Fast inference of polynomial invariants for imperative programs

David Cachera, Thomas Jensen, Arnaud Jobin, Florent Kirchner

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

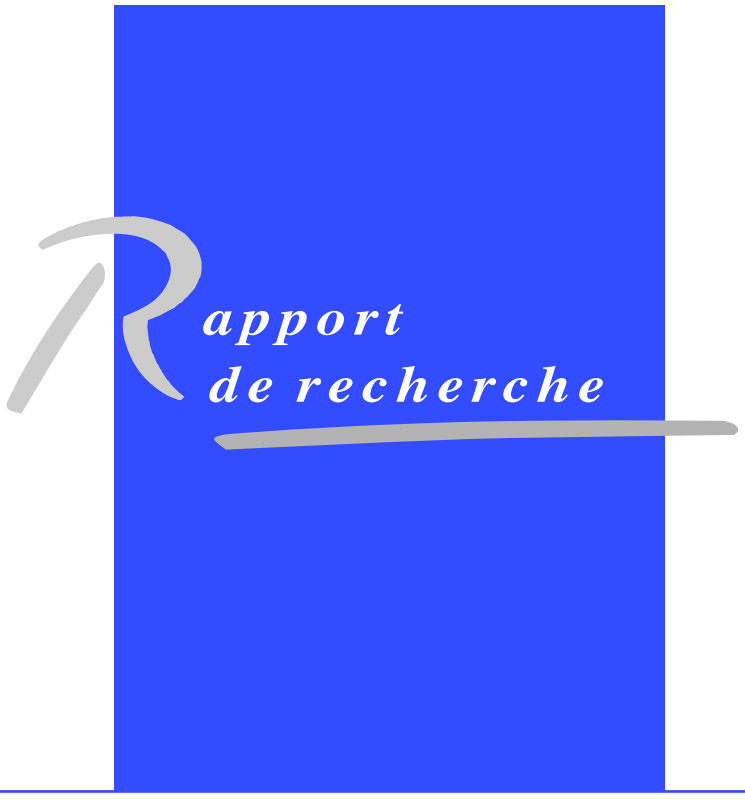
Fast inference of polynomial invariants for imperative programs

David Cachera — Thomas Jensen — Arnaud Jobin — Florent Kirchner

N° 7627

Mai 2011

Domaine 2



ISSN 0249-6399 ISRN INRIA/RR--7627--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Fast inference of polynomial invariants for imperative programs

David Cachera*, Thomas Jensen†, Arnaud Jobin‡, Florent Kirchner†

Domaine : Algorithmique, programmation, logiciels et architectures
Algorithmics, Programming, Software and Architecture
Équipe-Projet Celtique

Rapport de recherche n° 7627 — Mai 2011 — 31 pages

Abstract: We propose an abstract interpretation based method to compute polynomial invariants for imperative programs. Our analysis is a backward propagation approach that computes preconditions for equalities like $g = 0$ to hold at the end of execution. It extends previous work by Müller-Olm and Seidl to a language that includes both polynomial equalities and disequalities. Properties are expressed using ideals, a structure that satisfies the descending chain condition, enabling fixpoints computations to terminate without use of a widening operator. In the general case, termination is characterized using ideal membership tests and Gröbner bases computations. In order to optimize computational complexity, we propose a specialized analysis dealing with inductive invariants which ensures fast termination of fixpoints computations. The optimized procedure has been shown by experiments to work well in practice, and to be two orders of magnitude faster than the state of the art analyzer of Carbonell and Kapur.

Key-words: Static Analysis, Automatic Invariant Generation, Abstract Interpretation, Polynomial Invariants, Polynomial Ideals

* ENS Cachan Bretagne/IRISA
† INRIA Rennes - Bretagne Atlantique
‡ Université Rennes 1/IRISA

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Inférence rapide d'invariants polynomiaux pour programmes impératifs

Résumé : Nous proposons une méthode fondée sur la théorie de l'interprétation abstraite pour calculer des invariants polynomiaux de programmes impératifs. Notre analyse statique suit une approche arrière, qui calcule des préconditions pour qu'une égalité du type $g = 0$ soit vérifiée à la fin de l'exécution d'un programme. Cette approche étend des travaux précédents de Müller-Olm et Seidl à un langage qui inclut à la fois des égalités et des inégalités polynomiales. Les propriétés sont exprimées à l'aide d'idéaux de polynômes. La propriété de chaîne descendante de l'ensemble des idéaux garantit alors la terminaison des calculs de point fixe sans avoir besoin de faire appel à un opérateur d'élargissement. Dans le cas général, cette terminaison est caractérisée par un test d'inclusion reposant sur le calcul de bases de Gröbner. Afin d'améliorer la complexité, nous proposons une analyse spécialisée dans la découverte d'invariants inductifs, ce qui permet une terminaison rapide des calculs. Les expérimentations que nous avons menées montrent que cette procédure fournit des résultats satisfaisants en pratique, avec un temps de calcul de deux ordres de grandeur plus court qu'un analyseur à l'état de l'art, notamment celui de Carbonell et Kapur.

Mots-clés : Analyse statique, génération automatique d'invariants, interprétation abstraite, invariants polynomiaux, idéaux de polynômes

1 Introduction

The problem of automatically inferring polynomial (non-linear) invariants of programs is currently one of the major challenges in program verification. This is in contrast to the case for linear invariants where the initial work by Karr [8] and Cousot and Halbwachs [5] has lead to efficient implementations based on variants of the polyhedral domain.

Initial approaches to the non-linear problem proposed extensions of linear techniques [8, 5, 3, 18] to the non-linear case, made possible by imposing restrictions on the programming language. Sankaranarayanan et al. [19] proposed a constraint-based technique that generates non-linear invariants for programs containing only equality tests. A concomitant approach by Müller-Olm and Seidl [11, 12] defines an abstract interpretation method that can generate polynomial invariants for a restrictive class of guarded-loop programs where tests in conditionals are polynomial disequalities. Their analysis is a backward propagation based method: they start from p and compute the weakest preconditions of the satisfiability of the relation $p = 0$. More precisely, in order to prove that a polynomial relation $p = 0$ is valid at the end of a program, they show that the set of zeroes of a polynomial p can be exactly abstracted by a polynomial ideal. The restrictions imposed on the language are sufficiently strong to ensure that their method can be proven complete.

Rodríguez-Carbonell and Kapur develop another approach, casting the analysis as an abstract interpretation problem over a domain of ideals of variety. Their approach gave rise to two different procedures. The first one [14, 15, 17] consists in a complete method for computing all polynomial invariants for a rather restrictive class of programs: it considers simple loops with no conditional tests and whose assignments functions are required to be so called solvable mappings, *i.e.* a generalization of affine mappings. Thanks to these limitations, the generation procedure is proved to terminate without making use of a widening operator, with a number of iterations bounded by the number of variables of the program. In the second procedure [13, 16], these restrictions are removed and programs with polynomial equalities or disequalities guards are considered. As a result, their method is no more complete and may not terminate. To ensure termination, the authors use a widening operator that consists in dynamically removing polynomials of a degree greater than a fixed number. It was an open question at the time whether one could devise an analysis that conciliates both expressiveness and efficiency.

In this paper, we propose an abstract interpretation based method [4] to compute polynomial invariants that has both of the previous properties. Our analysis is a backward propagation approach that extends Müller-Olm and Seidl work [12] to a *general polynomial structured programming language* that includes **if** and **while** constructs where branching conditions are both polynomial equalities and inequalities. As in this previous approach, our analysis uses a form of weakest precondition calculus for showing that a polynomial relation $g = 0$ holds at the end of a program. Similarly, this verification scheme that applies to a fixed known polynomial can be derived in a generation scheme by using parametrized polynomials. Preconditions are expressed in the mathematical structure of ideals that satisfies the descending chain condition. We show that the backward approach, well adapted to polynomial disequality guards as Müller-Olm and Seidl already noticed, can be extended to equality guards by

```
1.  y1 := 0; y2 := 0; y3 := x1;
2.  while y3 ≠ 0 do
3.    if x2 = y2 + 1 then
4.      y1 := y1 + 1;   y2 := 0;   y3 := y3 − 1;
5.    else
6.      y2 := y2 + 1;   y3 := y3 − 1;
7.
```

(a) `mannadiv`.

```
1.  x := 0;
2.  y := n;
3.  while y ≠ 0 do
4.      x := x + n;
5.      y := y − 1;
6.
```

(b) `small`.

Figure 1: Two polynomial programs.

using polynomial division and greatest fixpoint computations which corresponds to a weakest liberal precondition calculus. Focusing on partial correctness has the consequence that *fixpoint iterations in the set of ideals are proved to terminate without the need for a widening operator*, as opposed to the general setting of Rodríguez-Carbonell and Kapur. In the general case, termination can be tested by solving the ideal membership problem, namely by Gröbner base computations. In order to reduce computational complexity, *we propose a specialized analysis that infers inductive invariants*. This restriction, similar to the one made by Sankaranarayanan *et al.* [19], presents the main advantage of ensuring termination of fixpoint computations in only one iteration. We have shown that it successfully infer invariants on a sizeable set of examples and gives excellent performance results.

The following example demonstrates the salient features of the approach. The algorithm in Figure 1a computes the Euclidean division of $\mathtt{x}_1$ by $\mathtt{x}_2$ [10].

The invariant computed for this example is non-linear, namely $\mathtt{y}_1 * \mathtt{x}_2 + \mathtt{y}_2 + \mathtt{y}_3 = \mathtt{x}_1$. Finding this invariant requires examining both branchs of the `if` construct, taking into account the information obtained by the guard. Our method is able to compute such invariants, and the relative simplicity of this example is reflected by the fact that we easily discover it, since we focus on inductive invariants. We thus need only one iteration for computing the invariant, on the contrary to Rodríguez-Carbonell and Kapur's approach that requires an heavier iterating method. Because the analysis by Sankaranarayanan *et al.* [19] has not been implemented, its performance on this type of programs is unknown.

The rest of the paper is organized as follows. In Section 2, we introduce the mathematical background needed throughout the paper, in particular multivariate polynomial algebra, division and remainder operators and the lattice structure of ideals. Section 3 defines the class of programs we are interested in, namely *polynomial programs*, as well as their backward collecting semantics. In Section 4, we dive into the analysis by defining the abstract semantics of polynomial programs and prove its correctness w.r.t. the concrete semantics.

Section 5 shows how the abstract semantics is used to automatically generate polynomial invariants. Finally, in Section 6 we propose a heuristic to drastically reduce computational complexity by making a natural assumption on invariants. We report on our implementation through a collection of benchmarks from the litterature.

2 Preliminaries

Our analysis builds a set of polynomial invariants for a given program. We consider polynomials in $\mathbb{R}[\mathtt{x_1}, \ldots, \mathtt{x_m}]$ where m represents the number of variables of the program[1]. In the rest of the paper, we will distinguish between x, element of $\mathbb{R}^m$, x_i element of $\mathbb{R}$, and $\mathtt{x_i}$ variable of the program.

A set of polynomial invariants $\{p_1 = 0, \ldots, p_s = 0\}$ naturally has an algebraic structure of ideal, as recalled by the following definition.

Definition 1 (Polynomial ideal) *A set $I \subseteq \mathbb{R}[\mathtt{x_1}, \ldots, \mathtt{x_m}]$ is a polynomial ideal if it contains 0, is stable under addition (if $p_1, p_2 \in I$ then $p_1 + p_2 \in I$) and stable under external multiplication (if $q \in \mathbb{R}[\mathtt{x_1}, \ldots, \mathtt{x_m}]$ and $p \in I$ then $q.p \in I$). We write $\mathcal{I}$ for the set of polynomial ideals of $\mathbb{R}[\mathtt{x_1}, \ldots, \mathtt{x_m}]$, and $<S>$ for the polynomial ideal generated by a set S of polynomials. By definition, $<S>$ is the smallest ideal containing all polynomials of S.*

The set $\mathcal{I}$ can be given a partial order structure by using the *reverse subset inclusion* between ideals. The upper bound of a set of polynomial ideals is then the intersection of its elements, while the lower bound is the ideal generated by the union of the elements [2].

Definition 2 (Lattice structure of $\mathcal{I}$) *Given I and J two polynomial ideals, we define $I \bigsqcup^\sharp J = I \bigcap J$, $I \bigsqcap^\sharp J = <I \bigcup J>$ and $\sqsubseteq^\sharp = \supseteq$. Operators $\bigsqcup^\sharp$ and $\bigsqcap^\sharp$ are extended in a standard fashion to range over sets of polynomial ideals. Equipped with these operators, $\mathcal{I}$ is a complete lattice, where the least element is $\bot^\sharp = <1>$ and the greatest element is $\top^\sharp = <0>$.*

One of the main benefit using polynomial ideals is that they are finitely generated. This property is known in the litterature as *Hilbert's theorem*.

Theorem 1 (Hilbert) *Every polynomial ideal $I \in \mathcal{I}$ is finitely generated, i.e., $I = <S>$ for a finite subset S of I.*

A standard result of multivariate polynomial theory states that Hilbert's theorem is equivalent to the fact that the set $\mathcal{I}$ satisfies the descending chain condition [6]. This is a key property that will be used to ensure termination of our analysis. Theorem 1 above also exhibits the tight link that exists between invariant sets and ideal structure. We have already seen that an invariant set can naturally be represented as an ideal. Conversely, any polynomial ideal can thus be represented by a finite set of polynomials, that can be seen as an invariant set. Therefore, in the rest of the paper, we will make no difference between an invariant set and the ideal generated by these invariants.

[1] A careful reader will see that our analysis can be set in any $\mathbb{F}[\mathtt{x_1}, \ldots, \mathtt{x_m}]$ where $\mathbb{F}$ is a noetherian ring, *i.e.* a ring satisfiying the ascending chain condition on its ideal set.

[2] The union set of two ideals is not an ideal in general.

The notion of *division* on multivariate polynomial ring will play an important role when defining the analysis. Contrary to the univariate case, $\mathbb{R}[\mathtt{x}_1, \ldots, \mathtt{x}_m]$ is not equipped with a Euclidean division. However, it is easy to define a division according to a monomial ordering [6]. In our case, we define a general division operator as follows.

Definition 3 (Division operator, remainder) *A* division operator $\mathtt{div}$ *is a function mapping a pair of polynomials* $(g, p) \in \mathbb{R}[\mathtt{x}_1, \ldots, \mathtt{x}_m]^2$ *to a pair* $(q, r) \in \mathbb{R}[\mathtt{x}_1, \ldots, \mathtt{x}_m]^2$ *such that* $g = pq + r$*. Polynomial* r *is called the* remainder *of* g *by* p *according to* $\mathtt{div}$*, and is noted* $\mathtt{Rem}(g, p, \mathtt{div})$ *or only* $\mathtt{Rem}(g, p)$ *if the division operator doesn't need to be explicitly given.*

Note that the usual divisions defined by a monomial ordering are division operators.

Finally, our concrete semantics will handle subsets of $\mathbb{R}^m$ [3] whereas our abstract semantics will deal with polynomial ideals. The link between these two domains is given by the following Galois connection $\mathcal{P}(\mathbb{R}^m) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{I}$:

$$\begin{array}{lrcl} \alpha: & \mathcal{P}(\mathbb{R}^m) & \to & \mathcal{I} \\ & X & \mapsto & \{u \in \mathbb{R}[\mathtt{x}_1, \ldots, \mathtt{x}_m] \mid \forall x \in X,\ u(x) = 0\} \\ \gamma: & \mathcal{I} & \to & \mathcal{P}(\mathbb{R}^m) \\ & I & \mapsto & \{x \in \mathbb{R}^m \mid \forall u \in I,\ u(x) = 0\} \end{array}$$

such that $\forall X \in \mathcal{P}(\mathbb{R}^m), \forall I \in \mathcal{I} :\ X \sqsubseteq \gamma(I) \Leftrightarrow \alpha(X) \sqsubseteq^\sharp I$.

3 Syntax and semantics of polynomial programs

Our analysis produces invariants of *polynomial programs*. We consider a variant of the IMP language [20] where assignments are polynomial and conditional tests are polynomial (dis)equalities.

Definition 4 (Syntax of polynomial programs)

$p \in \mathbb{R}[\mathtt{x}_1, \ldots, \mathtt{x}_m]$			*polynomials*
$\mathbb{V} \ni var$	$::=$	$\mathtt{x}_1 \mid \ldots \mid \mathtt{x}_m$	*program variables*
$\mathbb{T} \ni test$	$::=$	$p \bowtie 0$	*polynomial guards*
$\mathbb{P} \ni c$	$::=$	$var := p$	*polynomial assignments*
		$\mid c\ ;\ c$	*sequence*
		$\mid$ ***if*** $test$ ***then*** c ***else*** c	*conditional structure*
		$\mid$ ***while*** $test$ ***do*** c	*loop structure*
		$\mid$ ***skip***	*skip assertion*

where $\bowtie$ *stands for* $=$ *or* $\neq$*. We will also use* $\not\bowtie$ *for the negation of* $\bowtie$*.*

The concrete syntax will be decorated by line numbers in the implementation, which are omitted in the abstract syntax above. This language abides to the usual small-step operational semantics (SSOS). We will note $\to$: $\mathbb{P} \times \mathbb{R}^m \to$

[3] Recall that $\mathcal{P}(\mathbb{R}^m)$ is naturally equipped with a complete lattice structure $(\mathcal{P}(\mathbb{R}^m), \subseteq, \bigcup, \bigcap)$ where $\subseteq$, $\bigcup$ and $\bigcap$ represent respectively the inclusion, union and intersection of sets.

$\mathbb{P} \times \mathbb{R}^m$ this relation and $\rightarrow^+$ its transitive closure; the reader can refer to Appendix A.1 for details. The operational semantics allows us to define the following notions, linked to reachability.

Definition 5 (Terminating state) *A state $\sigma \in \mathbb{R}^m$ is called* terminating *for a program c if there exists a state $\sigma_f \in \mathbb{R}^m$ such that $\langle c, \sigma\rangle \rightarrow^+ \sigma_f$. A program c is said to be terminating if each state $\sigma \in \mathbb{R}^m$ is terminating for c.*

A polynomial is said to be invariant for a program if all its terminating states lead to states that are zeroes of the polynomial.

Definition 6 (Polynomial invariant) *A polynomial $g \in \mathbb{R}[\mathtt{x_1}, \ldots, \mathtt{x_m}]$ is said to be* invariant *at the end of a program c if, for all $\sigma \in \mathbb{R}^m$,*

$$\exists\ \sigma_f \in \mathbb{R}^m, \langle c, \sigma\rangle \rightarrow^+ \sigma_f \quad \Rightarrow \quad \sigma_f \in \gamma(g)$$

Note that, for programs with no terminating state, every polynomial will be invariant, which indicates the absence of information for such programs.

Backward semantics. As standard for abstract interpretation methods, we define a concrete semantics that we will use as an intermediate between the abstract semantics and the usual operational semantics. Its backward formulation will come in handy to prove the correctness of the analysis.

Definition 7 (Backward collecting semantics (BCS))

$B^\nu[\![c]\!] \ : \mathcal{P}(\mathbb{R}^m) \rightarrow \mathcal{P}(\mathbb{R}^m)$

$B^\nu[\![\mathtt{x_j} := p]\!]\ S = \{x \in \mathbb{R}^m \mid x[p(x)]_\mathtt{j} \in S\}$
where $x[p(x)]_\mathtt{j}$ is the element $(x_1, \ldots, x_{j-1}, p(x), x_{j+1}, \ldots, x_m)$

$B^\nu[\![\textbf{\textit{skip}}]\!]\ S = S$

$B^\nu[\![c_1; c_2]\!]\ S = B^\nu[\![c_1]\!]\ (B^\nu[\![c_2]\!]\ S)$

$B^\nu[\![\textbf{\textit{if}}\ p \bowtie 0\ \textbf{\textit{then}}\ c_1\ \textbf{\textit{else}}\ c_2]\!]\ S = (B^\nu[\![c_1]\!]\ S \bigcap [\![p \bowtie 0]\!]) \bigcup (B^\nu[\![c_2]\!]\ S \bigcap [\![p \not\bowtie 0]\!])$
where $[\![p \bowtie 0]\!] = \{x \in \mathbb{R}^m \mid p(x) \bowtie 0\}$

$B^\nu[\![\textbf{\textit{while}}\ p \bowtie 0\ \textbf{\textit{do}}\ c]\!]\ S = \nu F_{c,p,S}$
where $F_{c,p,S} = \lambda X.([\![p \not\bowtie 0]\!] \bigcap S) \bigcup ([\![p \bowtie 0]\!] \bigcap B^\nu[\![c]\!]\ X)$

Remark that this intermediate semantics has a greatest fixpoint definition. Compared to a least fixpoint, this provides less information, but the extra information a least fixpoint would give has to do with termination and is not essential to our analysis. In other words, we only deal with partial correctness. In order to investigate this issue, define the operator $\mathrm{B}^\mu[\![.]\!]$ by replacing greatest fixpoint computations by least fixpoints, and consider the following example.

Computing $\mathrm{B}^\mu[\![\mathtt{small}]\!]$ and $\mathrm{B}^\nu[\![\mathtt{small}]\!]$. The program in Figure 1b computes the square of a natural number n by successively adding the content of the variable $\mathtt{n}$ to a variable $\mathtt{x}$ initially containing 0. Thus, the polynomial $g = \mathtt{x} - \mathtt{n}^2$ is invariant at the end of the program. The weakest precondition of relation $g = 0$ holding at the end of the program is given by $\mathrm{B}^\mu[\![\mathtt{small}]\!]\ \gamma(g)$. We start by computing $\mathrm{B}^\mu[\![\textbf{while}\ \mathtt{y} \neq 0\ \textbf{do}\ \mathtt{x} := \mathtt{x} + \mathtt{n}; \mathtt{y} := \mathtt{y} - 1]\!]\ \gamma(g)$, obtaining the

least fixpoint by iterating the function $F = \lambda X.([\![\mathtt{y} \neq 0]\!] \bigcap \gamma(g)) \bigcup ([\![\mathtt{y} = 0]\!] \bigcap \mathrm{B}^{\mu}[\![c]\!]\ X)$ from $\{\}$.

$$\mathrm{B}^{\mu}[\![\textbf{while } \mathtt{y} \neq 0 \textbf{ do } \mathtt{x} := \mathtt{x} + \mathtt{n}; \mathtt{y} := \mathtt{y} - 1]\!]\ \gamma(g) \ = \bigcup\nolimits_{k \geq 0} F^k(\{\})$$

$$\begin{array}{lll} \text{where } F^k(\{\}) \ = \ \{(x,y,n) \in \mathbb{R}^3 \mid & & (y = 0 \wedge \ x = n^2) \\ & \vee & (y = 1 \ \wedge \ x + n = n^2) \\ & \vee & \ldots \\ & \vee & (y = k \ \wedge \ x + kn = n^2)\} \end{array}$$

Thus, we have

$$\begin{array}{rcl} \mathrm{B}^{\mu}[\![\mathtt{small}]\!]\ \gamma(g) & = & \mathrm{B}^{\mu}[\![\mathtt{x} := 0; \mathtt{y} := \mathtt{n}]\!]\ (\ \bigcup_{k \geq 0} F^k(\{\})\ ) \\ & = & \{(x,y,n) \in \mathbb{R}^3 \mid \exists k \in \mathbb{N} \text{ such that } (0,n,n) \in F^k(\{\})\} \\ & = & \{(x,y,n) \in \mathbb{R}^3 \mid \exists k \in \mathbb{N} \text{ such that } n = k\} \\ & = & \mathbb{R} \times \mathbb{R} \times \mathbb{N} \end{array}$$

This proves that, if initial condition $\sigma_i = (x,y,n) \in \mathbb{R} \times \mathbb{R} \times \mathbb{N}$ is satisfied, we eventually reach the end of the program in a state σ_f such that $g(\sigma_f) = 0$. Condition $x \in \mathbb{R}$ can easily be expressed with a polynomial (as $\{x \in \mathbb{R} \mid 0 = 0\}$). On the contrary, $n \in \mathbb{N}$ needs an infinite polynomial product to be expressed in $\mathbb{R}$ (as $\{n \in \mathbb{R} \mid \prod_{k \geq 0}(n-k) = 0\}$) which would make our analysis diverge.

On the other hand, we have $\mathrm{B}^{\nu}[\![\mathtt{small}]\!]\ \gamma(g) = \mathbb{R}^3$. This computation is an over-approximation of the least fixpoint and its knowledge may seem not to bring information. However, this must be read as: "relation $g = 0$ holds *for every final states* under no initial condition".

To sum up, a least fixpoint computation provides very precise information including termination conditions that have to be fulfilled for the invariant to hold at the end of the program. On the contrary, greatest fixpoint computations brings less precise but more relevant information : it reverses the point of view by focusing on relations satisfied by only final states. We traduce these practical observations in a formal way in the following paragraph.

Comparing backward semantics with `SSOS`. In order to stress the termination condition implicitly expressed in our concrete semantics, we compare least and greatest fixpoint definitions to `SSOS`. Theorem 2 states that least fixpoint semantics collects all terminating states.

Theorem 2 ($\mathrm{B}^{\mu}[\![.]\!]$ vs `SSOS`) *For all $S \in \mathcal{P}(\mathbb{R}^m)$ and for all polynomial program state $\sigma \in \mathbb{R}^m$,*

$$\sigma \in B^{\mu}[\![c]\!]\ S \ \Leftrightarrow \ \exists \sigma_f \in S, \langle c, \sigma \rangle \rightarrow^{+} \sigma_f$$

The greatest fixpoint semantics also contains the set of terminating states but may also contains non-terminating ones.

Theorem 3 ($\mathrm{B}^{\nu}[\![.]\!]$ vs `SSOS`) *For all $S \in \mathcal{P}(\mathbb{R}^m)$ and for all polynomial program state $\sigma \in \mathbb{R}^m$,*

$$\sigma \in B^{\nu}[\![c]\!]\ S \ \Rightarrow \ \left\{ \begin{array}{rl} \exists \sigma_f \in S, & \langle c, \sigma \rangle \rightarrow^{+} \sigma_f \\ \text{or} & \langle c, \sigma \rangle \rightarrow^{\infty} \end{array} \right.$$

The proofs of both theorems can be found in Appendix A.1. Obviously, if we only consider terminating programs, there is no difference between $\mathrm{B}^{\mu}[\![.]\!]$ and $\mathrm{B}^{\nu}[\![.]\!]$.

4 Abstract semantics

The fundamental idea behind abstraction is to correctly export concrete analysis in a coarser world were computations are easier. In our case, the concrete semantics is not computable because of the presence of fixpoint computations in the infinite lattice $\mathbb{R}^m$ that does not satisfy the ascending chain condition. In this section, we propose an abstract semantics that approximates the concrete semantics and whose underlying workset is $\mathcal{I}$, the set of polynomials ideals. As $\mathcal{I}$ is a lattice satisfying the descending chain condition, greatest fixpoints are computable.

We begin by defining the abstract semantics of polynomial programs and then prove that this abstraction correctly approximates the concrete semantics.

Definition 8 (Abstraction of polynomial programs)

$[\![c]\!]^\sharp : \mathcal{I} \to \mathcal{I}$

$[\![\mathtt{x_j} := p]\!]^\sharp I = \{q[\mathtt{x_j} \mapsto p], q \in I\}$
where $q[\mathtt{x_j} \mapsto p]$ *is the polynomial* $q(\mathtt{x_1}, \ldots, \mathtt{x_{j-1}}, p(\mathtt{x_1}, \ldots, \mathtt{x_m}), \mathtt{x_{j+1}}, \ldots, \mathtt{x_m})$

$[\![\textbf{\textit{skip}}]\!]^\sharp I = I$

$[\![s_1; s_2]\!]^\sharp I = [\![s_1]\!]^\sharp([\![s_2]\!]^\sharp I)$

$[\![\textbf{\textit{if}}\ p \neq 0\ \textbf{\textit{then}}\ c_1\ \textbf{\textit{else}}\ c_2]\!]^\sharp I = <p.([\![c_1]\!]^\sharp I), \mathtt{Rem}([\![c_2]\!]^\sharp I, p)>$

$[\![\textbf{\textit{if}}\ p = 0\ \textbf{\textit{then}}\ c_1\ \textbf{\textit{else}}\ c_2]\!]^\sharp I = <p.([\![c_2]\!]^\sharp I), \mathtt{Rem}([\![c_1]\!]^\sharp I, p)>$

$[\![\textbf{\textit{while}}\ p \neq 0\ \textbf{\textit{do}}\ c]\!]^\sharp I = \nu(F^\sharp_{c,p,I})$
where $F^\sharp_{c,p,I} = \lambda J.<p.([\![c]\!]^\sharp J), \mathtt{Rem}(I, p)>$

$[\![\textbf{\textit{while}}\ p = 0\ \textbf{\textit{do}}\ c]\!]^\sharp I = \nu(\overline{F}^\sharp_{c,p,I})$
where $\overline{F}^\sharp_{c,p,I} = \lambda J.<p.I, \mathtt{Rem}([\![c]\!]^\sharp J, p)>$

A few comments on this semantics: first, note that as our analysis acts backward, *assignment*-semantics only consists in a substitution and does not require the use of fresh variables. Let us elaborate on the presence of the Rem-operator in the semantics of the **if** construct. Considering an **if** statement guarded by a (dis)equality p, this semantics is based on the following simple idea: if we want to prove that relation $g = 0$ holds and we know that relation $p = 0$ holds, it suffices to compute $\mathtt{Rem}(g, p) = g - pq$ for a certain polynomial q and to prove that relation $\mathtt{Rem}(g, p) = 0$ holds. We point out that this property does not depend on the choice of q; in particular, this choice does not impact the correctness of our approach. However, some choices are not really relevant and may later lead to the generation of the trivial invariant "$0 = 0$". Additional details on how to find a suitable division operator are given in Section 6.

Remark 1 *In the special case of a program* $c \equiv$ ***if*** $p \neq 0$ ***then*** c_1 ***else*** *skip, the best choice for a quotient is given by* $q = 0$*, which defines* $\mathtt{Rem}(g, p) = g$*. The abstract semantics of* c *is then given by* $[\![c]\!]^\sharp I = <I, p.([\![c_1]\!]^\sharp I)>$*. This coincides with Müller-Olm and Seidl's abstract function, which is proven to be an exact abstraction of the concrete transfer function [12].*

The semantics for **while** is derived from that of **if**. Note that, similarly to the concrete semantics, it is defined by a greatest fixpoint definition. The

abstract transfer function for **while** is computed with a Kleene fixpoint iteration starting from $\top^\sharp = <0>$, the top element of the lattice $\mathcal{I}$. More precisely, for all $V \in \mathcal{I}$,

$$[\![\textbf{while } p \neq 0 \textbf{ do } c]\!]^\sharp V \quad = \quad \bigsqcap^\sharp_{n\geq 0} (F^\sharp_{c,p,V})^n(\top^\sharp)$$

According to Theorem 1, this iteration terminates in finitely many steps. As $\{(F^\sharp_{c,p,V})^n(<0>)\}_{n\in\mathbb{N}}$ is an increasing sequence[4], the stopping criterion consists in checking if the polynomials at step $n+1$ belong to the ideal generated at step n. Note that this ideal membership problem is decidable thanks to Gröbner base computations [6]. As incremental computations are involved here, we propose to use Faugère's $F5$ algorithm to this purpose [7].

Remark 2 *As the abstract transfer function for **while** is derived from $c \equiv$ **if** $p \neq 0$ **then** c_1 **else** skip, and following Remark 1, the abstract definition of a **while** statement guarded by a polynomial disequality is given by the trivial division operator that leaves its argument unchanged. This simplifies the definition of $F^\sharp_{c,p,I}$ into $\lambda J.<p.([\![c]\!]^\sharp J), I>$.*

Correctness of the analysis. We now have to prove that our abstract semantics is correct with respect to the concrete one. Both semantics are compared using the Galois connexion defined in Section 2.

Theorem 4 (Correctness) *Let g be a polynomial and c a polynomial program. Then:*

$$\gamma([\![c]\!]^\sharp <g>) \quad \subseteq \quad B^\nu[\![c]\!]\ \gamma<g>$$

We prove Theorem 4 by induction on the syntax of polynomial programs. The complete proof can be found in Appendix B. Note that this theorem, and its corollary hereafter, are key to show that our analysis indeed generates polynomial invariants.

Corollary 1 *Let g be a polynomial in $\mathbb{R}[\mathtt{x_1}, \ldots, \mathtt{x_m}]$ and c a polynomial program. If g satisfies $[\![c]\!]^\sharp <g> = <0>$, then g is a polynomial invariant at the end of c.*

Proof. For a given polynomial g, if $[\![c]\!]^\sharp <g> = <0>$, then $\gamma([\![c]\!]^\sharp <g>) = \mathbb{R}^m$. As $\gamma([\![c]\!]^\sharp <g>) \quad \subseteq \quad \mathrm{B}^\nu[\![c]\!]\ \gamma<g>$, we have $\mathrm{B}^\nu[\![c]\!]\ \gamma<g> \supseteq \mathbb{R}^m$, which proves that $\mathrm{B}^\nu[\![c]\!]\ \gamma<g> = \mathbb{R}^m$. Using Theorem 3, this implies that, for any initial state $\sigma \in \mathbb{R}^m$, if there exists σ_f such that $\langle c, \sigma\rangle \rightarrow^+ \sigma_f$, then $\sigma_f \in \gamma(g)$. Thus, g is a polynomial invariant. □

Corollary 1 provides a method to verify that a given polynomial g is invariant. It consists in computing the abstract semantics on $<g>$ and comparing the result with the zero ideal $<0>$. In the next section, we will show how to use Corollary 1 to generate polynomial invariants.

5 Generating polynomial invariants

As noticed with Corollary 1, the abstract semantics naturally provides a way to validate an invariant candidate. We now show how the additional use of parametrized polynomials allows for automatically generating invariants.

[4]Note that this sequence is decreasing if we consider the order induced by $\sqsubseteq^\sharp$.

Definition 9 (linear a_i-parametrized polynomial) *We denote by a_i ($i \in \mathbb{N}$) a set of parameters. A polynomial $g \in \mathbb{R}[\mathtt{x}_1, \ldots, \mathtt{x}_m]$ is said to be a* linear a_i-parametrized polynomial *(noted a_i-lpp) if its coefficients are a linear combination of the a_i-parameters. For example, $a_0 + a_1.\mathtt{x}_1 + a_2.\mathtt{x}_2 + a_3.\mathtt{x}_1\mathtt{x}_2 + a_4.{\mathtt{x}_1}^2 + a_5.{\mathtt{x}_2}^2$ is the most generic linear a_i-parametrized polynomial of $\mathbb{R}[\mathtt{x}_1, \mathtt{x}_2]$ of degree 2. An ideal is said to be a* linear a_i-parametrized ideal *if it is generated by linear a_i-parametrized polynomials.*

input : a polynomial program c, a natural number d
output: a set of polynomials $\mathcal{G}$

```
1 begin
2 |  start from g, the most generic a_i-lpp of degree d;
3 |  compute the abstract semantics [[c]]^♯<g>;
4 |  generate C_{g,c}, the set of constraints equivalent to: [[c]]^♯<g> = <0>;
5 |  compute S_{g,c}, the set of solutions of C_{g,c};
6 |  define G as the set of all instances of g by elements of S_{g,c};
7 end
```

Algorithm 1: Invariant set generation

Algorithm 1 generates polynomial invariants of degree d for a program c. Line 4 brings up the most significant difference with the verification scheme detailled by Corollary 1. Indeed, as we start from an a_i-*lpp* polynomial, $[\![c]\!]^\sharp<g>$ is a set of parametrized polynomials. More precisely, we can show by induction on the syntax of polynomial programs that $[\![c]\!]^\sharp<g>$ is a linear a_i-parametrized ideal. Now, proving that equality $[\![c]\!]^\sharp<g> = <0>$ holds is equivalent to proving that all polynomials in $[\![c]\!]^\sharp<g>$ are null. As a polynomial is null if all its coefficients are null, $\mathscr{C}_{g,c}$ contains the coefficients of all polynomials in $[\![c]\!]^\sharp<g>$. These coefficients are linear combination of the a_i parameters. Thus, $\mathscr{C}_{g,c}$ is an homogeneous system of a_i-linear equations, and $\mathscr{S}_{g,c}$ can be obtained by a simple Gaussian elimination algorithm. Thanks to $\mathscr{S}_{g,c}$, each a_i can be rewritten into a linear combination of a subset of the initial parameters, that corresponds to a basis of the linear system. Similarly, the a_i-*lpp* polynomial g is rewritten using this same basis, yielding the parametrized polynomial $\tilde{g}$. The remaining a_is being left unconstrained, any instantiation of these parameters constitute a program invariant: these instantiations form the set $\mathcal{G}$. These remarks are summarized in the following correctness theorem.

Theorem 5 *Let $c \in \mathbb{P}$ and $d \in \mathbb{N}$. Polynomials computed by Algorithm 1 are invariants for c of degree less or equal to d.*

An important remark here is that generating $\mathscr{C}_{g,c}$ can be costly, as the stopping criterion requires computing Gröbner bases on parametrized polynomials. We are exploring several avenues to deal with this complexity in the general case, starting with naïvely capping the number of iterations. This would allow us to produce polynomial invariant *candidates*, which are simpler to verify. In the next section, we propose another way to cope with this complexity issue. By making a natural assumption on the polynomial relations we want to generate, we prove that only one iteration suffices to compute fixpoints for **while** statements guarded by a polynomial disequality.

1. $\mathtt{r} := 0;$
2. $\mathtt{s} := 1;$
3. $\mathtt{t} := 1;$
4. **while** $\mathtt{s} \leq \mathtt{n}$ **do**
5. $\quad \mathtt{r} := \mathtt{r} + 1;$
6. $\quad \mathtt{t} := \mathtt{t} + 2;$
7. $\quad \mathtt{s} := \mathtt{s} + \mathtt{t};$
8.

I_1	$I_4[{}^1/_t; {}^1/_s; {}^0/_r]$
I_4	$I_8 \bigsqcap^\sharp I_5^1 \bigsqcap^\sharp I_5^2 \bigsqcap^\sharp I_5^3 \bigsqcap^\sharp I_5^4 \bigsqcap^\sharp I_5^5$
I_5^1	$<a_0+a_1(r+1)+a_2(s+t+2)+a_3(t+2)+a_4n+ a_5(r+1)^2+a_6(r+1)(s+t+2)+a_7(r+1)(t+2)+ a_8(r+1)n+a_9(s+t+2)^2+a_{10}(s+t+2)(t+2)+ a_{11}(s+t+2)n+a_{12}(t+2)^2+a_{13}(t+2)n+a_{14}n^2>$
I_8	$<a_0 + a_1r + a_2s + a_3t + a_4n + a_5r + a_6rs + a_7rt + a_8rn + a_9s^2 + a_{10}st + a_{11}sn + a_{12}t^2 + a_{13}tn + a_{14}n^2>$

Figure 2: A polynomial program to compute a square root: `sqrt`.

Example 1 (invariant for `sqrt`) *In this paragraph, we illustrate Algorithm 1 on the small and didactic example of Figure 2. We deliberately make the choice of a simple example, where the guard is not taken into account, to be able to display each step of Algorithm 1. The table on the right of Figure 2 contains key abstract values of the analysis. In this example, we aim at computing invariants of degree less than or equal to* 2*. Note that this is not indicative of a limitation of the analysis, which is in no way bound to the quadratic case. Examples in Section 6 deal with full guards and higher-degree invariants.*

Abstract computations start from ideal I_8 that only contains g, the most generic a_i-lpp polynomial of degree 2 *in variables r, s, t, n. As the guard of the **while** statement is not a polynomial (dis)equality, it is replaced by a non deterministic choice, and the loop is unfolded until a fixpoint is reached.*

$$I_4 = [\![\textbf{\textit{while}}\ (*)\ \textbf{\textit{do}}\ \ \mathtt{r} := \mathtt{r}+1; \mathtt{t} := \mathtt{t}+2; \mathtt{s} := \mathtt{s}+\mathtt{t}]\!]^\sharp g = <g, g_1, g_2, g_3, \ldots>$$

where g_i corresponds to the polynomial computed on line 5 *at step i ($g_0 = g$ and for $i \geq 1$, $g_i = I_5^i = [\![\mathtt{r} := \mathtt{r}+1; \mathtt{t} := \mathtt{t}+2; \mathtt{s} := \mathtt{s}+\mathtt{t}]\!]^\sharp(g_{i-1})$). By Theorem 1, we know that this iteration terminates in finitely many steps. Gröbner bases computation show that $g_6 \in <g, g_1, g_2, g_3, g_4, g_5>$, which is sufficient to prove that $I_4 = <g, g_1, g_2, g_3, g_4, g_5>$. Finally, I_1 is built by substituting the program variables in I_4 by the values given by the initial assignments. The next step of Algorithm 1 consists in obtaining the linear constraints equivalent to equality $I_1 = <0>$. For example, the constraint $a_0+a_2+a_3+a_9+a_{10}+a_{12} = 0$ is obtained by equating the coefficient of degree* 0 *in $g[{}^1/_t; {}^1/_s; {}^0/_r]$ to zero. For lack of space, we do not detail the whole linear system and skip directly to the computation of its solution (line 5 of Algorithm 1). Taking the set $\{a_0, a_2, a_7, a_{10}, a_{12}\}$ as basis parameters yields*

$$\mathscr{S}_{g,\mathtt{sqrt}} = \begin{cases} a_1 = 2a_0 - a_7 - 2a_{12} & a_6 = -2a_{10} \\ a_3 = -a_0 - a_2 - a_{10} - a_{12} & a_8 = -2a_{13} \\ a_4 = -a_{13} & a_9 = a_{11} = a_{14} = 0 \\ a_5 = -a_2 - 2a_7 - a_{10} - 4a_{12} & \end{cases}$$

Substituting the a_i-parameters according to this solution, we conclude that, for all $a_0, a_2, a_7, a_{10}, a_{12} \in \mathbb{R}$

$$\begin{gathered} a_0(1+2r-t) \quad + \quad a_2(s-t-r^2) \quad + \quad a_7(-r-2r^2+rt) \\ + a_{10}(-t-r^2-2rs+st) + a_{12}(-2r-t-4r^2+t^2) + a_{13}(-n-2rn+tn) = 0 \end{gathered}$$

Each polynomial that appears after parameters $a_0, \ldots, a_{13}$ in this formula is invariant at the end of program `sqrt`. These invariants are subsumed by $t = 2r+1$ and $s = (r+1)^2$.

As already noticed, the use of Gröbner base computations to decide termination can be avoided in certain cases, while retaining the precision of our approach. The following Section explores a subclass of program invariants, on which termination is obtained in a single iteration step.

6 Fast generation of inductive invariants for loops

The basic idea for accelerating convergence consists in restricting the set of invariant candidates by making a natural assumption: an invariant g is said to be inductive for a loop if the relation $g = 0$ is satisfied at the beginning of the loop and remains true when iterating.

Definition 10 (Inductive (loop) invariants) *Let us consider a loop program* $\mathtt{w} \equiv$ ***while*** b ***do*** c. *An invariant is said to be inductive for* $\mathtt{w}$ *if, for all* σ *in* $\gamma(g)$, *if there exists* $\sigma' \in \mathbb{R}^m$ *such that* $\langle c, \sigma\rangle \rightarrow^+ \sigma'$, *then* $\sigma' \in \gamma(g)$.

This definition can be rewritten using the backward concrete semantics, as stated by the following lemma, which is a direct consequence of Theorem 3.

Lemma 1 *Let* $\mathtt{w} \equiv$ ***while*** b ***do*** c *be a polynomial loop program and* g *a polynomial in* $\mathbb{R}[\mathtt{x_1}, \ldots, \mathtt{x_m}]$. *Then,* g *is an inductive invariant for* $\mathtt{w}$ *if and only if* $\gamma(g) \subseteq B^{\nu}[\![c]\!]\ \gamma(g)$.

The notion of inductive invariant now has to be translated into the abstract semantics. The following theorem gives a sufficient condition for having an inductive invariant.

Theorem 6 (Inductive hypothesis) *Assuming the notations of Lemma 1, if* $[\![c]\!]^\sharp <g> = <g>$, *then* g *is an inductive invariant.*

Proof. By Theorem 4, we have $\gamma([\![c]\!]^\sharp <g>) \subseteq \mathrm{B}^{\nu}[\![c]\!]\ \gamma<g>$. Hypothesis $[\![c]\!]^\sharp g = <g>$ and Lemma 1 allow to conclude the proof. □

This theorem has a direct consequence on fixpoint computations: by positing the inductive hypothesis, we can prove that iterating is not needed to compute abstract **while** statements guarded by a polynomial disequality.

Lemma 2 *Let* $g \in \mathbb{P}$ *and* $\mathtt{w} \equiv$ ***while*** $p \neq 0$ ***do*** c *be a polynomial loop program. Suppose that* $[\![c]\!]^\sharp <g> = <g>$. *Then* $[\![\mathtt{w}]\!]^\sharp <g> = <g>$.

Proof. According to Remark 2, we have

$$[\![\textbf{while } p \neq 0 \textbf{ do } c]\!]^\sharp <g> = <g, p.([\![c]\!]^\sharp g), \ldots>$$

By hypothesis, $[\![c]\!]^\sharp <g> = <g>$. Thus, $p.([\![c]\!]^\sharp <g>) = p<g> \subseteq <g>$, which proves that stabilization is reached and concludes the proof. □

In the case of loops wih positive guards, the analysis still needs to iterate through the successive remainder computations. The termination of this process is still ensured by the descending chain property of polynomial ideals, and the analysis proceeds as in the general case of Section 4. An alternative is to over-approximate the solution set by treating positive guards nondeterministically;

this brings us back to the quick single iteration case. In practice however, positive guarded loops are a rare occurrence: for instance, none of Carbonell and Kapur's polynomial program benchmarks [16, 17] include such conditionals.

The following example illustrates two important points: it explains the `Rem`-computations and demonstrates the use of the inductive hypothesis.

Example 2 (inductive invariant for `mannadiv`) *Recall algorithm from Figure 1a. As in Section 5, we call I_i the ideal generated from unfolding the semantics of the instruction at line i. The analysis starts from ideal $I_7 = <g>$, containing the most generic quadratic a_i-lpp polynomial g, and performs a series of variable instantiations before evaluating the* ***if*** *instruction of line* 3*. We note $g_1 = g[{}^{y_3-1}/_{y_3}; {}^{0}/_{y_2}; {}^{y_1+1}/_{y_1}]$ and $g_2 = g[{}^{y_3-1}/_{y_3}; {}^{y_2+1}/_{y_2}]$ the polynomials produced by these instantiations. Ideal I_3 is defined using the operation $\mathtt{Rem}(g_1, p)$ where $p = x_2 - y_2 - 1$. As already mentioned in Section 4, the choice of the division operator is crucial for our analysis and a bad choice may lead to a loss of precision. To avoid this issue, we perform division by a parametrized polynomial. In this example, computation of $\mathtt{Rem}(g_1, p)$ is done as follows. We introduce q the most generic b_i-lpp of degree $deg(g_1) - deg(p) = 1$, and define*

$$\mathtt{Rem}(g_1, p) = g_1 - q.p = g_1 - (b_0 + b_1\,x_1 + b_2\,x_2 + b_3\,y_1 + b_4\,y_2 + b_5\,y_3).(x_2 - y_2 - 1)$$

Note that the resulting polynomial is in $\{a_i, b_i\}$-lpp form, which is essential for the linearity of the constraints generated further. Ideal $I_3 = <(x_2 - y_2 - 1).g_2, \mathtt{Rem}(g_1, p)>$ is then built from this definition.

The `while` *instruction from line* 2 *triggers the use of the inductive hypothesis (Theorem 6), indicating that the loop body maintains the original invariant, constraining I_3 to be equal to I_7. This constraint can be satisfied by enforcing both $\mathtt{Rem}(g_1, p) = g$ and $g_2 = g$. Let $\mathcal{C}_{w_1}$ and $\mathcal{C}_{w_2}$ be these respective constraints. What is more, by Lemma 2 we have $I_2 = I_7$. The substitutions on line* 1 *operate on I_2; enforcing the nullness of I_1 generates another set of constraints $\mathcal{C}_i$. The important point here is that the constraints $\mathcal{C}_{w_*}$ and $\mathcal{C}_i$ express two different properties of the invariant (inductiveness, and initial nullness), and that solving only $\mathcal{C}_i$ would yield an incomplete solution.*

Step 4 of Algorithm 1 consists in solving the conjunct of all constraints:

$\mathcal{C}_i$	$\mathcal{C}_{w_2}$	$\mathcal{C}_{w_1}$	
$a_6 + a_{10} + a_{20} = 0$	$a_{20} = a_{18}$	$a_{10} = a_8$	$a_{18} + a_{13} = 0$
$a_1 + a_5 = 0$	$a_{19} = 2\,a_{18}$	$a_{20} = a_{15}$	$a_9 = a_{16} = a_{19} = 0$
$a_7 + a_{14} = 0$	$a_{17} = a_{16}$	$b_4 + a_{13} = 0$	$b_1 = b_2 = b_3 = b_5 = 0$
$a_0 = a_2 = a_{11} = 0$	$a_4 = a_5$	$b_0 = a_{12} - a_{14}$	
	$a_{10} = a_9$	$a_5 = a_3 - a_{14} + a_{12}$	
	$a_7 + a_{14}$	$a_4 = a_{12} - a_{13} - a_{14}$	
	$a_{14} = a_{13}$	$a_{17} = 2\,a_{15}$	

These resolve into $a_{12} = a_5 = a_4 = b_0 = -a_1$, and all other parameters equating to 0*. Finally, the direct instantiation of the a_i-lpp polynomial returns the single program invariant: $x_1 = y_1\,x_2 + y_2 + y_3$.*

Examples and benchmarks. Table 3 presents the results of the Maple implementation of our analysis. It is run on Rodríguez-Carbonell and Kapur's inductive benchmarks [17, 16], and measured to their best and worst results. Our method is systematically better, and on average 2.2 orders of magnitude faster. A few examples deserve additional comments: we have added `petter30`

Name	d	Var	Comparing performance				
			RCK approach		Fast Ind	RCK/Fast Ind	
			fastest	slowest	in seconds	worst	best
`cohencu`	2	4	0.7	1.88	0.009	78	209
`divbin`	2	5	0.99	2.1	0.005	198	420
`freire1`	2	3	0.38	0.7	0.006	63	116
`freire2`	2	4	0.7	1.0	0.007	100	143
`mannadiv`	2	5	1.12	2.96	0.005	224	592
`knuth`	3	9	2.61	55.4	0.084	31	659
`prod4br`	3	6	4.63	> 300	0.013	356	$> 23.10^3$
`euclidex2`	2	8	1.95	15.15	0.008	244	1894
`fermat1`	2	5	0.8	2.57	0.006	133	428
`fermat2`	2	5	0.92	1.49	0.006	153	248
`lcm1`	2	6	1.0	2.49	0.006	167	415
`lcm2`	2	6	1.21	2.05	0.006	202	342
`petter1`	2	2	0.5	1.0	0.003	83	167
`petter5`	6	2	1.4	> 300	0.006	233	$> 50.10^3$
`sqrt`	2	3	0.46	–	0.010	46	–
`z3sqrt`	2	4	0.82	–	0.006	137	–
`cohencu`	3	4	–	–	0.018	–	–
`freire2`	3	4	–	–	0.012	–	–
`petter30`	31	2	–	–	1.423	–	–

Figure 3: Performance results for the Maple implementation

that yields an invariant of degree 31, and run `cohencu` and `freire2` with a different set of parameters in order to find a more interesting, higher-order invariant (resp. $\mathtt{x} = \mathtt{n}^3$, and $4\mathtt{r}^3 - 6\mathtt{r}^2 + 3\mathtt{r} + 4\mathtt{x} - 4\mathtt{a} = 1$). Our analysis had more mixed results with the `knuth` program: we had to introduce two additional variables to encode the non-polynomial initial assignements, which significantly increased the size of the linear system. It is unclear how Rodríguez-Carbonell and Kapur dealt with these assignments. Unfortunately, we were not able to compare to Sankaranarayanan *et al.*' s approach, for lack of an implementation.

An OCaml implementation of this analysis for Java bytecode, using the Sawja[5] static analysis libraries to derive annotations and intermediate code representation, provides a fully automated invariant generation tool. Early results have been competitive with the Maple implementation, and a prototype is available at `www.irisa.fr/celtique/ext/polyinv`. A thorough analysis of our results can be found in Appendix C.

7 Related work

As already mentioned in the introduction of this paper, our approach extends Müller-Olm and Seidl's work [12] to a structured language with both polynomial equality and disequality guards. In the case of programs with negative guards only, our abstract function coincides with theirs, thus inheriting the completeness property. In terms of computational complexity, we propose a practical alternative to iteration-based methods, by extending the hypotheses of Sankaranarayanan *et al.* [19] to a more general setting without sacrificing efficiency. More precisely, the condition $[\![c]\!]^\sharp(<g>) = g$, which provides the most efficient invariant computation, corresponds to their notion of *constant value consecution*, and the condition $[\![c]\!]^\sharp(<g>) = <g>$, which is the inductive hypothesis of Theorem 6, corresponds to their notion of *constant-scale consecution*.

[5] `sawja.inria.fr`

Rodríguez-Carbonell and Kapur [16] propose a method adapted to both kinds of guards, but at the price of a high degree of computational and conceptual complexity. First, their abstract domain is the set of ideal varieties, *i.e.,* ideals such that $\alpha \circ \gamma(I) = I$ (called the IV property). The transfer function for disequality guards comes down to computing ideal quotients in this abstract domain. The IV property is costly to maintain, since it relies on the computation of radical ideals. By default, their implementation skips these computations and ignores disequality guards, thus inducing over-approximations. In their forward approach, abstract assignments are handled by fresh variable introduction and elimination. This requires computing ideal intersections. Abstract equality tests, which are easier to handle in a forward approach, still need IV computations due to the nature of the abstract domain. These are also generally skipped in practice. Finally, even if their approach can be adapted to bound the number of iterations by making strong assumptions on programs [17], it systematically demands iterative fixpoint computations. The process of computing all polynomial invariants for a given structure of programs was extended by Kovács in [9] which provides, again through iterative fixpoint computation, a complete invariant generation method for a specific loop pattern with nested conditionals.

8 Conclusion

We have presented a backwards polynomial invariant generation method that enjoys both programming language expressiveness and, for a large class of invariants, a fast generation procedure.

The proposed method constitutes a solid foundation for extensions to an analysis tool that covers a full-fledged language, such as the Java bytecode. The analysis has good scalability properties as the limiting factor is the number of variables and not the degree of the variables nor the size of the code. The integration of the analysis into the Sawja static analysis framework for Java yields good results on benchmarks from the litterature and will enable us to back these claims with experiments on large code bases.

In addition, there are at least two specific avenues we will investigate in further work. For one, the non inductive invariant generation case still requires Gröbner bases computation in order to know when loop unfolding has to be stopped. We are currently experimenting to identify which kind of algorithm is well adapted to our particular form of ideals. Second, on a more theoretical side, the abstract transfer functions for handling our **if** and **while** control structures are based on remainder computations, where any remainder function yields a correct abstract semantics. We have found a way to implicitly generate "good" remainder functions for inductive invariants, but an exact characterization of such functions in the general case is still to be found.

We also undertook the mechanized formalization of all the material of this paper with the Coq proof assistant, following Besson *et al.*'s approach [2] to linear invariant generation. In addition to the gain in confidence, this lays the groundwork for a certifying analysis toolchain, *i.e.*, the combination of an analyzer that generates certificates in predefined format, and a formally verified checker that validates them.

References

[1] Roland Carl Backhouse. Galois connections and fixed point calculus. In Roland Carl Backhouse, Roy L. Crole, and Jeremy Gibbons, editors, *ACMMPC*, volume 2297 of *Lecture Notes in Computer Science*, pages 89–148. Springer, 2000.

[2] Frédéric Besson, Thomas P. Jensen, David Pichardie, and Tiphaine Turpin. Certified result checking for polyhedral analysis of bytecode programs. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *TGC*, volume 6084 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2010.

[3] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2003.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM Press, 1977.

[5] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM Press, 1978.

[6] David Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms*. Undergraduate Texts in Mathematics. Springer, third edition, 2007. An introduction to computational algebraic geometry and commutative algebra.

[7] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In *ISSAC*, pages 75–83. ACM Press, 2002.

[8] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[9] Laura Kovács. A complete invariant generation approach for P-solvable loops. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2009.

[10] Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[11] Markus Müller-Olm and Helmut Seidl. Polynomial constants are decidable. In *SAS*, pages 4–19. Springer, 2002.

[12] Markus Müller-Olm and Helmut Seidl. Computing polynomial program invariants. *Information Processing Letters*, 91(5):233–244, 2004.

[13] Enric Rodríguez-Carbonell and Deepak Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In Roberto Giacobazzi, editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 280–295. Springer, 2004.

[14] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants: algebraic foundations. In *ISSAC*, pages 266–273. ACM Press, 2004.

[15] Enric Rodríguez-Carbonell and Deepak Kapur. Program verification using automatic generation of invariants. In *ICTAC*, volume 3407 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2004.

[16] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75, 2007.

[17] Enric Rodríguez-Carbonell and Deepak Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443–476, 2007.

[18] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Constraint-based linear-relations analysis. In Roberto Giacobazzi, editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2004.

[19] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In *POPL*, pages 318–329. ACM Press, 2004.

[20] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

$$\frac{}{\langle c_1; c_2, \sigma\rangle \sim \langle c_2, \langle c_1, \sigma\rangle\rangle}\ \textbf{\textit{seq}} \qquad \frac{\sigma \to \sigma'}{\langle c, \sigma\rangle \to \langle c, \sigma'\rangle}\ \textbf{\textit{congruence}}$$

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}\ \textbf{\textit{skip}} \qquad \frac{p(\sigma) = v}{\langle \mathtt{x_j} := p, \sigma\rangle \to \sigma[v]_{\mathtt{j}}}\ \textbf{\textit{assign}}$$

$$\frac{p(\sigma) \bowtie 0 \equiv \mathit{true}}{\langle \textbf{if } p \bowtie 0 \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \langle c_1, \sigma\rangle}\ \textbf{\textit{if}} \quad \textit{where} \quad \left.\begin{array}{l} 0 = 0 \\ v \neq 0 \end{array}\right\} \equiv \mathit{true}$$

$$\frac{p(\sigma) \bowtie 0 \equiv \mathit{false}}{\langle \textbf{if } p \bowtie 0 \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \langle c_2, \sigma\rangle}\ \textbf{\textit{if}} \quad \textit{where} \quad \left.\begin{array}{l} 0 \neq 0 \\ v = 0 \end{array}\right\} \equiv \mathit{false}$$

$$\frac{}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \sim \langle \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip}, \sigma\rangle}\ \textbf{\textit{while}}$$

Notation: given $\sigma = (\sigma_1, \ldots, \sigma_m)$, we note $\sigma[v]_{\mathtt{j}}$ the updated state $(\sigma_1, \ldots, \sigma_{j-1}, v, \sigma_{j+1}, \ldots, \sigma_m)$.

Figure 4: Small-step operational semantics.

A Semantical Frameworks and Relations

A.1 Wright-Felleisen small-step semantics of polynomial programs.

The semantics of polynomial programs is defined as a relation between closed terms c. We use a congruence rule to lift the primitive deduction rules to a standard, left-to-right, innermost-to-outermost interpretation of the language.

Definition 11 (Small-step operational semantics (SSOS)) *Let c be a polynomial program, v a program value, and $\sigma, \sigma' \in \mathbb{R}^m$ be states of the program c. The reduction semantics of our language is provided by the rules in Figure 4.*

Remark that we use the $\sim$ relation to denote macro-like expansion. We use this notation to refrain from introducing additional steps in the small-step reductions.

A.2 Relation between backward and operational semantics

A.2.1 Comparing $\mathrm{B}^{\mu}[\![.]\!]$ with SSOS.

We begin by comparing the backward least fixpoint collecting semantics $\mathrm{B}^{\mu}[\![.]\!]$ with SSOS. Theorem 7 states that least fixpoint semantics collects all terminating states.

Theorem 7 ($\mathrm{B}^{\mu}[\![.]\!]$ vs SSOS) *For all $S \in \mathcal{P}(\mathbb{R}^m)$ and for all polynomial program state σ,*

$$\sigma \in B^{\mu}[\![c]\!]\ S \ \Leftrightarrow\ \exists \sigma_f \in S, \langle c, \sigma\rangle \to^{+} \sigma_f$$

First, remark that, according to the definition of the SSOS, there is no program c and states $\sigma, \sigma_f \in \mathbb{R}^m$ such that $\langle c, \sigma\rangle \to^{0} \sigma_f$. Thus, if $\langle c, \sigma\rangle \to^{*} \sigma_f$ is satisfied

for $\sigma, \sigma_f \in \mathbb{R}^m$ then $\langle c, \sigma\rangle \to^+ \sigma_f$. This property will be used in further proofs without mentioning it.

In order to improve the readability of Theorem 7, we divide it in two lemmas stating one or the other side of the equivalence.

Lemma 3 *For all $S \in \mathcal{P}(\mathbb{R}^m)$, for all $\sigma \in \mathbb{R}^m$ and polynomial program $c \in \mathbb{P}$,*

$$\sigma \in B^\mu[\![c]\!]\ S \ \Leftarrow\ \exists \sigma_f \in S, \langle c, \sigma\rangle \to^+ \sigma_f$$

Proof. By mathematical induction on the length k of the derivation sequence, we prove the following property for all $k \in \mathbb{N}$:

$$\sigma \in \mathrm{B}^\mu[\![c]\!]\ S \ \Leftarrow\ \exists \sigma_f \in S, \langle c, \sigma\rangle \to^k \sigma_f$$

Induction base If $k = 0$ then the property trivially holds since there is no σ_f such that $\langle c, \sigma\rangle \to^0 \sigma_f$

Induction step We assume the property for $k \leq m$ and prove it holds for $m+1$. Assume there exists $\sigma_f \in S$ such that $\langle c, \sigma\rangle \to^{m+1} \sigma_f$.

If $c \equiv$ skip then $\langle c, \sigma\rangle \to^1 \sigma$. Thus $m = 0$ and $\sigma_f = \sigma$. As $\mathrm{B}^\mu[\![c]\!]\ S = S$, the property trivially holds for this case.

If $c \equiv \mathtt{x_j} := p$ then $\langle c, \sigma\rangle \to^1 \sigma[v]_j$ for $v = p(\sigma)$. Thus $m = 0$ and $\sigma_f = \sigma[v]_j$. As $\mathrm{B}^\mu[\![c]\!]\ S = \{\sigma \in \mathbb{R}^m \mid \sigma[p(\sigma)]_j \in S\}$, the property trivially holds for this case.

If $c \equiv c_1; c_2$ then $\langle c_1, \sigma\rangle \to^{k_1} \sigma'$ and $\langle c_2, \sigma'\rangle \to^{k_2} \sigma_f$ for k_1 and k_2 such that $1 \leq k_i \leq m$ and $k_1 + k_2 = m$. By induction hypothesis, $\sigma' \in \mathrm{B}^\mu[\![c_2]\!]\ S$ and $\sigma \in \mathrm{B}^\mu[\![c_1]\!]\ (\mathrm{B}^\mu[\![c_2]\!]\ S)$. As $\mathrm{B}^\mu[\![c]\!]\ S = \mathrm{B}^\mu[\![c_1]\!]\ (\mathrm{B}^\mu[\![c_2]\!]\ S)$, the property holds for this case.

If $c \equiv$ if $p \bowtie 0$ then c_1 else c_2

Recall $\mathrm{B}^\mu[\![c]\!]\ S = ([\![p \bowtie 0]\!] \bigcap \mathrm{B}^\mu[\![c_1]\!]\ S) \bigcup ([\![p \not\bowtie 0]\!] \bigcap \mathrm{B}^\mu[\![c_2]\!]\ S)$.

Assume $p(\sigma) \bowtie 0 \equiv \mathit{true}$ then $\langle c, \sigma\rangle \to^1 \langle c_1, \sigma\rangle \to^m \sigma_f$. By induction hypothesis, we get $\sigma \in \mathrm{B}^\mu[\![c_1]\!]\ S$. Thus, $\sigma \in [\![p \bowtie 0]\!] \bigcap \mathrm{B}^\mu[\![c_1]\!]\ S$ and as a consequence $\sigma \in \mathrm{B}^\mu[\![c]\!]\ S$.

Assume $p(\sigma) \bowtie 0 \equiv \mathit{false}$ then $\langle c, \sigma\rangle \to^1 \langle c_2, \sigma\rangle \to^m \sigma_f$. By induction hypothesis, we get $\sigma \in \mathrm{B}^\mu[\![c_2]\!]\ S$. Thus, $\sigma \in [\![p \not\bowtie 0]\!] \bigcap \mathrm{B}^\mu[\![c_2]\!]\ S$ and as a consequence $\sigma \in \mathrm{B}^\mu[\![c]\!]\ S$.

Thus the property holds for this case.

If $c \equiv$ **while** $p \bowtie 0$ **do** c_1

Assume $p(\sigma) \bowtie 0 \equiv true$ then $\langle c, \sigma\rangle \rightarrow^1 \langle c_1; c, \sigma\rangle \rightarrow^m \sigma_f$. By induction hypothesis, we get $\sigma \in \mathrm{B}^\mu[\![c_1; c]\!]\ S = \mathrm{B}^\mu[\![c_1]\!]\ (\mathrm{B}^\mu[\![c]\!]\ S)$. Moreover, by the fixpoint definition of the backward semantics of the **while**-statement, we have:

$$\begin{array}{rcl} \mathrm{B}^\mu[\![c]\!]\ S & = & F_{c,p,S}(\mathrm{B}^\mu[\![c]\!]\ S) \\ & = & ([\![p \not\bowtie 0]\!] \bigcap S) \bigcup ([\![p \bowtie 0]\!] \bigcap \mathrm{B}^\mu[\![c_1]\!]\ (\mathrm{B}^\mu[\![c]\!]\ S)) \end{array}$$

As $\sigma \in [\![p \bowtie 0]\!] \bigcap \mathrm{B}^\mu[\![c_1]\!]\ (\mathrm{B}^\mu[\![c]\!]\ S)$, we have $\sigma \in \mathrm{B}^\mu[\![c]\!]\ S$.

Assume $p(\sigma) \bowtie 0 \equiv false$ then $\langle c, \sigma\rangle \rightarrow^1 \langle \mathbf{skip}, \sigma\rangle \rightarrow^m \sigma_f$ and $m = 1$ and $\sigma_f = \sigma$. By induction hypothesis, we get $\sigma \in \mathrm{B}^\mu[\![\mathbf{skip}]\!]\ S = S$. Moreover, we have:

$$\begin{array}{rcl} \mathrm{B}^\mu[\![c]\!]\ S & = & \mu F_{c_1,p,S} \\ & = & \mu\lambda X.([\![p \not\bowtie 0]\!] \bigcap S) \bigcup ([\![p \bowtie 0]\!] \bigcap \mathrm{B}^\mu[\![c_1]\!]\ X) \\ & = & ([\![p \not\bowtie 0]\!] \bigcap S) \bigcup (\mu\lambda X.[\![p \bowtie 0]\!] \bigcap \mathrm{B}^\mu[\![c_1]\!]\ X) \end{array}$$

As $\sigma \in [\![p \not\bowtie 0]\!] \bigcap S$, we have $\sigma \in \mathrm{B}^\mu[\![c]\!]\ S$.

The property holds for this last case, which proves that it always holds. □

We can now prove the other side of the equivalence.

Lemma 4 *For all $\sigma \in \mathbb{R}^m$, for all $S \in \mathcal{P}(\mathbb{R}^m)$ and polynomial program c, we have:*

$$\sigma \in B^\mu[\![c]\!]\ S \ \Rightarrow\ \exists \sigma_f \in S, \langle c, \sigma\rangle \rightarrow^+ \sigma_f$$

Proof. By syntactic induction on polynomial programs. We proceed by case.

If $c \equiv \mathbf{skip}$ then we have both $\mathrm{B}^\mu[\![c]\!]\ S = S$, and for all $\sigma \in \mathbb{R}^m$,

$$\frac{}{\langle \mathbf{skip}, \sigma\rangle \rightarrow \sigma}\ \textbf{\textit{skip}}$$

which holds, *a fortiori*, for $\sigma \in \mathrm{B}^\mu[\![c]\!]\ S \subseteq \mathbb{R}^m$. Thus we have $\sigma_f = \sigma$.

If $c \equiv \mathtt{x_j} := p$ then we have both $\mathrm{B}^\mu[\![c]\!]\ S = \{x \in \mathbb{R}^m \mid x[p(x)]_\mathtt{j} \in S\}$, and for all $\sigma \in \mathbb{R}^m$,

$$\frac{p(\sigma) = v}{\langle \mathtt{x_j} := p, \sigma\rangle \rightarrow \sigma[v]_\mathtt{j}}\ \textbf{\textit{assign}} \quad \textit{where} \quad \sigma[v]_\mathtt{j} = (\sigma_1, \ldots, \sigma_{j-1}, v, \sigma_{j+1}, \ldots, \sigma_m)$$

Thus we have $\sigma_f = \sigma[v]_\mathtt{j}$.

If $c \equiv$ **if** $p \bowtie 0$ **then** c_1 **else** c_2 then we have:

$$\mathrm{B}^\mu[\![c]\!]\ S = (\mathrm{B}^\mu[\![c_1]\!]\ S \bigcap [\![p \neq 0]\!]) \bigcup (\mathrm{B}^\mu[\![c_2]\!]\ S \bigcap [\![p = 0]\!])$$

Now consider $\sigma \in \mathrm{B}^\mu[\![c]\!]\ S$; two cases arise:

either $p(\sigma) \bowtie 0 \equiv \mathit{true}$ thus $\sigma \in \mathrm{B}^{\mu}[\![c_1]\!]\ S$. The induction hypothesis states $\exists \sigma' \in S, \langle c_1, \sigma\rangle \rightarrow^{+} \sigma'$. And the small-step semantics provides the step:

$$\frac{p(\sigma) \bowtie 0 \equiv \mathit{true}}{\langle \textbf{if } p \bowtie 0 \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \rightarrow \langle c_1, \sigma\rangle}\ \textbf{\textit{if}}$$

or $p(\sigma) \bowtie 0 \equiv \mathit{false}$ is the mirror case: take $\sigma \in \mathrm{B}^{\mu}[\![c_2]\!]\ S$. The induction hypothesis states $\exists \sigma' \in S, \langle c_2, \sigma\rangle \rightarrow^{+} \sigma'$. What is more:

$$\frac{p(\sigma) \bowtie 0 \equiv \mathit{false}}{\langle \textbf{if } p \bowtie 0 \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \rightarrow \langle c_2, \sigma\rangle}\ \textbf{\textit{if}}$$

Thus, $\sigma_f = \sigma'$ and $\langle c, \sigma\rangle \rightarrow^{+} \sigma'$.

If $c \equiv \textbf{while } p \bowtie 0 \textbf{ do } c_1$ then we have, using the Kleene iteration theorem:

$$\mathrm{B}^{\mu}[\![c]\!]\ S = \mu F_{c_1,p,S} = \bigcup\nolimits_{n \geq 0} F^{n}_{c_1,p,S}(\{\})$$

Now consider $\sigma \in \mathrm{B}^{\mu}[\![c]\!]\ S$; thus, $\exists n_0 \in \mathbb{N}$ such that $\sigma \in F^{n_0}_{c_1,p,S}(\{\})$. To conclude this case, we prove by mathematical induction that:

$$\forall n \in \mathbb{N},\quad \sigma \in F^{n}_{c_1,p,S}(\{\}) \ \Rightarrow\ \exists \sigma_f \in S, \langle c, \sigma\rangle \rightarrow^{+} \sigma_f$$

- Case $n = 0$. As $F^{0}_{c_1,p,S}(\{\}) = \{\}$, the implication is valid because its premise is false.
- $n \rightarrow n+1$. Let $n \in \mathbb{N}$ such that $\sigma \in F^{n}_{c_1,p,S}(\{\}) \ \Rightarrow\ \exists \sigma_f \in S, \langle c_1, \sigma\rangle \rightarrow^{+} \sigma_f$. Let now suppose that $\sigma \in F^{n+1}_{c_1,p,S}(\{\})$. We have:

$$\begin{array}{rcl} F^{n+1}_{c_1,p,S}(\{\}) & = & F_{c_1,p,S}(F^{n}_{c_1,p,S}(\{\})) \\ & = & ([\![p \not\bowtie 0]\!] \bigcap S) \bigcup ([\![p \bowtie 0]\!] \bigcap \mathrm{B}^{\mu}[\![c_1]\!]\ (F^{n}_{c_1,p,S}(\{\}))) \end{array}$$

Two cases can arise:

either $p(\sigma) \bowtie 0 \equiv \mathit{false}$ then after macro expansion, the small-step semantics provides the step:

$$\frac{p(\sigma) \bowtie 0 \equiv \mathit{false}}{\langle c, \sigma\rangle \rightarrow \langle \textbf{skip}, \sigma\rangle}\ \textbf{\textit{if}}$$

Thus $\sigma_f = \sigma$ and $\langle c, \sigma\rangle \rightarrow^{+} \sigma_f$.

or $p(\sigma) \bowtie 0 \equiv \mathit{true}$ then $\sigma \in \mathrm{B}^{\mu}[\![c_1]\!]\ (F^{n}_{c_1,p,S}(\{\}))$. By induction hypothesis on c_1, there exists σ'' such that $\langle c_1, \sigma\rangle \rightarrow^{+} \sigma''$, and by induction hypothesis on n, we get that $\langle c_1, \sigma''\rangle \rightarrow^{+} \sigma_f$. What is more:

$$\frac{p(\sigma) \bowtie 0 \equiv \mathit{true}}{\langle c, \sigma\rangle \rightarrow \langle c_1; c, \sigma\rangle}\ \textbf{\textit{if}}$$

Thus, $\langle c, \sigma\rangle \rightarrow^{+} \sigma_f$.

This proves that the property holds for this last case and thus always holds.

□

A.2.2 Comparing SSOS with $\mathrm{B}^{\nu}[\![.]\!]$.

The greatest fixpoint semantics also contains the set of terminating states but may also contains non-terminating states, putting aside termination condition.

Theorem 8 ($\mathrm{B}^{\nu}[\![.]\!]$ vs SSOS) *For all $S \in \mathcal{P}(\mathbb{R}^m)$ and for all polynomial program state σ,*

$$\sigma \in B^{\nu}[\![c]\!]\ S \ \Rightarrow\ \left\{ \begin{array}{rl} \exists \sigma_f \in S, & \langle c, \sigma\rangle \rightarrow^{+} \sigma_f \\ \text{or} & \langle c, \sigma\rangle \rightarrow^{\infty} \end{array} \right.$$

Proof. By syntactic induction on the polynomial program c. We proceed by case.

If $c \equiv$ skip then we have both $\mathrm{B}^{\nu}[\![c]\!]\ S = S$, and for all $\sigma \in \mathbb{R}^m$,

$$\frac{}{\langle \mathbf{skip}, \sigma\rangle \rightarrow \sigma}\ \textbf{\textit{skip}}$$

which holds, *a fortiori*, for $\sigma \in \mathrm{B}^{\nu}[\![c]\!]\ S \subseteq \mathbb{R}^m$. Thus we have $\sigma_f = \sigma$.

If $c \equiv \mathtt{x_j} := p$ then we have both $\mathrm{B}^{\nu}[\![c]\!]\ S = \{x \in \mathbb{R}^m \mid x[p(x)]_{\mathtt{j}} \in S\}$, and for all $\sigma \in \mathbb{R}^m$,

$$\frac{p(\sigma) = v}{\langle \mathtt{x_j} := p, \sigma\rangle \rightarrow \sigma[v]_{\mathtt{j}}}\ \textbf{\textit{assign}} \quad \textit{where} \quad \sigma[v]_{\mathtt{j}} = (\sigma_1, \ldots, \sigma_{j-1}, v, \sigma_{j+1}, \ldots, \sigma_m)$$

Thus we have $\sigma_f = \sigma[v]_{\mathtt{j}}$.

If $c \equiv c_1; c_2$ then we have

$$\mathrm{B}^{\nu}[\![c_1; c_2]\!]\ S = \mathrm{B}^{\nu}[\![c_1]\!]\ (\mathrm{B}^{\nu}[\![c_2]\!]\ S)$$

By induction hypothesis, we have:

1. $\sigma' \in \mathrm{B}^{\nu}[\![c_2]\!]\ S \ \Rightarrow\ \left\{ \begin{array}{rl} \exists \sigma'' \in S, & \langle c_2, \sigma'\rangle \rightarrow^{+} \sigma'' \\ \textit{or} & \langle c_2, \sigma'\rangle \rightarrow^{\infty} \end{array} \right.$
2. $\sigma \in \mathrm{B}^{\nu}[\![c_1]\!]\ (\mathrm{B}^{\nu}[\![c_2]\!]\ S) \ \Rightarrow\ \left\{ \begin{array}{rl} \exists \sigma' \in \mathrm{B}^{\nu}[\![c_2]\!]\ S, & \langle c_1, \sigma\rangle \rightarrow^{+} \sigma' \\ \textit{or} & \langle c_1, \sigma\rangle \rightarrow^{\infty} \end{array} \right.$

Thus the compound application of the ***seq*** and ***congruence*** rules yields the following derivation tree:

$$\langle c_1; c_2, \sigma\rangle \longrightarrow^{1} \langle c_2, \langle c_1, \sigma\rangle\rangle \longrightarrow^{+} \langle c_2, \sigma'\rangle \longrightarrow^{+} \sigma''$$
$$\langle c_1, \sigma\rangle \searrow^{\infty} \qquad \langle c_2, \sigma'\rangle \searrow^{\infty}$$

Hence, either c diverges, or $\sigma_f = \sigma''$.

If $c \equiv$ if $p \bowtie 0$ then c_1 else c_2 then we have:

$$\mathrm{B}^{\nu}[\![c]\!]\ S = (\mathrm{B}^{\nu}[\![c_1]\!]\ S \bigcap [\![p \bowtie 0]\!]) \bigcup (\mathrm{B}^{\nu}[\![c_2]\!]\ S \bigcap [\![p \not\bowtie 0]\!])$$

Now consider $\sigma \in \mathrm{B}^{\nu}[\![c]\!]\ S$; two cases arise:

either $p(\sigma) \bowtie 0 \equiv \mathit{true}$ thus $\sigma \in \mathrm{B}^\nu[\![c_1]\!]\ S$. The induction hypothesis states $\exists \sigma' \in S, \langle c_1, \sigma\rangle \rightarrow^+ \sigma'$ or $\langle c_1, \sigma\rangle \rightarrow^\infty$. And the small-step semantics provides the first step:

$$\frac{p(\sigma) \bowtie 0 \equiv \mathit{true}}{\langle \textbf{if } p \bowtie 0 \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \rightarrow \langle c_1, \sigma\rangle}\ \textbf{\textit{if}}$$

or $p(\sigma) \bowtie 0 \equiv \mathit{false}$ is the mirror case: take $\sigma \in \mathrm{B}^\nu[\![c_2]\!]\ S$. The induction hypothesis states $\exists \sigma' \in S, \langle c_2, \sigma\rangle \rightarrow^+ \sigma'$ or $\langle c_2, \sigma\rangle \rightarrow^\infty$. What is more:

$$\frac{p(\sigma) \bowtie 0 \equiv \mathit{false}}{\langle \textbf{if } p \bowtie 0 \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \rightarrow \langle c_2, \sigma\rangle}\ \textbf{\textit{if}}$$

Thus, either c diverges, or $\sigma_f = \sigma'$.

If $c \equiv$ **while** $p \bowtie 0$ **do** c_1 then the backward semantics yields $\sigma \in \nu F_{c_1,p,S}$ where:

$$F_{c_1,p,S} = \lambda X.([\![p \not\bowtie 0]\!] \bigcap S) \bigcup ([\![p \bowtie 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ X)$$

Backward concrete *while*-semantics is obtained by iterating from the top element $\mathbb{R}^m$. Thus, we have

$$\mathrm{B}^\nu[\![c]\!]\ S = \bigcap\nolimits_{n\geq 1} F^n_{c_1,p,S}(\mathbb{R}^m)$$

and, as $\sigma \in \mathrm{B}^\nu[\![c]\!]\ S$, then $\forall i \in \mathbb{N}, \sigma \in F^i_{c_1,p,S}(\mathbb{R}^m)$.

Since $\mathrm{B}^\nu[\![c_1]\!]$ is a $\bigcup$-morphism, we have:

$$\begin{aligned}
F^i_{c_1,p,S}(\mathbb{R}^m) \quad = \quad & [\![p \not\bowtie 0]\!] \bigcap S \\
\bigcup \quad & (\ [\![p \bowtie 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ ([\![p \not\bowtie 0]\!] \bigcap S)\ ) \\
\bigcup \quad & (\ [\![p \bowtie 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ ([\![p \bowtie 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ ([\![p \not\bowtie 0]\!] \bigcap S))\ ) \\
\bigcup \quad & \cdots \\
\vdots & \\
\bigcup \quad & \cdots \\
\bigcup \quad & (\ [\![p \bowtie 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ ([\![p \bowtie 0]\!] \bigcap \cdots \bigcap \mathrm{B}^\nu[\![c_1]\!]\ ([\![p \not\bowtie 0]\!] \bigcap S)) \cdots))\ ) \\
\bigcup \quad & (\ [\![p \bowtie 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ ([\![p \bowtie 0]\!] \bigcap \cdots \bigcap \mathrm{B}^\nu[\![c_1]\!]\ ([\![p \bowtie 0]\!]) \bigcap \mathrm{B}^\nu[\![c_1]\!]\ (\mathbb{R}^m)) \cdots))\ )
\end{aligned}$$

$F^i_{c_1,p,S}(\mathbb{R}^m)$ is defined by the union of $i+1$ sets, corresponding to the successive unfoldings of the loop. In the rest of the proof, we need to know precisely in which of these $i+1$ sets we can found σ. For that matter, we note ${}_j.F^i$ the set appearing at line j of $F^i_{c_1,p,S}(\mathbb{R}^m)$. To remove any

ambiguity, we can formally define ${}_j.F^i$ by the following recurrence:

$$\begin{cases} {}_0.F^1 &= [\![p \not\bowtie 0]\!] \bigcap S \\ {}_j.F^{j+1} &= [\![p \bowtie 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ ({}_{j-1}.F^j),\ \forall j \geq 1 \\ \\ {}_1.F^1 &= [\![p \bowtie 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ (\mathbb{R}^m) \\ {}_j.F^j &= [\![p \bowtie 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ ({}_{j-1}.F^{j-1}),\ \forall j \geq 2 \\ \\ {}_t.F^{j+1} &= {}_t.F^{t+1},\ \forall 1 \leq t \leq j \end{cases}$$

The proof is based on the following idea. If we are able to find a derivation of $\langle c, \sigma\rangle$ that reaches the set $[\![p \not\bowtie 0]\!]$, then, by unfolding the loop once more, we make this derivation terminate in one more step. This case is characterized by the fact that σ belongs to a ${}_t.F^i$ for a $0 \leq t \leq i-1$. Note that, thanks to the last line of the above recurrence, this property can be reformulated as $\sigma \in {}_{i-1}.F^i$ for a certain $i \geq 1$. If we can't find such a derivation, it means that we keep infinitely deriving in the set $[\![p \bowtie 0]\!]$ and never leave the loop. This case is characterized by the fact that for all $i \geq 1$, σ belongs to ${}_i.F^i$. Note that either one of this characterization is realized. The rest of the proof consists in formalizing this idea.

The following lemma expresses what occurs if the first case is satisfied.

Lemma 5 (First case) *Let $c \equiv$ **while** $p \bowtie 0$ **do** c_1. Then,*

$$\forall \sigma \in \mathbb{R}^m, \forall j \geq 1, \left(\sigma \in {}_{j-1}.F^j \Rightarrow \left\{ \begin{array}{rl} \exists \sigma_f \in S, & \langle c, \sigma\rangle \to^+ \sigma_f \\ \text{or} & \langle c, \sigma\rangle \to^\infty \end{array} \right.\right)$$

Proof. By recurrence on j. Let $\sigma \in \mathbb{R}^m$.

- If $j = 1$. Then $\sigma \in {}_0.F^1 = [\![p \not\bowtie 0]\!] \bigcap S$ and
$$\frac{p(\sigma) \bowtie 0 \equiv \mathit{false}}{\langle c, \sigma\rangle \to \sigma}\ \textbf{\textit{if}}$$
Defining $\sigma_f = \sigma$ concludes this case.
- Suppose that property holds at rank j and let $\sigma \in {}_j.F^{j+1} = [\![p \bowtie 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ ({}_{j-1}.F^j)$. Then, we have
$$\frac{p(\sigma) \bowtie 0 \equiv \mathit{true}}{\langle c, \sigma\rangle \to \langle (c_1; c), \sigma\rangle}\ \textbf{\textit{if}}$$
and finally
$$\langle c_1; c, \sigma\rangle \ \sim\ \langle c, \langle c_1, \sigma\rangle\rangle \longrightarrow {}^+\langle c, \sigma'\rangle \longrightarrow {}^+\sigma''$$
$$\qquad\qquad\qquad\searrow \infty \qquad\qquad\qquad\searrow \infty$$
First step is just a macro expansion. Second step is the result of syntactic induction: as $\sigma \in {}_j.F^{j+1} \subseteq \mathrm{B}^\nu[\![c_1]\!]\ ({}_{j-1}.F^j)$, either there exists a state σ' such that $\sigma' \in {}_{j-1}.F^j$ and $\langle c_1, \sigma\rangle \to^+ \sigma'$ or $\langle c_1, \sigma\rangle \to^\infty$. The final step results from the recurrence hypothesis. Defining $\sigma_f = \sigma''$ concludes this case.

□

The second case is treated with help to the following lemma.

Lemma 6 (Second case) *Let $c \equiv$ **while** $p \bowtie 0$ **do** c_1. Then,*

$$\forall \sigma \in \mathbb{R}^m, \forall j \geq 1, \left(\sigma \in {}_j.F^j \Rightarrow \left\{ \begin{array}{l} \exists i_j \geq j-1, \sigma' \in [\![p \bowtie 0]\!], \quad \langle c, \sigma \rangle \rightarrow^{i_j} \langle c, \sigma' \rangle \\ \text{or} \quad \langle c, \sigma \rangle \rightarrow^{\infty} \end{array} \right. \right)$$

Proof. By recurrence on j. Let $\sigma \in \mathbb{R}^m$.

- If $j = 1$. Then $\sigma \in {}_1.F^1 = [\![p \bowtie 0]\!] \bigcap \mathrm{B}^{\nu}[\![prog_1]\!]$ $(\mathbb{R}^m)$. As $\langle c, \sigma \rangle \rightarrow^0 \langle c, \sigma \rangle$, defining $\sigma' = \sigma$ concludes the case.
- Suppose that property holds at rank j and let $\sigma \in {}_{j+1}.F^{j+1} = [\![p \bowtie 0]\!] \bigcap \mathrm{B}^{\nu}[\![c_1]\!]$ $({}_j.F^j)$. Then, we have

$$\frac{p(\sigma) \bowtie 0 \equiv \mathit{true}}{\langle c, \sigma \rangle \rightarrow \langle (c_1; c), \sigma \rangle}\ \textbf{\textit{if}}$$

which provides, as in the previous lemma

$$\langle c_1; c, \sigma \rangle \ \sim\ \langle c, \langle c_1, \sigma \rangle \rangle \longrightarrow {}^{+}\langle c, \sigma' \rangle \longrightarrow {}^{i_j}\sigma''$$

$$\langle c, \langle c_1, \sigma \rangle \rangle \searrow \infty \qquad \langle c, \sigma' \rangle \searrow \infty$$

First step is just a macro expansion. Second step is the result of syntactic induction: as $\sigma \in {}_{j+1}.F^{j+1} \subseteq \mathrm{B}^{\nu}[\![c_1]\!]$ $({}_j.F^j)$, either there exists a state σ' such that $\sigma' \in {}_j.F^j$ and $\langle c_1, \sigma \rangle \rightarrow^+ \sigma'$ or $\langle c_1, \sigma \rangle \rightarrow^{\infty}$. The final step results from the recurrence hypothesis. Note that the whole derivation is done in more than j steps, which concludes this case.

□

Now that we have stated these two lemmas, it remains to conclude the proof. To do it, we consider the set $Aux = \{i \in \mathbb{N} \mid \sigma \in {}_{i-1}.F^i\}$. Then, two cases can arise.

1. Either Aux is not empty, and there exists $k \in Aux$.
   Then, $\sigma \in {}_{k-1}.F^k$ and Lemma 5 allows to conclude this case.
2. Or Aux is empty.
   Then, $\forall j \geq 1, \sigma \in {}_j.F^j$. By Lemma 6, we get that either $\exists j \geq 1$ such that $(\sigma \in {}_j.F^j \Rightarrow \langle c, \sigma \rangle \rightarrow^{\infty})$ or $\forall j \geq 1, \exists i_j \geq j-1, \langle c, \sigma \rangle \rightarrow^{i_j} \langle c, \sigma' \rangle$. This means that we can always find a derivation as long as we want. In other words, $\langle c, \sigma \rangle \rightarrow^{\infty}$. This concludes this last case and the proof.

□

B Correctness

Theorem 9 (Correctness) *Let I be a polynomial ideal and c a polynomial program. Then:*

$$\gamma([\![c]\!]^{\sharp} I) \quad \subseteq \quad B^{\nu}[\![c]\!]\ \gamma(I)$$

We now prove Theorem 9 by syntactic induction on polynomial programs.

Proof. Let I be a polynomial ideal. Note that, by Theorem 1, I is finitely generated. Thus, there exist $g_1, \ldots, g_s \in \mathbb{R}[\mathtt{x}_1, \ldots, \mathtt{x}_m]$ such that $I = <g_1, \ldots, g_s>$.

If $c \equiv$ **skip** then

$$\begin{array}{rcl} [\![c]\!]^\sharp I & = & I \\ \mathrm{B}^\nu[\![c]\!]\ \gamma(I) & = & \gamma(I) \end{array}$$

And thus, this case is trivial.

If $c \equiv \mathtt{x_j} := p$ then

$$\begin{array}{rcl} [\![c]\!]^\sharp I & = & I[\mathtt{x_j} \mapsto p] \\ \mathrm{B}^\nu[\![c]\!]\ \gamma(I) & = & \{y \in \mathbb{R}^m \mid y[p(y)]_\mathtt{j} \in \gamma(I)\} = \gamma(I[\mathtt{x_j} \mapsto p]) \end{array}$$

where we define $I[\mathtt{x_j} \mapsto p] = <g_1[\mathtt{x_j} \mapsto p], \ldots, g_s[\mathtt{x_j} \mapsto p]>$.
This case is thus also trivial.

If $c \equiv c_1; c_2$ then

$$\begin{array}{rcl} [\![c]\!]^\sharp I & = & [\![c_1]\!]^\sharp([\![c_2]\!]^\sharp(I)) \\ \mathrm{B}^\nu[\![c]\!]\ \gamma(I) & = & \mathrm{B}^\nu[\![c_1]\!]\ (\mathrm{B}^\nu[\![c_2]\!]\ \gamma(I)) \end{array}$$

By induction on c_1, we have:

$$\gamma([\![c_1]\!]^\sharp([\![c_2]\!]^\sharp I)) \quad \subseteq \quad \mathrm{B}^\nu[\![c_1]\!]\ \gamma([\![c_2]\!]^\sharp I)$$

Now, by induction on c_2, we have:

$$\gamma([\![c_2]\!]^\sharp I) \quad \subseteq \quad \mathrm{B}^\nu[\![c_2]\!]\ \gamma(I)$$

As the concrete semantics $\mathrm{B}^\nu[\![c]\!]$ is a monotonic function, we conclude this case by applying $\mathrm{B}^\nu[\![c_1]\!]$ to each member of the subset inequality.

If $c \equiv$ **if** $p \neq 0$ **then** c_1 **else** c_2 then

$$\begin{array}{rcl} [\![c]\!]^\sharp I & = & < p.([\![c_1]\!]^\sharp I),\ \mathtt{Rem}([\![c_2]\!]^\sharp I, p) > \\ \mathrm{B}^\nu[\![c]\!]\ \gamma(I) & = & ([\![p \neq 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ \gamma(I)) \\ & & \bigcup\ ([\![p = 0]\!] \bigcap \mathrm{B}^\nu[\![c_2]\!]\ \gamma(I)) \end{array}$$

The variety of the abstract semantics is:

$$\begin{array}{rcl} & \gamma(<p.([\![c_1]\!]^\sharp I),\ \mathtt{Rem}([\![c_2]\!]^\sharp I, p)>) & \\ = & \gamma(p.([\![c_1]\!]^\sharp I)) \bigcap \gamma(\mathtt{Rem}([\![c_2]\!]^\sharp I, p)) & \\ = & (\gamma(p) \bigcup \gamma([\![c_1]\!]^\sharp I)) \bigcap \gamma(\mathtt{Rem}([\![c_2]\!]^\sharp I, p)) & \\ = & \gamma(p) \bigcap \gamma(\mathtt{Rem}([\![c_2]\!]^\sharp I, p)) & \} A \\ \bigcup & \gamma([\![c_1]\!]^\sharp I) \bigcap \gamma(\mathtt{Rem}([\![c_2]\!]^\sharp I, p)) & \} B \end{array}$$

In order to improve the readability of the proof, we introduce the capital letters A and B (on the right of the previous formula). Now, writing $\mathbb{R}^m$ as $([\![p \neq 0]\!] \bigcup \gamma(p))$, we have:

$$\begin{array}{rcl} & A \bigcup B & \\ = & (A \bigcup B) \bigcap ([\![p \neq 0]\!] \bigcup \gamma(p)) & \\ = & (\ A \bigcap ([\![p \neq 0]\!] \bigcup \gamma(p))\ ) \bigcup (\ B \bigcap ([\![p \neq 0]\!] \bigcup \gamma(p))\ ) & \\ = & A \bigcup (B \bigcap [\![p \neq 0]\!]) \bigcup (B \bigcap \gamma(p)) & \\ = & A \bigcup (B \bigcap [\![p \neq 0]\!]) & \end{array}$$

The former equality is a direct consequence of the fact that:

$$B \bigcap \gamma(p) \quad = \quad A \bigcap \gamma([\![c_1]\!]^\sharp I) \quad \subseteq \quad A$$

To conclude the proof, we prove that:

$$B \bigcap [\![p \neq 0]\!] \quad \subseteq \quad ([\![p \neq 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ \gamma(I)) \tag{1}$$

$$A \quad \subseteq \quad ([\![p = 0]\!] \bigcap \mathrm{B}^\nu[\![c_2]\!]\ \gamma(I)) \tag{2}$$

We first prove inequality (1). By induction hypothesis on c_1, we have:

$$\gamma([\![c_1]\!]^\sharp I) \subseteq \mathrm{B}^\nu[\![c_1]\!]\ \gamma(I)$$

Then, intersecting each member of the inclusion by $[\![p \neq 0]\!]$, we get:

$$([\![p \neq 0]\!] \bigcap \gamma([\![c_1]\!]^\sharp I)) \subseteq ([\![p \neq 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ \gamma(I))$$

which proves (1).

We now prove inequality (2). Let x be an element of A and u be an element of $[\![c_2]\!]^\sharp I$. We note $t = \mathtt{Rem}(u, p)$ and q the quotient associated to this division: $u = q.p + t$. As $x \in A$, $x \in \gamma(p)$ and so $p(x) = 0$. We also have $x \in \gamma(\mathtt{Rem}([\![c_2]\!]^\sharp I, p))$. Thus, $t(x) = 0$, which implies that $u(x) = 0$. We proved that for all $u \in [\![c_2]\!]^\sharp I$, $u(x) = 0$, which means that $x \in \gamma([\![c_2]\!]^\sharp I)$ and demonstrates that $A \subseteq [\![p = 0]\!] \bigcap \gamma([\![c_2]\!]^\sharp I)$. We conclude the proof of inequality (2) thanks to the induction hypothesis on c_2.

If $c \equiv$ **if** $p = 0$ **then** c_1 **else** c_2 : this case is symmetric to the previous one and can be treated in the same way. We do not develop it here.

If $c \equiv$ **while** $p \neq 0$ **do** c_1 then

$$\begin{array}{rcl} [\![c]\!]^\sharp I & = & \nu\lambda J. < p.([\![c_1]\!]^\sharp(J)), \mathtt{Rem}(I, p) > \\ \mathrm{B}^\nu[\![c]\!]\ \gamma(I) & = & \nu\lambda X.([\![p = 0]\!] \bigcap \gamma(I)) \bigcup ([\![p \neq 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ X) \end{array}$$

To deal with this case, very similar to the previous one, we use the following theorem on fixpoints [1].

Lemma 7 *Transfer lemma*
Let $(\mathcal{A}, \sqsubseteq), (\mathcal{A}^\sharp, \sqsubseteq^\sharp)$ be two complete lattices and $\gamma : \mathcal{A}^\sharp \to \mathcal{A}$ a function. Let $f : \mathcal{A} \to \mathcal{A}$ and $f^\sharp : \mathcal{A}^\sharp \to \mathcal{A}^\sharp$ be two monotonic functions such that

$$\gamma \circ f^\sharp \dot{\sqsubseteq} f \circ \gamma$$

Then we have:

$$\gamma(\nu f^\sharp) \sqsubseteq \nu f$$

Proof. Let us apply the hypothesis to $\nu f^\sharp$. We obtain $\gamma(f^\sharp(\nu f^\sharp)) \sqsubseteq f(\gamma(\nu f^\sharp))$. $\nu f^\sharp$ being a fixpoint of $f^\sharp$, we have $f^\sharp(\nu f^\sharp) = \nu f^\sharp$. Thus, we have $\gamma(\nu f^\sharp) \sqsubseteq f(\gamma(\nu f^\sharp))$ meaning that $\gamma(\nu f^\sharp)$ is a prefixpoint of f. By definition of νf, we conclude that $\gamma(\nu f^\sharp) \sqsubseteq \nu f$.

□

We note f the concrete function and $f^\sharp$ the abstract function defined by:

$$\begin{array}{lcl} f & = & \lambda X.([\![p=0]\!] \bigcap \gamma(I)) \bigcup ([\![p\neq 0]\!] \bigcap \mathrm{B}^\nu[\![c_1]\!]\ X) \\ f^\sharp & = & \lambda J. < p.([\![c_1]\!]^\sharp(J)), \mathtt{Rem}(I,p) > \end{array}$$

and show that $\gamma \circ f^\sharp \dot{\subseteq} f \circ \gamma$.

Let J_0 be a polynomial ideal. One the one hand, we have:

$$\begin{array}{ll} & f \circ \gamma(J_0) \\ = & ([\![p=0]\!] \bigcap \gamma(I)) \bigcup ([\![p\neq 0]\!] \bigcup \mathrm{B}^\nu[\![c_1]\!]\ \gamma(J_0)) \end{array}$$

On the other hand, we have:

$$\begin{array}{lllll} & \gamma \circ f^\sharp(J_0) & & & \\ = & \gamma(<p.([\![c_1]\!]^\sharp J_0), \mathtt{Rem}(I,p)>) & & & \\ = & (\ \gamma(p) \bigcup \gamma([\![c_1]\!]^\sharp J_0)\ ) & \bigcap & \gamma(\mathtt{Rem}(I,p)) & \\ = & (\ \gamma(p) \bigcap \gamma(\mathtt{Rem}(I,p))\ ) & \bigcup & (\ \gamma([\![c_1]\!]^\sharp J_0) \bigcap \gamma(\mathtt{Rem}(I,p))\ ) & \\ = & (\ \gamma(p) \bigcap \gamma(\mathtt{Rem}(I,p))\ ) & \bigcup & (\ [\![p\neq 0]\!] \bigcap \gamma([\![c_1]\!]^\sharp J_0) \bigcap \gamma(\mathtt{Rem}(I,p))\ ) & \end{array}$$

The former equality is obtained reasoning the same way as in the **if**-case, writing $\mathbb{R}^m = [\![p=0]\!] \bigcup [\![p\neq 0]\!]$. Now, by induction hypothesis on c_1, we have $\gamma([\![c_1]\!]^\sharp J_0) \subseteq \mathrm{B}^\nu[\![c_1]\!]\ \gamma(J_0)$. Reasoning again as in the **if**-case, we get $\gamma(p) \bigcap \gamma(\mathtt{Rem}(I,p)) \subseteq \gamma(p) \bigcap \gamma(I)$. Combining these two inclusions, we get $\gamma \circ f^\sharp \dot{\subseteq} f \circ \gamma$, which proves by Lemma 7:

$$\begin{array}{ccc} \gamma(\nu f^\sharp) & \subseteq & \nu f \\ \shortparallel & & \shortparallel \\ \gamma([\![c]\!]^\sharp I) & \subseteq & \mathrm{B}^\nu[\![c]\!]\ \gamma(I) \end{array}$$

This concludes this case.

If $c \equiv$ **while** $p=0$ **do** c_1 : this case is symmetric to the previous one and can be treated in the same way. We do not develop it here.

□

C Benchmarks

Table 5 presents the results of the Maple implementation of our analysis. Our tests were run on a 2.8 GHz Intel Core 2 Duo sporting 4 Go of DDR3 RAM, on Rodríguez-Carbonell and Kapur's inductive benchmarks [17, 16], and measured to their best and worst results. Our method is systematically better, and on average 2.2 orders of magnitude faster.

A few examples deserve additional comments. We found an higher-order invariant ($\mathtt{x} = \mathtt{n}^3$) for `cohencu` starting from the most generic a_i-*lpp* of degree

Name	d	Var	Comparing performance				
			RCK approach		Fast Ind	RCK/Fast Ind	
			fastest	slowest	in seconds	worst	best
cohencu	2	4	0.7	1.88	0.009	78	209
divbin	2	5	0.99	2.1	0.005	198	420
freire1	2	3	0.38	0.7	0.006	63	116
freire2	2	4	0.7	1.0	0.007	100	143
mannadiv	2	5	1.12	2.96	0.005	224	592
knuth	3	9	2.61	55.4	0.084	31	659
prod4br	3	6	4.63	> 300	0.013	356	$> 23.10^3$
euclidex2	2	8	1.95	15.15	0.008	244	1894
fermat1	2	5	0.8	2.57	0.006	133	428
fermat2	2	5	0.92	1.49	0.006	153	248
lcm1	2	6	1.0	2.49	0.006	167	415
lcm2	2	6	1.21	2.05	0.006	202	342
petter1	2	2	0.5	1.0	0.003	83	167
petter5	6	2	1.4	> 300	0.006	233	$> 50.10^3$
sqrt	2	3	0.46	–	0.010	46	–
z3sqrt	2	4	0.82	–	0.006	137	–
cohencu	3	4	–	–	0.018	–	–
freire2	3	4	–	–	0.012	–	–
petter30	31	2	–	–	1.423	–	–

Figure 5: Performance results for the Maple implementation

3 and not restricting ourselves to degree 2 as it is done in Rodríguez-Carbonell and Kapur's work. The same way, we found the higher-order invariant $4\mathtt{r}^3 - 6\mathtt{r}^2 + 3\mathtt{r} + 4\mathtt{x} - 4\mathtt{a} = 1$ for `freire2`. In Figure 6, we formally define `petter30`, a polynomial program that computes the sum of the first n numbers to the power 30 and provides the following invariant of degree 31 at the end of the program.

1. $\mathtt{x} := 0;$
2. $\mathtt{y} := 0;$
3. `while` $\mathtt{y} \neq n$ `do`
4. $\quad \mathtt{x} := \mathtt{x} + \mathtt{y}^{30};$
5. $\quad \mathtt{y} := \mathtt{y} + 1;$
6.

Figure 6: `petter30`, a polynomial program.

The `knuth` program has non-polynomial initial assignments: $r = n\%d$; $rp = n\%(d-2)$; $q = 4 * (n/(d-2) - n/d)$; where % denote the modulo operator and / denote the integer division. We introduce two variables n_1 and n_2 to code the integer division and rewrite the assignments into: $r = n - n_1 * d$; $rp = n - n_2 * (d-2)$; $q = 4 * (n_2 - n_1)$;. These new variables are necessary to generate the sufficient initial constraints to infer the program invariant.

Finally, Figure 7a and 7b define two programs derived from `mannadiv`. Their main interest is to introduce non linear polynomial guards in the *loop* or in the *if* statements. Proceeding the same way as it is done in the `mannadiv` example, we infer respectively the polynomial invariants $\mathtt{x}^2 + 2\mathtt{t} + \mathtt{y} - \mathtt{n} = 0$ and $\mathtt{x}^3 + 3\mathtt{t} + \mathtt{y} - \mathtt{n} = 0$ at the end of the program. These changes do not cause a significative loss of performance of our method since they are respectively generated in 0.005 seconds and 0.008 seconds.

```
1.  y := n;
2.  x := 0;
3.  t := 0;
4.  while y.(y − 1) ≠ 0 do
5.     if t = x then
6.        y := y − 1;
7.        t := 0;
8.        x := x + 1;
9.     else
10.       y := y − 2;
11.       t := t + 1;
12.
```

(a) `mannadiv2`.

```
1.  y := n;
2.  x := 0;
3.  t := 0;
4.  while y > 0 do
5.     if t = x^2 then
6.        y := y − 3x − 1;
7.        t := 0;
8.        x := x + 1;
9.     else
10.       y := y − 3;
11.       t := t + 1;
12.
```

(b) `mannadiv3`.

Figure 7: Two polynomial programs derived from `mannadiv`.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399