 2. Therefore:

Property 13 *Let A a linear transformation, J its Jordan normal form. Let J_k denote the k^{th} Jordan block associated to the eigenvalue 1, J_{ki} its i^{th} line (starting from the bottom). Let x_{ki} the i^{th} variable of the Jordan block J_k (also starting from the bottom).*

- *For every k, k', c, $x_{ki} - x_{k'i} = c$ is a semi-invariant of J*
- *If there exists a Jordan block J_k of size 3 or more, then there exists a semi-invariant of degree $m + 1$ where m is the size of the largest Jordan block.*

Proof.

- We proceed by induction on the position on the Jordan block. From equation (5.4), we know that $x_{k1} = \mathbb{1}$ for all k. Therefore for all k, $x_{k1} = x_{11}$ is a semi-invariant of J.
- Let us take a Jordan block J_k. After n iterations, the variable $\{x_k\}_i$ will be associated to a polynomial of degree at most $i + 1$ (cf equation (5.4)). Indeed, the i^{th} line contains a polynomial of degree at most i in n (the loop counter), and each coefficient is multiplied by a variable the matrix is applied to a variable vector, which returns a polynomial of degree

```
(x,y) = (non_det(-1,1),non_det(-1,1));
while(*) do
  (x,y) = (0.68 * (x-y), 0.68 * (x+y));
done
```

FIGURE 5.3: Simple affine loop

$i+1$. If the Jordan block has a size of 3 or more, this loop counter automatically appears as the variable $\{x_k\}_2$. Therefore, it is possible to replace every occurence of the loop counters in the polynomial relations by $\{x_k\}_2$.

□

Variables in the Jordan normal form of a linear transformation represent linear combinations of variables in the original form of the matrix. Therefore, this property proves the existence of polynomial invariants up to a calculable bound, i.e. the maximal size of the Jordan blocks. In the base of Jordan, this property also explicits invariant relations between and beside blocks.

5.6 Inequalities

5.6.1 Convergence and divergence

Being an inductive invariant requires for a formula F to be true after an iteration of the loop under the hypothesis that F holds before the iteration. The left eigenspace of a linear transformation (i.e. the eigenspace of the dual transformation) is exactly its set of exact invariants (cf Definition 21). So far, we only studied equality relations between variables. Let us introduce now the concept of *convergent* and *divergent* invariants:

Definition 24 *Convergence*
$\varphi \in \mathbb{K}^n$ *is a* convergent inductive invariant *for a linear mapping* f *iff*

$$\forall X \in \mathbb{K}^n, \forall k \in \mathbb{K}, |\langle \varphi, X \rangle| \leqslant k \Rightarrow |\langle \varphi, f(X) \rangle| \leqslant k \tag{5.5}$$

Definition 25 *Divergence*
$\varphi \in \mathbb{K}^n$ *is a* divergent inductive invariant *for a linear mapping* f *iff*

$$\forall X \in \mathbb{K}^n, \forall k \in \mathbb{K}| \langle \varphi, X \rangle| \geqslant k \Rightarrow |\langle \varphi, f(X) \rangle| \geqslant k \tag{5.6}$$

5.6.2 Convergent invariants and eigenvectors

By linear algebra

$$|\langle \varphi, X \rangle| \leqslant k \Rightarrow |\langle f^*(\varphi), X \rangle| \leqslant k \tag{5.7}$$

is strictly equivalent to the Definition 24 of convergent semi-invariants. The set of X such that $|\langle\varphi, X\rangle| \leqslant k$ represent what we call a *domain described by* φ, i.e. a polynomial relation. The previous constraint specify that the domain described by φ is stable by f.

The loop in figure 5.3 admits the invariant $x^2+y^2 \leqslant 2$, a domain described by $\varphi = (0,0,0,1,0,1)^t$ in the base $(\mathbb{1}, x, xy, x_2, y, y_2)$[2] where x_2 represents x^2, xy represents $x * y$ and y_2 represents y^2. As φ is a left-eigenvector of f, it is an exact semi-invariant of the loop. Therefore, it generates a vectorial space of exact semi-invariants $I = \{k.(x^2+y^2) = 0\ |, k \in \mathbb{K}\}$, which is a very poor result as x^2+y^2 is constant only if it starts at 0 (otherwise $k = 0$ and we don't know anything about x^2+y^2). Let us focus now on the eigenvalue associated to φ on f^*, which is 0.9248. We can replace $|\langle f^*(\varphi), X\rangle|$ by $|\lambda|.|\langle\varphi, X\rangle|$ in (5.7), which returns :

$$|\langle\varphi, X\rangle| \leqslant k \Rightarrow |\lambda|.|\langle\varphi, X\rangle| \leqslant k$$

As $|\lambda| < 1$, the vector φ satisfies the equation, thus φ is a convergent semi-invariant. Knowing the maximal initial value of x^2+y^2 allows to determine the value of k, which is 2. More generally, we have :

Property 14 *φ is a convergent semi-invariant* $\Leftrightarrow \exists\lambda, |\lambda| \leqslant 1, f^*(\varphi) = \lambda.\varphi$

Proof. If $|\lambda| \leqslant 1$, then φ is a convergent semi-invariant (see introduction of section 5.6). We will prove the following Lemma:

Lemma 8 $(\forall k, |\langle\varphi, X\rangle| \leqslant k \Rightarrow |\langle\varphi, f(X)\rangle| \leqslant k) \Rightarrow$ *φ is a left-eigenvector of f.*

Proof. With $k = 0$, we end up with the exact semi-invariant of Definition 21, whose solutions are eigenvectors of f^* by Theorem 4. □

As the exact semi-invariants set of f is the union of the eigenspaces of f^*, we can deduce that this set is a superset of all the relations satisfying (5.5). Moreover by Lemma 8, we have

$$(|\langle\varphi, X\rangle| \leqslant k \Rightarrow |\langle\varphi, f(X)\rangle| \leqslant k) \Rightarrow (|<\varphi, X>| \leqslant k \Rightarrow |\lambda|.|<\varphi, X>| \leqslant k)$$

For $k = |<\varphi, X>|$ it is true if and only if $|\lambda| \leqslant 1$. □

Divergent invariants and eigenvectors The same reasoning applies for the generation of divergent invariants. For example, an eigenvalue λ such that $|\lambda| > 1$ associated to a semi-invariant φ implies that $|\langle\varphi, X\rangle| \geqslant k$ is an inductive invariant. Thus, we also have

Property 15 $\exists\lambda, |\lambda| \geqslant 1, f^*(\varphi) = \lambda.\varphi \Rightarrow$ *φ is a divergent semi-invariant*

[2]The representation of monomials of variables in a linear transformation refers to the elevation process introduced in Chapter 3, Section 3.1.

```
while (*) do
  N = non_det(-0.1,0.1);
  (x,y) = (0.68 * (x-y) + N, \
    0.68(x+y) + N);
done
```

FIGURE 5.4: Non deterministic variant of the Figure 5.3

Proof. If there exists λ such that $f^*(\varphi) = \lambda.\varphi$, then we have that

$$| < \varphi, X > | \geqslant k \Rightarrow | < \varphi, f(X) > | \geqslant k$$

is equivalent to

$$| < \varphi, X > | \geqslant k \Rightarrow |\lambda|.| < \varphi, X > | \geqslant k$$

If we also have that $|\lambda| > 1$, then the previous equation is true. □

Note that this is only an implication this time. For example, the transformation $f(x, \mathbb{1}) = (x + \mathbb{1}, \mathbb{1})$ admits $x \geqslant x_{init}$ as a divergent invariant but the only left eigenvector of f is $(0, 1)$, which correspond to the invariant "$\mathbb{1}$ *is constant*". Moreover, not all invariants of the form $P(X) \leqslant k$ are generated : the loop with the only assignment $x = x - 1$ admits the (non-convergent) invariant $x \leqslant x_{init}$. This invariant does not enter the scope of our setting as $|x| \leqslant x_{init}$ is false for $2x_{init} + 1$ iterations of $x = x - 1$.

5.7 Non determinism

5.7.1 Non deterministic transformations

Some programs depend on inputs given all along their execution, for example linear filters. More generally, an important part of program analysis consists in studying non-deterministic assignments. As an example let us consider the program in figure 5.4, a slightly modified version of the program in figure 5.3. Our previous reasoning is not applicable now because, due to the non-determinism of N, the loop is no longer a linear mapping.

Idea. Intuitively, we will represent this loop by a matrix parametrized by N. For that purpose we use the concept of abstract mapping introduced in [JSS14].

Definition 26 *An abstract linear mapping $f : \mathbb{K}^q \mapsto \mathcal{M}_n(\mathbb{K})$ is a mapping associating a vector $N \in \mathbb{K}^q$ to a matrix. We call f^* the dual mapping of f (i.e. the mapping such that $f^*(N) = (f(N))^t$). The expression of the parametrized matrix with respect to an abstract linear mapping will be called the* abstract matrix.

In our setting, the parameters are the non-deterministic values. For example, the previous loop can be represented by the abstract matrix M_N :

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ N & 0.68 & 0 & 0 & -0.68 & 0 \\ N^2 & 1.36N & 0 & 0.462 & 0 & -0.462 \\ N^2 & 1.36N & 0.925 & 0.462 & -1.36N & 0.462 \\ N & 0.68 & 0 & 0 & 0.68 & 0 \\ N^2 & 1.36N & 0.925 & 0.462 & 1.36N & 0.462 \end{pmatrix}$$

Remark. Similarly to deterministic solvable mappings defined in Chapter 3, non deterministic solvable mappings can be linearized to an abstract matrix. By considering non deterministic parameters as constants, the problem is reduced to the linearization of deterministic solvable mappings.

5.7.2 Generation of a candidate invariant

We have shown in section 5.6 that M_0 admits the invariant $e_0 = (0, 0, 0, 1, 0, 1)$ associated to the eigenvalue $\lambda_0 = 0.9248$. By decomposing M_N as the sum of M_0 and $(M_N - M_0)$, we also have $e_0.M_N = e_0.M_0 + e_0.(M_N - M_0) = \lambda_0.e_0 + \delta_0^N$, where $\delta_0^N = e_0.(M_N - M_0) = (2N^2, 2.72N, 0, 0, 0, 0)$. As the eigenvalue λ_0 is smaller than 1, we are looking for relations φ such that $\forall X, |\langle \varphi, X \rangle| \leqslant k \Rightarrow |\langle M_N^T.\varphi, X \rangle| \leqslant k$. We will call e_0 a *candidate invariant* for M_N. For e_0 to be a proper invariant for this transformation, the following property must hold:

$$\forall X, |\langle e_0, X \rangle| \leqslant k \Rightarrow |\lambda_0 \langle e_0, X \rangle + \langle \delta_0^N, X \rangle| \leqslant k \tag{5.8}$$

Intuitively, multiplying $\langle e_0, X \rangle$ by λ_0 reduces its norm strictly under k. We need to make sure that adding $\langle \delta_0^N, X \rangle$ does not contradict the induction criterion by increasing the result over k. The variables of the program depend on k, as does $\langle \delta_0^N, X \rangle$. If it increases faster than $|\lambda_0 \langle e_0, X \rangle|$ when k is increased, then no value of k will make the candidate invariant inductive. In particular, if $\langle e_0, X \rangle$ is a polynomial P of degree d, we need to be able to give an upper bound to $\langle \delta_0^N, X \rangle$ knowing that $|P(X)| < k$. If the degree of $\langle \delta_0^N, X \rangle$ is strictly smaller than d, then it will grow asymptotically slower than $|P(X)|$, thus for a big enough k the induction criterion will be respected.

Property 16

$$\big(\forall X, |\langle e_0, X \rangle| \leqslant k \Rightarrow |\langle \delta_0^N, X \rangle| \leqslant (1 - |\lambda_0|).k\big) \Rightarrow \tag{5.9}$$

$$|\langle e_0, X \rangle| \leqslant k \text{ is an invariant of the loop.}$$

Proof. We work with the hypothesis $\forall X, |\langle e_0, X \rangle| \leqslant k$.

$$
\begin{aligned}
|\left\langle \delta_0^N, X \right\rangle| \leqslant (1 - |\lambda_0|)k \quad &\Rightarrow \quad |\left\langle \delta_0^N, X \right\rangle| + |\lambda_0.k| \leqslant k \\
&\Rightarrow \quad |\left\langle \delta_0^N, X \right\rangle| + |\lambda_0.\left\langle e_0, X \right\rangle| \leqslant k \\
&\Rightarrow \quad |\left\langle \delta_0^N, X \right\rangle + \lambda_0.\left\langle e_0, X \right\rangle| \leqslant k
\end{aligned}
$$

□

In our example, $\left\langle \delta_0^N, X \right\rangle = 2.72 * N * x + 2 * N^2$. The polynomial x is of degree 1 while $< e_0, X >= x^2 + y^2$ is of degree 2. We need to find a k such that

$$-0.0752 * k \leqslant 2.72 * N * x + 2 * N^2 \leqslant 0.0752 * k \tag{5.10}$$

5.7.3 Optimizing expressions

We will now maximize and minimize $2.72 * N * x + 2 * N^2$, knowing that $x^2 + y^2 \leqslant k$ and $-0.1 \leqslant N \leqslant 0.1$. Solving this problem is very close to solving a constrained polynomial optimization (CPO) problem [Ber14]. CPO techniques provide ways to find values minimizing and maximizing expressions under a set of inequalities constraints. The main issue is related to the parameter k that must be known in order to use CPO directly. This Chapter will not investigate how CPO works in detail, but how we can reduce the problem of finding an optimal k to the CPO problem.

Assuming we have a function *min* computing the minimum, if it exists, of an expression under polynomial constraints, the algorithm in Figure 5.5 finds a value of k and refines it to get as small as possible. The idea is to find k by dichotomy.

- If k doesn't satisfy the constraints, we try a bigger one.
- If we find a k satisfying the two conditions, then it is a potential candidate. We can still try to refine it by searching for a smaller k.

We can improve this algorithm by guessing an upper value of k instead of taking an arbitrary maximal value *MAX_INT*. For our example, we started at $k = 50$ and found that $k = 14.9$ respects all the constraints.

- $x^2 + y^2 \leqslant 14.9 \Rightarrow |x| \leqslant 3.9$
- $|N| \leqslant 0.1$
- $|2.72 * x * N + 2 * N^2| \leqslant 1.08$, and $k * (1 - |\lambda|) = 1.12$.

5.7.4 Convergence

Note however that the existence of a k satisfying (5.10) is not guaranteed. For example, the set $S = \{(x, y, N) | x^2 + y^2 \leqslant k \wedge -0.1 \leqslant N \leqslant 0.1\}$ is a compact set for any value of k, which means that x, y and N have maximum and minimum values. This implies the existence of a lower and an upper

Data:
λ : float
Q : objective function
P : polynomial constraint
non_det_c : non deterministic constraints
N : int
Result: k such that $\forall X, P(X) \leqslant k \Rightarrow f(X) \leqslant (1 - |\lambda|).k$
low_k = 0;
up_k = MAX_INT;
k = MAX_INT / 2;
i = 0;
while *i<N* $\wedge$ *up_k* $\neq$ *MAX_INT* **do**
  i = i+1;
  Pk = function (x $\rightarrow$ P(x) + k);
  min = min(Q,[Pk] $\cup$ non_det_c);
  max = min(-1*Q,[Pk] $\cup$ non_det_c);
  if *min > (-1+|λ|) * k and max < (1-|λ|)*k* **then**
    up_k = k;
  else
    low_k = k;
  end
  /* Check the overflow for the next statement ! */
  k = (low_k + up_k) / 2;
end

FIGURE 5.5: Dichotomy search of a k satisfying the condition of Property 16

bound for every expression composed with x, y and N, but the value of those expressions may be always higher than k such as for $x^2 + y^2 + 1$ bounded by $k + 1$.

Property 17 *Let P and Q two polynomials and $M > 0 \in \mathbb{R}$.*
If $\lim_{\|X\| \to +\infty} |\frac{Q(X)}{P(X)}| < M$, then there exists $k \in \mathbb{R}^+$ such that for all $k' \geqslant k$

$$|P(X)| \leqslant k' \Rightarrow |Q(X)| \leqslant M.k'$$

Proof. If $\lim_{\|X\| \to +\infty} |\frac{Q(X)}{P(X)}| < M$, then there exists X' such that for all X with $\|X'\| \leqslant \|X\|$, we have $|\frac{Q(X)}{P(X)}| \leqslant M$ We will first prove this property for $\|X\| \leqslant \|X'\|$, then for $\|X\| \geqslant \|X\|$.

- For X such that $\|X\| \leqslant \|X'\|$, both P and Q are bounded. Therefore, there exists a positive k such that $|P(X)| \leqslant k$ and $|Q(X)| \leqslant M.k$.
- Now for X such that $\|X\| \leqslant \|X'\|$, let us assume that $|P(X)| \leqslant k$. Then we have that for all k

$$|\frac{Q(X)}{P(X)}| \leqslant M \Rightarrow \frac{|Q(X)|}{k} \leqslant M$$

□

By taking $M = (1 - |\lambda_0|)$, this theorem gives us a sufficient condition to guarantee the convergence of the algorithm in figure 5.5.

Corollary. *If the objective has a lower degree in the deterministic variables than the candidate invariant, then the algorithm converges. If it has the same degree, then it depends on the main coefficients.*

As we are dealing with two polynomials P and Q, then if P (the candidate invariant) has a higher degree than Q (the objective function) in all its variables, the limit of $\frac{Q(X)}{P(X)}$ will be 0, which is enough to ensure the convergence of the method. If we come back to the objective function for the loop of figure 5.3, $Q(X) = 2.72.x.N + 2.N^2$ is a polynomial of degree 1 in x and 0 in y, thus $\lim_{\|X\| \to +\infty} |\frac{Q(X,N)}{P(X)}| = 0$ and we can be sure that the optimization will converge.

On the other hand, if we have $X = (x, y)$, $P(X) = x^2 + y^2$ and $Q(X, N) = 10.N(x^2+y^2+1)$, with $|N| \leqslant 0.1$, the optimization procedure may not produce a result by theorem 17 because $\lim_{\|X\| \to +\infty} |\frac{Q(X,N)}{P(X)}| = 10N$ is higher than $1 - |\lambda|$ for $N = 0.1$.

5.7.5 Initial state

The knowledge of the initial state is not one of our hypotheses yet, but the previous theorem provides the necessary information we need to treat the case where the initial state is strictly higher than the minimal k we found. The previous theorem tells us that there exists a k such that for all $k' \geqslant k$,

k' is a solution to the optimization problem. Our optimization algorithm is searching for a value of k for which the set is inductive, though, and this solution may be only local : there may be a $k' > k$ which is not a solution of the optimization procedure. If the value of $P(X_{init})$ is strictly higher than k, there are two possibilities :

- it satisfies the objective (5.10), optimization is then not necessary as $k = P(X_{init})$ is correct, and we directly have a solution.
- it doesn't satisfy the objective, we have to find a $k > P(X_{init})$ satisfying it.

In both cases, we can enhance the optimization algorithm by first testing the objective (5.10) with $k = P(X_{init})$. If it does not respect the objective, then starting the dichotomy with $low\_k = P(X_{init})$ will return a solution (guaranteed by the property 17) strictly higher than $P(X_{init})$.

Chapter 6

How precise can invariants be ?

Contents

Requirements: Linear algebra (Section 2.1)

Generating invariants is not a goal in itself, but a mean to prove the correctness of a program. Chapter 5 gave a new characterization of linear loop invariants, and a simple algorithm to generate them. But these invariants are generated out of any proof context, and they are sometimes insufficient to prove the correctness of a program. Though Theorem 4 states that eigenvectors are *exactly* invariants of linear loops, it is not clear what is achievable or not with them. We will show in this Chapter a possible application of eigenvectors as invariants for the *Kannan-Lipton Orbit problem* [KL80; KL86]. This chapter is based on work presented on [Oli+18].

6.1 The Orbit Problem

6.1.1 The Kannan-Lipton Orbit problem

The *Kannan-Lipton Orbit problem* can be stated as follows :

Given a square matrix $A \in \mathcal{M}_d(\mathbb{Q})$ of size d and
two vectors $X, Y \in \mathbb{Q}^d$, determine if there exists n such that $A^n X = Y$.

This problem is decidable in polynomial time [KL86]. In case an instance of the problem has no solution (in other words, Y is not reachable from X), [Fij+17] studies the existence of non-reachability semialgebraic certificates for a given instance of the Orbit Problem where Y is not reachable. Semialgebraic certificates are sets described by conjunctions and disjunctions of polynomial inequalities with integer coefficients that include the reachable set of states but not the objective Y. Those certificates allow to quickly prove the non-reachability of the given vector Y and all vectors outside of the certificate.

[Fij+17] concludes on the existence of such certificates under simple hypotheses on the eigenvalue decomposition of A. To sum up, if there exist eigenvalues with modulus different than 1 or if the transformation is not diagonalizable, there exists certificates of non reachability that involves inequalities of polynomials. Otherwise, it depends if the objective belong to a certain set described by equalities of polynomials.

These hypotheses are surprisingly similar to the hypotheses of *PILA* as, when $|\lambda| \neq 1$, left-eigenvectors represent polynomial inequality invariants while [Fij+17] uses certificates defined by *polynomial inequalities*. Eigenvectors were sometimes unable to infer invariants, especially when the studied matrix was non-diagonalizable with all its eigenvalues λ such that $|\lambda| = 1$, while [Fij+17] was able to infer certificates.

Interests of eigenvectors. In terms of complexity, eigenvectors and the argument of [Fij+17] using the Jordan Normal form of the matrix are equivalent, as the Jordan Normal form of a matrix can be calculated in polynomial time given eigenvectors and generalized eigenvectors. It is however necessary to compute *all eigenvectors and generalized eigenvectors* of a transformation to get the Jordan Normal form, which slows the analysis (this is especially true for generalized eigenvector that are harder to calculate).

6.1.2 Eigenvectors as certificates

In this Chapter, we investigate the connections between the construction of certificates for the Orbit Problem and the invariants characterization of Chapter 5. We show that for an instance of the Orbit Problem for the transformation A of dimension n, the problem of generating a certificate can be reduced to the search of eigenvectors. Particularly,

- in the first hypothesis, there exists a linear transformation of dimension $O(n^2)$ (resp. $O(2^n)$) computing an equivalent image of A such that its eigenvectors can be used as real certificates (resp. semialgebraic certificates) for the non reachability of the given instance;
- in the second hypothesis, there exists a linear transformation of dimension $O(n^2)$ (resp. $O(2^n)$) computing an equivalent image of A such that its *generalized* eigenvectors can be used as real certificates (resp. semialgebraic certificates) for the non reachability of the given instance;

- in a more general case, a semialgebraic certificate for the Orbit Problem in $\mathbb{Z}$ always exists.

Remark. It is worth noting that there exists no proof about the decidability of the existence of linear certificates directly on the transformation A.

6.2 Certificate sets of the rational Orbit Problem

This chapter focuses on $\mathbb{A} \subset \mathbb{C}$, the field of algebraic numbers. Elements of $\mathbb{A}$ are roots of polynomials with integer coefficients. Indeed, the linear transformations we consider are in $\mathbb{Q}^d \to \mathbb{Q}^d$, thus their eigenvalues (as roots of the characteristic polynomial) are in $\mathbb{A}$. Let $f : \mathbb{Q}^d \to \mathbb{Q}^d$ be a linear transformation. We refer to the Orbit Problem of A_f with an initial state $X \in \mathbb{Q}^d$ and an objective state $Y \in \mathbb{Q}^d$ as $\mathcal{O}(A, X, Y)$. In other words, $\mathcal{O}(A, X, Y) = (\exists n \in \mathbb{N}. Y = A^n X)$. As we are studying non-reachability, every instance of the problem is assumed to be false unless stated otherwise.

Definition 27 *A* non-reachability certificate *or just* certificate *is a pair* $(N, P) \in \mathbb{N} \times \mathcal{P}(\mathbb{Q}^d)$ *of an instance* $\mathcal{O}(A, X, Y)$ *such that :*

- $\forall n \in \mathbb{N}, n < N \Rightarrow A^n X \neq Y$
- $\forall n \in \mathbb{N}, n \geqslant N \Rightarrow A^n X \in P$
- $Y \notin P$

N *is called the* certificate index *and* P *the* certificate set.

When the certificate set is described by a combination of linear (resp. polynomial) relations between variables, the certificate is called linear (resp. polynomial). Irrational, semialgebraic and rational certificates are linear or polynomial certificates whose coefficients are respectively irrationals, algebraic integers or rationals.

Semi-algebraic certificates, are always equivalent to rational certificates. Indeed, every coefficient $\varphi_i \in \mathbb{A}$ is nullified by a polynomial Q with integer coefficients. It is then possible to replace φ_i by a free variable that is constrained to be a root of Q. For example, $P = \{x | \sqrt{2}x \leqslant 2\} = \{x | \exists y. y^2 = 2 \wedge y \geqslant 0 \wedge yx \leqslant 2\}$.

Remarks. This definition of certificates is slightly different than the notion of certificates of [Fij+17] as it does not require an inductivity criterion. We have choosen this notation so as to simplify the notations.

The certificate sets we generate are *future invariants* of the transformation, in the sense that $f^n(X)$ eventually reaches the set for some n and always remains in it, whereas Y is outside the invariant. Different choices of X and Y may delay the number of iterations needed to reach it. The certificate index solves this issue by expressing the number of iterations necessary for $f^n(X)$

to reach the certificate set. This information is crucial for the practical use of certificates, as a solver can use it to shorten its analysis.

The existence of such a pair implies the non reachability of Y as A^nX is either different from Y or belongs to a set to which Y does not. For example, if Y does not belong to the reachable set of states $R = \{A^nX \mid n \geqslant 0\}$, the pair $(0, R)$ is a certificate. However, typically, R can not be described in a *non-enumerative* way. We are interested in *simple* certificates, i.e. where proving that the objective Y does not belong to the reachable set of states is straightforward. That means that membership in P should be easy to solve. For example, let $R' = \{(v_1, ..., v_n) | v_1 + v_2 \geqslant 0\}$ and assume $R \subset R'$. Testing whether Y is in R' or not is easy as this set is described by a linear combination of variables. If $Y \notin R'$, then R' is generally a *better* (simpler) certificate set than R. On the other hand, finding a good certificate index may be harder.

Generation of certificates. The decidability of the existence or the non-existence of semialgebraic certificates for the Orbit Problem for rational linear transformations is proven in [Fij+17]. It classifies four categories of rational linear transformations $f : \mathbb{Q}^d \to \mathbb{Q}^d$:

- f admits null eigenvalues;
- f has at least an eigenvalue of modulus strictly greater or less than 1;
- f has all its eigenvalues of modulus 1, but it is not diagonalisable;
- f has all its eigenvalue of modulus 1 and is diagonalisable.

In the second and third case, linear transformations always admit a non reachability certificate if the Orbit problem has no solution. The intuition behind this result is to consider the Jordan normal form f_J of the transformation f. Let V be a vector of variables and V_J the vector of variables in the base of J. In this form, there exists a variable v_J (representing a linear combination of variables of V) such that $f_J(V_J)_{|v_J} = \lambda v_J$. Applied k times, the new value of v_J is $\lambda^k v_J$, which diverges towards infinity or converges towards 0 when $|\lambda| \neq 1$. Checking if a value y is reachable or not can then be done by checking if there exists $k \in \mathbb{N}$ such that $\lambda^k v_J = y$. We are now left to compute those certificates.

Case 1: there exist null eigenvalues

This particular case leads to degenerate instances of the orbit problem. When a linear transformation admits a null eigenvalue, there exists a linear combination of variables that is always null. In other words, there exists a variable v that can be expressed as a linear combination of the other variables. Therefore, this variable doesn't provide any useful information on the transformation other than an easily checkable constraint on v. If the linear constraint is satisfied, we get rid of this case by using Lemma 4 of [Fij+17], stating the following:

The problem of generating non-reachability certificates for an orbit instance $\mathcal{O}(A, X, Y)$ can be reduced to the problem of generating reachability certificates for an orbit instance $\mathcal{O}(A', X', Y')$ where A' is invertible.

6.2.1 Case 2: there exist eigenvalues λ and $|\lambda| \neq 1$.

Real eigenvalues.

The key of the following property lies in **[pilat_nd_long]**, stating that λ-left eigenvectors φ of a linear transformation f are its invariants. More precisely, we can see that if φ is a left-eigenvector of a linear transformation A, then by definition the following holds:

$$\forall v \in \mathbb{K}^d, \langle \varphi, Av \rangle = \lambda \langle \varphi, v \rangle \tag{6.1}$$

If $|\lambda| > 1$ (resp. $|\lambda| < 1$), then the sequence $(| \langle \varphi, A^n v \rangle |)$ (for $n \in \mathbb{N}$) is *strictly increasing* (resp. *strictly decreasing*),

Property 18 *Let $A \in \mathcal{M}_d(\mathbb{Q})$ a linear transformation and $\mathcal{O}(A, X, Y)$ an instance of the Orbit problem with no solution. Searching for a non-reachability certificate of an instance of the Orbit problem when A admits real eigenvalues λ such that $|\lambda| \neq 0$ and $|\lambda| \neq 1$ can be reduced to computing the eigenvector decomposition of A.*

More precisely, if there exists φ a λ-left-eigenvector of A with $|\lambda| \neq 0$ and $|\lambda| \neq 1$, then there necessarily exists N such that the couple (N, P) defined as follows is a non-reachability certificate of $\mathcal{O}(A, X, Y)$.

1. *If $| \langle \varphi, X \rangle | \neq 0$ and $| \langle \varphi, Y \rangle | = 0$, then $N = 0$ and $P = \{v : \langle \varphi, v \rangle \neq 0\}$*
2. *If $| \langle \varphi, X \rangle | = 0$ and $| \langle \varphi, Y \rangle | \neq 0$, then $N = 0$ and $P = \{v : \langle \varphi, v \rangle = 0\}$.*
3. *If $| \langle \varphi, X \rangle | \neq 0$ and $| \langle \varphi, Y \rangle | \neq 0$, $N = max(1, \lfloor \frac{ln(|\langle \varphi, Y \rangle|) - ln(|\langle \varphi, X \rangle|)}{ln(|\lambda|)} \rfloor + 1)$ and*
   - *If $|\lambda| > 1$, then $P = \{v : | \langle \varphi, v \rangle | \geqslant |\lambda . \langle \varphi, Y \rangle |\}$.*
   - *If $|\lambda| < 1$, then $P = \{v : | \langle \varphi, v \rangle | \leqslant |\lambda . \langle \varphi, Y \rangle |\}$.*
4. *Otherwise, if $d > 1$ there exist a transformation $B \in \mathcal{M}_{d-1}(\mathbb{Q})$ such that the problem of finding a certificate for $\mathcal{O}(A, X, Y)$ can be reduced to the problem of finding a certificate for $\mathcal{O}(B, X, Y)$.*

   If $d = 1$, then $\mathcal{O}(A, X, Y)$ has a solution.

The certificate is semi-linear iff $\lambda \in \mathbb{Q}$.

Proof. Let φ be a left-eigenvector of A associated to the eigenvalue λ. We know that for all v, $\langle \varphi, v \rangle = k \Rightarrow \langle \varphi, Av \rangle = \lambda.k$. Let $U_n = | \langle \varphi, A^n X \rangle |$ be the module of the n-th reachable state from X. If $|\lambda| < 1$ (resp. $|\lambda| > 1$), then (U_n) is strictly decreasing (resp. strictly increasing).

1. Let $k_v = |\langle \varphi, v \rangle|$. If $k_X \neq 0$ and $k_Y = 0$, then the sequence (U_n) never reaches k_Y, as for all n, $U_n \neq 0$. In other words, $|U_n| > 0$ for all $n \in \mathbb{N}$. Then it is clear that $P = \{X : |\langle \varphi, X \rangle| \neq 0\}$ is a valid certificate set of index $N = 0$.

2. Similarly, if $k_X = 0$ and $k_Y \neq 0$, then $P = \{X : |\langle \varphi, X \rangle| = 0\}$ and $N = 0$.

3. Assume now that $k_X \neq 0$ and $k_Y \neq 0$. If $k_X < k_Y$ and $|\lambda| < 1$ (respectively $k_X > k_Y$ and $|\lambda| > 1$), then $(1, \{v : |\langle \varphi, v \rangle| \leqslant |\lambda|.k_Y\})$ is a valid certificate set (respectively $(1, \{v : |\langle \varphi, v \rangle| \geqslant |\lambda|.k_Y\})$). Otherwise, let us assume $|\lambda| < 1$ and $k_X \geq k_Y$. U_n is strictly decreasing, so there exist a N such that $U_N \geqslant k_Y$ and $U_{N+1} < k_Y$. This implies that Y can only be reachable after a finite number of iterations N. We also have that $U_{N+1} \geqslant |\lambda|.k_Y$ and $U_{N+2} < |\lambda|.k_Y$. If for all $n < N + 1$, $Y \neq A^n X$, we can define $P = \{v : |\langle \varphi, v \rangle| < |\lambda|.k_Y\}$, and obtain $Y \notin P$ and $\{A^{N+1+n}X | n \in \mathbb{N}\} \subset P$. Therefore, the couple $(N + 1, P)$ is a non-reachability certificate of $\mathcal{O}(A, X, Y)$. A similar proof for $|\lambda| > 1$ is valid as the sequence U_n is now strictly increasing and the couple $(N, \{|\langle \varphi, X \rangle| \geqslant |\lambda|.k_Y\})$ is the corresponding certificate.

   We will now study the exact value of N. If Y is reachable, then there exists a unique value of N such that $|\lambda|^N |\langle \varphi, X \rangle| = k_Y$. This value is precisely $\frac{ln(|\langle \varphi, Y \rangle|) - ln(|\langle \varphi, X \rangle|)}{ln(|\lambda|)}$. If for every value of $n \leqslant N$, Y is not reached and as Y noes not belong to the certificate set P, the couple $(max(0, \lfloor N \rfloor), P)$ is a non-reachability certificate.

4. Assume $k_X = k_Y = 0$. In this case for every n, $\langle \varphi, A^n X \rangle = 0$, thus the linear combination of variables $\varphi.X$ is always equal to 0. There exists a base $\mathcal{B}$ of the transformation in which there exists a variable v which remains null for every iteration of the transformation. In other words, there exist A', Q such that $A' = Q.A.Q^{-1}$.

   Assume $d > 1$ and let $B' = A'_{|_{V \setminus v}}$ and $Q' = Q_{|_{V \setminus v}}$ the transformations restricted to all variables but v (by removing both the associated line and column). Finding a certificate for A is reduced to finding a certificate for $B = Q'^{-1} B' Q'$.

   If $d = 1$ and there exist a linear combination φ of X such that $\langle \varphi, X \rangle = 0$, then $X = 0$. Similarly, $Y = 0$.

Concerning the linearity of the certificate, if $\lambda \in \mathbb{Q}$, then every coefficient of φ also belongs to $\mathbb{Q}$. Indeed A has rational coefficients, so does $\varphi A = \lambda.\varphi$. Similarly, if φ has rational coefficients, $\varphi.A = \lambda.\varphi$ also does.

In the case of $k_X \neq 0$ and $k_Y \neq 0$, we also have to get rid of the absolute value around $\langle \varphi, v \rangle$ in the definition of the certificate set. If $|\lambda| > 1$, the certificate set $\{v : (\langle \varphi, v \rangle \geqslant |\lambda \langle \varphi, Y \rangle|) \wedge (\langle \varphi, v \rangle \leqslant -|\langle \varphi, Y \rangle|)\}$ is semilinear. A similar set can be found for $|\lambda| < 1$.

□

Certificate index.

Being able to minimize the number of necessary unrollings to prove the non reachability is useful. In this regard, notice that the certificate index value N of Property **??** is such that for every $n < N$, $\langle \varphi, A^n X \rangle \notin P$. In other words, it is minimal for its associated certificate set.

Example. Consider the Orbit Problem $\mathcal{O}(A, X, Y)$ with

$$\text{subsubsection } A = \begin{pmatrix} 0 & 3 & 0 & 0 \\ -3 & 3 & 1 & 0 \\ 0 & 0 & 2 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

A admits two real eigenvalues $\lambda_1 \approx 0.642$ and $\lambda_2 \approx 2.48$ respectively associated to the left-eigenvectors $\varphi_1 = (-0.522, 0.355, -0.261, 0.73)$ and $\varphi_2 = (0.231, -0.36, -0.749, -0.506)$. This is enough to build two preliminary certificate sets that only depend on Y : $P_1 = \{v. | \langle \varphi_1, v \rangle | \leqslant \lambda_1 . | \langle \varphi_1, Y \rangle |\}$ and $P_2 = \{v. | \langle \varphi_2, v \rangle | \geqslant \lambda_2 . | \langle \varphi_2, Y \rangle |\}$. Those can be used for any initial valuation of X.

Let's now set $X = (1, 1, 1, 1)$ and $Y = (-9, -7, 28, 7)$. We have then

- $\langle \varphi_1, X \rangle = 0.302$ and $\langle \varphi_1, Y \rangle = 0.015$, so $N = 7$.
- $\langle \varphi_2, X \rangle = -1.384$ and $\langle \varphi_2, Y \rangle = -24.073$, so $N = 4$.

We can easily verify that for any $n \leqslant 7$, $A^n X \neq Y$, so the certificates $(7, P_1)$ and $(4, P_2)$ are sufficient to prove the non reachability of Y.

Complex eigenvalues.

The treatment of complex eigenvalues can be reduced to the Case 1 by the *elevation* method described in Chapter 3. Let us recall the idea of elevation. If variables evolves linearly (or affinely) then any monomial of those variables also evolves linearly (or affinely). For example, given $f(x) = x + 1$, then the new value of x^2 after application of f is $(x + 1)^2 = x^2 + 2x + 1$, which is an affine combination of x^2, x and 1. f can be *elevated* to the degree 2 by expressing this new monomial : $f_2(x_2, x) = (x_2 + 2x + 1, x + 1)$. We denote $\Psi_k(A)$ a transformation A elevated to the degree k and, by extension, $\Psi_k(v)$ a vector v elevated to the degree k.

A and $\Psi_d(A)$ represents the same application, except that $\Psi_d(A)$ also calculates monomial values of variables manipulated by A. Hence, certificates of $\mathcal{O}(\Psi_d(A), \Psi_d(X), \Psi_d(Y))$ are also certificates for $\mathcal{O}(A, X, Y)$,

The product of all eigenvalues is the determinant of the transformation, which is by construction a rational. By Property 8, the elevation to the degree n where n is the size of the matrix admits at least one rational eigenvalue. We can deduce from this the following theorem.

Theorem 6 *Let $\mathcal{O}(A, X, Y)$ be an unsatisfiable instance of the Orbit problem with $A \in \mathcal{M}_n(\mathcal{Q})$ admitting at least one eigenvalue $\lambda \in \mathbb{C}$ such that $|\lambda| \neq 0$ and $|\lambda| \neq 1$. Then left eigenvectors of $\Psi_d(A)$ provide :*

- *real linear semialgebraic certificates for* $d = 1$ *(*$\Psi_1(A) = A$*) if there exist real eigenvalues;*
- *real semialgebraic certificates of degree* 2 *for* $d = 2$ *if there exist complex eigenvalues;*
- *at least one rational certificate of degree* n *for* $d = n$ *if* $|det(A)| \neq 1$.

Proof. We treat each case separately:

- The case where A admits real eigenvalues is treated by Property 18;
- If A admits a complex eigenvalue λ, A also admits its conjugate $\bar{\lambda}$ as eigenvalue. By Property 8, $\Psi_2(A)$ admits $\lambda.\bar{\lambda}$ as a real eigenvalue, which is treated by Property 18;
- The product of all eigenvalues of a rational matrix is rational. As such, Ψ_n necessarily admit a rational eigenvalue which implies the existence of an associated rational eigenvector that can be used, according to Property 18, as a certificate.

□

Remark. The image of $A \in \mathcal{M}_d(\mathbb{K})$ is a projection of the image of $\Psi_k(A)$ for any k, and semialgebraic certificates of A are, by extension, semilinear certificates of $\Psi_n(A)$. The size of $\Psi_k(A)$ is $\binom{d+k}{k}$, which is $O(d^2)$ when $k = 2$ and $O(d^d)$ when $d = k$. An eigenvector computation has a polynomial time complexity (slightly better than $O(d^3)$). The two first cases of Theorem 6 are thus computable in polynomial time in the number of variables.

Example. The matrix from the previous example admits two complex eigenvalue $\lambda \approx 1.439 + 2.712i$ and $\bar{\lambda}$. As $\lambda\bar{\lambda} \approx 9.425$, it also admits a polynomial invariant φ. As we know that $\langle \varphi, X \rangle = 0.220$ and $\langle \varphi, Y \rangle = 195.738$, the associated index is 4.

6.2.2 Case 3: all eigenvalues have a modulus equal to 1 and the matrix is not diagonalisable

Real eigenvalues.

This case is trickier as eigenvectors do not give information about the convergence or the divergence of the linear combination of variables they represent. For example, let us study the orbit problem $\mathcal{O}(A, X, Y)$ where A is the matrix associated with the mapping $f(x, \mathbb{1}) = (x + 2 * \mathbb{1}, \mathbb{1})$, $X = (0, 1)$ and $Y = (5, 1)$. x_Y is odd, thus Y is not reachable. f admits only $\varphi = (0, 1)$ as left-eigenvector associated to the eigenvalue $\lambda = 1$, meaning that $\langle (0, 1), (x, \mathbb{1}) \rangle = \langle (0, 1), f(x, \mathbb{1}) \rangle$ for any x. As $\langle (0, 1), (x, \mathbb{1}) \rangle = \mathbb{1}$, we are left with the invariant $\mathbb{1} = 1$. This invariant is clearly insufficient to prove that Y is not reachable.

f thankfully admits a generalized left-eigenvector $\mu = (\frac{1}{2}, 1)$ associated to 1. More precisely, $\mu A = \mu + \varphi$, which implies that $\mu A^n X = (\mu + n\varphi).X$. In other words, we have $\frac{1}{2}x + 1 = \frac{1}{2}x_X + 1 + n$ which simplifies into $\frac{1}{2}x = n$. The couple $(3, \{(x, y) : \exists n > 3, \frac{1}{2}x = n\})$ is a non reachability certificate.

Property 19 *Let A a non-diagonalisable linear transformation and $\{e_i\}_{i<N}$ N linked 1-left eigenvectors[1] (i.e. $e_0 A = e_0$ and for $0 < i < N$, $e_i A = e_i + e_{i-1}$).*

Then for all $i < N$, $\langle e_i A^k, X \rangle = P_i(k, X)$, where $P_i(k, X)$ is a polynomial of degree i in the variable k and 1 in each variable of X.

Proof. Let $\{e_i\}_{i<N}$ a family of N linked 1-left eigenvectors. We can compute $P_i(k, X)$ by induction on i. For $i = 0$, e_0 verifies $e_0 A^k = e_0 = \binom{k}{i} e_{N-i-1}$. Assume now $e_i A^k = P_i(k)$ are vectors of polynomials of degree at most i. Then, we have $e_{i+1}.A^{k+1} = (e_{i+1} + e_i).A^k = e_{i+1}A^k + P_i(k)$ Now, let $U_{n+1} = U_k + P_i(n)$. Then for anu U_0, $U_k = U_0 + \sum_{l=0}^{k} P_i(l)$ is a vector of polynomials of degree at most $i + 1$.

□

As every polynomial eventually diverges, there exists a linear combination of variables of X that diverges. This is enough to certify the non reachability of the Orbit Problem *for non diagonalizable matrices with the eigenvalue* $\lambda = 1$.

Remark. Even if the first eigenvector is enough to represent a non-reachability certificate, every generalized eigenvector also can. By Property 19, the value of the linear combination described by a generalized eigenvector φ evolves polynomially, thus it eventually always decrease or increase (after the highest root of its derivate). That is why for a given objective Y there exist a finite number of n such that $|\varphi Y| \leqslant |\varphi A^n X|$, thus after this n, $\{v : |\varphi v| > |\varphi Y|\}$ is a certificate.

Complex eigenvalues.

If $\lambda \in \mathbb{C}$, we will use the same trick we used for complex eigenvalues of Case 2. As for every complex eigenvalue λ of A, $\bar{\lambda}$ is also an eigenvalue, then $\lambda.\bar{\lambda} = 1$ is an eigenvalue of $\Psi_2(A)$ by Property 8. Thus :

Theorem 7 *Let $\mathcal{O}(A, X, Y)$ be a non satisfiable instance of the Orbit Problem such that for all eigenvalue λ of A, $|\lambda| = 1$ and A is not diagonalisable. Then there exist a family of linked 1-left-eigenvectors $\mathcal{F} = \{e_0, ..., e_n\}$ of $\Psi_2(A)$ such that for all $1 \leqslant i \leqslant n$, $Q_i(n) = \langle e_i, \Psi_2(A)^n \Psi_2(X) \rangle$ is a polynomial and (N, P) is a non reachability certificate with:*

- $N = \lfloor max(\{0\} \cup \{x \in \mathbb{R}.Q_i(x) = \langle e_i, \Psi_2(A^x)\Psi_2(Y)\rangle\}) \rfloor$
- $P = \{v : | \langle e_i, \Psi_2(A)^n \Psi_2(v) \rangle | \geqslant |Q_i(N)|\}$

[1] The existence of such a family with $N > 1$ is guaranteed by the non diagonalisability of A.

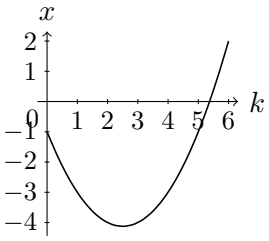


FIGURE 6.1: Graph of the polynomial y = $\frac{1}{2}k^2 - \frac{5}{2}k - 1$

Proof. Let $\mathcal{O}(A, X, Y)$ be an instance of the Orbit Problem. We will reduce the problem to the case where A has positive rational eigenvalues, i.e. $\lambda = 1$ and A admits a family $\mathcal{F}$ of linked left-eigenvectors of size $|\mathcal{F}| > 1$. In this case, by Property 19 we know that there exists a linear combination of variables v following a polynomial evolution described by Q such that $deg(Q) > 0$. As Q eventually diverges, there exists a N such that for all $N' > N$, $|v(A^{N'}X)| > |v(Y)|$. This N is the maximum between 0 and the highest value of x such that $Q(x) = v(Y)$ as, for any higher value of x, $|Q(x)| > |v(Y)|$. Also, the set $\{v.|\langle e_i, \Psi_2(A)^n \Psi_2(v)\rangle \mid \geqslant |Q(N)|\}$ contains all reachable configurations but does not contain Y, thus (N, P) is a valid certificate.

In the general case where $\lambda \in \mathbb{C}$, Property 9 guarantees the existence of generalized eigenvector on $\Psi_2(A)$ if A is not diagonalizable (i.e. also admits generalized eigenvectors)

Example. We consider the Orbit problem $\mathcal{O}(A, X, Y)$ with $A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$, $X = (-2, -1, 1)^t$ and $Y = (2, 6, 1)^t$. A admits as 1-generalized-left-eigenvectors: $\{e_0 = (0, 0, 1); e_1 = (0, 1, 0); e_2 = (1, 0, 0)\}$. By the previous property, we know that $e_2 A^k = e_2 + k.e_1 + \frac{k(k-1)}{2}.e_0$, thus

$$\begin{aligned} \langle e_2 A^k, (x_X, y_X, \mathbb{1})\rangle &= y_X + k x_X + \tfrac{k(k-1)}{2} \\ &= \tfrac{1}{2}k^2 - \tfrac{5}{2}k - 1 \end{aligned}$$

As we can see in Figure 6.1, from $k = 3$, the value of x is strictly increasing and after $k = 7$, the value of x is strictly superior to 2. Thus we have to check a finite number of iterations before reaching $x > 2$, which is the certificate set constraint of the non-reachability of Y. For $k \in [0, 6]$, Y is not reached. The couple $(7, \{(x, y, \mathbb{1}).x > 2\})$ is thus a certificate of non reachability of Y.

6.2.3 Case 4: eigenvalues all have a modulus equal to 1 and the transformation is diagonalizable

Some transformations do not admit generalized eigenvectors, namely diagonalizable transformations. The previous theorem is then irrelevant if for every eigenvalue λ, $|\lambda| = 1$. Such transformations are *rotations* : they remain in the same set around the origin. Take as example the transformation A of Figure 6.2, taken from [Fij+17].

It defines a counterclockwise rotation around the origin by angle θ =arctan($\frac{3}{5}$), and $\frac{\theta}{\pi}$ is not rational. The reachable set of states from X, i.e. $\{X, AX, A^2X, ...\}$ is strictly included in its closure, i.e. the set of reachable states and their neighbourhood. As Y is not on the closure of the set, then we can easily provide a non-reachability semi-algebraic invariant certificate of Y, that is the equation of the circle. However, we cannot give such a certificate for Z though it is not reachable. If it were reachable, there would exist a n such that $A^nX = Z$, thus $A^{2n}X = X$. n would also satisfy $\theta * n = 0[2\pi]$, which is impossible as $\frac{\theta}{\pi}$ is not rational. More generally, the closure of the reachable set of states of diagonalisable transformations with eigenvalues of modulus 1 is a semialgebraic set [Fij+17]. Semialgebraic certificates for such transformations exist if and only if Y does not belong to this closure [Fij+17].

Theorem 8 *For a given instance $\mathcal{O}(A, X, Y)$ such that A is diagonalizable and all its eigenvalues have a modulus of 1, eigenvectors can be used as semialgebraic certificates iff Y is not in the closure.*

Proof. Let $\mathcal{O}(A, X, Y)$ be an instance of the Orbit Problem with A a diagonalizable matrix only admitting eigenvalues λ such that $|\lambda| = 1$ Let φ an eigenvector of A, we denote $R = \{v|\exists k.A^kX = v\}$ the reachable set.

Lemma 9 *Let (λ_i, φ_i) be d couples of eigenvalue / left-eigenvector of a diagonalizable matrix A of size d. Then $R = \{v|\exists k, \forall 1 \geqslant i \geqslant d, \langle \varphi_i, v\rangle = \lambda_i^k . \langle \varphi_i, X\rangle\}$*

Proof. Let $R' = \{v|\exists k, \forall 1 \geqslant i \geqslant d, \langle \varphi_i, v\rangle = \lambda_i^k \langle \varphi_i, X\rangle\}$. By the definitions of R and φ_i, the inclusion $R \subset R'$ is trivially true. Now take $v \in R'$. As there exist d different and independent eigenvectors, v is a solution of the following relation: $\exists k.\Phi v = (\lambda_1^k x_1, ...\lambda_d^k x_d)^t$, where Φ is an invertible matrix whose lines are directly defined by eigenvectors. As Φ is invertible, there exists only one solution for each k. As v is one of those solutions, then $v \in R$.

By lemma 9, for any i between 1 and d, every element v of R verifies $|\langle \varphi_i, v\rangle| = |\langle \varphi_i, X\rangle|$, thus $R \subset R_\varphi = \{v : |\langle \varphi_i, v\rangle| = |\langle \varphi_i, X\rangle|\}$. Note that this inclusion is strict, as $X' = A^{-1}X \in R_\varphi$ but $X' \notin R$. If Y does not belong to R_φ, then $(0, R_\varphi)$ is a non reachability certificate.

□

$$\begin{array}{rcl} A & = & \frac{1}{5}\begin{pmatrix} 4 & -3 \\ 3 & 4 \end{pmatrix} \\ X & = & (1, 0) \\ Y & = & (1.5, 0.7) \\ Z & = & (-1, 0) \end{array}$$

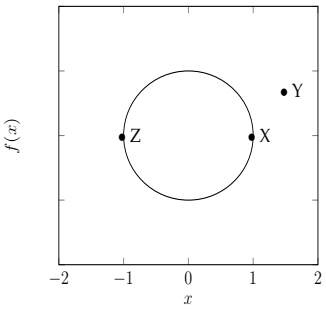


FIGURE 6.2: Closure of the reachable set of A starting with X.

6.3 General existence of a certificate for the integer Orbit Problem

The Orbit Problem is originally defined on $\mathbb{Q}$, but most programs only work on integers. Though $\mathbb{Z}$ is not a field, it is still possible to define linear transformations on $\mathbb{Z}$. Basic matrix operations involving divisions (such as inversion) are forbidden, but the only relevant operation in our case is multiplication (does there exist a n such that $A^nX = Y$?) which is consistent for integer matrices.

When dealing with linear transformations manipulating integers, things are quite different. Indeed, the following property holds for integer matrices.

Property 20 *Let $A \in \mathcal{M}_n(\mathbb{Z})$. If all its eigenvalue λ have a modulus inferior or equal to 1, then there exists $n > 1$ such that $\lambda^n = \lambda$.*

Proof. $|\lambda| \leqslant 1$.

If $\lambda = 0$, then we can conclude right away ($0^2 = 0$).

The characteristic polynomial $P \in \mathbb{Z}[X]$ of A is monic, i.e. its leading coefficient is 1. Thus by definition, every eigenvalue is an algebraic integer. We will use the Kronecker theorem [SZ+65], stating that if a non null algebraic integer α has all its rational conjugates (i.e. roots of its rational minimal polynomial) admitting a modulus lower or equal to 1, then α is a root of unity.

Each eigenvalue λ admits a minimal rational polynomial Q. We can show that Q necessarily divides P by performing an euclidian division : there exist $D, R \in \mathbb{Q}[X]$ such that $P(X) = Q(X)D(X) + R(X)$, with the degree of R strictly inferior to Q. We know that $P(\lambda) = 0$ and $Q(\lambda) = 0$, thus $R(\lambda) = 0$. If $R \neq 0$, then R is the minimal polynomial of λ as its degree is inferior to the degree of Q, which is absurd by hypothesis. Thus, the set of rational conjuguates of λ are roots of P, by hypothesis of modulus inferior or equal to 1. By the Kronecker theorem, λ is a root of unity, i.e. $\exists n > 1.\lambda^n = \lambda$.

□

This result is fundamental in the proof of the following theorem.

Theorem 9 *Any non-reachable instance of the Orbit problem $\mathcal{O}(A, X, Y)$ where $A \in \mathcal{M}_n(\mathbb{Z})$ admit a closed semi-algebraic invariant.*

Proof. We already treated the case where the matrix has an eigenvalue whose modulus is different from 1 (Property 18) and the case where the matrix is not diagonalizable (Property 19). We are left with the hypothesis of the Property 20.

Let A be a transformation such that all its eigenvalue are either 0 or roots of unity. A represents a finite-monoïd transformation, i.e. its reachable set of space is *finite*. More precisely, there exist N, p such that $\forall n > N, A^{n+p} = A^n$ Let $P = \{A^N X, A^{N+1}X, ..., A^{N+p-1}X\}$. If Y is not reachable, then the couple (P, N) is a non-reachability certificate.

The closure of such a certificate comes from the same eigenvalue argument. The only case we had a non-closed certificate comes from Property 18

when $|\lambda| \neq 0$, $|\lambda| \neq 1$, $|\langle \varphi, X \rangle| \neq 0$ and $|\langle \varphi, Y \rangle \neq 1$. As we also have $|\lambda| \geqslant 1$ for integer matrices, the certificate set $\{v : |\langle \varphi, v \rangle| \geqslant |\langle \varphi, X \rangle|\}$ is a valid closed certificate set.

□

6.4 Perspectives

The generation of certificates is a useful tool for automatic provers that attempts to prove the non reachability of certain invalid states. Still, provers often try to prove the non reachability of set of states described by one or multiple predicates instead of the non reachability of specific states. Certificate sets of transformations of the two first cases treated in Section 6.2 ($|\lambda| \neq 1$) are totally independent of the initial state X, which widens the possible uses of certificates. It is possible to use the same certificate set for differents values of X and Y, allowing to treat specific kind of vector sets (coefficients of X and Y as closed intervals for example, which are encountered more often in program verification than precise values). Interesting axis of development are to find certificates independent of X and Y in the general case and to study in detail which kind of vector sets can the certificate search be of use.

As this Chapter explores the Orbit Problem for rationals, it is worth noting that certificates may not necessarily be relevant for real-life programs manipulating floats. For example, the Orbit problem $(x \mapsto \frac{x}{2}, 1, 0)$ has a solution for some floating point implementations due to limited precision. The question of synthesizing certificates for such problems is also an interesting challenge.

Part III

Implementation and experimentations

Introduction to Frama-C

So far, we studied methods for helping the challenge of verification as generating invariants and certificates are possible on the programming model defined in Figure 2.2. This semantic is not expressive enough for real-life needs. Functions, memory and type system are examples of issues we did not focus on Chapter 6 as working on exact rationals is simpler than working on floats for example. Also, synthesizing invariants (Chapter 4 and Chapter 5) is a mean, not a goal in itself. It must serve the user's motives for formally verifying a program, and invariants are generally not sufficient. Usually, they are provided (either by the user or by a synthesier) so that other tools can narrow their results.

To this end, the *Frama-C* framework [Kir+15] provide means to allow collaboration between differents techniques.

Frama-C *Frama-C* (**FRA**mework for **M**odular **A**nalysis of **C** programs) is a collaborative and extensive open-source framework dedicated to the analysis of C programs. It contains different plug-ins that easies program verification, such as *EVA* [BBY17] for abstract interpretation or *E-ACSL* [SKV17] for dynamic analysis. Its kernel and all its plug-ins are developped in *OCaml*.

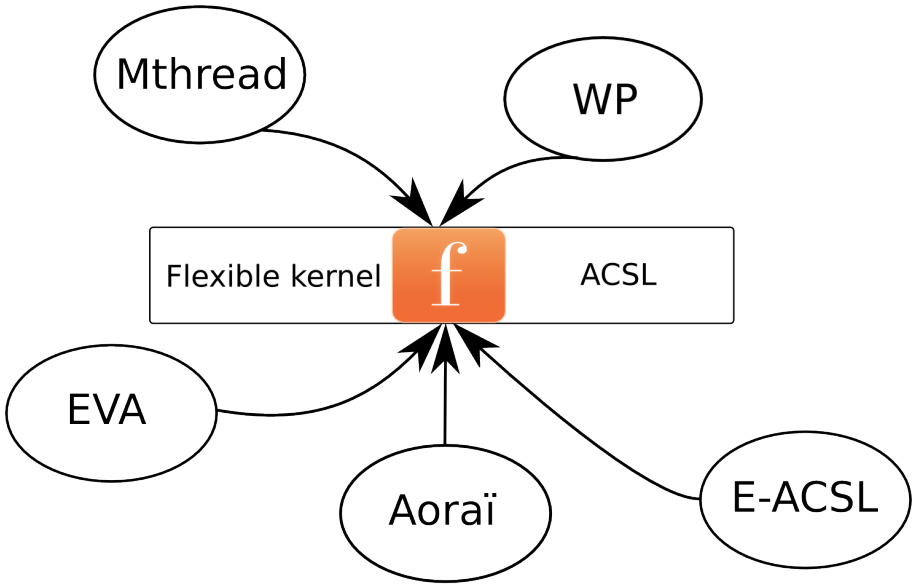


FIGURE 6.3: An overview of Frama-C

Its set of plugins share their results through two mechanisms.

- A flexible kernel allows any plug-in to communicate through a documented API. Every plugin results can be reached simultaneously by an *OCaml* script for achieving a strong proof obligation that couldn't be treated without collaboration.

- The *ACSL* specification language formalizes proof obligations. The proof of each specification can be performed independently by each plug-in and saved for later analyses.

The next chapters will introduce two new plus-ins of *Frama-C*. First, Chapter 7 the *Pilat* tool implements the *PILA* technique described in Chapter 5 with all its extensions. The invariants generated are used by the second plug-in, called *CaFE* (Chapter 8). This plug-in is a model-checker using abstract interpretation and loop invariants to build an automaton which is matched to a *CaRet* [AEM04] specification.

Chapter 7

Pilat: A polynomial invariant synthesizer

Contents

Requirements: PILA method (Chapter 5)

Chapters 4 and 5 introduced two linear invariant synthesis methods. The algorithm described in Chapter 4 is implemented as a part of the *Fluctuat* suite [Gou13]. This chapter will focus on the tool *Pilat*, a plug-in of the *Frama-C* [Kir+15] framework, implementing the eigenvector method of Chapter 5. It will present the architecture of *Pilat* and how it interacts with other plug-ins of the framework.

7.1 Pilat tool

7.1.1 Architecture overview

The *Pilat* initial algorithm described in Chapter 5 is based on three independent steps.

- **Linearization:** Chapter 3 presents a loop semantic transformation that replaces *solvable loops* (a sub class of polynomial loops, Definition 12) by linear loops. Algorithms to test the solvability of a loop and the linearization process are studied in Section 3.3. After this step, the studied loop is composed of conditions, nested loops and linear assignments.
- **Invariant computation:** Theorem 4 of Chapter 5 describe sets of linear loop invariants as the *eigenvectors of the dual of the studied transformation*. Inequality invariants can also be deduced from eigenvectors as described in Section 5.6. When conditions or nested loops occurs, intersections of vectorial spaces are performed.

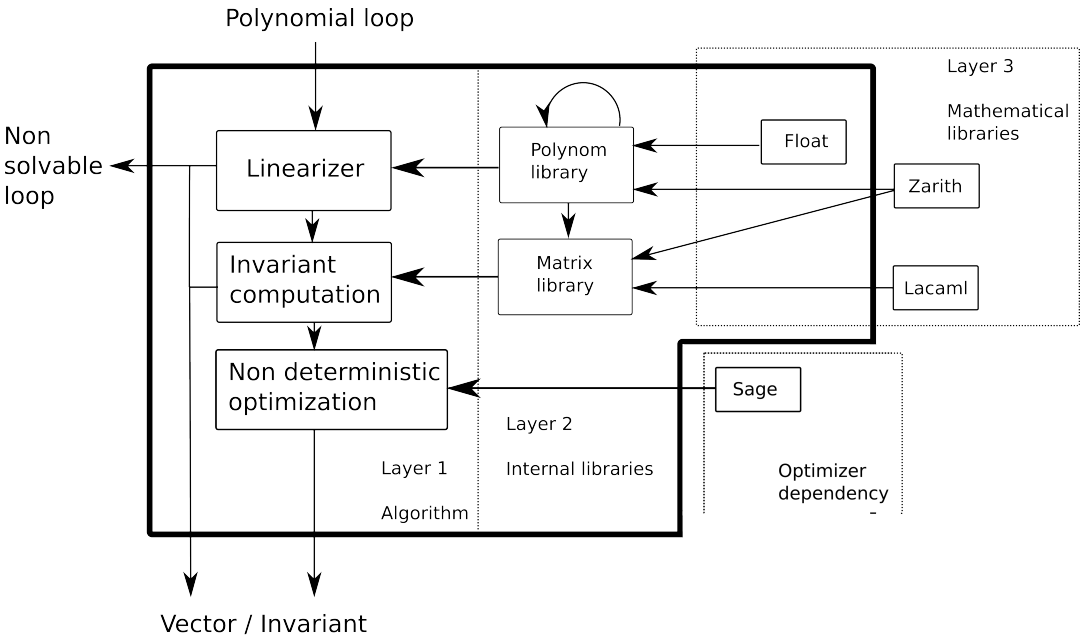


FIGURE 7.1: *Pilat* plug-in architecture

- **Non deterministic optimization:** The non deterministic optimization corresponds to the extension described Section 5.7. After generation of all constraints, a *Sage* script is called for the resolution of the polynomial constraints.

  Remark. *Sage* [Ste+08] is an open-source mathematical library developed in *Python*. It proposes multiple functions for solving the polynomial optimization problem required for the treatment of non deterministic loops. However, it doesn't propose a function to find directly a bound of the invariant (called k in Section 5.7), but can only minimize certain values given polynomial constraints. Therefore, the dichotomy algorithm of Figure 5.5 is necessary to find bounds.

7.1.2 Layers

Frama-C expresses C programs with an exhaustive AST expressing much information about types, memory, locations in the source code, etc.. *Pilat* is only interested in specific information of this AST, namely assignments, coefficients and polynomial expressions. It is divided into three layers.

Third layer: mathematical libraries. *Pilat* relies on multiple mathematical libraries to express coefficients of the loop. These libraries have the same

signature[1], called *Ring* (cf Appendix A.1) The *Pilat* tool implements two different *Ring* modules. When dealing with integer loops, i.e. that do not involve floating point operations nor integer divisions, *Pilat* will always use the *Zarith* library, allowing to represent unbounded size rationals. Otherwise, when dealing with float loops or non deterministic loops, *Pilat* will use the *OCaml* representation of floats for polynomials with an in-place float library.

Second layer: internal libraries. Polynomial representation of assignments and matrix representation of linear transformation are respectively at the core of the linearization and the invariant generation procedures. Similarly to the *Ring* signature, a *Matrix* and a *Polynomial* signature are implemented. Both the *Matrix* signature (Appendix A.2) and the *Polynomial* signature (Appendix A.3) are consistent with the *Ring* module, in the sense that matrices as well as polynomials are usable as rings.

In practice, those modules are instantiated as functors[2]. The functor creating modules of signature *Matrix* depends on a module of signature *Ring*, which allows creating matrices with polynomial coefficients (as *Polynomial* is compatible with the *Ring* signature). Similarly, the functor creating modules of signature *Polynomial* depends on two modules: a *Ring* module and a *Variable* module, containing printing utilities only.

Those functors are used in concert with the libraries of the previous layer to implement the necessary matrix and polynomial utilities. When dealing with float loops, *Pilat* privileges the use of *Lacaml*, a *Matrix* module which is an *OCaml* binding of the FORTRAN *Lapack* library.

First layer: Algorithm. Pilat algorithm requires three main components: a linearizer that transforms a polynomial loop into a linear loop; a invariant synthesizer; a polynomial optimizer for non deterministic assignments. Loop bodies are represented by a couple (variable,polynomial) while conditions are represented as a list of loop bodies.

The linearization algorithms of Figure 3.4 and Figure 3.8 respectively check the solvability of a polynomial transformation and linearize the loop if and only if it is solvable. *Pilat* merges those two algorithms by linearizing the loop while checking for its solvability. The idea is to attempt to linearize a transformation, and verify that no monomial ever depend on itself, which implies by Property 3 that the loop is not solvable.

Algorithms for computing eigenvectors of a matrix abound in the literature [PC99]. Many different types of matrices possess algorithms to solve this problem. A generic one is the following: for a loop body f, this step computes the roots λ of the characteristic polynomial $P(x)$ of f (which is defined as the determinant of $f - xId$). These roots are exactly the eigenvalues of f, it is now sufficient to compute the nullspace $K_\lambda = ker(f - \lambda.Id)$. This

[1] An *OCaml* signature is a contract over a set of function, called *module*. This contract must specify the type of elements defined in the implementation of the module that will be available by other modules.

[2] In *OCaml*, functors define modules parametrized by other modules.

algorithm is used in *Pilat* when using the *Zarith* library. As *Zarith* manipulates only rationals, it only searches for rational roots of $P(x)$, which is a polynomial with rational coefficients if the matrix has rational coefficients. The *Rational Root Theorem* allows catching rational roots easily:

Property 21 *Rational roots of* $P(x) = \sum_{i=0}^{n} a_i x^i$ *are of the form* $\frac{p}{q}$ *with:*

- p *divides* a_0*;*
- q *divides* a_n*.*

Proof. Assume there exists a rational root $\frac{p}{q}$ for P, with p and q coprimes. It is clear that

$$a_n(\frac{p}{q})^n + a_{n-1}(\frac{p}{q})^{n-1} + ...a_1\frac{p}{q} = -a_0$$

We multiply everything by q^n:

$$a_n p^n + a_{n-1}pq + ... + a_1 pq^{n-1} = -a_0 q^n$$

As p and q are coprimes, then p divides a_0. Also, we have

$$a_{n-1}pq + ... + a_1 pq^{n-1} + a_0 q^n = -a_n p^n$$

hence q divides a_n. □

In other words, it is enough to get all divisors of the extremity of the polynomial and test all possible combinations. Though this test is exponential in practice, integer loops admit in general a simple characteristic polynomial. If the characteristic polynomial is too complex, *Pilat* only checks roots with small numerators and denominators (bounded by a tool option).
This technique doesn't work when working on float loops, as they admit real eigenvalues. In this case, *Lacaml* has a primitive for generating eigenvalues. However, experimentations have shown eigenvalues generated this way can be imprecise when generated by *Lacaml*, especially when they are not rational. Generating useful invariants associated to irrational eigenvalues is quite complex as it implies that the invariant have irrational coefficients as well. Hence, they have no representation in C. Property 8 gives a partial answer to this issue, as it allows introducing product of eigenvalues in the matrix. As the product of every eigenvalue is rational for a rational matrix, the existence of a rational eigenvalue is guaranteed in the matrix expressing all monomials of degree n, where n is the size of the matrix. The complexity of generating these monomials is $O(n^n)$, which makes the algorithm unusable when $n = 4$. The search of geometric relations between eigenvalues (finding rational products of algebraic values) is a possible solution for this problem. Searching for eigenvalues with *Lacaml* is only made to check if an eigenvalue is higher or lower than 1. This check has shown to be valid on all experimentations presented in Section 7.2, at the end of this Chapter. Each vector of K_λ is a λ-eigenvector, hence an invariant of f. Computing the nullspace of a linear transformation is a well known algorithm [Fos86] that

will not be detailed here. When an irrational eigenvalue is used to search for invariants, *Pilat* fails to generate a basis for K_λ because of precision issues. Hence, it doesn't generate invariants with irrational, which is exactly what is expected.

When dealing with non deterministic loops (such as linear filters), a candidate invariant is generated and has the form $P(X) \leqslant k$. When dealing with deterministic loops, such invariants are inductive for all k, but for non deterministic cases, it is only inductive for certain values of k (cf Property 16). The last component for the treatment of non deterministic loops is a small script implementing the algorithm of Figure 5.5 to find a value of k for which $P(X) \leqslant k$ is inductive. In practice, the *min* implemented in *Sage* is unsound, in the sense that k can be underapproximated or overapproximated. This function takes as argument a starting value for k and tries to refine it. The unsoundness is then solved by repeating the process until the difference between two iterations is small enough to guarantee correctness. This difference can be set as an option of the tool (to a minimum of 0 which represents finding the exact value of k).

7.2 Experimentations and comparison with existing tools

In order to test our method, we implemented an invariant generator as a plugin of Frama-C [Kir+15], a framework for the verification of C programs written in OCaml. Tests have been made on a Dell Precision M4800 with 16GB RAM and 8 cores. Time does not include parsing time of the code, but only the invariant computation from the Frama-C representation of the program to the formulas. Two different benchmarks have been used for testing the method. A first one is specialized with integer programs (available at [Car08]), while the second manipulates floating point transformation (more details on Appendix B). Results are respectively in Table 7.1 and Table 7.2.

All the tested functions are examples for which the presence of a polynomial invariant is compulsory for their verification. The choice of high degree for some functions is motivated by our will to show the efficiency of our tool to find high degree invariants as choosing a higher degree induces computing a bigger set of relations. In the other cases, degree is choosen for its usefulness.

For example in figure 7.2 we were interested in finding the invariant $x + qy = k$ for eucli_div. That's why we set the degree to 2. Let X be the vector of variables $(x, y, q, xq, xy, qy, y_2, x_2, q_2, \mathbb{1})$. The matrix A representing the loop in figure 7.2 has only one eigenvalue : 1. There exist 4 eigenvectors $\{e_i\}_{i\in[1;4]}$ associated to 1 in A, so $\left\langle \sum_{i=1}^{4} k_i e_i, X \right\rangle = 0$ is a semi-invariant. One of these eigenvectors, let's say e_1, correspond to the constant variable, i.e. $e_1.X = \mathbb{1} = 1$, thus we have $\left\langle \sum_{i=2}^{4} k_i e_i, X \right\rangle = -k_1$ as invariant. In our case, $\langle e_2, X\rangle = y$, $\langle e_3, X\rangle = x + yq$ and $\langle e_4, X\rangle = y_2$. We can remove $(y = k)$ and $(y_2 = k)$

Program			Time (in ms)		
Name	Var	Degree	*Aligator* [Kov08]	*Fastind* [Cac+14]	*Pilat*
divbin	5	2	80	6	2.5
hard	6	2	89	13	2
mannadiv	5	2	27	6	2
sqrt	4	2	33	5	1.5
djikstra	5	2	279	31	4
euclidex2	8	2	1759	10	6
lcm2	6	2	175	6	3
prodbin	5	2	100	6	2.5
prod4	6	2	13900	–	8
fermat2	5	2	30	9	2
knuth	9	3	O.O.T.	347	192
eucli_div	3	2	13	6	2
cohencu	5	2	90	5	2
read_writ	6	2	82	–	12
illinois	4	2	O.O.T.	–	8
mesi	4	2	620	–	4
moesi	5	2	O.O.T.	–	8
petter_4	2	10	19000	37	3
petter_5	2	10	O.O.T.	37	2
petter_6	2	10	O.O.T.	37	2

TABLE 7.1: Performance results with our implementation *Pilat* for deterministic integer loops. The second column of the table represents the number of variables used in the program. The third column represents the invariant degree used for *Pilat* and *Fastind*. The last three columns are the computation time of the tools in *ms*. O.O.T. represents an aborted ten minutes computation and – indicates that no invariant is found.

	PILAT	Input	Results			Abs. Int.
Program	Var	Degree	# invariants	Generation (in s)	Optimization (in s)	Proof (in s)
Deterministic						
Example 1	2	2	1	0.003	–	1.6
Dampened oscillator	2	2	1	0.007	–	0.036
Harmonic oscillator	2	2	1	0.004	–	0.035
Sympletic oscillator	2	2	1	0.002	–	0.008
[AGG12] filter	2	1	1	0.0035	–	0.0017
Non deterministic						
Simple linear filter	2	2	1	0.0015	1.3	6.5
Example 3	2	2	1	0.003	1.7	4.3
Linear filter	2	2	1	0.0019	1	1
Lead-lag controller	2	1	2	0.002	2.5	6
Gaussian regulator	3	2	1	0.007	2.5	–
Controller	4	2	5	0.066	14	–
Low-pass filter	5	2	2	0.06	7	–

TABLE 7.2: Performance results with our implementation *Pilat* for deterministic and non deterministic linear filters. The first part represents deterministic loops (thus, no optimization is necessary). The second part of the benchmark are non deterministic loops. Tests with abstract interpretation have been performed with the fixpoint solver described in [MBR16] by attempting to prove goals implied by the invariants our tool synthesizes when they were compatible.

that are evident because y does not change inside the loop. The remaining invariant is $x + yq = k$.

Input : degree = 2

```
int eucli_div(int x, int y){
  int q = 0;
  while (x > y) {
    x = x-y;
    q ++;
  }
  return q;
}
```

Frama-C output :

```
  int eucli_div(int x, int y){
    int q = 0;
    int k = x + y*q;
    // invariant x + y*q = k;
    while (x > y) {
x = x-y;
q ++;
    }
    return q;
  }
```

FIGURE 7.2: Euclidean division C loop and generation of its associated invariants.

Chapter 8

CaFE: model checking

Contents

8.1 Motivation

Requirements: Model-checking (Section 2.3)

The right sequencing of events along time is a widely studied topic of program analysis, for example when studying distributed systems or information exchange protocols. In particular, temporal logics [Pnu77] allows describing formally the expected behavior of a system as a succession of distinct actions. In *Frama-C*, the specification language *ACSL* is used by many plug-ins as a proof-sharing system. In general, LTL properties are difficult to express using only *ACSL*, particularily when studying multiple function calls. On the other hand, temporal logics [Pri57] are apropriate tools for such properties. Besides, they allow the study of infinite execution paths that *ACSL* is not designed to specify. The Linear Temporal Logic [Pnu77], or LTL, interprets time as a linear sequence of actions and modelizes properties for such a sequence. Those properties are translated into *automatons* that are matched against the program behavior.

Aoraï [SP11] is a Frama-C plug-in allowing the verification of LTL formulas over a finite C program. The tool generates *ACSL* predicates at every call and return site of each function such that *the verification of such predicates is equivalent to the verification of the initial LTL formula* [Gio+08]. These *ACSL* specifications are independent of the initial program as they involve newly created variables stating the position on the automaton. In order to solve them, other tools must infer the relations between the program variables and the new variables, which may not be solved easily.

Also, let us recall the three temporal properties of interest introduced in Section 2.3.1:

- during a given event, the lock must not be taken (safety requirement);
- if an event occurs in the function f, then when f has been called, the lock was not taken (contextual requirement);
- every function must free the lock before it returns (liveness requirement).

While Aoraï/LTL is unable to catch such properties, the tool *CaFE* has been developped. *CaFE* (for *CaRet Frama-c*'s extension) is a model-checker based on the *CaRet* temporal logic [AEM04]. The *CaRet* temporal logic has tuned operators allowing to express properties on specific program points and on the call stack. The original algorithm of [AEM04] can decide in finite time (complexity 2-*EXP*) the validity of a *CaRet* formula over a Recursive State Machine (or *RSM*). This is not the case for programs, as the *CaRet* language can express the halting problem which is known to be undecidable. The idea of *CaFE* is to generate informations on the program for guiding the analysis and apply *generalized model-checking* [BG00]. Generalized model-checking is a method to adapt model-checking algorithms to work on partial information models. This chapter will introduce the *Abstract Recursive State Automata*, or *A RSM*, and the *CaRet* temporal logic for partial information models in Section 2.3.5. Then, Section 8.2 adresses the problem of adapting the original algorithm to work on *A RSM* and remain correct, in the sense that a positive answer of an algorithm can be trusted (but not a negative answer). At last, the architecture of *CaFE* working with different plug-ins of *Frama-C* is presented in Section 8.3.

A full description of this work can be found in [OPB].

8.2 CaFE : a model checker of CaRet formulas

Soundness.

The model-checking algorithm of *CaRet* properties over *RSM* is already well known [AEM04]. For a *RSM* R on a proposition set AP and a formula φ, it is possible to compute the product automaton $S_{\neg\varphi} = R \times \neg\varphi$ that accept every word that correspond to an execution of R and that verifies $\neg\varphi$. Each node of this automaton holds the set of properties that must be required at a given program point. Any accepting path of this new automaton is a counter example of the checked property. If $\mathcal{L}(S_{\neg\varphi})$ is empty, then every execution of R verifies φ (or $R \vDash \varphi$). The general idea is to execute both automatons at the same time, create a new state including the location on the *RSM* and which properties are verified, and check whether the state is consistent or not with respect to η. If it is, the state is preserved, otherwise it is deleted. Being able to decide at any node $n \in N$ of R which property is consistent and which is not is crucial to this construction. For a *RSM* R, a state of $R \times \neg\varphi$ representing the node $n \in N$ is valid when all the properties $p \in AP$ it holds verify $\eta(n, p) = \top$ [AEM04].

Abstract recursive state machine. It is not possible to exactly represent a program by a *RSM*, mostly because it is not possible, for a given proposition set AP, to have a complete function η.

Definition 28 *An abstract recursive state machine (or ARSM) is exactly a RSM, except its labeling function η is allowed to return $\star$ when the validity of a property is unknown.*

In the case of an *ARSM* A, the incompleteness of the labeling function forbids to have a straight forward interpretation of consistency. The general idea of generalized model checking [BG00] is to give an interpretation of the unknown answer $\star$.

- Considering $\star$ as incoherent is similar to not taking any risk. As soon as the labelling function cannot decide, the state is invalid and hence, not added to the product automaton. We are left only with states that are decided.
- Considering $\star$ as possible allows preserving states of the product automaton that contradicts themselves by stating that, on a given $n \in N$ of A, both p and $\neg p$ are possible.

$S =$	$A \times \varphi$	$A \times \neg\varphi$
No-risk	no unsatisfying executions of A	no satisfying executions of A
Sound	at least all satisfying executions of A	at least all unsatisfying executions of A

TABLE 8.1: Interpretation of the product automaton paths in the no-risk and the sound strategies.

One important guarantee to provide in model-checking is soundness, or in other words the guarantee of the validity of positive results. Table 8.1 interprets the set of paths of the product automatons $A \times \varphi$ and $A \times \neg\varphi$. Choosing an automaton that contains all satisfying executions or no unsatisfying execution is irrelevant for soundness. An automaton S with no satisfying execution doesn't bond an effective unsatisfying execution to belong to S. That is why the choice of $A \times \neg\varphi$ with the sound strategy is the more efficient choice in this case.

Property 22 *Let A an ARSM and φ a CaRet formula. If $\mathcal{L}(R \times \neg\varphi) = \emptyset$ with the sound strategy, then $A \vDash \varphi$*

Proof. If $A \nvDash \varphi$, there exist a path π satisfying $\neg\varphi$. As $S_{\neg\varphi} = A \times \neg\varphi$ represents at least all the unsatisfying executions of A, $\mathcal{L}(S_{\neg\varphi})$ will at least admit π. □

Comparison of similar automatons

Definition 29 *Let $A = (M, \{R_m\}_{m\in M}, \eta, init)$ and $A' = (M', \{R'_m\}_{m\in M'}, \eta', init')$ two ARSM. A and A' are* equivalent *if $M = M'$, $\forall m \in M, R_m = R'_m$ and $init = init'$.*

Definition 30 *Let A_1 and R_2 two ARSM. Let $Fib_a(A) = \{(s,p)|\eta(s,p) = a\}$ the fiber of a by η. A_1* approximates *A_2, or $A_1 \geqslant A_2$ if and only if:*

- *A_1 and A_2 are equivalent*
- *$Fib_\top(A_1) \subseteq Fib_\top(A_2)$ and $Fib_\bot(A_1) \subseteq Fib_\bot(A_2)$*

Remark. If $A_1 \geqslant A_2$, then $Fib_\star(A_2) \subseteq Fib_\star(A_1)$. If A_2 is a classic *RSM*, then $Fib_\star(A_2) = \emptyset$ by definition. Every approximation of A_2 is an *A RSM* whose decided properties have the same truth value than A_2 according to η.

Property 23 *Let A_1, A_2 two ARSM such that $A_1 \geqslant A_2$ and φ CaRet formula. Let $S_1 = A_1 \times \neg\varphi$ and $S_2 = A_2 \times \neg\varphi$. Then $\mathcal{L}(S_2) \subseteq \mathcal{L}(S_1)$.*

If $A_1 \geqslant A_2$, then $Fib_a(A_1) \subseteq Fib_a(A_2)$ for $a \in \{\top, \bot\}$. Let $\pi \in \mathcal{L}(S_2)$. π can be seen as a sequence of nodes of S_2 that have been accepted during the product of A_2 and $\neg\varphi$. This means that the properties that each node holds have been either validated by η ($\top$) or that it can't conclude ($\star$) as the sound strategy considers coherent what is unknown. As $A_1 \geqslant A_2$, then let n an element of π. η cannot refute n to belong to S_1 as $Fib_\bot(R_1) \subseteq Fib_\bot(A_2)$. Therefore, every node of π belong to S_1. □

8.3 Overview of CaFE

CaFE (CaRet Frama-C's extension) is a model-checker based on the previous properties. interactions with different plug-ins of the framework and an external prover, *Z3*.

CaFE. The transformation of a C program to a *RSM* is made in-place by Frama-C, that parses the source code to a CFG described in C Intermediate Language [Nec+02] (or CIL). A CIL control flow graph is structurally more expressive than a *A RSM*. Predecessors, successors, functions entry and exit points are expressed in CIL, which is enough to represent the main frame of an *A RSM*. There only misses the labelling function η. So as to solve this issue and the characteristic combinatory explosion of model checking, *CaFE* work along with some other plug-ins of *Frama-C*.

- *EVA* [BBY17]: an abstract interpreter able to combine multiple numerical domains. Its results are saved as numerical invariants for each program points. It is possible to check the possible abstract value of an expression on any program point. This is the main source of information for the η function.

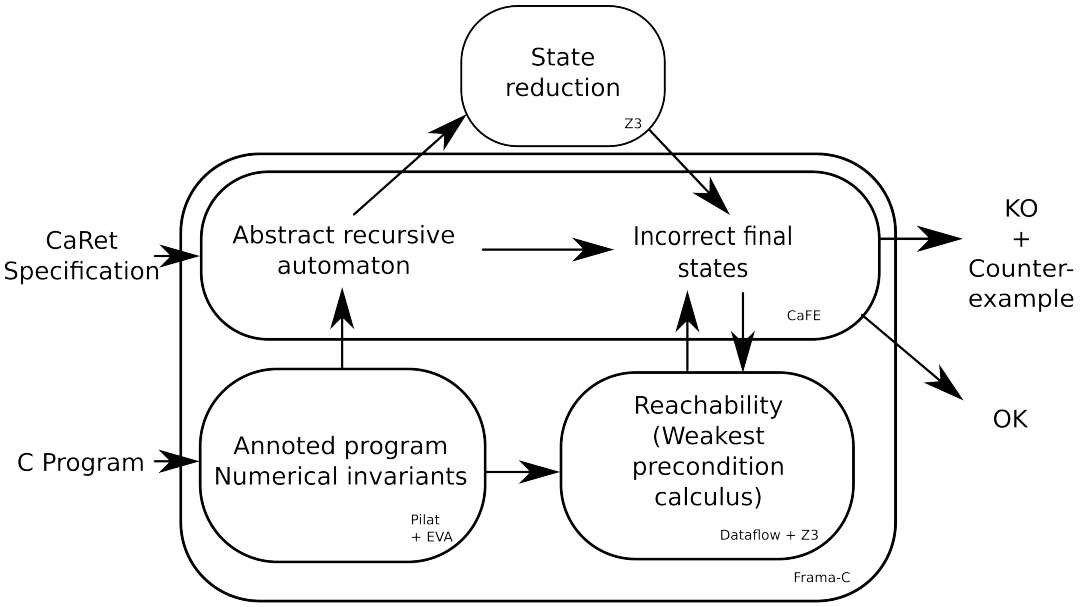


FIGURE 8.1: Functionning of *CaFE*

- *Pilat*: the invariant synthesizer described in Chapter 7. It adds *ACSL* annotations to linear loops that can be used to refine the η function for loops that *EVA* often overapproximate too widely.
- The Dataflow module: a generic tool for iterating over a CIL control flow graph (forwards and backwards)

The product of the program seen as a *RSM* and the negation of the *CaRet* specification uses the results of the two first tools to delete states that are inconsistent with respect to an effective execution of the program. When both *EVA* and *Pilat* cannot conclude on the consistency of a state, the SMT solver *Z3* [MB08] is used in addition. The case where *Z3* also cannot conclude is the case where η responds $\star$.

The new automaton S is then simplified by deleting non-accepting paths and unreachable nodes. If there is still accepting states, a weakest precondition calculusis launched by the Dataflow module from the accepting final states that reuses the results of *EVA* and *Pilat*. The validity of each path is tested by *Z3*. Inconsistent paths with respect to this calculus are removed, while others are output as counter-examples.

Limitations. The tool *CaFE* still presents some limitations. First, non solvable loops are not treated by *Pilat*, therefore they are not given additional loop invariants. Even if *EVA* overapproximates the loop state, this state is often imprecise. When such situation occurs, *CaFE* returns many false positives. The user may solve this issue by changing *EVA* options to get a satisfying level of precision.

Also, the complexity of model-checking limits the size of case studies. In general, the problem is linked to the size of the *CaRet* property to verify. It is necessary to manually divide the formula into multiple smaller formulas.

For example, it is in general more efficient to prove a and b separately than verifying $a \wedge b$.

8.4 Application to concurrency

With the current arrival of multi core processors, the shared memory principle raised many issues, especially the access by a core of a data in the cache that is being processed by another core. The MOESI is a cache coherence procotol solving this issue. Each line of the cache is attributed a state: ***m**odified*, ***o**wned*, ***e**xclusive*, ***s**hared* or ***i**nvalid*. When a core needs to access a cache line, it performs a request to ensure the exclusivity of the line. If the status of each line can be modified, the protocol must ensure there are always the same total number of lines.

For this experimentation, a sequential representation of the protocol in Figure 8.2 inspired from [Car08] has been used. This is simply an infinite loop simulating after every step the call to a function manipulating the content of a cached data. The goal of this experimentation is to compare two different specifications testing if the initial ressources are conserved during the program execution under two hypotheses:

- each instruction is observed sequentially: $G^g(m + o + e + s + i = c)$;
- function calls are considered atomic: $G^a(m + o + e + s + i = c)$.

In ACSL, those specifications can be represented by a set of assertions over the *main* function, where each assertion must be proven separately. This process is inefficient for the automatic treatment of large programs. In *CaFE*, the program is directly assimiled to a *RSM* as represented in Figure 8.2. The use of the full *Frama-C* framework is vital for the treatment of the temporal specifications.

1. *Pilat* starts by generating the unique linear inductive invariant of the loop: $m + o + e + s + i = k$, where k is a unknown constant.
2. *EVA* analyses the full program, proving in particular that the *main* does not ends (the exit of *main* is unreachable) and adds to each instruction a numerical invariant in a choosen domain (here, only the interval domain is used).
3. Thanks to this data, *CaFE* applies the model-checking technique described in Section 8.2. Unreachable and inconsistent states are deleted from the generated product automaton with respect to the results of *Pilat* and *EVA* It also uses Z3 in order to refine the simplification.
4. There still exist accepting final states. A backward analysis of the program by the Dataflow module with the use of Z3 quickly proves those states cannot be reached. The corresponding states are deleted as well as the possible paths leading to them.

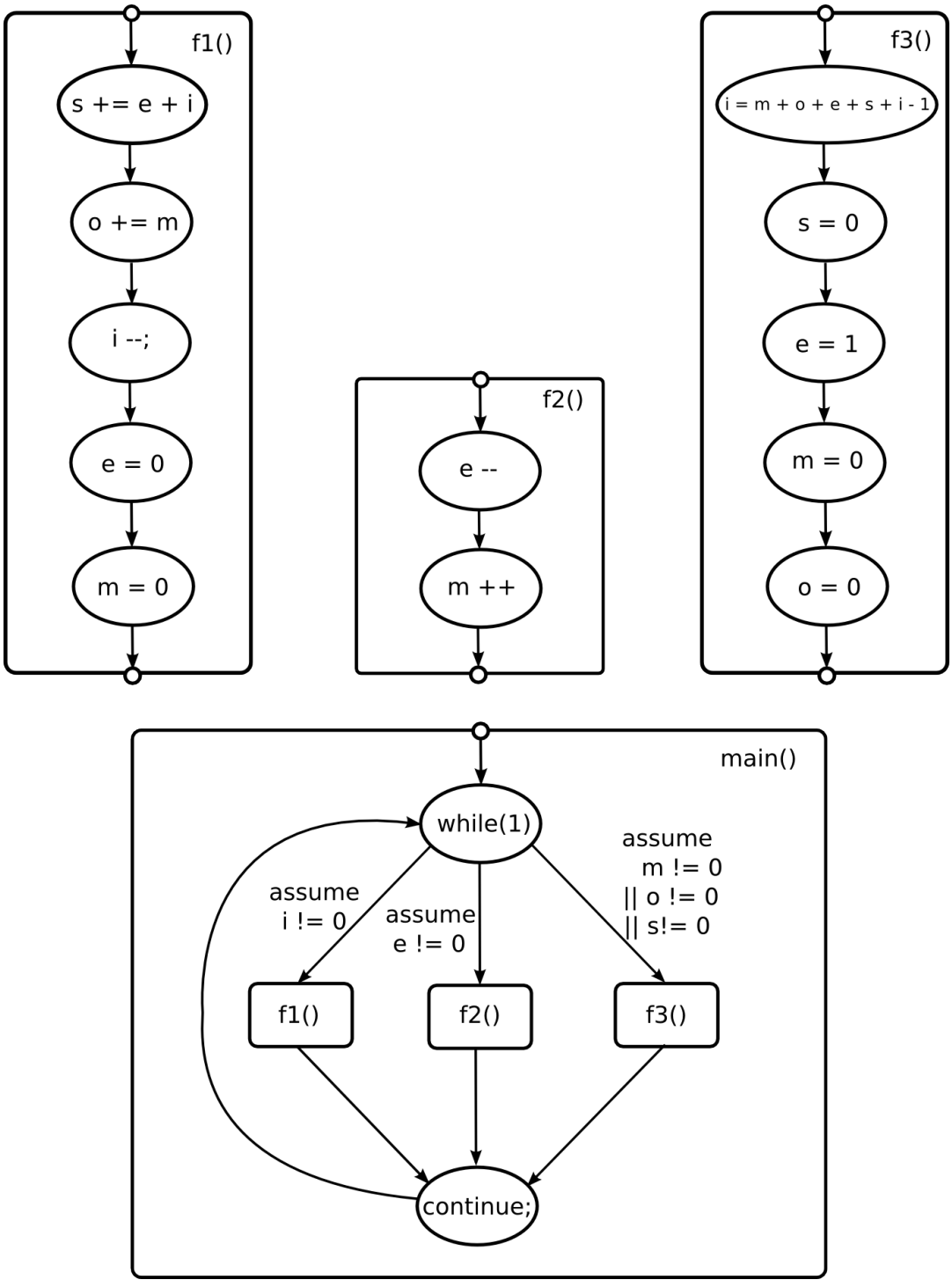


FIGURE 8.2: Recursive state machine representing the MOESI protocol.

Initial state :
$m = [0, 15]$, $o = [0, 15]$, $e = [0, 15]$, $s = [0, 15]$, $i = [0, 15]$, $c = m + o + e + s + i$;

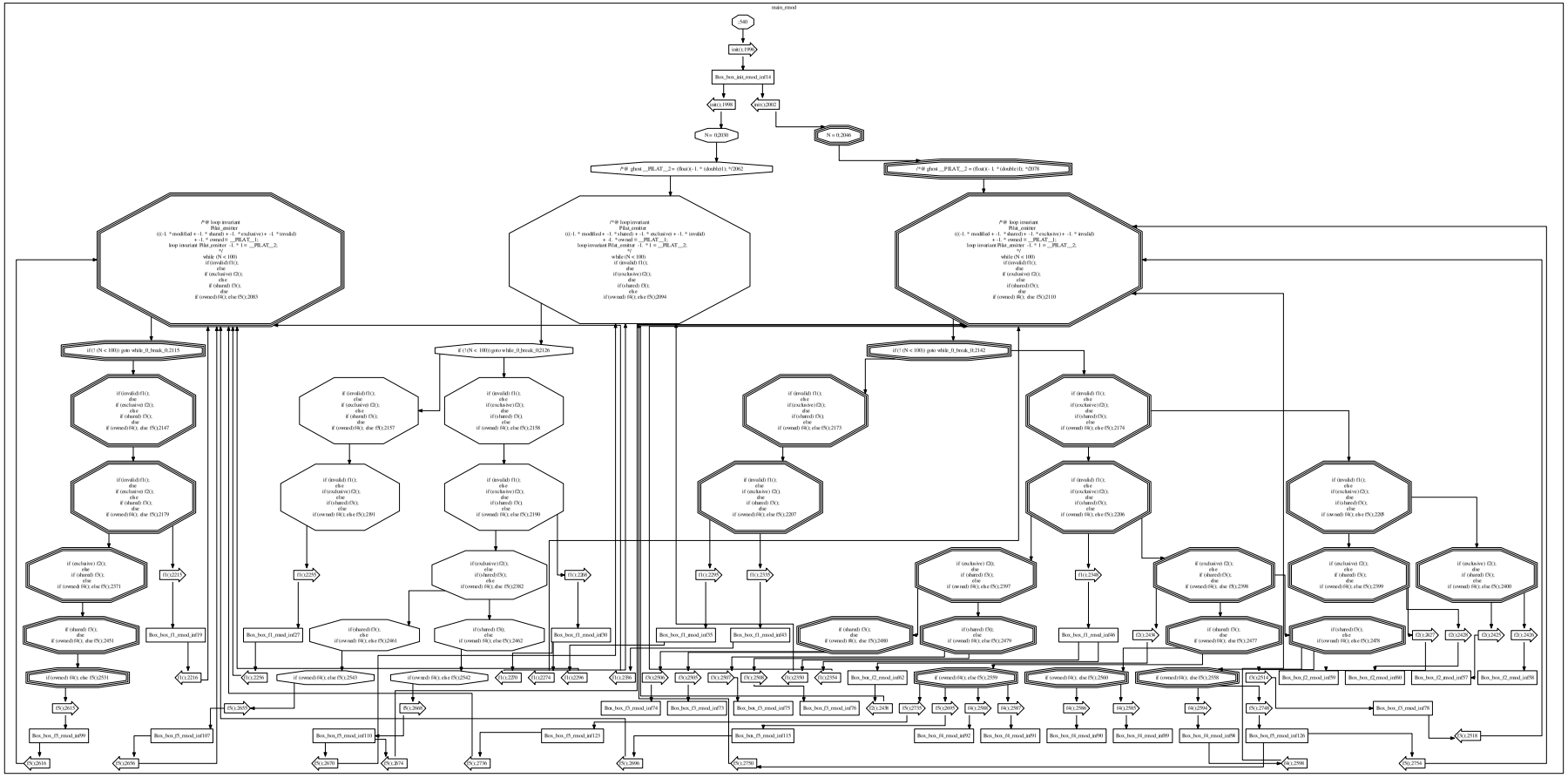


FIGURE 8.3: *CaFE* result: the set of program executions not satisfying the formula $G^g(m + o + e + s + i = c)$. It contains multiple accepting states, therefore the formula is considered false by *CaFE*. In the case of $G^a(m + o + e + s + i = c)$, there exist no counterexample left, so the formula is indeed true. donc vérifiée.

Observations. The result of the proof of the two specifications is available on Figure 8.3. The specification $G^g(m + o + e + s + i = c)$ is clearly incorrect as the function, not being considered atomic, modify sequentially the state of the cache during an update. *CaFE* is able to generale real (but unproven) counterexamples to this specification under the form of a *RSM* containing multiple accepting states. As each of those states corresponds to an instruction, it is possible to extract a set of possible executions leading to a final state where the property have been violated. Conversely, the invariant generated by *Pilat* proves the set of ressources is preserved between two loop steps. The formula $G^a(m + o + e + s + i = c)$ specifying the invariance of the quantity of ressources between the beginning and the end of each step of the protocol is then verified.

Part IV

Perspectives

Chapter 9

Conclusion

This thesis developped new insights on linear invariants by investigating three problematics.

9.1 Solvability

The first contribution of this thesis is the explicitation of the solvable polynomial class as the only polynomials that are expressible by finite linear transformations. Linear arithmetic in the integers is known to be a decidable theory in SMT solving, while polynomial arithmetic is not. As there exists a technique generating all invariants of solvable loops [RK07], it was difficult to tell which problem is decidable and which is not. This new result brings more insights on the class of solvable transformations and it is a nice step in the classification of decidable problems for loops.

9.1.1 Polynomial similarity

The elevation principle is a very simple concept. So simple that it feels strange that it has never been formalized before. Similarity is a key concept in linear algebra, that allows to associate different linear transformations that perform the same operation, but on different bases. Linear algebra restricts itself (rightfully) to linear transformation, hence changing the base is also a linear transformation. In this thesis, similarity has been generalized to polynomial base changing applications in the context of linear invariant generation. Fundamental properties of these transformations has also been investigated, but some questions remain unanswered:

- what is the nature of the operator Ψ_d ?
- equivalence[1] generalizes similarity in linear algebra, is there an equivalent relation for polynomial similarity ?
- what are the properties of $\Psi_d(A)$ that are deducible from A ?

This last question has been partly solved by the study of properties of the eigenvalues of $\Psi_d(A)$, but there is few information on its rank, its characteristic polynomial or its determinant for example.

[1] A and B are equivalent if there exist P and Q invertible such that $A = QBP^{-1}$

9.1.2 Infinite systems

The Carleman linearization procedure [KS91] inspiring this work is initially developped for linearizing any finite system of polynomial differential equation to an infinite system of linear differential equations by the exact same procedure. Each monomial of variable is seen as an independent function associated to a new linear differential equation, just as we associated to monomials new variables evolving linearly. In the non-solvable case, the procedure doesn't end as we restricted our study to finite linear transformations. The relevance of linearizing non-solvable transformations to infinite linear transformations is still open, and it has been shown to be unsound in the case of Carleman [Ste89] (there exists solutions of the infinite linear system that are not solutions of the finite polynomial system).

9.2 Invariant generation

The second contribution of this thesis are the new invariant synthesis methods. Two very different approaches have been studied in this thesis. The first approach is based on abstract intepretation and constraint solving. It has shown to be precise and adapted to the analysis of linear filters by generating approximations of ellipsoids, which are in general good invariants for such programs. The second approach was based on the eigenvector decomposition of linear loops. This technique is more generic, as all linear loops can be studied, including loops with contitions, nested loops and non deterministic assignments.

9.2.1 Generalization of the parametrized widening operator

Abstract interpretation generally lacks precision when it comes to handling loops, while acceleration lacks genericity. The parametrized widening operator described in Chapter 4 is a good trade off between the two techniques and gives good results in term of precision. In general, techniques based on parametrization rely on SMT solvers to be fully functional (in our case, to find a valuation of the parameters). We developped this technique under the scope of the zonotope abstract domain, but it is not clear whether this is the only abstract domain in which this technique could be applied. We can imagine generalizing this technique to more classical domains. The challenge is not really to be able to apply this technique to other domains, but to apply it to different kind of loops. In the case of linear filters and zonotopes, we assumed the nice property of slowly converging toward an invariant, which may not be the case for other loops. Determining what conditions are necessary for this kind of approach to succeed in synthesizing an invariant is a possible axis of development for widening operators.

9.2.2 Spectral theory

The eigenvector characterization of linear invariants of Chapter 5 in based on vector space with finite dimenson. It has the advantage to provide a simple way to generate and prove linear loop invariants. As we said above, linearization could be generalized to handle any kind of polynomial loops by using an infinite number of variables. Spectral theory [EE87] generalizes the eigenvalue/eigenvector decomposition problem to vector spaces of infinite dimension. Invariant properties of eigenvectors are suspected to be preserved on any vector space.

9.3 Usefulness of eigenvectors

The last contribution of this thesis relies on the practical use of invariants. Along with the generation of certificates, invariants are necessary in program proofs. In the context of C programs, the tool *Pilat* generates them as ACSL loop invariants that can be used by the different plug-ins of Frama-C.

9.3.1 Complete characterization of certificates

Eigenvectors have not only shown to be invariants easy to generate, but also expressive enough to solve the Kannan-Lipton Orbit problem. The study of the eigenvalues role in the generation and the degree of those certificates is a key in understanding their shape. The generation of linear certificates with algebraic coefficients is still an open problem. The complete characterization of invariants coupled with this certificate study gives us necessary clues in solving it.

9.3.2 *Pilat* extensions

Some extensions of the method have not been implemented, such as the generation of inequality invariants when the linear transformation admits generalized eigenvectors of order at least 2. When their associated eigenvalue is 1, then we saw that it is possible to generate invariants in Chapter 6 by considering their polynomial behavior. The question of how to generate invariants from generalized eigenvector associated to different eigenvalues has not been raised neither.

9.3.3 Temporal logic

The usefulness of these invariants have been proven by the generation of certificates and by the model-checker *CaFE*. This model-checker, based on the abstract interpreter *EVA*, uses the results of *Pilat* to precise its results. However, it is not efficient as the model-checking algorithm is 2-EXP in the size of the CaRet specification. CaRet is a powerful language that allows to express a large number of behavor. A lot of them are nonetheless practically never used, hence the question of the relevancy of the choice of CaRet remains.

The model-checking algorithm is basically the same for every tool (based on the automaton product). An interesting axis of developmenty of *CaFE* is its potential ability to handle different temporal logics with different operators. For example, as *Pilat* uses different libraries of temporal logic, *CaFE* could use different instanciations of temporal logics or automatons to perform a model-checking procedure. It could also be enhanced by counter-example verification, which is not the case in the current implementation.

Bibliography

[AEM04] Rajeev Alur, Kousha Etessami, and P. Madhusudan. "A Temporal Logic of Nested Calls and Returns". In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. 2004, pp. 467–481. DOI: 10.1007/978-3-540-24730-2_35. URL: https://doi.org/10.1007/978-3-540-24730-2_35.

[AGG12] Assalé Adjé, Stéphane Gaubert, and Eric Goubault. "Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis". In: *Logical Methods in Computer Science* 8.1 (2012). DOI: 10.2168/LMCS-8(1:1)2012. URL: https://doi.org/10.2168/LMCS-8(1:1)2012.

[AM09] Rajeev Alur and Parthasarathy Madhusudan. "Adding nesting structure to words". In: *Journal of the ACM (JACM)* 56.3 (2009), p. 16.

[Bar+05] Sébastien Bardin et al. "Flat Acceleration in Symbolic Model Checking". In: *Automated Technology for Verification and Analysis, Third International Symposium, ATVA 2005, Taipei, Taiwan, October 4-7, 2005, Proceedings*. 2005, pp. 474–488. DOI: 10.1007/11562948_35. URL: https://doi.org/10.1007/11562948_35.

[Bar+11] Clark Barrett et al. "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14. URL: https://doi.org/10.1007/978-3-642-22110-1_14.

[Bau+16] Patrick Baudin et al. *ACSL: ANSI C Specification Language, version 1.12*. 2016.

[BBY17] Sandrine Blazy, David Bühler, and Boris Yakobowski. "Structuring Abstract Interpreters Through State and Value Abstractions". In: *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*. 2017, pp. 112–130. DOI: 10.1007/978-3-319-52234-0_7. URL: https://doi.org/10.1007/978-3-319-52234-0_7.

[Ber14] Dimitri P Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.

[BG00] Glenn Bruns and Patrice Godefroid. "Generalized Model Checking: Reasoning about Partial State Spaces". In: *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*. 2000, pp. 168–182. DOI: 10.1007/3-540-44618-4_14. URL: https://doi.org/10.1007/3-540-44618-4_14.

[Cac+14] David Cachera et al. "Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases". In: *Sci. Comput. Program.* 93 (2014), pp. 89–109. DOI: 10.1016/j.scico.2014.02.028. URL: https://doi.org/10.1016/j.scico.2014.02.028.

[Car08] E.R. Carbonell. "Polynomial invariant generation". http://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/list.html. 2008.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: http://doi.acm.org/10.1145/512950.512973.

[CH78] Patrick Cousot and Nicolas Halbwachs. "Automatic discovery of linear restraints among variables of a program". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1978, pp. 84–96.

[Cla+00] Edmund M. Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. 2000, pp. 154–169. DOI: 10.1007/10722167_15. URL: https://doi.org/10.1007/10722167_15.

[Con13] John H Conway. "On unsettleable arithmetical problems". In: *The American Mathematical Monthly* 120.3 (2013), pp. 192–198.

[Del+09] David Delmas et al. "Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software". In: *Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings*. 2009, pp. 53–69. DOI: 10.1007/978-3-642-04570-7_6. URL: https://doi.org/10.1007/978-3-642-04570-7_6.

[Deu03] Alain Deutsch. "Static verification of dynamic properties". In: *Polyspace white paper* (2003), p. 45.

[EE87] David Eric Edmunds and W Desmond Evans. *Spectral theory and differential operators*. Vol. 15. Clarendon Press Oxford, 1987.

[Ern+01] Michael D. Ernst et al. "Dynamically Discovering Likely Program Invariants to Support Program Evolution". In: *IEEE Trans. Software Eng.* 27.2 (2001), pp. 99–123. DOI: 10.1109/32.908957. URL: https://doi.org/10.1109/32.908957.

[Fij+17] Nathanaël Fijalkow et al. "Semialgebraic Invariant Synthesis for the Kannan-Lipton Orbit Problem". In: *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany*. 2017, 29:1–29:13. DOI: 10.4230/LIPIcs.STACS.2017.29. URL: https://doi.org/10.4230/LIPIcs.STACS.2017.29.

[Flo67] Robert W Floyd. "Assigning meanings to programs". In: *Mathematical aspects of computer science* 19.19-32 (1967), p. 1.

[Fos86] Leslie V Foster. "Rank and null space calculations using matrix decomposition without column interchanges". In: *Linear Algebra and its Applications* 74 (1986), pp. 47–71.

[GGP09] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. "The Zonotope Abstract Domain Taylor1+". In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. 2009, pp. 627–633. DOI: 10.1007/978-3-642-02658-4_47. URL: https://doi.org/10.1007/978-3-642-02658-4_47.

[Gio+08] Alain Giorgetti et al. "Verification of class liveness properties with Java modeling language". In: *IET Software* 2.6 (Dec. 2008), pp. 500–514. DOI: 10.1049/iet-sen:20080008. URL: http://dx.doi.org/10.1049/iet-sen:20080008.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. "dReal: An SMT Solver for Nonlinear Theories over the Reals". In: *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*. 2013, pp. 208–214. DOI: 10.1007/978-3-642-38574-2_14. URL: https://doi.org/10.1007/978-3-642-38574-2_14.

[Gou13] Eric Goubault. "Static Analysis by Abstract Interpretation of Numerical Programs and Systems, and FLUCTUAT". In: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. 2013, pp. 1–3. DOI: 10.1007/978-3-642-38856-9_1. URL: https://doi.org/10.1007/978-3-642-38856-9_1.

[GP15] Eric Goubault and Sylvie Putot. "A zonotopic framework for functional abstractions". In: *Formal Methods in System Design* 47.3 (2015), pp. 302–360. DOI: 10.1007/s10703-015-0238-z. URL: https://doi.org/10.1007/s10703-015-0238-z.

[GPV12] Eric Goubault, Sylvie Putot, and Franck Védrine. "Modular Static Analysis with Zonotopes". In: *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*. 2012, pp. 24–40. DOI: 10.1007/978-3-642-33125-1_5. URL: https://doi.org/10.1007/978-3-642-33125-1_5.

[GS14] Laure Gonnord and Peter Schrammel. "Abstract acceleration in linear relation analysis". In: *Sci. Comput. Program.* 93 (2014), pp. 125–153. DOI: 10.1016/j.scico.2013.09.016. URL: https://doi.org/10.1016/j.scico.2013.09.016.

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[ILR17] Hugo Illous, Matthieu Lemerre, and Xavier Rival. "A Relational Shape Abstract Domain". In: *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*. 2017, pp. 212–229. DOI: 10.1007/978-3-319-57288-8_15. URL: https://doi.org/10.1007/978-3-319-57288-8_15.

[JM09] Bertrand Jeannet and Antoine Miné. "Apron: A Library of Numerical Abstract Domains for Static Analysis". In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52. URL: https://doi.org/10.1007/978-3-642-02658-4_52.

[JSS14] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. "Abstract acceleration of general linear loops". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 2014, pp. 529–540. DOI: 10.1145/2535838.2535843. URL: http://doi.acm.org/10.1145/2535838.2535843.

[Kar76] Michael Karr. "Affine Relationships Among Variables of a Program". In: *Acta Inf.* 6 (1976), pp. 133–151. DOI: 10.1007/BF00268497. URL: https://doi.org/10.1007/BF00268497.

[Kir+15] Florent Kirchner et al. "Frama-C: A software analysis perspective". In: *Formal Asp. Comput.* 27.3 (2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7. URL: https://doi.org/10.1007/s00165-014-0326-7.

[KL80] Ravindran Kannan and Richard J. Lipton. "The Orbit Problem is Decidable". In: *Proceedings of the 12th Annual ACM Symposium on Theory of Computing, April 28-30, 1980, Los Angeles, California, USA*. 1980, pp. 252–261. DOI: 10.1145/800141.804673. URL: http://doi.acm.org/10.1145/800141.804673.

[KL86] Ravindran Kannan and Richard J. Lipton. "Polynomial-time algorithm for the orbit problem". In: *J. ACM* 33.4 (1986), pp. 808–821. DOI: 10.1145/6490.6496. URL: http://doi.acm.org/10.1145/6490.6496.

[Koc+18] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *arXiv preprint arXiv:1801.01203* (2018).

[Kov08] Laura Kovács. "Aligator: A Mathematica Package for Invariant Generation (System Description)". In: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*. 2008, pp. 275–282. DOI: 10.1007/978-3-540-71070-7_22. URL: https://doi.org/10.1007/978-3-540-71070-7_22.

[KS91] Krzysztof Kowalski and W-H Steeb. *Nonlinear dynamical systems and Carleman linearization*. World Scientific, 1991.

[LA04] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665. URL: https://doi.org/10.1109/CGO.2004.1281665.

[Lan97] Gérard Le Lann. "An analysis of the Ariane 5 flight 501 failure-a system engineering perspective". In: *1997 Workshop on Engineering of Computer-Based Systems (ECBS '97), March 24-28, 1997, Monterey, CA, USA*. 1997, pp. 339–246. DOI: 10.1109/ECBS.1997.581900. URL: https://doi.org/10.1109/ECBS.1997.581900.

[Mau04] Laurent Mauborgne. "AstrÉe: Verification of Absence of Runtime Error". In: *Building the Information Society*. Springer, 2004, pp. 385–392.

[MB08] Leonardo MendonℂCa de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 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*. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3_24.

[MBR16] Antoine Miné, Jason Breck, and Thomas W. Reps. "An Algorithm Inspired by Constraint Solvers to Infer Inductive Invariants in Numeric Programs". In: *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 2016, pp. 560–588. DOI: 10.1007/978-3-662-49498-

1_22. URL: https://doi.org/10.1007/978-3-662-49498-1_22.

[Min06] Antoine Miné. "The octagon abstract domain". In: *Higher-Order and Symbolic Computation* 19.1 (2006), pp. 31–100. DOI: 10.1007/s10990-006-8609-1. URL: https://doi.org/10.1007/s10990-006-8609-1.

[Min10] Hermann Minkowski. *Geometrie der zahlen*. Vol. 40. 1910.

[Min67] Marvin L Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967.

[Mon10] David Monniaux. "Quantifier Elimination by Lazy Model Enumeration". In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. 2010, pp. 585–599. DOI: 10.1007/978-3-642-14295-6_51. URL: https://doi.org/10.1007/978-3-642-14295-6_51.

[MS04] Markus Müller-Olm and Helmut Seidl. "A Note on Karr's Algorithm". In: *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*. 2004, pp. 1016–1028. DOI: 10.1007/978-3-540-27836-8_85. URL: https://doi.org/10.1007/978-3-540-27836-8_85.

[Nec+02] George C. Necula et al. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs". In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. 2002, pp. 213–228. DOI: 10.1007/3-540-45937-5_16. URL: https://doi.org/10.1007/3-540-45937-5_16.

[OBP16] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. "Polynomial Invariants by Linear Algebra". In: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*. 2016, pp. 479–494. DOI: 10.1007/978-3-319-46520-3_30. URL: https://doi.org/10.1007/978-3-319-46520-3_30.

[OBP17] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. "Synthesizing Invariants by Solving Solvable Loops". In: *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*. 2017, pp. 327–343. DOI: 10.1007/978-3-319-68167-2_22. URL: https://doi.org/10.1007/978-3-319-68167-2_22.

[Oli+18] Steven de Oliveira et al. "Left-eigenvectors are certificates of the Orbit Problem (to be submitted)". In: *CoRR* abs/1803.09511 (2018). arXiv: 1803.09511. URL: http://arxiv.org/abs/1803.09511.

[OPB] Steven de Oliveira, Virgile Prevosto, and Saddek Bensalem. "CaFE: un model-checker collaboratif". In: *Approches Formelles dans l'Assistance au Développement de Logiciels 2017, Proceedings* ().

[PC99] Victor Y. Pan and Zhao Q. Chen. "The Complexity of the Matrix Eigenproblem". In: *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*. 1999, pp. 507–516.

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32. URL: http://dx.doi.org/10.1109/SFCS.1977.32.

[Pri57] A. N. Prior. "Time and Modality". In: *Clarendon Press* (1957).

[RG13] Pierre Roux and Pierre-Loïc Garoche. "Integrating Policy Iterations in Abstract Interpreters". In: *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*. 2013, pp. 240–254. DOI: 10.1007/978-3-319-02444-8_18. URL: https://doi.org/10.1007/978-3-319-02444-8_18.

[Ric53] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[RK07] Enric Rodríguez-Carbonell and Deepak Kapur. "Generating all polynomial invariants in simple loops". In: *J. Symb. Comput.* 42.4 (2007), pp. 443–476. DOI: 10.1016/j.jsc.2007.01.002. URL: https://doi.org/10.1016/j.jsc.2007.01.002.

[Rou+12] Pierre Roux et al. "A generic ellipsoid abstract domain for linear time invariant systems". In: *Hybrid Systems: Computation and Control (part of CPS Week 2012), HSCC'12, Beijing, China, April 17-19, 2012*. 2012, pp. 105–114. DOI: 10.1145/2185632.2185651. URL: http://doi.acm.org/10.1145/2185632.2185651.

[Ser+12] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. 2012, pp. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[SKV17] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. "E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper)". In: *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*. 2017, pp. 164–173. URL: http://www.easychair.org/publications/paper/t6tV.

[SP11] N. Stouls and V. Prevosto. *Aoraï plug-in tutorial, version Nitrogen-20111001*. http://frama-c.com/download/frama-c-aorai-manual.pdf. Oct. 2011.

[Spo82] Fausto Spoto. "Julia: A generic static analyser for the java bytecode". In: *Part XXX*. Citeseer. 1982.

[Ste+08] William Stein et al. "Sage: Open source mathematical software". In: *7 December 2009* (2008).

[Ste89] W-H Steeb. "A note on Carleman linearization". In: *Physics Letters A* 140.6 (1989), pp. 336–338.

[SZ+65] Andrzej Schinzel, Hans Zassenhaus, et al. "A refinement of two theorems of Kronecker". In: *Michigan Math. J* 12 (1965), pp. 81–85.

[Ven12] Arnaud Venet. "The Gauge Domain: Scalable Analysis of Linear Inequality Invariants". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 2012, pp. 139–154. DOI: 10.1007/978-3-642-31424-7_15. URL: https://doi.org/10.1007/978-3-642-31424-7_15.

[Wag+06] Ferdinand Wagner et al. *Modeling software with finite state machines: a practical approach*. CRC Press, 2006.

[WBR13] John Henry Wilkinson, Friedrich Ludwig Bauer, and C Reinsch. *Linear algebra*. Vol. 2. Springer, 2013.

Appendix A

Pilat architecture

A.1 The Ring signature

```
module type Ring = sig

  (* type of elements of the ring *)
  type t


  (* neutral elements for the addition and the
     multiplication *)
  val zero : t
  val one : t

  (* Basic operations *)
  val add : t -> t -> t
  val sub : t -> t -> t
  val mul : t -> t -> t
  val div : t -> t -> t

  (* Comparison of elements *)
  val equal : t -> t -> bool
  val leq : t -> t -> bool
  val geq: t -> t -> bool
  val lt : t -> t -> bool
  val gt : t -> t -> bool
  val compare : t -> t -> int

  (* Printers *)
  val pp_print : Format.formatter -> t -> unit
  val to_str : t -> string
  val of_str : string -> t

end
```

A.2 The Matrix signature

```
module type Ring = sig

  (* type of elements of the ring *)
  type t
```

```
  (* neutral elements for the addition and the
     multiplication *)
  val zero : t
  val one : t

  (* Basic operations *)
  val add : t -> t -> t
  val sub : t -> t -> t
  val mul : t -> t -> t
  val div : t -> t -> t

  (* Comparison of elements *)
  val equal : t -> t -> bool
  val leq : t -> t -> bool
  val geq: t -> t -> bool
  val lt : t -> t -> bool
  val gt : t -> t -> bool
  val compare : t -> t -> int

  (* Printers *)
  val pp_print : Format.formatter -> t -> unit
  val to_str : t -> string
  val of_str : string -> t

end
```

A.3 The Polynomial signature

```
module type Ring = sig

  (* type of elements of the ring *)
  type t


  (* neutral elements for the addition and the
     multiplication *)
  val zero : t
  val one : t

  (* Basic operations *)
  val add : t -> t -> t
  val sub : t -> t -> t
  val mul : t -> t -> t
  val div : t -> t -> t

  (* Comparison of elements *)
  val equal : t -> t -> bool
  val leq : t -> t -> bool
  val geq: t -> t -> bool
  val lt : t -> t -> bool
  val gt : t -> t -> bool
  val compare : t -> t -> int

  (* Printers *)
  val pp_print : Format.formatter -> t -> unit
  val to_str : t -> string
```

```
  val of_str : string -> t

end
```

Appendix B

Pilat results on deterministic and non deterministic loops

B.1 Example 1

```
int main(){
  float x,y;
  while(x < 4){
    x = 0.68 * (x-y);
    y = 2*0.68*y + x;
  }

  return 1;

}
```

Invariant generated : $-cst \leqslant 1. * (x * x) + 1. * (y * y) \leqslant cst$

B.2 Dampened oscillator

```
 int main(){
  float x0,x1,tx0,tx1;
  while(1){
    tx0 = x0 + 0.01 * x1;
    tx1 = -0.1 * x0 + 0.99*x1;

    x0 = tx0;
    x1 = tx1;
  }

  return 1;

}
```

Invariant generated :
$-cst \leqslant (1. * (x1 * x0) + 10. * (x0 * x0)) + 1. * (x1 * x1) \leqslant cst$

B.3 Harmonic oscillator

```
int main(){
  float x0,x1,tx0,tx1;
  while(1){
    tx0 = 0.95 * x0 + 0.09975 * x1;
```

```
    tx1 = -0.1 * x0 + 0.95*x1;

    x0 = tx0;
    x1 = tx1;
  }
```

Invariant generated :
$-cst \leqslant 1.00250626566 * (x0 * x0) + 1. * (x1 * x1) \leqslant cst$

B.4 Symplectic SEU Oscillator

```
int main(){
  float v,x;
  while (v >= 1/2) {
    x = (1 - 0.05) * x + (0.1 - 0.00025 ) * v;
    v = -0.1 *x+(1-0.05 )* v ;
  }
}
```

Invariant generated : $-cst \leqslant 0.105 * (x * v) + 1.05 * (v * v) + 1. * (x * x) \leqslant cst$

B.5 [AGG12] filter

```
float float_interval(float, float);

int main(){
  float x,y;
  while(1){
    x = (0.75) * x - (0.125)* y;
    y = x;
  }
  return 0;

}
```

Invariants generated : $-cst \leqslant -6. * x + 1. * y \leqslant cst$

B.6 Simple filter

```
int main(){
  float x,y;
  float k;
  while(x < 4){
    k=float_interval(-0.1,0.1); /* 0 */
    x = 0.68 * (x-y) + k;
    y = 2*0.68*y + x;

  }

  return 1;

}
```

Invariant generated : $|1. * (x * x) + 1. * (y * y)| \leqslant 14.892578125$

B.7 Example 3

```
int main(){
  float s0 = 0,s1 = 0,r;
  while(1){
    r = 1.5*s0 - 0.7*s1 + float_interval(-0.1,0.1);
    s1 = s0;
    s0 = r;
  }
  return 0;
}
```

Invariant generated : $|(-2.14285714286 * (s1 * s0) + 1.42857142857 * (s0 * s0)) + 1. * (s1 * s1)| \leqslant 0.830078125;$

B.8 Linear filter

```
int main(){
  float s0 = 0,s1 = 0,r;
  int N = 50;
  while(N > 0){
    r = 1.5*s0 - 0.7*s1 + float_interval(-1.6,1.6);
    s1 = s0;
    s0 = r;
    N--;
  }

  return 0;

}
```

Invariant generated : $|(-2.14285714286 * (s0 * s1) + 1.42857142857 * (s1 * s1)) + 1. * (s0 * s0)| \leqslant 137.451171875$

B.9 Lead lag controller

```
 int main(){
  float x0p,x1p,x0,x1;
  while(1){/*
    x1 = 0.01*x0 + x1;
    x0 = 0.499*x0 - 0.05*x1 + 0.0005*x0 + float_interval(-1,1);
     */
    x0p = x0; x1p = x1;

    x0 = 0.499*x0p - 0.05*x1p + float_interval(-1,1);
    x1 = 0.010*x0p + x1p;


  }

  return 1;

}
```

Invariants generated :

$0.02 * x0 + 1. * x1 <= 70.1172$
$10. * x0 + 1. * x1 <= 20.1172$

B.10 Gaussian regulator

```
int main(){
  float x0,x1,x2,tx0,tx1,tx2,in;
  while(1){
    in = float_interval(-1,1);
    tx0 = 0.9379 * x0 - 0.0381 * x1 - 0.0414 * x2 + 0.0237 * in;
    tx1 = -0.0404 * x0 + 0.968 * x1 - 0.0179 * x2 + 0.0143 * in;
    tx2 = 0.0142 * x0 - 0.0197 * x1 + 0.9823 * x2 + 0.0077 * in;
    x0 = tx0;
    x1 = tx1;
    x2 = tx2;

  }

  return 0;

}
```

Invariants generated :
$|(1.2187798948 * x0 + 1.16137161588 * x1) + 1. * x2| \leqslant 1.171875$
$|-2.45498840354 * x1 * x0 + -0.788574527791 * x2 * x0 + 0.868152061813 * x0 * x0 + 1.12295787049 * x2 * x1 + 1.73559323995 * x1 * x1 + 1. * x2 * x2| \leqslant 10.7422$

B.11 Controller

```
int main(){
  float x0,x1,x2,x3,tx0,tx1,tx2,tx3,in0,in1;
  while(1){
    in0 = float_interval(-1,1);
    in1 = float_interval(-1,1);
    tx0 = 0.6227 * x0 + 0.3871 * x1 - 0.113 * x2 + 0.0102 * x3
        + 0.3064 * in0 + 0.1826 * in1;
    tx1 =  -0.3407 * x0 + 0.9103 * x1 - 0.3388 * x2 + 0.0649 * x3
        - 0.0054 * in0 + 0.6731 * in1;
    tx2 =  0.0918 * x0 - 0.0265 * x1 - 0.7319 * x2 + 0.2669 * x3
        + 0.0494 * in0 + 1.6138 * in1;
    tx3 = 0.2643 * x0 - 0.1298 * x1 - 0.9903 * x2 + 0.3331 * x3
        - 0.0531 * in0 + 0.4012 * in1;

    x0 = tx0;
    x1 = tx1;
    x2 = tx2;
    x3 = tx3;

  }

  return 1;

}
```

$|-0.00203592622443 * x1 * x0 + 0.0881698609486 * x2 * x0 + -0.0355121282196 * x3 * x0 + 0.000315277812672 * x0 * x0 + -0.284681200032 * x2 * x1 + 0.114660896235 *$

$x3*x1+0.00328678028134*x1*x1+-4.96561965553*x3*x2+6.16434464085*x2*x2+1.*x3*x3| \leqslant 20.166015625;$
$|-0.00861093083991*x1*x0+0.298520354653*x2*x0+-0.126853461757*x3*x0+0.00193714038684*x0*x0+-0.418765740617*x2*x1+0.190035991969*x3*x1+0.00760806829668*x1*x1+-4.0401517826*x3*x2+3.86658391065*x2*x2+1.*x3*x3| \leqslant 18.212890625;$
$|-0.0289556589751*x1*x0+0.339803909038*x2*x0+-0.218194795294*x3*x0+0.0119022421734*x0*x0+-0.413335822158*x2*x1+0.265411087703*x3*x1+0.017610761369*x1*x1+-3.11468390967*x3*x2+2.42531396428*x2*x2+1.*x3*x3| \leqslant 5.17578125;$
$|-0.0177560641098*x0+0.0573304481174*x1+-2.48280982777*x2+1.*x3| \leqslant 5005.46875;$
$|-0.109097397647*x0+0.132705543852*x1+-1.55734195483*x2+1.*x3| \leqslant 582.03125;$

B.12 Low pass filter

```
int main(){

  float x0,x1,x2,x3,x4,tx0,tx1,tx2,tx3,tx4,in0;

  while(1){
    in0 = float_interval(-1,1);
    x0 = 0.4250 * tx0 + 0.8131 * in0;
    x1 = 0.3167 * tx0 + 0.1016 * tx1 - 0.4444* tx2
   + 0.1807 * in0;
    x2 = 0.1278 * tx0 + 0.4444 * tx1 +0.8207 * tx2
   + 0.0729 * in0;
    x3 = 0.0365 * tx0 + 0.1270 * tx1 + 0.5202 * tx2
   + 0.4163 * tx3 - 0.5714 * tx4 + 0.0208 * in0;
    x4 = 0.0147 * tx0 + 0.0512 * tx1 + 0.2099 * tx2
   + 0.57104 * tx3 + 0.7694 * tx4 + 0.0084 * in0;

    tx0 = x0;
    tx1 = x1;
    tx2 = x2;
    tx3 = x3;
    tx4 = x4;
  }

}
```

Invariants generated : $|1.00164325052*tx1*tx0+0.000140144389337*tx2*tx0+-1.6191550573*tx3*tx0+-1.00126210564*tx4*tx0+0.72511353991*tx0*tx0+2.62046247703*tx2*tx1+-0.618060865742*tx3*tx1+-2.00110233268*tx4*tx1+1.00110265182*tx1*tx1+1.00020053093*tx3*tx2+-2.61878326237*tx4*tx2+2.61995131748*tx2*tx2+0.617955897795*tx4*tx3+0.999369968498*tx3*tx3+1.*tx4*tx4| <= 17.724609375;$
$|1.00123175519*tx1*tx0+-0.999123490687*tx2*tx0+2.61966963703*tx0*tx0+1.61813681368*tx2*tx1+1.*tx1*tx1+1.*tx2*tx2| <= 8.642578125;$

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Recherche de constance dans les routines linéaires

Mots-clefs : Vérification de programmes, génération d'invariants, model-checking, propriétés temporelles

Résumé : La criticité des programmes dépasse constamment de nouvelles frontières car l'informatique est de plus en plus utilisée dans la prise de décision (voitures autonomes, robots chirurgiens, etc.). Développer des programmes sûrs et vérifier les programmes existants est devenu indispensable.
Afin de vérifier formellement le bon fonctionnement d'un programme donné, il faut faire face aux défis de la mise à l'échelle et de la décidabilité. Programmes composés de millions de lignes de code, complexité de l'algorithme, concurrence, et même de simples expressions polynomiales font partis des problèmes que la vérification formelle doit savoir gérer. Pour y arriver, les méthodes formelles travaillent sur des abstractions des programmes étudiés afin d'analyser des approximations de leur comportement.
L'analyse des boucles est un axe entier de la vérification formelle car elles sont encore aujourd'hui peu comprises. Certaines d'entre elles peuvent facilement être traitées, pourtant il existe des exemples apparemment très simples mais dont le comportement n'a encore aujourd'hui pas été résolu (on ne sait toujours pas pourquoi la suite de Syracuse, simple boucle linéaire, converge toujours vers 1). L'approche la plus commune afin de gérer les boucles de manière approchée est l'utilisation d'invariants de boucles, c'est à dire de relations sur les variables manipulées par une boucle qui sont vraies à chaque fois que la boucle recommence.
En général, les invariants utilisent des expressions similaires à celles utilisées dans la boucle : si elle manipule explicitement la mémoire par exemple, on s'attend à utiliser des invariants portant sur la mémoire. Cependant, il existe des boucles contenant uniquement des affectations linéaires qui n'admettent pas d'invariants linéaires, mais polynomiaux.
Cette thèse présente de nouvelles propriétés sur les boucles linéaires et polynomiales. Il est déjà connu que les boucles linéaires sont polynomialement expressives, au sens ou si plusieurs variables évoluent linéairement dans une boucle, alors n'importe quel monôme de ces variables évolue linéairement. La première contribution de cette thèse est la caractérisation d'une classe de boucles polynomiales équivalentes aux boucles linéaires, au sens où il existe une boucle linéaire avec le même comportement.
Ensuite, deux nouvelles méthodes de génération d'invariants sont présentées. La première méthode est basée sur l'interprétation abstraite et s'intéresse aux filtres linéaires convergents. Ces filtres jouent un rôle important dans de nombreux systèmes embarqués (par exemple dans l'avionique) et requièrent l'utilisation de flottants, un type de valeurs qui peut mener à des erreurs d'imprécision. Aussi, la présence d'affectations aléatoires dans ces filtres rend leur analyse encore plus complexe.
La seconde méthode traite d'une approche basée sur la génération d'invariants pour n'importe quel type de boucles linéaires. Elle part d'un nouveau théorème présenté dans cette thèse qui caractérise les invariants de boucles comme étant les vecteurs propres du dual de la transformation linéaire traitée. Cette méthode est généralisée pour prendre en compte les conditions, les boucles imbriquées et le non déterminisme dans les affectations.
La génération d'invariants n'est pas un but en soi, mais un moyen. Cette thèse s'intéresse au genre de problèmes que peut résoudre les invariants générés par la seconde méthode. Le premier problème traité est problème de l'orbite (Kannan-Lipton Orbit problem), dont il est possible de générer des certificats de non accessibilité en utilisant les vecteurs propres de la transformation considérée. En outre, les vecteurs propres sont mis à l'épreuve en pratique par leur utilisation dans le model-checker CaFE basé sur la vérification de propriétés temporelles sur des programmes C.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title : Finding constancy in linear routines

Keywords : Program verification, invariant generation, model-checking, temporal properties

Abstract : The criticality of programs constantly reaches new boundaries as they are relied on to take life-or-death decisions in place of the user (autonomous cars, robot surgeon, etc.). This raised the need to develop safe programs and to verify the already existing ones. Anyone willing to formally prove the soundness of a program faces the two challenges of scalability and undecidability. Million of lines of code, complexity of the algorithm, concurrency, and even simple polynomial expressions are part of the issues formal verification have to deal with. In order to succeed, formal methods rely on state abstraction to analyze approximations of the behavior of the analyzed program.

The analysis of loops is a full axis of formal verification, as this construction is still today not well managed. Though some of them can be easily handled when they perform simple operations, there still exist some seemingly basic loops whose behavior has not been solved yet (the Syracuse sequence for example is suspected to be undecidable). The most common approach for the treatment of loops is the use of loop invariants, i.e. relations on variables that are true at the beginning of the loop and after every step.

Intuitively, invariants are expected to use the same set of expressions used in the loop: if a loop manipulates the memory on a structure for example, invariants will naturally use expressions involving memory operations. However, there exist loops containing only linear instructions that admit only polynomial invariants (for example, the sum on integers can be computed by a linear loop and is a degree 2 polynomial in n), hence using expressions that are syntactically absent of the loop. The intuition stated above is thus a bit naive and we should seek for more relations between invariants and loop instructions.

This thesis presents new insights on loops containing linear and polynomial instructions. It is already known that linear loops are polynomially expressive, in the sense that if a variable evolves linearly, then any monomial of this variable evolves linearly. The first contribution of this thesis is the extraction of a class of polynomial loops that is exactly as expressive as linear loops, in the sense that there exists a linear loop with the exact same behavior.

Then, two newmethods for generating invariants are presented.

The first method is based on abstract interpretation and is focused on a specific kind of linear loops called linear filters. Linear filters play a role in many embedded systems (plane sensors for example) and require the use of floating point operations, that may be imprecise and lead to errors if they are badly handled. Also, the presence of non deterministic assignments makes their analysis even more complex.

The second method treats of a more generic subject by finding a complete set of linear invariants of linear loops that is easily computable. This technique is based on the linear algebra concept of eigenspace. It is extended to deal with conditions, nested loops and non determinism in assignments.

Generating invariants is an interesting topic, but it is not an end in itself, it must serve a purpose. This thesis investigates the expressivity of invariants generated by the second method by generating counter examples for the Kannan-Lipton Orbit problem. It also presents the tool PILAT implementing this technique and compares its efficiency technique with other state-of-the-art invariant synthesizers. The effective usefulness of the invariants generated by PILAT is demonstrated by using the tool in concert with CaFE, a model-checker for C programs based on temporal logics.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France