Automatic Generation of Polynomial Invariants of Bounded Degree using Abstract Interpretation

E. Rodríguez-Carbonell [a,*], D. Kapur [b]

[a] *Software Department, Technical University of Catalonia, Jordi Girona, 1-3 08034 Barcelona (Spain)*

[b] *Department of Computer Science, University of New Mexico, Albuquerque NM 87131-0001 (USA)*

Abstract

A method for generating polynomial invariants of imperative programs is presented using the abstract interpretation framework. It is shown that for programs with polynomial assignments, an invariant consisting of a conjunction of polynomial equalities can be automatically generated for each program point. The proposed approach takes into account tests in conditional statements as well as in loops, insofar as they can be abstracted into polynomial equalities and disequalities. The semantics of each program statement is given as a transformation on polynomial ideals. Merging of execution paths is defined as the intersection of the polynomial ideals associated with each path. For loop junctions, a family of widening operators based on selecting polynomials up to a certain degree is proposed. The presented method has been implemented and successfully tried on many programs. Heuristics employed in the implementation to improve its efficiency are discussed, and tables providing details about its performance are included.

Key words:
Abstract interpretation, Invariant, Ideal of polynomials, Gröbner basis

---

* Corresponding author.
Email addresses: `erodri@lsi.upc.edu` (E. Rodríguez-Carbonell), `kapur@cs.unm.edu` (D. Kapur).
URLs: `www.lsi.upc.edu/~erodri` (E. Rodríguez-Carbonell), `www.cs.unm.edu/~kapur` (D. Kapur).

1 Introduction

There has recently been a surge of interest in research on automatic generation of invariants in imperative programs. This is perhaps due to the successful development of powerful automated reasoning tools including BDD packages, SAT solvers, model checkers, decision procedures for common data structures in applications (such as numbers, lists, arrays, ...), as well as theorem provers for first-order logic, higher-order logic and induction. These tools have been successfully used in application domains such as hardware circuits and designs, software and protocol analysis.

A method for generating polynomial invariants for imperative programs is developed in this paper. It is analogous to the approach proposed in [12] for finding linear inequalities as invariants based on the abstract interpretation framework [9]. The proposed method, in contrast, generates polynomial equalities as invariants by soundly abstracting the semantics of programming language constructs in terms of ideal-theoretic operations. It is shown that, for programs with polynomial assignments, an invariant consisting of a conjunction of polynomial equalities can be automatically generated for each program point.

The presented approach is able to handle nested loops [1] and also takes into account tests in conditional statements and loops, insofar as they can be abstracted into polynomial equalities and disequalities. Merging of execution paths in a program is defined as the intersection of the polynomial ideals associated with each path. In order to ensure termination, a family of widening operators [9,12] is proposed based on retaining only the polynomials of degree $\leq d$ in the intersection of ideals (at the merging of paths). This is achieved by computing a Gröbner basis [13,14] with a graded term ordering and keeping only those polynomials in the basis with degree $\leq d$, where both the term ordering and the degree bound are parameters of the analysis.

The proposed method has been implemented using Macaulay 2 [19], an algebraic geometry tool that supports operations on polynomial ideals such as the computation of Gröbner bases. Using this implementation, loop invariants for several programs have been successfully generated automatically.

The method does not need pre/postconditions for deriving invariants. Further, if tests in conditional statements and loops are ignored, and if all right-hand sides of assignments are linear, the technique finds all polynomial invariants of degree $\leq d$, where d is the degree bound in the widening. In that sense, the method is *sound* and *complete.*

[1] The method also works for unnested loops with unstructured control flow, using Bourdoncle's algorithm [2] to find adequate widening points in the flow graph.

The rest of the paper is organized as follows. In Section 2, related work is briefly reviewed. Section 3 gives background information on polynomial ideals, operations on them and special bases of polynomial ideals called Gröbner bases. Section 4 introduces a simple programming language used in the paper for presenting the method (although, in fact, the proposed techniques are independent of the programming model and can be applied in more general settings). Section 5 discusses the abstraction and concretization functions from variable values to ideals and viceversa, so that the framework of abstract interpretation is applicable. Section 6 gives the semantics of programming constructs using ideal-theoretic operations. For each kind of statement, it is shown how the output polynomial ideal can be obtained from the input polynomial ideals. Most importantly, Subsection 6.5 presents the semantics of loop junction nodes using widening operators. Section 7 shows that, if tests in conditionals and loops are ignored and all assignments are linear, the proposed method is sound and complete in the sense that, for every program point, all of the invariants of degree $\leq d$ are discovered, where d is the parameter used in the widening operator. Section 8 illustrates the application of the method on some examples, which have been analyzed by a prototype invariant generator that has been implemented using the algebraic geometry tool Macaulay 2 [19]. A number of heuristics to improve the efficiency of the implementation are provided, together with tables comparing these heuristics on a benchmark of programs taken from the literature. Finally, Section 9 concludes and discusses ideas for extending this research.

2 Related Work

As stated above, the approach in this paper complements the method proposed by Cousot and Halbwachs [12], who applied the framework of abstract interpretation [9] to the generation of linear inequalities as invariants. That work extended Karr's algorithm [24] for finding invariant linear equalities at any program point.

Recently, there has been a surge of interest in automatically deriving polynomial invariants of imperative programs. For programs with linear assignments, Müller-Olm and Seidl [29] have proposed an interprocedural method for computing polynomial equalities of bounded degree as invariants. The same authors [27] have also found a technique for discovering all the polynomial invariants of bounded degree in a program with polynomial assignments and just disequality tests. In contrast, our method can deal with non-linear polynomial assignments and both equality and disequality tests, at the cost of losing completeness.

In [31,32], we have developed an abstract framework for generating invariants of loops without nesting. This framework has been instantiated to generate conjunctions of polynomial equalities as invariants. The method uses the Gröbner basis algorithm to compute such invariants, and can be shown to be sound and complete. However, it cannot handle nested loops; furthermore, tests in conditional statements and loops are ignored.

In [34], Sankaranarayanan et al. have presented a method for generating nonlinear polynomials as invariants, which starts with a template polynomial with undetermined coefficients and attempts to find values for the coefficients so that the template is invariant by means of the Gröbner basis algorithm. Kapur has proposed a related approach using quantifier elimination [22]. Unlike these techniques, our method has been implemented and tried on many examples with considerable success.

Finally, at the 2004 Static Analysis Symposium in Verona, Italy, we have learned about an approach similar to ours [30] by Colón [7], based on abstract interpretation and the concept of pseudo-ideal. Whereas Colón's method can only consider equality tests, our method can handle both equality and disequality tests. This difference can be crucial in certain applications, like the analysis of Petri nets [5,27] and cache coherence protocols [15], as can be seen with the examples in Section 8.

3 Preliminaries

3.1 Ideals and Varieties

Given a field $\mathbb{K}$, let $\mathbb{K}[\bar{x}] = \mathbb{K}[x_1, ..., x_n]$ denote the ring of polynomials in the variables $x_1, ..., x_n$ with coefficients from $\mathbb{K}$. An *ideal* is a set $I \subseteq \mathbb{K}[\bar{x}]$ that contains 0, is closed under addition and is such that if $p \in \mathbb{K}[\bar{x}]$ and $q \in I$, then $pq \in I$. Given a set of polynomials $S \subseteq \mathbb{K}[\bar{x}]$, the *ideal spanned by S* is

$$\{f \in \mathbb{K}[\bar{x}] \mid \exists k \geq 1 \; f = \sum_{j=1}^{k} p_j q_j \text{ with } p_j \in \mathbb{K}[\bar{x}], q_j \in S\}\,.$$

This is the minimal ideal containing S, and we denote it by $\langle S \rangle_{\mathbb{K}[\bar{x}]}$ or simply by $\langle S \rangle$. For an ideal $I \subseteq \mathbb{K}[\bar{x}]$, a set $S \subseteq \mathbb{K}[\bar{x}]$ such that $I = \langle S \rangle$ is called a *basis* of I, and we say that S *generates* I.

Given two ideals $I,\, J \subseteq \mathbb{K}[\bar{x}]$, their *intersection* $I \cap J$ is an ideal. However, the union of ideals is, in general, not an ideal. The *sum* of I and J, $I + J = \{p + q \mid p \in I, q \in J\}$, is the minimal ideal that contains $I \cup J$. The quotient of I into J is the ideal $I : J = \{p \mid \forall q \in J,\, pq \in I\}$.

For any set S of polynomials in $\mathbb{K}[\bar{x}]$, the *variety* of S over $\mathbb{K}^n$ is defined as its set of zeroes,

$$\mathbf{V}(S) = \{\bar{\omega} \in \mathbb{K}^n \mid \forall p \in S,\ p(\bar{\omega}) = 0\}.$$

When referring to varieties, we can assume S to be an ideal, since $\mathbf{V}(\langle S \rangle) = \mathbf{V}(S)$. For $A \subseteq \mathbb{K}^n$, the ideal

$$\mathbf{I}(A) = \{p \in \mathbb{K}[\bar{x}] \mid \forall \bar{\omega} \in A,\ p(\bar{\omega}) = 0\}$$

is called the *ideal of* A. We write $\mathbf{IV}(S)$ instead of $\mathbf{I}(\mathbf{V}(S))$.

Ideals and varieties are dual concepts, in the sense that given ideals I, J, $\mathbf{V}(I \cap J) = \mathbf{V}(I) \cup \mathbf{V}(J)$ and $\mathbf{V}(I + J) = \mathbf{V}(I) \cap \mathbf{V}(J)$. Moreover, if $I \subseteq J$ then $\mathbf{V}(I) \supseteq \mathbf{V}(J)$. Analogously, if $A, B \subseteq \mathbb{K}^n$ (in particular, if A, B are varieties), then $\mathbf{I}(A \cup B) = \mathbf{I}(A) \cap \mathbf{I}(B)$ and $A \subseteq B$ implies $\mathbf{I}(A) \supseteq \mathbf{I}(B)$. However, in general for any two varieties V, W, the inclusion $\mathbf{I}(V \cap W) \supseteq \mathbf{I}(V) + \mathbf{I}(W)$ holds and may be strict; but $\mathbf{I}(V \cap W) = \mathbf{IV}(\mathbf{I}(V) + \mathbf{I}(W))$ is always true. For any ideal I, the inclusion $I \subseteq \mathbf{IV}(I)$ holds; $\mathbf{IV}(I)$ represents the largest set of polynomials with the same zeroes as I. Since any I satisfying $I = \mathbf{IV}(I)$ is the ideal of the variety $\mathbf{V}(I)$, we say that any such I is an *ideal of variety*. For any $A \subseteq \mathbb{K}^n$, it can be seen that the ideal $\mathbf{I}(A)$ is an ideal of variety. For further detail on these concepts, see [13,14].

3.2 Gröbner Bases: Special Bases of Ideals

A *term* in a tuple $\bar{x} = (x_1, ..., x_n)$ of variables is an expression of the form $\bar{x}^{\bar{\alpha}} = x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$, where $\bar{\alpha} = (\alpha_1, ..., \alpha_n) \in \mathbb{N}^n$. The set of terms is denoted by $\mathcal{T}$. A *monomial* is an expression of the form $c \cdot p$, with $c \in \mathbb{K}$ and $p \in \mathcal{T}$. The *degree* of a monomial $c \cdot \bar{x}^{\bar{\alpha}}$ with $c \neq 0$ is $\deg(c \cdot \bar{x}^{\bar{\alpha}}) = \alpha_1 + \cdots + \alpha_n$. The degree of a non-null polynomial is the maximum of the degrees of its monomials. Let $\mathbb{K}_d[\bar{x}]$ denote the set of all polynomials of $\mathbb{K}[\bar{x}]$ of degree $\leq d$.

An *admissible term ordering* $\succ$ is a relation over $\mathcal{T}$ such that:

(1) $\succ$ is a total ordering over $\mathcal{T}$.
(2) If $\bar{\alpha}, \bar{\beta}, \bar{\gamma} \in \mathbb{N}^n$ and $\bar{x}^{\bar{\alpha}} \succ \bar{x}^{\bar{\beta}}$, then $\bar{x}^{\bar{\alpha}+\bar{\gamma}} \succ \bar{x}^{\bar{\beta}+\bar{\gamma}}$.
(3) $\forall \bar{\alpha} \in \mathbb{N}^n$, $\bar{x}^{\bar{\alpha}} \succeq 1 = \bar{x}^{\bar{0}}$.

Moreover, $\succ$ is called a *graded term ordering* if $\forall \bar{\alpha}, \bar{\beta} \in \mathbb{N}^n$, $\deg(\bar{x}^{\bar{\alpha}}) > \deg(\bar{x}^{\bar{\beta}})$ implies $\bar{x}^{\bar{\alpha}} \succ \bar{x}^{\bar{\beta}}$.

Term orderings extend to monomials by ignoring the coefficients and comparing the corresponding terms. The most common term orderings are defined as follows, assuming that $x_1 \succ x_2 \succ \cdots \succ x_n$:

- **Lexicographical Ordering (lex).** If $\bar{\alpha}, \bar{\beta} \in \mathbb{N}^n$, then $\bar{x}^{\bar{\alpha}} \succ_{\text{lex}} \bar{x}^{\bar{\beta}}$ iff the leftmost non-zero entry in $\bar{\alpha} - \bar{\beta}$ is positive.
- **Graded Lexicographical Ordering (grlex).** If $\bar{\alpha}, \bar{\beta} \in \mathbb{N}^n$, then $\bar{x}^{\bar{\alpha}} \succ_{\text{grlex}} \bar{x}^{\bar{\beta}}$ iff $\deg(\bar{x}^{\bar{\alpha}}) > \deg(\bar{x}^{\bar{\beta}})$, or $\deg(\bar{x}^{\bar{\alpha}}) = \deg(\bar{x}^{\bar{\beta}})$ and $\bar{x}^{\bar{\alpha}} \succ_{\text{lex}} \bar{x}^{\bar{\beta}}$.
- **Graded Reverse Lexicographical Ordering (grevlex).** If $\bar{\alpha}, \bar{\beta} \in \mathbb{N}^n$, then $\bar{x}^{\bar{\alpha}} \succ_{\text{grevlex}} \bar{x}^{\bar{\beta}}$ iff $\deg(\bar{x}^{\bar{\alpha}}) > \deg(\bar{x}^{\bar{\beta}})$, or $\deg(\bar{x}^{\bar{\alpha}}) = \deg(\bar{x}^{\bar{\beta}})$ and the rightmost non-zero entry in $\bar{\alpha} - \bar{\beta}$ is negative.

The orderings **grlex** and **grevlex** are examples of graded term orderings.

Term orderings are used to define a notion of *reduction* over polynomials. For any polynomial $g \in \mathbb{K}[\bar{x}]$, let $\mathrm{lm}(g)$ denote the largest monomial (with respect to a fixed term ordering) among all the monomials in g, which we call the *leading monomial* of g. Given polynomials $f, g \in \mathbb{K}[\bar{x}]$, the reduction relation is defined as $f \xrightarrow{g} f'$ iff there exists a monomial m in f such that $\mathrm{lm}(g)$ divides m and

$$f' = f - \frac{m}{\mathrm{lm}(g)} g.$$

The purpose of this reduction is to eliminate the monomial m from f using g. Given a finite set of polynomials G, we write

$$f \xrightarrow{G} f'$$

if $f \xrightarrow{g} f'$ for some $g \in G$, and also

$$f \xrightarrow{G}{}^{*} f'$$

if f reduces to f' in finitely many reduction steps. If a polynomial cannot be further reduced, we say that it is *in normal form* modulo G. Thanks to the properties of term orderings, for any polynomial f a normal form of f modulo G can be computed in a finite number of steps. However, in general, normal forms are not necessarily unique.

Given an ideal I, a *Gröbner basis* of I is a finite set of polynomials G satisfying $I = \langle G \rangle$ and such that normal forms modulo G are unique, i.e., for any $f \in \mathbb{K}[\bar{x}]$

$$(\ f \xrightarrow{G}{}^{*} f_1\ , \quad f \xrightarrow{G}{}^{*} f_2\ , \quad f_1 \text{ and } f_2 \text{ are in normal form }) \Longrightarrow \quad f_1 = f_2\,.$$

If $G = \{g_1, ..., g_k\}$ is a Gröbner basis of an ideal I, then $\forall f \in I$, the normal form of f modulo G is 0. Thus, by applying reduction steps repeatedly, we can decompose f in terms of the polynomials in G: there exist q_1, ..., $q_k \in \mathbb{K}[\bar{x}]$ such that $f = \sum_{j=1}^{k} q_j g_j$ and $\forall j : 1 \leq j \leq k$, $q_j \neq 0$ implies $\mathrm{lm}(f) \succeq \mathrm{lm}(q_j g_j)$.

Example 1 *Consider the set of polynomials* $S = \{x_2^2 - x_1, -x_2^2 + x_2\} \subset$

$\mathbb{K}[x_1, x_2]$ *and the term ordering* **grevlex** *with* $x_1 \succ x_2$. *Let us reduce the polynomial* x_2^2 *with respect to* S. *Since we have that* $\mathrm{lm}(x_2^2 - x_1) = x_2^2$ *and* $\mathrm{lm}(-x_2^2 + x_2) = -x_2^2$, *we can rewrite* x_2^2 *in two different ways:*

$$x_2^2 \xrightarrow{S} x_1 \qquad \textit{since } x_2^2 - (x_2^2 - x_1) = x_1 \ ,$$

and

$$x_2^2 \xrightarrow{S} x_2 \qquad \textit{since } x_2^2 + (-x_2^2 + x_2) = x_2 \ .$$

In fact, as neither x_1 *nor* x_2 *can be further reduced with respect to* S, *both are normal forms of* x_2^2, *and so* S *is not a Gröbner basis of* $\langle S \rangle$. *Nevertheless, we can eliminate the conflict by forcing that* x_1 *rewrites into* x_2: *if we consider the polynomial* $x_1 - x_2 = -(x_2^2 - x_1) - (-x_2^2 + x_2)$ *and define* $G = \{x_1 - x_2, x_2^2 - x_1, -x_2^2 + x_2\}$, *then* x_2 *is the only normal form of* x_2^2 *with respect to* G, *since* $x_1 \xrightarrow{G} x_2$. *Indeed,* G *is a Gröbner basis of* $\langle G \rangle = \langle S \rangle$. *However,* G *is redundant: if we define* $G' = \{x_1 - x_2, -x_2^2 + x_2\}$, *we have that*

$$x_2^2 - x_1 \xrightarrow{G'} x_2^2 - x_2 \xrightarrow{G'} 0 \ ,$$

which corresponds with the fact that $x_2^2 - x_1 = -(x_1 - x_2) - (-x_2^2 + x_2)$. *Therefore,* G' *is also a Gröbner basis of* $\langle S \rangle$ *(which is irredundant).*

4 Programming Model

In order to simplify the presentation, programs are represented as finite connected flowcharts with one entry node and assignment, test, junction and exit nodes, as in [12]. We also assume that the evaluation of arithmetic and boolean expressions has no side effects and so does not affect the values of program variables, which are denoted by $x_1, ..., x_n$.

Formally, nodes for flowcharts are taken from a set **Nodes**, which is partitioned into the following subsets (we show the respective symbol used in figures for each subset in parentheses below):

(1) **Entry ($\rhd\!\!\longrightarrow$).** There is just one entry node, which has no predecessors and one successor. It means where the flow of the program begins.
(2) **Assignments ($\square$).** Assignment nodes have one predecessor and one successor. Every assignment node is labelled with a variable x_i and an expression $f(\bar{x})$, thus representing the assignment $x_i := f(\bar{x})$.
(3) **Tests ($\bigcirc$).** A test node has a predecessor and two successors, corresponding to the *true* and *false* paths. It is labelled with a boolean expression $C(\bar{x})$, which is evaluated when the flow reaches the node.

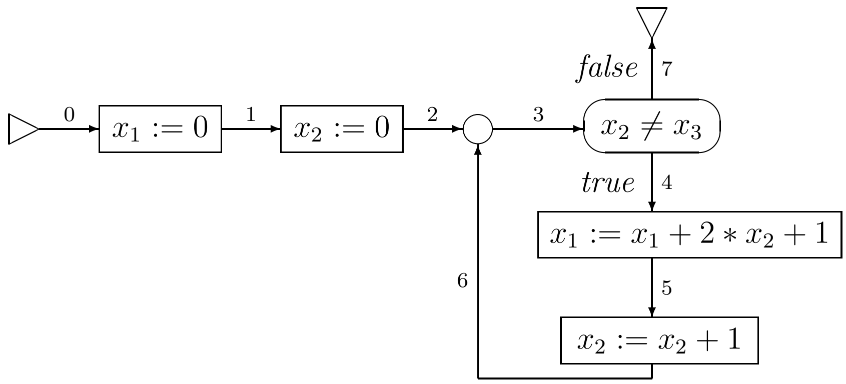

Fig. 1. Example of a program

(4) **Junctions (◯).** Junction nodes have one successor and more than one predecessor. They involve no computation and only represent the merging of execution paths (in conditional and loop statements).
(5) **Exits ($\rightarrow\!\!\lhd$).** Exit nodes have just one predecessor and no successors. They represent where the flow of the program halts.

For example, the program in Figure 1 incrementally computes the sequence of squares of the first x_3 natural numbers, stored in the variable x_1.

5 Ideals of Variety as Abstract Values

A state of the computation at any given program point is any tuple of values program variables can take. A set of states, considered as a subset of $\mathbb{K}^n$, can be abstracted into the ideal consisting of all polynomials that vanish in those states. This is how the abstraction function is intuitively defined.

At the abstract level, we work with polynomial ideals (more specifically, with ideals of variety, i.e., with ideals I such that $I = \mathbf{IV}(I)$). To each arc a of the flowchart (which represents a program point), we attach an assertion P_a of the form $P_a = \{\wedge_{j=1}^{k} p_{aj}(\bar{x}) = 0\}$, or equivalently the ideal $I_a = \langle p_{a1}(\bar{x}), ..., p_{ak}(\bar{x})\rangle$, where the $p_{aj} \in \mathbb{K}[\bar{x}]$ are polynomials. The abstraction function,

$$\alpha = \mathbf{I} : 2^{\mathbb{K}^n} \to \mathcal{I},$$

is the ideal operator, which yields the ideal of the polynomials that vanish at the points in the given subset of $\mathbb{K}^n$; and the concretization function,

$$\gamma = \mathbf{V} : \mathcal{I} \to 2^{\mathbb{K}^n},$$

is the variety operator (where $2^{\mathbb{K}^n}$ denotes the powerset of $\mathbb{K}^n$ and $\mathcal{I}$ is the set of ideals of variety in $\mathbb{K}[\bar{x}]$). Both $(2^{\mathbb{K}^n}, \subseteq, \cup, \emptyset, \mathbb{K}^n)$ and $(\mathcal{I}, \supseteq, \cap, \langle 1\rangle, \{0\})$ are semi-lattices, and the functions defined above are morphisms between these semi-lattices. These operators form a Galois connection, as $\forall A \subseteq \mathbb{K}^n\ \forall I \in \mathcal{I}$, $\mathbf{I}(A) \supseteq I \Leftrightarrow A \subseteq \mathbf{V}(I)$. The semantics of program constructs for abstract values is given in Section 6 as transformations on polynomial ideals.

Our goal is to compute the ideal of the polynomials that vanish at the reachable states at each program point, in other words the invariant ideal at each program point. This can be done as follows. The output ideal of the entry node represents the precondition, i.e., what is known about the variables at the start of the execution of the program. After initializing the reachable states of any arc to the empty set (i.e., $I_a = \langle 1 \rangle$, the bottom of $\mathcal{I}$), we propagate the precondition ideal around the flowchart by application of the semantics until stabilization, i.e., until a fixpoint is reached. To carry out this forward propagation, a particular iteration strategy can be chosen among several fixpoint algorithms [2,10]. In order to guarantee termination, we assume that each cycle in the graph contains a special junction node, called *loop junction node*, for which the reachable states are extrapolated by means of a *widening operator* ∇. Intuitively, loop junction nodes correspond to loops, whereas simple junction nodes are associated with conditional statements.

6 Transformation of Ideals of Variety by Language Constructs

This section develops a semantics of programs in terms of ideals of variety; i.e., for each kind of program node, we show how the output ideal of variety can be obtained from the input ideals of variety and the relevant information attached to the node.

6.1 Program Entry Node

If we are given a conjunction of polynomial equalities as a precondition for the procedure to be analyzed, the $\mathbf{IV}(\cdot)$ of the polynomials in it can be used as the output ideal of variety for the program entry node. Otherwise, if the variables are assumed not to be initialized, they do not satisfy any constraints and the tuple of their values may be any point in $\mathbb{K}^n$. This is represented by the zero ideal $\{0\} = \mathbf{I}(\mathbb{K}^n)$, whose corresponding assertion is the tautology $0 = 0$.

In the program from Section 4, if we do not assume anything about the initial values of x_1, x_2 and x_3, we take $I_0 = \{0\}$.

Let $I = \langle p_1, ..., p_k \rangle$ be the input ideal of variety of the assignment node, x_i be the variable that is assigned and $f(\bar{x})$ be the right-hand side of the assignment.

The strongest postcondition of the assertion $\{ \wedge_{j=1}^{k} p_j(\bar{x}) = 0 \}$ after the assignment $x_i := f(\bar{x})$ is

$$\{ \exists x'_i \, (x_i = f(x_i \leftarrow x'_i) \wedge (\wedge_{j=1}^{k} p_j(x_i \leftarrow x'_i) = 0)) \} \, ,$$

where intuitively x'_i stands for the value of the assigned variable previous to the assignment, and $\leftarrow$ denotes substitution of variables. Our goal now is to express this formula in terms of ideals of variety.

Assuming $f(\bar{x}) \in \mathbb{K}[\bar{x}]$, we translate the equality $x_i = f(x_i \leftarrow x'_i)$ into the polynomial $x_i - f(x_i \leftarrow x'_i)$ and consider the ideal

$$I' = \langle x_i - f(x_i \leftarrow x'_i), p_1(x_i \leftarrow x'_i), ..., p_k(x_i \leftarrow x'_i) \rangle_{\mathbb{K}[x'_i, \bar{x}]} \, .$$

This ideal I' captures the effect of the assignment, with the drawback that a new variable x'_i has been introduced. We have to eliminate this variable x'_i from I' and then compute the corresponding ideal of variety; in other words, we need to compute all those polynomials in I' that depend only on the variables in $\bar{x}$, i.e., $I' \cap \mathbb{K}[\bar{x}]$, and then take $\mathbf{IV}(I' \cap \mathbb{K}[\bar{x}])$. By Lemma 15 in Appendix A, $\mathbf{IV}(I' \cap \mathbb{K}[\bar{x}]) = I' \cap \mathbb{K}[\bar{x}]$, so the output is $I' \cap \mathbb{K}[\bar{x}]$.

In our running example, assume that $I_0 = \{0\}$ and that we want to compute the output ideal I_1 of the assignment $x_1 := 0$. Applying the above semantics, we take $\langle x_1 \rangle \cap \mathbb{K}[\bar{x}] = \langle x_1 \rangle$. This means that, if we do not know anything about the variables and apply the assignment $x_1 := 0$, then $x_1 = 0$ after the assignment.

Although the ideas just presented are general and can be applied to any polynomial assignment, there is a common particular case of assignment that can be dealt with more efficiently than described above, and is thus worth taking into account. Consider a right-hand side of the following form:

$$f(\bar{x}) = cx_i + f'(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n) \, ,$$

where $c \in \mathbb{K}$, $c \neq 0$ and f' does not depend on x_i. Then the assignment is invertible, and we can express the previous value of the variable x_i in terms of its new value. It is easy to see that in this case

$$I' = \langle x'_i - c^{-1}(x_i - f'(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)), p_1(x_i \leftarrow x'_i), ..., p_k(x_i \leftarrow x'_i) \rangle \, .$$

To eliminate x'_i from I', we substitute x'_i by $c^{-1}(x_i - f'(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n))$

in the p_j. The output is then

$$\langle \cup_{j=1}^{k} \{ p_j(x_i \leftarrow c^{-1}(x_i - f'(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n))) \} \rangle .$$

For instance, assume that $I_4 = \langle x_1, x_2 \rangle$ (i.e., $x_1 = x_2 = 0$) and that we want to compute the output ideal I_5 of the assignment $x_1 := x_1 + 2 * x_2 + 1$. As the right-hand side of the assignment has the required form, we take $I_5 = I_4(x_1 \leftarrow x_1 - 2x_2 - 1) = \langle x_1 - 2x_2 - 1, x_2 \rangle = \langle x_1 - 1, x_2 \rangle$. Then, at program point 5, the variables satisfy $x_1 = 1$ and $x_2 = 0$, which is consistent with the result of applying $x_1 := x_1 + 2 * x_2 + 1$ to $(x_1, x_2) = (0, 0)$.

6.3 Test Nodes

Let $C = C(\bar{x})$ be the boolean condition attached to a test node with input ideal $I = \langle p_1, ..., p_k \rangle$. Then the strongest postconditions for the *true* and *false* paths are respectively

$$\{ C(\bar{x}) \wedge (\wedge_{j=1}^{k} p_j(\bar{x}) = 0) \} , \qquad \{ \neg C(\bar{x}) \wedge (\wedge_{j=1}^{k} p_j(\bar{x}) = 0) \} .$$

For simplicity, below we just show how to express the assertion for the *true* path in terms of ideals when C is an atomic formula. More complex boolean expressions can be handled easily [21].

6.3.1 Polynomial Equalities

If C is a polynomial equality, i.e., it is of the form $q = 0$ with $q \in \mathbb{K}[\bar{x}]$, then the states of the *true* path are the points that belong to both $\mathbf{V}(I)$ and $\mathbf{V}(q)$, that is to say $\mathbf{V}(I) \cap \mathbf{V}(q)$; in this case we take as output

$$\mathbf{I}(\mathbf{V}(I) \cap \mathbf{V}(q)) = \mathbf{IV}(I + \langle q \rangle) = \mathbf{IV}(\langle p_1, ..., p_k, q \rangle) ,$$

since $\mathbf{V}(I) \cap \mathbf{V}(q) = \mathbf{V}(I + \langle q \rangle)$.

For instance, assume that in our example $I_3 = \langle x_1 - x_2^2 \rangle$ and we want to compute the output ideal I_7 of the *false* path. Now $C(\bar{x}) = (x_2 \neq x_3)$, and so $\neg C(\bar{x}) = (x_2 = x_3)$. According to our discussion above, then $I_7 = \mathbf{IV}(x_1 - x_2^2, x_2 - x_3) = \langle x_1 - x_2^2, x_2 - x_3 \rangle$, which means that at program point 7, $x_2 = x_3$ and $x_1 = x_2^2$.

6.3.2 Polynomial Disequalities

If C is a polynomial disequality, i.e., it is of the form $q \neq 0$ with $q \in \mathbb{K}[\bar{x}]$, then the states of the *true* path are the points that belong to $\mathbf{V}(I)$ but not to $\mathbf{V}(q)$, in other words $\mathbf{V}(I) \setminus \mathbf{V}(q)$. So the output should be the ideal of the polynomials vanishing in this set difference, $\mathbf{I}(\mathbf{V}(I) \setminus \mathbf{V}(q))$. The following result allows us to characterize this ideal:

Lemma 2 *If $I, J \subseteq \mathbb{K}[\bar{x}]$ are ideals and $I = \mathbf{IV}(I)$, then*

$$\mathbf{I}(\mathbf{V}(I) \setminus \mathbf{V}(J)) = I : J \ .$$

PROOF. We recall that the quotient of two ideals I and J is defined as $I : J = \{p \mid \forall q \in J, pq \in I\}$. For the $\subseteq$ inclusion, by definition we have to show that for arbitrary $p \in \mathbf{I}(\mathbf{V}(I) \setminus \mathbf{V}(J))$ and $q \in J$, $pq \in I$. Since $I = \mathbf{IV}(I)$, it is enough to prove that $\forall \bar{\omega} \in \mathbf{V}(I)$, $p(\bar{\omega})q(\bar{\omega}) = 0$. This is the case since $\bar{\omega} \in \mathbf{V}(J)$ implies $q(\bar{\omega}) = 0$, and $\bar{\omega} \in \mathbf{V}(I) \setminus \mathbf{V}(J)$ implies $p(\bar{\omega}) = 0$.

As regards the $\supseteq$ inclusion, we have to see that given any $p \in I : J$ and $\bar{\omega} \in \mathbf{V}(I) \setminus \mathbf{V}(J)$, $p(\bar{\omega}) = 0$. As $\bar{\omega} \notin \mathbf{V}(J)$, there exists a polynomial $q \in J$ such that $q(\bar{\omega}) \neq 0$. Since $pq \in I$ by definition of the quotient of ideals and $\bar{\omega} \in \mathbf{V}(I)$, we have $p(\bar{\omega})q(\bar{\omega}) = 0$ and therefore necessarily $p(\bar{\omega}) = 0$. $\square$

Thus, as $\mathbf{I}(\mathbf{V}(I) \setminus \mathbf{V}(q)) = I : \langle q \rangle$, we take $I : \langle q \rangle$ as output ideal.

For example, if the input ideal of a test node with condition $C(\bar{x}) = (x_1 \neq 0)$ is $I = \langle x_1 x_2 \rangle$ (either $x_1 = 0$ or $x_2 = 0$), the output for the *true* path is

$$\langle x_1 x_2 \rangle : \langle x_1 \rangle = \langle x_2 \rangle \ ,$$

which means that, after the test, we have that $x_2 = 0$ on the *true* path.

6.4 Simple Junction Nodes

Typically, simple junction nodes correspond to the merging of execution paths of conditional statements. If we denote the input ideals of variety by $I_i = \langle p_{i1}, ..., p_{ik_i} \rangle$ for $1 \leq i \leq l$ (for a certain $l \geq 2$), then the strongest postcondition after the simple junction is

$$\{ \vee_{i=1}^{l} (\wedge_{j=1}^{k_i} p_{ij}(\bar{x}) = 0) \} \, .$$

Then the output ideal of variety is $\mathbf{I}(\cup_{i=1}^{l} \mathbf{V}(I_i)) = \cap_{i=1}^{l} \mathbf{IV}(I_i) = \cap_{i=1}^{l} I_i$, since I_i's are ideals of variety and so satisfy $I_i = \mathbf{IV}(I_i)$.

6.5 Loop Junction Nodes

A loop junction node represents the merging of the execution paths of a loop statement. As the following example illustrates, if we treat loop junctions as simple junctions, the forward propagation procedure may not terminate. That implies that we need to extrapolate.

For instance consider the loop junction in the running example, with input arcs 2, 6 and output arc 3. Assume that $I_2 = \langle x_1, x_2 \rangle$ (so $x_1 = x_2 = 0$), $I_3 = \langle x_1 - x_2^2, x_2(x_2-1) \rangle$ (either $x_1 = x_2 = 0$ or $x_1 = x_2 = 1$) and $I_6 = \langle x_1 - x_2^2, (x_2-1)(x_2-2) \rangle$ (either $(x_1, x_2) = (1,1)$ or $(x_1, x_2) = (4,2)$). Then the new value for I_3 is

$$I_2 \cap I_6 = \langle x_1, x_2 \rangle \cap \langle x_1 - x_2^2, (x_2-1)(x_2-2) \rangle =$$
$$= \langle x_1 - x_2^2, x_2(x_2-1)(x_2-2) \rangle \,.$$

Notice that the zeroes of the polynomials above are such that $x_1 = x_2^2$ and either $x_2 = 0$ or $x_2 = 1$ or $x_2 = 2$; this is consistent with the behaviour of the program, since the semantics captures the effect of executing the loop body ≤ 2 times.

At the following step of the forward propagation procedure, $I_2 = \langle x_1, x_2 \rangle$, $I_3 = \langle x_1 - x_2^2, x_2(x_2-1)(x_2-2) \rangle$ and $I_6 = \langle x_1 - x_2^2, (x_2-1)(x_2-2)(x_2-3) \rangle$. So the next value for I_3 is

$$I_2 \cap I_6 = \langle x_1 - x_2^2, x_2(x_2-1)(x_2-2)(x_2-3) \rangle \,.$$

It is not difficult to see that, after t iterations of the forward propagation procedure,

$$I_3 = \langle x_1 - x_2^2, \prod_{s=0}^{t+1} (x_2 - s) \rangle \,.$$

It is clear, however, that only the first polynomial $x_1 - x_2^2$ yields an invariant for the loop, as it persists to be in I_3 after arbitrarily many iterations of the forward propagation. In [31], we presented an algorithm in which ideal-theoretic manipulations were employed to consider the effect of executing a path arbitrarily many times. Below, we develop an approximate method by using *widening operators*, similarly as in the approach in [12] for linear inequalities based on abstract interpretation [9].

6.5.1 Widening Operator

Let I be the output ideal of variety associated with a loop junction node, I^{prev} be its previously computed value and $J_1, ..., J_l$ be the input ideals of variety going into the loop junction. We need to perform an upper approximation of the set of states $\mathbf{V}(I^{\text{prev}}) \cup (\cup_{i=1}^{l} \mathbf{V}(J_i))$, or by duality a lower approximation of $I^{\text{prev}} \cap (\cap_{i=1}^{l} J_i)$; that is to say, we have to sift the polynomials in the intersection so that:

(1) the result is still sound, i.e., all values of variables possible at the loop junction are accounted for,
(2) the procedure for computing invariants terminates; and
(3) the method is powerful enough to generate useful invariants.

Formally, we introduce a widening operator ∇ so that $I = I^{\text{prev}} \nabla (\cap_{i=1}^{l} J_i)$. In this context:

Definition 3 *A* widening ∇ *is an operator between ideals of variety such that:*

(1) Given two ideals of variety I and J, then $I \nabla J \subseteq I \cap J$ (so that $\mathbf{V}(I \nabla J) \supseteq \mathbf{V}(I) \cup \mathbf{V}(J)$, as we do not wish to miss any state).
(2) For any decreasing chain of ideals of variety $J_0 \supseteq J_1 \supseteq ... \supseteq J_j \supseteq ...$, the chain defined as $I_0 = J_0$, $I_{j+1} = I_j \nabla J_{j+1}$ is not an infinite strictly decreasing chain.

These two properties take care of the conditions (1) and (2) mentioned earlier. As regards condition (3), in Sections 7 and 8, we will give evidence that our widening operators are quite powerful.

Definition 4 *Given two ideals of variety $I, J \subseteq \mathbb{K}[\bar{x}]$, $d \in \mathbb{N}$ and a graded term ordering $\succ$ (such as* **grlex**, **grevlex***), we define $I \nabla_d J$ as*

$$I \nabla_d J = \mathbf{IV}(\{p \in \mathrm{GB}(I \cap J, \succ) \mid \deg(p) \leq d\}) = \mathbf{IV}(\mathrm{GB}(I \cap J, \succ) \cap \mathbb{K}_d[\bar{x}]),$$

where $\mathrm{GB}(I \cap J, \succ)$ stands for a Gröbner basis of $I \cap J$ with respect to the term ordering $\succ$.

Theorem 5 *For any $d \in \mathbb{N}$ and any graded term ordering $\succ$, the operator ∇_d is a widening.*

PROOF. *Property (1).* Given two ideals of variety $I, J \subseteq \mathbb{K}[\bar{x}]$, then $I \nabla_d J \subseteq I \cap J$:

$$I \nabla_d J = \mathbf{IV}(\mathrm{GB}(I \cap J, \succ) \cap \mathbb{K}_d[\bar{x}]) \subseteq \mathbf{IV}(\mathrm{GB}(I \cap J, \succ)) =$$

$$= \mathbf{IV}(I \cap J) = \mathbf{IV}(I) \cap \mathbf{IV}(J) = I \cap J\,.$$

Property (2). Now let us prove that, for any decreasing chain of ideals $J_0 \supseteq J_1 \supseteq ... \supseteq J_j \supseteq ...$, the chain defined as $I_0 = J_0$, $I_{j+1} = I_j \nabla_d J_{j+1}$ is not an infinite strictly decreasing chain. Since $I_0 \supseteq I_1 \supseteq ... \supseteq I_j \supseteq ...$, we also have the decreasing chain

$$I_0 \cap \mathbb{K}_d[\bar{x}] \supseteq I_1 \cap \mathbb{K}_d[\bar{x}] \supseteq ... \supseteq I_j \cap \mathbb{K}_d[\bar{x}] \supseteq ... .$$

But each $I_j \cap \mathbb{K}_d[\bar{x}]$ is a $\mathbb{K}$-vector space: if $p, q \in I_j \cap \mathbb{K}_d[\bar{x}]$, then $p+q \in I_j \cap \mathbb{K}_d[\bar{x}]$, as I_j is an ideal and $\mathbb{K}_d[\bar{x}]$ is closed under addition; and if $p \in I_j \cap \mathbb{K}_d[\bar{x}]$ and $\lambda \in \mathbb{K}$, we can consider $\lambda \in \mathbb{K}[\bar{x}]$ and since I_j is an ideal, $\lambda \cdot p \in I_j \cap \mathbb{K}_d[\bar{x}]$. So taking dimensions (as vector spaces), we have that

$$\dim(I_0 \cap \mathbb{K}_d[\bar{x}]) \geq \dim(I_1 \cap \mathbb{K}_d[\bar{x}]) \geq ... \geq \dim(I_j \cap \mathbb{K}_d[\bar{x}]) \geq ... .$$

But this chain of natural numbers cannot strictly decrease indefinitely, as it is bounded from below by 0. Therefore there exists $i \in \mathbb{N}$ such that $\forall j > i$ $\dim(I_i \cap \mathbb{K}_d[\bar{x}]) = \dim(I_j \cap \mathbb{K}_d[\bar{x}])$. We can assume that $i \geq 1$ without loss of generality. As $I_i \cap \mathbb{K}_d[\bar{x}] \supseteq I_j \cap \mathbb{K}_d[\bar{x}]$ and the two vector spaces have the same dimension, we get the equality $I_i \cap \mathbb{K}_d[\bar{x}] = I_j \cap \mathbb{K}_d[\bar{x}]$. Since $i \geq 1$,

$$I_i = \mathbf{IV}(\mathrm{GB}(I_{i-1} \cap J_i, \succ) \cap \mathbb{K}_d[\bar{x}]) \subseteq \mathbf{IV}(I_i \cap \mathbb{K}_d[\bar{x}]) =$$

$$= \mathbf{IV}(I_j \cap \mathbb{K}_d[\bar{x}]) \subseteq \mathbf{IV}(I_j) = I_j ,$$

as I_j is an ideal of variety. But by construction, we know $I_i \supseteq I_j$; so $I_i = I_j$, which implies that the chain must stabilize in a finite number of steps. □

As the reader may have noticed, we are thus defining a family of widening operators parameterized by the degree bound and the graded term ordering under consideration.

6.5.2 Applying the Widening

Let us apply the widening operator for $d = 2$ to our running example. Assume that

$$I_2 = \langle x_1, x_2 \rangle,$$

$$I_3 = \langle x_1 - x_2^2, \prod_{s=0}^{3}(x_2 - s) \rangle = \langle x_1 - x_2^2, x_1^2 - 6x_2x_1 + 11x_1 - 6x_2 \rangle,$$

$$I_6 = \langle x_1 - x_2^2, \prod_{s=1}^{4}(x_2 - s) \rangle = \langle x_1 - x_2^2, x_1^2 - 10x_1x_2 + 35x_1 - 50x_2 + 24 \rangle.$$

Taking the graded term ordering $\succ$=**grevlex** with $x_1 \succ x_2 \succ x_3$,

$$I_3 \, \nabla_2 (I_2 \cap I_6) = \mathbf{IV}(\mathrm{GB}(I_3 \cap I_2 \cap I_6, \succ) \cap \mathbb{K}_2[\bar{x}]) =$$

$$= \mathbf{IV}(\{x_1 - x_2^2, x_1^2x_2 - 10x_1^2 + 35x_1x_2 - 50x_1 + 24x_2,$$

$x_1^3 - 65x_1^2 + 300x_1x_2 - 476x_1 + 240x_2\} \cap \mathbb{K}_2[\bar{x}]) = \mathbf{IV}(x_1 - x_2^2) = \langle x_1 - x_2^2 \rangle\,.$

So, in this case, the widening operator accomplishes its purpose, and at the following iteration of the forward propagation the output stabilizes, as we will see next in Example 6.

Example 6 *Below we give the first two iterations of Gauss-Seidel's fixpoint algorithm [10] on our running example for $d = 2$. Our goal is to solve the following fixpoint equation, which has been obtained from the program semantics presented above (where we have considered that nothing is assumed about the values of the variables at the entry point):*

$I_0 = \{0\}$;

$I_1 = (\langle x_1 \rangle + \langle I_0(x_1 \leftarrow x_1') \rangle) \cap \mathbb{K}[\bar{x}]$;

$I_2 = (\langle x_2 \rangle + \langle I_1(x_2 \leftarrow x_2') \rangle) \cap \mathbb{K}[\bar{x}]$;

$I_3 = I_3 \nabla_2 (I_2 \cap I_6)$;

$I_4 = \langle I_3 \rangle : \langle x_2 - x_3 \rangle$;

$I_5 = I_4(x_1 \leftarrow x_1 - 2x_2 - 1)$;

$I_6 = I_5(x_2 \leftarrow x_2 - 1)$;

$I_7 = \mathbf{IV}(\langle x_2 - x_3 \rangle + I_3)$.

*The computation is performed using $\succ$=**grevlex** with $x_1 \succ x_2 \succ x_3$. Initially, $\forall j : 0 \leq j \leq 7$, $I_j^{(0)} = \langle 1 \rangle$. As nothing is assumed about the values of the variables at the entry point, $I_0^{(1)} = \{0\}$. After the assignments $x_1 := 0$ and $x_2 := 0$ (which are not invertible), respectively*

$I_1^{(1)} = (\langle x_1 \rangle + \{0\}) \cap \mathbb{K}[\bar{x}] = \langle x_1 \rangle,$

$I_2^{(1)} = (\langle x_2 \rangle + \langle x_1 \rangle) \cap \mathbb{K}[\bar{x}] = \langle x_1, x_2 \rangle.$

When dealing with the loop junction, since $I_3^{(0)} = I_6^{(0)} = \langle 1 \rangle$,

$I_3^{(1)} = I_3^{(0)} \nabla_2 (I_2^{(1)} \cap I_6^{(0)}) = I_2^{(1)} = \langle x_1, x_2 \rangle.$

When taking the true output path,

$I_4^{(1)} = I_3^{(1)} : \langle x_2 - x_3 \rangle = \langle x_1, x_2 \rangle.$

The assignments $x_1 := x_1 + 2 * x_2 + 1$ and $x_2 := x_2 + 1$ are invertible, and so:

$I_5^{(1)} = I_4^{(1)}(x_1 \leftarrow x_1 - 2x_2 - 1) = \langle x_1 - 2x_2 - 1, x_2\rangle,$

$I_6^{(1)} = I_5^{(1)}(x_2 \leftarrow x_2 - 1) = \langle x_1 - 2x_2 + 1, x_2 - 1\rangle.$

Finally, when taking the false output path of the loop test, we add the condition $x_2 - x_3$*:*

$I_7^{(1)} = \mathbf{IV}(I_3^{(1)} + \langle x_2 - x_3\rangle) = \mathbf{IV}(x_1, x_2, x_2 - x_3) = \langle x_1, x_2, x_3\rangle.$

Notice that, from this iteration onwards, the values of I_0*,* I_1 *and* I_2 *remain the same. As regards the other program points, at the next iteration:*

$I_3^{(2)} = I_3^{(1)}\nabla_2(I_2^{(2)} \cap I_6^{(1)}) = \langle x_1, x_2\rangle\nabla_2\langle x_1 - x_2^2, x_2(x_2 - 1)\rangle =$

$= \langle x_1 - x_2^2, x_2(x_2 - 1)\rangle.$

$I_4^{(2)} = \langle I_3^{(2)}\rangle : \langle x_2 - x_3\rangle = \langle x_1 - x_2^2, x_2(x_2 - 1)\rangle.$

$I_5^{(2)} = I_4^{(2)}(x_1 \leftarrow x_1 - 2x_2 - 1) = \langle x_1 - x_2^2 - 2x_2 - 1, x_2(x_2 - 1)\rangle.$

$I_6^{(2)} = I_5^{(2)}(x_2 \leftarrow x_2 - 1) = \langle x_1 - x_2^2, (x_2 - 1)(x_2 - 2)\rangle.$

$I_7^{(2)} = \mathbf{IV}(I_3^{(2)} + \langle x_2 - x_3\rangle) = \mathbf{IV}(x_1 - x_2^2, x_2(x_2 - 1), x_2 - x_3) =$

$= \langle x_1 - x_2^2, x_2(x_2 - 1), x_2 - x_3\rangle.$

Of the following iterations we just show the computation of $I_3^{(5)}$*, which corresponds to the example in Section 6.5.2 illustrating the application of the widening operator:*

$I_3^{(5)} = I_3^{(4)}\nabla_2(I_2^{(5)} \cap I_6^{(4)}) = \mathbf{IV}(\{x_1 - x_2^2, x_1^2x_2 - 10x_1^2 + 35x_1x_2 - 50x_1 + 24x_2,$

$x_1^3 - 65x_1^2 + 300x_1x_2 - 476x_1 + 240x_2\} \cap \mathbb{K}_2[\bar{x}]) = \mathbf{IV}(x_1 - x_2^2) = \langle x_1 - x_2^2\rangle\,.$

In the next iteration we get that $\forall i : 0 \leq i \leq 7$*,* $I_i^{(6)} = I_i^{(5)}$*. So the algorithm stabilizes in 6 iterations and we obtain the loop invariant* $\{x_1 = x_2^2\}$*.*

7 Completeness

We show in this section that the method is complete for finding polynomial invariants up to degree d, where d is the parameter in the widening, provided we restrict to a simplified class of programs: firstly, conditions in test nodes are ignored[2]; further, all assignments are assumed to be linear (i.e., of the form $x_i := f(\bar{x})$, with f a polynomial of degree 1).

As seen in Example 6, the ideal-theoretic semantics of program constructs is used to associate a fixpoint equation $\bar{I} = F(\bar{I})$ to a program, where the unknown $\bar{I}$ stands for the tuple of invariant ideals for each program point, and F is an expression involving sum, intersection and quotient of ideals and elimination of variables. The least of the solutions to this fixpoint equation with respect to $\supseteq$ yields the optimal invariants[3]; unfortunately, in general, it cannot be computed in a finite number of steps by applying forward propagation. To solve this problem, the widenings proposed in Section 6.5.1 extrapolate the intersection of ideals when handling loop junction nodes, at the cost of losing completeness. However, Theorem 7 shows that the widenings are precise enough so as to keep all those polynomials of degree $\leq d$ of any fixpoint, in particular the least fixpoint:

Theorem 7 *Let $\bar{I}^*$ be a fixpoint of the application F given by the semantics of a program (without widening). Let $\bar{I}^{(i)}$ be the approximation obtained at the i-th iteration of the forward propagation using ∇_d at loop junctions. Then $\forall i \in \mathbb{N}$ and $\forall a$ program point, $I_a^* \cap \mathbb{K}_d[\bar{x}] \subseteq I_a^{(i)}$.*

The proof of this theorem, which is given below, is based on the key property of the proposed widening operators: the extrapolation includes all polynomials of degree $\leq d$ of the intersection; in other words, given I, J ideals of variety, $I \cap J \cap \mathbb{K}_d[\bar{x}] \subseteq I \nabla_d J$. In order to prove this, we first need to show that, given any ideal I, if we only take the polynomials of $\mathrm{GB}(I, \succ)$ of degree $\leq d$ (with $\succ$ a graded term ordering), then we can still generate all polynomials of I of degree $\leq d$.

Lemma 8 *Let I be an ideal of $\mathbb{K}[\bar{x}]$ and $\succ$ be a graded term ordering. Then $\forall d \in \mathbb{N}$, $I \cap \mathbb{K}_d[\bar{x}] \subseteq \langle \mathrm{GB}(I, \succ) \cap \mathbb{K}_d[\bar{x}] \rangle$.*

PROOF. Given $d \in \mathbb{N}$, we can write $\mathrm{GB}(I, \succ) = \{p_1, ..., p_s\}$ so that the leading monomials are ordered increasingly (that is to say, $\mathrm{lm}(p_{i+1}) \succeq \mathrm{lm}(p_i)$). Now, if $\deg(p_s) \leq d$, by construction $\mathrm{GB}(I, \succ) = \mathrm{GB}(I, \succ) \cap \mathbb{K}_d[\bar{x}]$ and therefore $I \cap \mathbb{K}_d[\bar{x}] \subseteq I = \langle \mathrm{GB}(I, \succ) \cap \mathbb{K}_d[\bar{x}] \rangle$.

Thus, we can assume that $\deg(p_s) > d$ and that there exists r such that

[2] However, as we have seen in previous sections, the method can deal with polynomial dis/equalities in test nodes, and therefore can be applied in more general circumstances.

[3] The semi-lattice of ideals of variety $(\mathcal{I}, \supseteq)$ can be shown to be a complete lattice with $\mathrm{meet}(\{I_i\}_{i \in X}) = \mathbf{IV}(\sum_{i \in X} I_i)$ and $\mathrm{join}(\{I_i\}_{i \in X}) = \mathbf{IV}(\cap_{i \in X} I_i)$ (join $= \cap$ if X is finite). Thus, Knaster-Tarski theorem guarantees the existence of a least fixpoint.

$0 \leq r < s$ and $\deg(p_r) \leq d < \deg(p_{r+1})$ (we define $p_0 = 0$ for notational convenience). Given any $q \in I \cap \mathbb{K}_d[\bar{x}]$, from Gröbner bases theory, we know we can write $q = \sum_{i=0}^{s} g_i p_i$ for certain $g_i \in \mathbb{K}[\bar{x}]$ such that if $g_i \neq 0$, then $\mathrm{lm}(q) \succeq \mathrm{lm}(g_i p_i)$. Therefore $\forall i : r < i \leq s$, if $g_i \neq 0$ then $d \geq \deg(q) = \deg(\mathrm{lm}(q)) \geq \deg(\mathrm{lm}(g_i p_i)) = \deg(g_i) + \deg(p_i) \geq \deg(p_i)$, which is impossible. So $\forall i : r < i \leq s$, $g_i = 0$ and $q = \sum_{i=0}^{r} g_i p_i$. Thus $I \cap \mathbb{K}_d[\bar{x}] \subseteq \langle \mathrm{GB}(I, \succ) \cap \mathbb{K}_d[\bar{x}] \rangle$. $\square$

The next result is the key property of the widening $I \nabla_d J$: though it approximates from below $I \cap J$, the approximation is good enough so as to include all polynomials of degree $\leq d$ that belong to both I and J.

Lemma 9 *Let I, J be ideals of variety of $\mathbb{K}[\bar{x}]$. Then $\forall d \in \mathbb{N}$, $I \cap J \cap \mathbb{K}_d[\bar{x}] \subseteq I \nabla_d J$.*

PROOF. By Lemma 8,

$$I \cap J \cap \mathbb{K}_d[\bar{x}] \subseteq \langle \mathrm{GB}(I \cap J, \succ) \cap \mathbb{K}_d[\bar{x}] \rangle \subseteq$$

$$\subseteq \mathbf{IV}(\mathrm{GB}(I \cap J, \succ) \cap \mathbb{K}_d[\bar{x}]) = I \nabla_d J. \qquad \square$$

Finally, we give the formal proof of Theorem 7.

PROOF (of Theorem 7)

For the sake of simplicity, we prove the theorem for the naive forward propagation algorithm $\bar{I}^{(i+1)} = F(\bar{I}^{(i)})$; the reasoning can be extended easily to other fixpoint iteration strategies.

The proof is by induction over i. The inductive step is proved by considering all possible cases of program points and checking that all polynomials of degree $\leq d$ of the fixpoint are retained.

For $i = 0$, by construction we have that for all program points a, $I_a^{(0)} = \langle 1 \rangle = \mathbb{K}[\bar{x}]$; and obviously $I_a^* \cap \mathbb{K}_d[\bar{x}] \subseteq \mathbb{K}[\bar{x}] = I_a^{(0)}$.

Now let us assume that the theorem is true for i, and let us show that then it holds for $i+1$. We distinguish several cases depending on the kind of program point a:

- If a is the output arc of the entry node, by definition the invariant ideal of variety at this point is a constant precondition ideal I_{ent}. Thus $I_a^{(i+1)} = I_a^{(i)} = I_a^* = I_{\mathrm{ent}}$. So trivially $I_a^* \cap \mathbb{K}_d[\bar{x}] \subseteq I_a^{(i+1)}$.

- Let us assume that a is the output arc of an assignment node with input arc b labelled with $x_i := f(x_i, \bar{x}_i)$ (for simplicity, we split the $\bar{x}$ variables into the assigned variable x_i and the tuple of all those variables that are not changed, $\bar{x}_i = x_1, ..., x_{i-1}, x_{i+1}, ..., x_n$).

  Then, by construction

$$I_a^{(i+1)} = (\langle x_i - f(x_i', \bar{x}_i)\rangle + \langle I_b^{(i)}(x_i \leftarrow x_i')\rangle) \cap \mathbb{K}[x_i, \bar{x}_i]\,,$$

  where $\langle\cdot\rangle = \langle\cdot\rangle_{\mathbb{K}[x_i, x_i', \bar{x}_i]}$. As $\bar{I}^* = F(\bar{I}^*)$, we also have that

$$I_a^* = (\langle x_i - f(x_i', \bar{x}_i)\rangle + \langle I_b^*(x_i \leftarrow x_i')\rangle) \cap \mathbb{K}[x_i, \bar{x}_i]\,.$$

  Finally, by Lemma 16 in Appendix A and by the induction hypothesis $I_b^* \cap \mathbb{K}_d[x_i, \bar{x}_i] \subseteq I_b^{(i)}$:

$$I_a^* \cap \mathbb{K}_d[x_i, \bar{x}_i] = (\langle x_i - f(x_i', \bar{x}_i)\rangle + \langle I_b^*(x_i \leftarrow x_i')\rangle) \cap \mathbb{K}_d[x_i, \bar{x}_i] \subseteq$$

$$\subseteq (\langle x_i - f(x_i', \bar{x}_i)\rangle + \langle I_b^*(x_i \leftarrow x_i') \cap \mathbb{K}_d[x_i', \bar{x}_i]\rangle) \cap \mathbb{K}[x_i, \bar{x}_i] \subseteq$$
$$\subseteq (\langle x_i - f(x_i', \bar{x}_i)\rangle + \langle I_b^{(i)}(x_i \leftarrow x_i')\rangle) \cap \mathbb{K}[x_i, \bar{x}_i] = I_a^{(i+1)}\,.$$

- Now assume that a_t (a_f) is the *true* (*false*) output arc of a test node with input arc b. Since we ignore the conditions in test nodes, $I_{a_t}^{(i+1)} = I_{a_f}^{(i+1)} = I_b^{(i)}$ and $I_{a_t}^* = I_{a_f}^* = I_b^*$. By induction hypothesis

$$I_{a_t}^* \cap \mathbb{K}_d[\bar{x}] = I_b^* \cap \mathbb{K}_d[\bar{x}] \subseteq I_b^{(i)} = I_{a_t}^{(i+1)}\,,$$

$$I_{a_f}^* \cap \mathbb{K}_d[\bar{x}] = I_b^* \cap \mathbb{K}_d[\bar{x}] \subseteq I_b^{(i)} = I_{a_f}^{(i+1)}\,.$$

- If a is the output arc of a simple junction with input arcs $b_1, ..., b_l$, by construction $I_a^{(i+1)} = \cap_{j=1}^l I_{b_j}^{(i)}$. And since $\bar{I}^* = F(\bar{I}^*)$, $I_a^* = \cap_{j=1}^l I_{b_j}^*$. By induction hypothesis $\forall j : 1 \leq j \leq l$, $I_{b_j}^* \cap \mathbb{K}_d[\bar{x}] \subseteq I_{b_j}^{(i)}$. Then

$$I_a^* \cap \mathbb{K}_d[\bar{x}] = (\cap_{j=1}^l I_{b_j}^*) \cap \mathbb{K}_d[\bar{x}] = \cap_{j=1}^l (I_{b_j}^* \cap \mathbb{K}_d[\bar{x}]) \subseteq \cap_{j=1}^l I_{b_j}^{(i)} = I_a^{(i+1)}\,.$$

- Finally, let us assume that a is the output arc of a loop junction with input arcs $b_1, ..., b_l$. Since $\bar{I}^* = F(\bar{I}^*)$, $I_a^* = \cap_{j=1}^l I_{b_j}^*$. By construction, $I_a^{(i+1)} = I_a^{(i)} \nabla_d (\cap_{j=1}^l I_{b_j}^{(i)})$. And by induction hypothesis, $I_a^* \cap \mathbb{K}_d[\bar{x}] \subseteq I_a^{(i)}$ and $\forall j : 1 \leq j \leq l$, $I_{b_j}^* \cap \mathbb{K}_d[\bar{x}] \subseteq I_{b_j}^{(i)}$; the latter implies that

$$I_a^* \cap \mathbb{K}_d[\bar{x}] = \cap_{j=1}^l (I_{b_j}^* \cap \mathbb{K}_d[\bar{x}]) \subseteq \cap_{j=1}^l I_{b_j}^{(i)} \cap \mathbb{K}_d[\bar{x}]\,.$$

  Then by Lemma 9,

$$I_a^* \cap \mathbb{K}_d[\bar{x}] \subseteq I_a^{(i)} \cap (\cap_{j=1}^l I_{b_j}^{(i)}) \cap \mathbb{K}_d[\bar{x}] \subseteq I_a^{(i)} \nabla_d (\cap_{j=1}^l I_{b_j}^{(i)}) = I_a^{(i+1)}\,.$$

This concludes the proof. □

As said above, the fixpoint $\bar{I}^*$ in the statement of the theorem may be the least fixpoint of F with respect to $\supseteq$. Therefore, on termination of the approximate forward propagation with widening, the theorem guarantees that we have computed all the invariant polynomials of degree $\leq d$.

The previous theorem makes the strong assumption that tests in conditional statements and loops are ignored, and that assignments are linear. Nonetheless, there are severe theoretical limits on the improvement on this completeness result: as shown in [28], for undecidability reasons there *cannot* exist a complete method even for finding all the *linear equality* invariants in imperative programs with linear assignments and linear equality conditions.

8 Examples and Experimental Evaluation

Based on the program semantics presented in Section 6, we have implemented a polynomial invariant generator prototype using the algebraic geometry tool Macaulay 2 [19] on a Pentium 4 with a 3.4 GHz processor and 1 Gb of memory. Next, we show in detail the results of applying our implementation to several programs extracted from the literature; later on, we will discuss the performance of the invariant generator and the heuristics it employs.

8.1 Some Illustrative Examples

In this subsection several examples taken from the literature are analyzed by means of our polynomial invariant generator. Since we are interested in determining non-linear loop invariants, the default value of the parameter d in the widening is 2. If that does not work, then d is incremented.

Example 10 *The first example has been extracted from [34]. It is a program that, given two natural numbers a and b, computes simultaneously the* gcd *and the* lcm*, which on termination are x and $u+v$ respectively. Notice that the program has nested loops.*

```
var a, b, x, y, u, v: integer end var
 (x, y, u, v):=(a, b, b, 0);
 while x ≠ y do
       while x > y do (x, v):=(x − y, u + v); end while
```

```
        while x < y do (y, u):=(y − x, u + v); end while
    end while
```

Our implementation gives the same invariant for the three loops,

$$\{xu + yv = ab\} \ ,$$

which is computed in 1.22 seconds (using $d = 2$*). On termination of the outer loop, for which the invariant* $\{\gcd(x,y) = \gcd(a,b)\}$ *can be found by other methods [4], we have* $x = y \wedge \gcd(x,y) = \gcd(a,b) \wedge xu + yv = ab$*, which implies* $u + v = \mathrm{lcm}(a,b)$*.*

Example 11 *The next example is an implementation of extended Euclid's algorithm to compute Bezout's coefficients* (p,r) *of two natural numbers* x*,* y *(see [25]), using a division program extracted from [6]. Note that it has several levels of nested loops and also non-linear polynomial assignments.*

```
   var x, y, a, b, p, q, r, s: integer end var
    (a, b, p, q, r, s):=(x, y, 1, 0, 0, 1);
    while b ≠ 0 do
        var c, k: integer end var
        (c, k):=(a, 0);
        while c ≥ b do
            var u, v: integer end var
            (u, v):=(1, b);
            while c ≥ 2v do  (u, v):=(2u, 2v); end while
            (c, k):=(c − v, k + u);
        end while
        (a, b, p, q, r, s):=(b, c, q, p − qk, s, r − sk);
    end while
```

We get the following invariants in 8.53 seconds using $d = 2$*:*

(1) Outermost loop:

$$\{px + ry = a \wedge qx + sy = b\} \, .$$

(2) Middle loop:

$$\{px + ry = a \wedge qx + sy = b \wedge kb + c = a\} \, .$$

(3) Innermost loop:

$$\{px + ry = a \wedge qx + sy = b \wedge kb + c = a \wedge ub = v \wedge vk + uc = ua\} \, .$$

The invariant of the outermost loop $\{px + ry = a \wedge qx + sy = b\}$ *ensures that* (p,r) *is a pair of Bezout's coefficients for* x*,* y *on termination of the program.*

Example 12 *The following example is a version of a program in [25], which tries to find a divisor m of a natural number n using a parameter p:*

```
var n, p, m, r, t, q: integer end var
(m, r, t, q):=(p, n mod p, n mod (p − 2), 4(n div (p − 2) − n div p));
while m ≤ ⌊√n⌋ ∧ r ≠ 0 do
      if 2r − t + q < 0 then
            (m, r, t, q):=(m + 2, 2r − t + q + m + 2, r, q + 4);
      else if 0 ≤ 2r − t + q < m + 2 then
            (m, r, t):=(m + 2, 2r − t + q, r);
      else if m + 2 ≤ 2r − t + q < 2m + 4 then
            (m, r, t, q):=(m + 2, 2r − t + q − m − 2, r, q − 4);
      else
            (m, r, t, q):=(m + 2, 2r − t + q − 2m − 4, r, q − 8);
      end if
end while
```

This is one of the most non-trivial programs we have analyzed. With $d = 2$*, after 1.17 seconds, the invariant generator terminates returning just the trivial loop invariant* $0 = 0$*. When attempted with* $d = 3$*, the loop invariant*

$$\{m(mq - 4r + 4t - 2q) + 8r = 8n\}$$

is found in 2.61 seconds. This invariant, together with other non-polynomial invariants, is crucial to prove that on termination, if $r = 0$ *then* m *is a divisor of* n*.*

Example 13 *The last example, which has been taken from [15], illustrates the need of taking into account both polynomial equality and disequality tests. It is a program modelling the Illinois cache coherence protocol, in which all variables are non-negative by construction of the model. Using the notation of Dijkstra's guarded command language [16] for representing non-deterministic tests, the program can be written as follows:*

```
var x, y, u, v: natural end var
(x, y, u):=(0, 0, 0);
while true do
     if x = y = u = 0 ∧ v ≠ 0 → (v, x):=(v − 1, x + 1);
     [] v ≠ 0 ∧ y ≠ 0 → (v, y, u):=(v − 1, y − 1, u + 2);
     [] v ≠ 0 ∧ u ≠ 0 → (v, u, x):=(v − 1, u + x + 1, 0);
     [] v ≠ 0 ∧ x ≠ 0 → (v, u, x):=(v − 1, u + x + 1, 0);
     [] x ≠ 0 → (x, y):=(x − 1, y + 1);
     [] u ≠ 0 → (v, y, u):=(v + u − 1, y + 1, 0);
     [] v ≠ 0 → (v, x, u, y):=(v + x + y + u − 1, 0, 0, 1);
     [] y ≠ 0 → (y, v):=(y − 1, v + 1);
```

```
        [] u ≠ 0 → (u,v):=(u − 1, v + 1);
        [] x ≠ 0 → (x,v):=(x − 1, v + 1);
        end if
    end while
```

The protocol is safe if and only if the formula $(y = 0 \ \vee \ u = 0) \ \wedge \ (y \leq 1)$ *is a loop invariant. Unfortunately, this property is not inductive and thus it requires an auxiliary invariant to be proved. When generating this auxiliary invariant from the program, a non-linear analysis*[4] *taking into account just polynomial equality tests [7], or just polynomial disequality tests [27], yields only trivial invariants; it is necessary to handle both kinds of tests. Our invariant generator gives in* 7.68 *seconds, the following loop invariant for* $d = 2$*:*

$$\{yu = 0 \wedge y^2 = y \wedge x^2 = x \wedge xu = 0 \wedge xy = 0\} \ .$$

Notice that $(yu = 0)$ *implies that* $(y = 0 \ \vee \ u = 0)$*; and that* $(y^2 = y)$ *implies* $(y = 0 \ \vee \ y = 1)$*. Therefore, the generated invariant suffices to prove the safety requirements.*

8.2 Heuristics and Experimental Evaluation

In this subsection we perform an experimental evaluation of our polynomial invariant generator and the heuristics it employs. Table 1 summarizes the results obtained after applying the tool to a benchmark of examples[5] . There is a row for each program; the columns provide the following information:

- 1st column is the name of the program.
- 2nd column explains what the program does.
- 3rd column gives the source where the program was picked from (the entry ($\star$) is for the examples developed up by the authors).
- 4th column is the bound d for the widening operator.
- 5th column gives the number of variables in the program.
- 6th column gives the number of conditional statements.
- 7th column is the number of loops.
- 8th column is the maximum depth of nested loops.
- 9th column is the number of polynomials in the loop invariant for each loop.
- 10th column gives the time taken by our implementation (in seconds).

[4] The linear invariants produced by the linear invariant generator StInG [33], obtained by abstract interpretation with convex polyhedra and also by applying constraint-based invariant generation, do not suffice to prove the safety properties of the system either.

[5] These examples are available at `www.lsi.upc.edu/~erodri` .

Table 1
Table of examples

program	computing	from	d	var	if	loop	dep	inv	time
`cohencu`	cube	[6]	2	5	0	1	1	3	0.94
`cohendiv`	division	[6]	2	6	0	2	2	1-3	0.65
`wensley`	division	[36]	2	6	1	1	1	3	0.99
`divbin`	division	[20]	2	5	1	2	1	2-1	0.99
`mannadiv`	division	[26]	2	6	0	2	2	1-3	1.12
`hard`	division	[34]	2	6	1	2	1	3-3	1.31
`prod4br`	product	($\star$)	3	6	3	1	1	1	4.63
`euclidex1`	extended gcd	[25]	2	10	0	2	2	3-4	5.63
`euclidex2`	extended gcd	($\star$)	2	8	1	1	1	5	1.95
`euclidex3`	extended gcd	[25]	2	12	0	3	3	2-3-5	8.53
`fermat1`	divisor	[3]	2	5	0	3	2	1-1-1	0.89
`fermat2`	divisor	[3]	2	5	1	1	1	1	0.92
`knuth`	divisor	[25]	3	7	3	1	1	1	2.61
`lcm1`	lcm	[34]	2	6	0	3	2	1-1-1	1.22
`lcm2`	lcm	[16]	2	6	1	1	1	1	1.21
`sqrt`	square root	[26]	2	3	0	1	1	2	0.46
`z3sqrt`	square root	[35]	2	4	1	1	1	1	0.82
`dijkstra`	square root	[16]	2	5	1	2	1	2-1	1.31
`freire1`	square root	[17]	2	3	0	1	1	1	0.38
`freire2`	cubic root	[17]	2	4	0	1	1	4	0.85
`readers`	simulation	[34]	2	6	3	1	1	3	1.95
`illinois`	protocol	[15]	2	4	9	1	1	5	7.68
`mesi`	protocol	[15]	2	3	2	1	1	2	2.65
`moesi`	protocol	[15]	2	4	3	1	1	5	4.28
`berkeley`	protocol	[15]	2	4	3	1	1	4	2.74
`firefly`	protocol	[15]	2	4	6	1	1	5	5.01

Notice that the invariants are always obtained in less than 9 seconds, even for the examples that require $d = 3$, and that, on average, it takes just over 2 seconds for the tool to analyze the programs.

Next, we briefly describe the techniques that we have employed to speed up the computation of invariants. To begin with, we have implemented an algorithm for computing fixpoints that incrementally selects branches of conditional statements: first, just one branch at each conditional statement that is encountered is taken into account, and invariants for every program point are computed; then, the invariant obtained at the previous stage is used as the initial condition for a new fixpoint computation, in which new execution paths are considered by analyzing both branches of one of the conditional statements for which just one branch had been considered so far; finally, this process is repeated iteratively until all branches of all conditional statements are taken into account.

Further, a sound heuristic employing finite fields is also employed when the number of conditionals in the program to analyze is greater than a parameter that can be set by the user (its default value is 3). The idea is to generate the invariant ideals in a finite field $\mathbb{Z}_p$ (with $p = 32749$ by default), and then use these ideals as initial conditions for the computation of the fixpoint in the complex numbers $\mathbb{C}$. This technique, together with the incremental fixpoint algorithm, has proved crucial to handling the worst-case complexity of Gröbner bases computations, which may be doubly-exponential in the number of variables (although, as it is well-known in the literature [1], in practice Gröbner bases can be computed much faster for well-structured problems, which is most often our case).

Apart from implementing these ideas, the invariant generator also allows the user to decide whether to deal with polynomial disequalities at program tests by means of quotients of ideals, or rather to ignore them as it is done in general with tests which are not polynomial dis/equalities. By default, the tool ignores disequalities: all programs in Table 1 have been analyzed without computing quotients of ideals, except for the last five entries. The tool also allows skipping the $\mathbf{IV}(\cdot)$ computations [6]; in fact, the default option is precisely to avoid these computations. Of course, these two overapproximations come at the theoretical cost of losing precision. Table 2 shows the results of studying the trade-off between efficiency and accuracy. Again, there is a row for each program (we have only considered those programs in the benchmark that have polynomial tests in conditionals/loops); the meaning of the columns is as follows:

- 1st column is the name of the program.
- 2nd column is the bound d for the widening operator.
- 3rd column shows the number of polynomial tests in the program.
- 4th column indicates the time (in seconds) taken by the tool with the default

[6] Given an ideal $I \subseteq \mathbb{C}[\bar{x}]$, we compute $\mathbf{IV}(I)$ by using that $\mathbf{IV}(I) = \mathrm{Rad}(I)$, the *radical* of I, which is the set of polynomials p such that there exists $k \in \mathbb{N}$ satisfying $p^k \in I$. If we are working in a finite field, then $\mathrm{Rad}(I)$ is an approximation of $\mathbf{IV}(I)$.

Table 2
Table showing the effect of computing $\mathbf{IV}(\cdot)$ and/or quotients

program	d	tests	time	quot		IV		quot + IV	
				time	prec	time	prec	time	prec
`cohencu`	2	1	0.94	1.02	✗	1.82	✗	1.88	✗
`divbin`	2	1	0.99	1.09	✗	1.52	✗	1.55	✗
`mannadiv`	2	2	1.12	1.45	✗	2.05	✗	2.96	✗
`hard`	2	1	1.31	1.39	✗	2.26	✗	2.32	✗
`prod4br`	3	1	4.63	4.88	✗	TO	—	TO	—
`euclidex1`	2	1	5.63	5.74	✗	TO	—	TO	—
`euclidex2`	2	1	1.95	2.01	✗	15.03	✗	15.15	✗
`euclidex3`	2	1	8.53	8.60	✗	TO	—	TO	—
`fermat1`	2	1	0.89	0.94	✗	2.52	✗	2.57	✗
`fermat2`	2	1	0.92	0.99	✗	1.41	✗	1.49	✗
`knuth`	3	2	2.61	2.62	✗	3.05	✗	3.11	✗
`lcm1`	2	1	1.22	1.31	✗	2.44	✗	2.49	✗
`lcm2`	2	1	1.21	1.32	✗	1.98	✗	2.05	✗
`dijkstra`	2	1	1.31	1.39	✗	1.86	✗	1.93	✗
`readers`	2	4	1.95	1.98	✗	2.75	✗	2.77	✗
`illinois`	2	10	3.59	7.68	✓	4.10	✗	9.84	✓
`mesi`	2	4	2.59	2.65	✓	3.19	✗	3.30	✓
`moesi`	2	5	4.05	4.28	✓	4.94	✗	5.30	✓
`berkeley`	2	4	2.18	2.74	✓	2.56	✗	3.48	✓
`firefly`	2	7	3.51	5.01	✓	4.40	✗	6.44	✓

options, i.e., without computing neither quotients nor $\mathbf{IV}(\cdot)$.

- 5th-6th columns refer to just computing quotients. Whereas 5th column indicates the time taken by the prototype (timeouts are 300 seconds and are represented by TO), 6th column compares the generated invariants with those obtained with the default options: we denote that the invariants have been improved by ✓, otherwise we use ✗.
- 7th-8th columns are as the previous two, but when computing just $\mathbf{IV}(\cdot)$.
- 9th-10th columns are as the previous two, but when computing quotients and $\mathbf{IV}(\cdot)$.

Notice that the computation of $\mathbf{IV}(\cdot)$ never improved the precision of the analysis, even though the involved overhead is sometimes prohibitive. On the other hand, the overhead when computing quotients is not so significant, and in some cases (like in the last five programs) it pays off; nevertheless, in general taking into account disequality tests does not provide any improvement on the invariants generated with the default options.

Finally, the prototype also allows the choice between different term orderings for the widening operator. For instance, the user can employ either the graded reverse lexicographical ordering **grevlex**, or the graded lexicographical ordering **grlex** (the former is the default). Independently from this, it is also possible to decide how variables that have been declared in different blocks are ordered: either we consider that the outermost variables are the greateast ones, or that the innermost variables are the greatest ones (the former is the default). Table 3 shows the effect of the different term orderings on the timing (for all cases, the obtained invariants were the same, and so there was no effect on precision). The columns provide the following information:

- 1st column is the name of the program.
- 2nd column is the bound d for the widening operator.
- 3rd column shows the timings with **grevlex** and the outermost strategy.
- 4th column shows the timings with **grevlex** and the innermost strategy.
- 5th column shows the timings with **grlex** and the outermost strategy.
- 6th column shows the timings with **grlex** and the innermost strategy.

From the results in the table, one can conclude that the choice between **grevlex** or **grlex** is not very relevant, since the difference between timings is rather small; however, it seems that **grevlex** tends to be better than **grlex**. On the other hand, the underlying ordering between variables does have a potential impact on the performance of the tool, as can be seen with `prod4br`, `euclidex1` or `euclidex3`. Unfortunately, it does not seem easy to decide which is the best strategy to follow in general.

9 Conclusions

We have presented a sound method based on abstract interpretation for generating polynomial invariants of imperative programs. The technique has been implemented using the algebraic geometry tool Macaulay 2 [19]. The implementation has successfully computed invariants for many non-trivial programs. Its performance is satisfactory as can be seen in Table 1.

In the proposed method, the semantics of program statements is soundly expressed using ideal-theoretic operations. Obviously, only certain kinds of

Table 3
Table comparing different term orderings

program	d	time grevlex		time grlex	
		outermost	**innermost**	**outermost**	**innermost**
`cohencu`	2	0.94	1.03	0.99	1.04
`cohendiv`	2	0.65	0.64	0.66	0.64
`wensley`	2	0.99	0.99	0.99	0.99
`divbin`	2	0.99	0.99	1.00	1.00
`mannadiv`	2	1.12	1.00	1.13	1.01
`hard`	2	1.31	1.31	1.31	1.31
`prod4br`	3	4.63	19.74	5.11	21.04
`euclidex1`	2	5.63	4.46	5.58	4.55
`euclidex2`	2	1.95	2.46	1.96	2.80
`euclidex3`	2	8.53	6.51	8.50	6.76
`fermat1`	2	0.89	0.86	0.93	0.87
`fermat2`	2	0.92	0.86	0.94	0.88
`knuth`	3	2.61	2.65	2.68	2.66
`lcm1`	2	1.22	1.30	1.26	1.44
`lcm2`	2	1.21	1.21	1.26	1.22
`sqrt`	2	0.46	0.46	0.46	0.46
`z3sqrt`	2	0.82	0.83	0.83	0.84
`dijkstra`	2	1.31	1.29	1.35	1.31
`freire1`	2	0.38	0.39	0.38	0.40
`freire2`	2	0.85	0.89	0.87	0.88
`readers`	2	1.95	1.93	1.99	1.96
`illinois`	2	7.68	7.73	7.72	7.87
`mesi`	2	2.65	2.63	2.66	2.66
`moesi`	2	4.28	4.22	4.28	4.22
`berkeley`	2	2.74	2.74	2.74	2.74
`firefly`	2	5.01	5.01	5.00	5.01

statements can be considered this way; in particular, restrictions on tests in conditional statements and loops, as well as on assignments, must be imposed. However, using the approach discussed in [23], where an ideal-theoretic interpretation of first-order predicate calculus is presented, it might be possible to give an algebraic semantics of arbitrary program constructs using ideal-theoretic operations. This needs further investigation.

Another issue for further research is the widening operator for ensuring termination. The widenings discussed in this paper, which retain polynomials of degree less than or equal to a certain a priori bound, work very well. But we will miss out invariants if the guess made for the upper bound on the degree of the invariants is incorrect. In that sense, the proposed method is complementary to our earlier work in [31], in which no a priori bound on the degree of polynomial invariants needs to be assumed.

Also, since the method here introduced is based on abstract interpretation, it should be possible to integrate it with the well-known approaches for generating invariant intervals [8], linear inequalities [12], etc. More specifically, the semantics presented in Section 6 can be viewed as the description of the abstract domain of ideals of variety; under this more formal point of view, the concept of reduced product of abstract domains [11] provides us with the theoretical framework for this integration of invariant generation methods. However, it is not clear how the interface between the abstract domains should be beyond sharing linear equalities, as in some cases non-linear equalities implicitly imply linear inequalities, and viceversa: for instance, as shown in Example 13, $y^2 = y$ implies $0 \leq y \leq 1$ (and, assuming that y is an integer, the formulas are actually equivalent); this requires further research too. At any rate, the combination of both techniques would be an effective powerful method for generating invariants expressed as a combination of linear inequalities and polynomial equalities, thus handling a large class of programs. In contrast, we do not see how this is feasible with the recent approaches presented in [34]. The use of the abstract interpretation framework is also likely to open the door to extending our approach to programs manipulating complex data structures including arrays, records and recursive data structures.

Acknowledgements

This research was partially supported by an NSF ITR award CCR-0113611, the Prince of Asturias Endowed Chair in Information Science and Technology at the University of New Mexico, an FPU grant from the Spanish Secretaría de Estado de Educación y Universidades, ref. AP2002-3693, and the Spanish "LogicTools" project CICYT TIN 2004-03382. The authors would also like to thank R. Clarisó, R. Nieuwenhuis, A. Oliveras and the anonymous referees of

the previous versions of this paper for their help and advice.

A Appendix: Results on Assignment Nodes

The next lemma is a technical result needed to deal with assignment nodes.

Lemma 14 *Let z be a variable and $\bar{y}$ be a tuple of variables different from z. Given $f \in \mathbb{K}[\bar{y}]$ and $p \in \mathbb{K}[z, \bar{y}]$, $p(f(\bar{y}), \bar{y}) - p(z, \bar{y}) \in \langle z - f(\bar{y})\rangle$.*

PROOF. For any $m \in \mathbb{N}$,

$$z^m = (z - f(\bar{y}) + f(\bar{y}))^m = \sum_{n=0}^{m} \binom{m}{n} (z - f(\bar{y}))^n f(\bar{y})^{m-n} ,$$

which implies

$$z^m - f(\bar{y})^m = \sum_{n=1}^{m} \binom{m}{n} (z - f(\bar{y}))^n f(\bar{y})^{m-n} \in \langle z - f(\bar{y})\rangle .$$

If k is the number of variables in $\bar{y}$, we write $p(z, \bar{y}) = \sum_{m\in\mathbb{N}} \sum_{\bar{\alpha}\in\mathbb{N}^k} c_{m,\bar{\alpha}} z^m \bar{y}^{\bar{\alpha}}$, with only a finite number of the $c_{m,\bar{\alpha}}$ different from 0. Then $\forall m \in \mathbb{N}\ \forall \bar{\alpha} \in \mathbb{N}^k$, $c_{m,\bar{\alpha}} z^m \bar{y}^{\bar{\alpha}} - c_{m,\bar{\alpha}} f(\bar{y})^m \bar{y}^{\bar{\alpha}} = c_{m,\bar{\alpha}}(z^m - f(\bar{y})^m)\bar{y}^{\bar{\alpha}} \in \langle z - f(\bar{y})\rangle$. Finally, by adding up, $\forall m \in \mathbb{N}\ \forall \bar{\alpha} \in \mathbb{N}^k$, we have $p(f(\bar{y}), \bar{y}) - p(z, \bar{y}) \in \langle z - f(\bar{y})\rangle$. □

The following lemma guarantees that the output ideal of an assignment node, as we have defined it, is indeed an ideal of variety:

Lemma 15 *Let z, z' be variables and $\bar{y} = (y_1, ..., y_k)$ be a tuple of variables different from z,z'. Given $f \in \mathbb{K}[z', \bar{y}]$ and $I \subseteq \mathbb{K}[z', \bar{y}]$ ideal of variety,*

$$\mathbf{IV}((\langle z - f(z', \bar{y})\rangle + \langle I\rangle) \cap \mathbb{K}[z, \bar{y}]) = (\langle z - f(z', \bar{y})\rangle + \langle I\rangle) \cap \mathbb{K}[z, \bar{y}] ,$$

where $\langle\cdot\rangle = \langle\cdot\rangle_{\mathbb{K}[z,z',\bar{y}]}$.

PROOF. Since in this proof we need to consider ideals and varieties with different sets of variables, we will add a subscript representing the ambient ring to the **V** and **IV** operators. Now, as the $\supseteq$ inclusion is trivial, it is enough to show the $\subseteq$ inclusion, i.e.,

$$\mathbf{IV}_{\mathbb{K}[z,\bar{y}]}((\langle z - f(z', \bar{y})\rangle + \langle I\rangle) \cap \mathbb{K}[z, \bar{y}]) \subseteq (\langle z - f(z', \bar{y})\rangle + \langle I\rangle) \cap \mathbb{K}[z, \bar{y}] .$$

First, let us see that if $(\zeta', \bar{\omega}) \in \mathbf{V}_{\mathbb{K}[z',\bar{y}]}(I)$ (where $\bar{\omega} = (\omega_1, ..., \omega_k)$), then

$$(f(\zeta', \bar{\omega}), \bar{\omega}) \in \mathbf{V}_{\mathbb{K}[z,\bar{y}]}((\langle z - f(z', \bar{y})\rangle + \langle I\rangle) \cap \mathbb{K}[z, \bar{y}]) .$$

Let $p(z, \bar{y}) \in (\langle z - f(z', \bar{y})\rangle + \langle I\rangle) \cap \mathbb{K}[z, \bar{y}]$. Then we can write

$$p(z, \bar{y}) = Q(z, z', \bar{y})(z - f(z', \bar{y})) + \sum_{i=1}^{m} R_i(z, z', \bar{y}) r_i(z', \bar{y})$$

for certain $m \in \mathbb{N}$, $Q, R_i \in \mathbb{K}[z, z', \bar{y}]$ and $r_i \in I$. Given $(\zeta', \bar{\omega}) \in \mathbf{V}_{\mathbb{K}[z',\bar{y}]}(I)$, if we substitute $z = f(\zeta', \bar{\omega})$, $z' = \zeta'$ and $\bar{y} = \bar{\omega}$, then

$$p(f(\zeta', \bar{\omega}), \bar{\omega}) = Q(f(\zeta', \bar{\omega}), \zeta', \bar{\omega}) \cdot 0 + \sum_{i=1}^{m} R_i(f(\zeta', \bar{\omega}), \zeta', \bar{\omega}) r_i(\zeta', \bar{\omega}) = 0$$

and our claim holds, as p is arbitrary.

Now let us see that for $g \in \mathbf{IV}_{\mathbb{K}[z,\bar{y}]}((\langle z - f(z', \bar{y})\rangle + \langle I\rangle) \cap \mathbb{K}[z, \bar{y}])$, we have that $g(f(z', \bar{y}), \bar{y}) \in I = \mathbf{IV}_{\mathbb{K}[z',\bar{y}]}(I)$. Indeed, if $(\zeta', \bar{\omega}) \in \mathbf{V}_{\mathbb{K}[z',\bar{y}]}(I)$, then we have proved that $(f(\zeta', \bar{\omega}), \bar{\omega}) \in \mathbf{V}_{\mathbb{K}[z,\bar{y}]}((\langle z - f(z', \bar{y})\rangle + \langle I\rangle) \cap \mathbb{K}[z, \bar{y}])$, and thus $g(f(\zeta', \bar{\omega}), \bar{\omega}) = 0$. As $(\zeta', \bar{\omega}) \in \mathbf{V}_{\mathbb{K}[z',\bar{y}]}(I)$ is arbitrary, $g(f(z', \bar{y}), \bar{y}) \in I$.

But by Lemma 14, finally $g(f(z', \bar{y}), \bar{y}) - g(z, \bar{y}) \in \langle z - f(z', \bar{y})\rangle$, and therefore

$$g = g(z, \bar{y}) \in (\langle z - f(z', \bar{y})\rangle + \langle I\rangle) \cap \mathbb{K}[z, \bar{y}] . \qquad \square$$

The next lemma is needed in the proof of completeness when showing that, after assignment nodes, all polynomials of the fixpoint of degree $\leq d$ are retained.

Lemma 16 *Let z, z' be variables and $\bar{y}$ be a tuple of variables different from z,z'. Given $f \in \mathbb{K}_1[z', \bar{y}]$ polynomial of degree 1 and I ideal of $\mathbb{K}[z', \bar{y}]$,*

$$(\langle z - f(z', \bar{y})\rangle + \langle I\rangle) \cap \mathbb{K}_d[z, \bar{y}] \subseteq (\langle z - f(z', \bar{y})\rangle + \langle I \cap \mathbb{K}_d[z', \bar{y}]\rangle) \cap \mathbb{K}[z, \bar{y}] ,$$

where $\langle\cdot\rangle = \langle\cdot\rangle_{\mathbb{K}[z,z',\bar{y}]}$.

PROOF. Let $p(z, \bar{y}) \in (\langle z - f(z', \bar{y})\rangle + \langle I\rangle) \cap \mathbb{K}_d[z, \bar{y}]$. Then $\deg(p) \leq d$, and we can write $p(z, \bar{y}) = Q(z, z', \bar{y})(z - f(z', \bar{y})) + \sum_{i=1}^{m} R_i(z, z', \bar{y}) r_i(z', \bar{y})$ for certain $m \in \mathbb{N}$, $Q, R_i \in \mathbb{K}[z, z', \bar{y}]$ and $r_i \in I$. By Lemma 14, we can obtain from this another decomposition of the form $p(z, \bar{y}) = q(z, z', \bar{y})(z - f(z', \bar{y})) + r(z', \bar{y})$ for a certain $q \in \mathbb{K}[z, z', \bar{y}]$ and $r \in I$. It is enough to see that $\deg(r) \leq d$.

Let us assume the contrary, that is to say $\deg(r) > d$, and we will get a contradiction. Under this hypothesis, let us take $\succ = $ **grlex** with $z \succ z' \succ$

$y_1 \succ \cdots \succ y_k$. As $\deg(p) \leq d$, $\mathrm{lm}(r)$ has to be cancelled by some monomial in $q(z, z', \bar{y})(z - f(z', \bar{y}))$; more specifically, $\mathrm{lm}(r)$ has to be cancelled by a monomial in $q(z, z', \bar{y})f(z', \bar{y})$, since $\mathrm{lm}(r)$ does not have any z. Then

$$\deg(q(z, z', \bar{y})f(z', \bar{y})) \geq \deg(\mathrm{lm}(r)) = \deg(r) > d\,.$$

But

$$\deg(q(z, z', \bar{y})f(z', \bar{y})) = \deg(q(z, z', \bar{y})) + \deg(f(z', \bar{y})) =$$
$$= \deg(q(z, z', \bar{y})) + 1 = \deg(q(z, z', \bar{y})) + \deg(z) = \deg(\mathrm{lm}(q(z, z', \bar{y}) \cdot z))\,.$$

So $\deg(\mathrm{lm}(q(z, z', \bar{y}) \cdot z)) > d$. But as $\mathrm{lm}(q(z, z', \bar{y}) \cdot z)$ cannot be cancelled we have $\deg(p) > d$, which is a contradiction. $\square$

References

[1] M. Bardet, J.-C. Faugère, and B. Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *Proceedings of the International Conference on Polynomial System Solving*, pages 71–74, 2004. Conference held in Paris, France in honour of Daniel Lazard.

[2] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer-Verlag, 1993.

[3] D. M. Bressoud. *Factorization and Primality Testing*. Springer-Verlag, 1989.

[4] R. Chadha and D. A. Plaisted. On the mechanical derivation of loop invariants. *Journal of Symbolic Computation*, 15(5-6):705–744, 1993.

[5] R. Clarisó, E. Rodríguez-Carbonell, and J. Cortadella. Derivation of non-structural invariants of Petri Nets using abstract interpretation. In G. Ciardo and P. Darondeau, editors, *Application and Theory of Petri Nets and Other Models of Concurrency: Proceedings of the 26th International Conference*, volume 3536 of *Lecture Notes in Computer Science*, pages 188–207. Springer-Verlag, 2005.

[6] E. Cohen. *Programming in the 1990s*. Springer-Verlag, 1990.

[7] M. Colón. Approximating the algebraic relational semantics of imperative programs. In Giacobazzi [18], pages 296–311.

[8] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In B. Robinet, editor, *Proceedings of the 2nd International Symposium on Programming*, pages 106–130, 1976.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[10] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 1–12, 1977.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, 1978.

[13] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer-Verlag, 1996.

[14] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra: Systems and Algorithms for Algebraic Computation.* Academic Press, 1988.

[15] G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.

[16] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[17] P. Freire. Pedro Freire creations. SQRT, 2002. Available at `http://www.pedrofreire.com/crea2_en.htm`.

[18] R. Giacobazzi, editor. *Static Analysis, Proceedings of the 11th International Symposium*, volume 3148 of *Lecture Notes in Computer Science.* Springer-Verlag, 2004.

[19] D. R. Grayson and M. E. Stillman. Macaulay 2, a software system for research in algebraic geometry. Available at `http://www.math.uiuc.edu/Macaulay2`.

[20] A. Kaldewaij. *Programming. The Derivation of Algorithms.* Prentice-Hall, 1990.

[21] D. Kapur. A refutational approach to geometry theorem proving. *Artificial Intelligence*, 37:61–93, 1988.

[22] D. Kapur. Automatically generating loop invariants using quantifier elimination. In *Proceedings of the 10th International Conference on Applications of Computer Algebra*, 2004. Also published as Technical Report TR-CS-2003-58, Department of Computer Science, University of New Mexico, Albuquerque, USA, 2003.

[23] D. Kapur and P. Narendran. An equational approach to theorem proving in first-order predicate calculus. In A. K. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 1146–1153. Morgan Kaufmann, 1985.

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

[25] D. E. Knuth. *The Art of Computer Programming. Volume 2, Seminumerical Algorithms*. Addison-Wesley, 1969.

[26] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

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

[28] M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 1016–1028. Springer-Verlag, 2004.

[29] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 330–341. ACM Press, 2004.

[30] E. Rodríguez-Carbonell and D. Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In Giacobazzi [18], pages 280–295.

[31] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In J. Gutiérrez, editor, *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, pages 266–273. ACM Press, 2004.

[32] E. Rodríguez-Carbonell and D. Kapur. Program verification using automatic generation of invariants. In Z. Liu and K. Araki, editors, *Theoretical Aspects of Computing, Proceedings of the 1st International Colloquium*, volume 3407 of *Lecture Notes in Computer Science*, pages 325–340. Springer-Verlag, 2005.

[33] S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In Giacobazzi [18], pages 53–68.

[34] S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 318–329. ACM Press, 2004.

[35] B. J. Shelburne. Zuse's Z3 square root algorithm. Available at `http://www4.wittenberg.edu/academics/mathcomp/bjsdir/ZuseZ3Talk.pdf`.

[36] B. Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–112, 1974.