An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants* **

Enric Rodríguez-Carbonell and Deepak Kapur

Technical University of Catalonia, Barcelona
www.lsi.upc.es/~erodri
University of New Mexico, Albuquerque
www.cs.unm.edu/~kapur

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 to be polynomial equalities and disequalities. The semantics of each statement is given as a transformation on polynomial ideals. Merging of paths in a program is defined as the intersection of the polynomial ideals associated with each path. For a loop junction, a widening operator based on selecting polynomials up to a certain degree is proposed. The algorithm for finding invariants using this widening operator is shown to terminate in finitely many steps. The proposed approach has been implemented and successfully tried on many programs. A table providing details about the programs is given.

1 Introduction

There has recently been a surge of interest in research on automatic generation of loop invariants of 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 [6] for finding

* 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 project MCYT TIC2001-2476-C03-01.

linear inequalities as invariants based on the abstract interpretation framework [5]. The proposed method, in contrast, generates polynomial equations as invariants by interpreting the semantics of programming language constructs in terms of ideal-theoretic operations, which we consider by itself as an exciting novel contribution of this paper. The semantics of each statement is given as a transformation on polynomial ideals. 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 is able to handle nested loops[1] and also takes into account tests in conditional statements and loops, insofar as they can be abstracted to be polynomial equalities and disequalities. Merging of paths in a program is defined as the intersection of the polynomial ideals associated to each path. For ensuring the termination of the invariant generation procedure, a *widening operator* is proposed. This widening operator is based on retaining only the polynomials of degree $\leq d$ in the intersection; this is achieved by computing a Gröbner basis [7] with a graded term ordering and keeping only those polynomials in the basis with degree $\leq d$. The procedure for finding invariants using this widening operator is shown to terminate in finitely many steps.

The proposed algorithm has been implemented using Macaulay2 [12], an algebraic geometry tool that supports operations on polynomial ideals such as the computation of Gröbner bases. Using this implementation, loop invariants for several numerical programs have been successfully generated automatically.

The method, as well as the implementation, do not need pre/postconditions for deriving loop invariants. Further, under conditions on the semantics of programs, it 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.

The rest of the paper is organized as follows. In the next subsection, related work is briefly reviewed. Section 2 gives background information on polynomial ideals, operations on them and special bases of polynomial ideals called Gröbner bases. Section 3 introduces a simple programming language used in the paper for presenting the method. Section 4 discusses abstraction and concretization functions from variable values to ideals and viceversa, so that the framework of abstract interpretation is applicable. Section 5 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 constructed from the input polynomial ideals. Most importantly, Subsection 5.5 discusses the semantics of loop junction nodes using a widening operator. Section 6 shows that, under specific conditions of the semantics, the proposed method is sound and complete in the sense that, for every program point, our algorithm indeed finds all the invariants of degree $\leq d$, where d is the parameter used in the widening operator. Section 7 illustrates the application of the method on some examples; this is followed by a table giving details of programs successfully tried for which our implementation

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

discovers loop invariants. Section 8 concludes and discusses ideas for extending this research.

1.1 Related Work

As stated above, the proposed approach is a complement of the method proposed by Cousot and Halbwachs [6], who applied the framework of abstract interpretation [5] for finding invariant linear inequalities. That work extended Karr's algorithm in [16] for finding invariant linear equalities at any program point.

Recently, there has been a renewed surge of interest in automatically deriving invariants of imperative programs. In [4] Colón et al. have used non-linear constraint solving based on Farkas' lemma to attack the problem of finding invariant linear inequalities. Extending Karr's work, for programs with affine assignments Müller-Olm and Seidl [18] proposed an interprocedural method for computing polynomial equations of bounded degree as invariants. In [20], we developed an abstract framework for generating invariants of loops. This framework was instantiated to generate conjunctions of polynomial equations as invariants for loop programs. The method used the Gröbner basis algorithm for computing such invariants, and was shown to be sound and complete. However, that method cannot handle nested loops; furthermore, tests in conditional statements and loops are abstracted to be *true*. In [22], a method is proposed 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 using the Gröbner basis algorithm.

In contrast, not only can the method proposed in this paper generate invariants of programs with nested loops, but also it is not necessary to know a priori the structure of the polynomials appearing as invariants. However, the widening operator does need as input the degree of the polynomial invariants of interest to the user. Furthermore, unlike the methods of [22], the proposed method has been implemented and tried on many examples with considerable success. We believe that the technique discussed in [6] for linear inequalities can be easily integrated with our method since they share the framework, thus resulting in a powerful effective method for automatically generating loop invariants expressible using linear inequalities and polynomial equalities.

2 Preliminaries

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 $I \subseteq \mathbb{K}[\bar{x}]$ which contains 0, is closed under addition and 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, $\mathbf{V}(S) = \{\bar{\omega} \in \mathbb{K}^n | \ p(\bar{\omega}) = 0 \ \forall p \in S\}$. When taking varieties we can assume S to be an ideal, since $\mathbf{V}(\langle S \rangle) = \mathbf{V}(S)$. On the other hand, if $A \subseteq \mathbb{K}^n$ the ideal $\mathbf{I}(A) = \{p \in \mathbb{K}[\bar{x}] | \ p(\bar{\omega}) = 0 \ \forall \bar{\omega} \in A\}$ is called the *ideal of* A. We write $\mathbf{IV}(S)$ instead of $\mathbf{I}(\mathbf{V}(S))$ and $\mathbf{VI}(A)$ instead of $\mathbf{V}(\mathbf{I}(A))$.

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 the inclusion $I \subseteq \mathbf{IV}(I)$ holds; $\mathbf{IV}(I)$ represents the largest set of polynomials with the same set of 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. Moreover, if I is an ideal of variety, then $\mathbf{I}(\mathbf{V}(I) - \mathbf{V}(J)) = I : J$, where $-$ denotes difference of sets. For further detail on these concepts, see [8, 7].

A *term* in a set $\bar{x} = (x_1, ..., x_n)$ 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. We denote the set of all polynomials of $\mathbb{K}[\bar{x}]$ of degree $\leq d$ by $\mathbb{K}_d[\bar{x}]$.

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

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

Given a term ordering $\succ$, for any polynomial $f \in \mathbb{K}[\bar{x}]$, $\mathrm{lm}(f)$ stands for the leading monomial of f with respect to $\succ$. Given an ideal I different from $\{0\}$, a *Gröbner basis* for I is a finite set of polynomials $G = \{g_1, ..., g_k\}$ satisfying $\langle\{\mathrm{lm}(p) \mid p \in I\}\rangle = \langle \mathrm{lm}(g_1), ..., \mathrm{lm}(g_k)\rangle$. For such a set G, $I = \langle G\rangle$ holds (and so G is indeed a basis for I).

3 Programming Model

To simplify presentation, programs are represented as finite connected flowcharts with one entry node, assignment, test, junction and exit nodes, as in [6].

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**, partitioned into the following subsets (we show between parentheses the symbol for pictures):

1. **Entry ($\triangleright\!\!\rightarrow$).** 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 an identifier x_i and an expression $f(\bar{x})$, thus representing the assignment $x_i := f(\bar{x})$.
3. **Tests ($\subset\!\supset$).** 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.
4. **Junctions ($\bigcirc$).** 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 ($\rightarrow\!\!\triangleleft$).** Exit nodes have just one predecessor and no successors. They represent where the flow of the program halts.

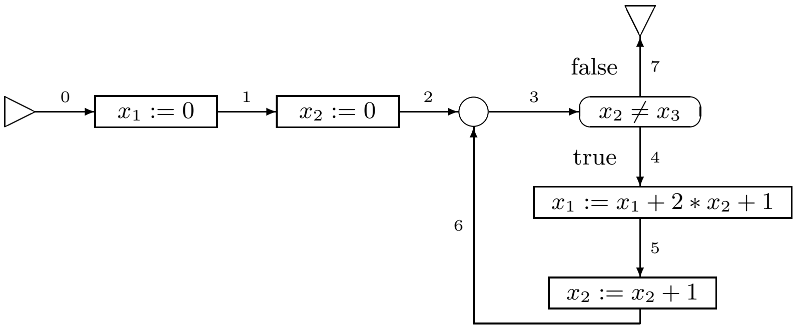

Fig. 1. Example of program

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

4 Ideals of Varieties as Abstract Values

The state of the computation at any given node in a program is the set of values each program variable can take. This is represented as a subset of $\mathbb{K}^n$. Program constructs change the state. A state can be abstracted to the ideal consisting of all polynomials that vanish in that state. 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, we attach an assertion P_a of the form $P_a = \{\bigwedge_{j=1}^{k} p_{aj}(\bar{x}) = 0\}$, 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 $(\mathcal{I}, \supseteq, \cap, \langle 1\rangle, \langle 0\rangle)$ 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 later on as transformations on polynomial ideals.

The algorithm for computing the invariant ideal of variety for each program point works 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. Assuming at first that variables are undefined on any arc (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. 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 respective assertion is approximated using a *widening operator* ∇. Intuitively, loop junction nodes correspond to loops, whereas simple junction nodes are associated to conditional statements.

5 Transformation of Ideals of Variety by Language Constructs

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

5.1 Program Entry Node

If we are given a precondition for the procedure to be analyzed, the $\mathbf{IV}(\cdot)$ of the polynomial equations 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 vector of their values may be any point in $\mathbb{K}^n$. This is represented by the zero ideal $\langle 0 \rangle = \mathbf{I}(\mathbb{K}^n)$, whose corresponding assertion is the tautology $0 = 0$.

5.2 Assignments

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 $\{\bigwedge_{j=1}^{k} p_j(\bar{x}) = 0\}$ 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 translate this formula in terms of ideals of variety.

Let us assume $f(\bar{x}) \in \mathbb{K}[\bar{x}]$. 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 $\bar{x}$ variables, i.e. $I' \cap \mathbb{K}[\bar{x}]$, and then take $\mathbf{IV}(I' \cap \mathbb{K}[\bar{x}])$. As it can be proved that $\mathbf{IV}(I' \cap \mathbb{K}[\bar{x}]) = I' \cap \mathbb{K}[\bar{x}]$, the final output is $I' \cap \mathbb{K}[\bar{x}]$.

In our running example, assume that $I_0 = \langle 0 \rangle$ and that we want to compute the output ideal I_1 of the assignment $x_1 := 0$. Applying the ideas above, we take $\langle x_1 \rangle \cap \mathbb{K}[\bar{x}] = \langle x_1 \rangle$. This means that if we know nothing about the variables and apply the assignment $x_1 := 0$, then $x_1 = 0$ after the assignment.

Invertible Assignments. A common particular case is the following: $f(\bar{x}) = c \cdot x_i + f'(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)$, 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 $c^{-1} \cdot (x_i - f'(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n))$ in the p_j. The output is $\langle \cup_{j=1}^{k} \{p_j(x_i \leftarrow c^{-1} \cdot (x_i - f'(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)))\} \rangle$.

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 $I_5 = I_4(x_1 \leftarrow x_1 - 2x_2 - 1) = \langle x_1 - 2x_2 - 1, x_2 \rangle$. Then, at program point 5, the variables satisfy $x_1 = 2x_2 + 1$ and $x_2 = 0$, which is consistent with the result of applying $x_1 := x_1 + 2 * x_2 + 1$ to $(x_1, x_2) = (0, 0)$.

5.3 Test Nodes

Let $C = C(\bar{x})$ be the boolean condition attached to a test node with the 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 [14].

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 $\mathbf{V}(q) \cap \mathbf{V}(I)$; in this case we take as output

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

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

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 $I_7 = \mathbf{IV}(x_2 - x_3, x_1 - x_2^2) = \langle x_2 - x_3, x_1 - x_2^2 \rangle$, which means that at program point 7, $x_2 = x_3$ and $x_1 = x_2^2$.

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 $\mathbf{V}(I) - \mathbf{V}(q)$. So the output should be the ideal of the polynomials vanishing in this difference of sets, $\mathbf{I}(\mathbf{V}(I) - \mathbf{V}(q))$. As it can be proved that $\mathbf{I}(\mathbf{V}(I) - \mathbf{V}(q)) = I : \langle q \rangle$, we take $I : \langle q \rangle$ as output.

For example, if the input ideal of the 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.

Polynomial Inequalities. Over $\mathbb{K} = \mathbb{R}, \mathbb{Q}$, a polynomial inequality $q > 0$ (or $q < 0$) cannot be made equivalent to a boolean combination of polynomial equalities. In this case we have to perform an approximation of C to polynomial dis/equalities. For both $q > 0$ or $q < 0$, we approximate it by $q \neq 0$.

5.4 Simple Junction Nodes

Typically, simple junction nodes correspond to the merging of the execution paths of conditional statements. In general, if the input ideals of variety $I_1, ..., I_l$ are such that for $1 \leq i \leq l$, we have $I_i = \langle p_{i1}, ..., p_{ik} \rangle$, then the strongest postcondition after the execution of the simple junction node is

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

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

5.5 Loop Junction Nodes

Intuitively, a loop junction node represents the merging of the execution paths of a `while` 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 approximate.

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 $(x_1, x_2) = (4,2)$). The new value for I_3 should be

$$I_3 \cap I_2 \cap I_6 =$$

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

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

Notice that the solutions for the polynomials above are such that $x_1 = x_2^2$ and either $x_2 = 0$ or $x_2 = 1$ or $x_2 = 2$, which is consistent with the behaviour of the loop, since the semantics captures the effect after the loop body has been executed ≤ 2 times.

At the next 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 $I_6 = \langle x_1 - x_2^2, (x_2-1)(x_2-2)(x_2-3) \rangle$. Then the next value for I_3 should be

$$I_3 \cap I_2 \cap I_6 = \langle x_1 - x_2^2, x_2(x_2-1)(x_2-2)(x_2-3) \rangle \, .$$

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 executions of the loop. In [21], we gave an approach 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 a *widening operator*, similarly as in the approach in [6] for linear inequalities based on abstract interpretation [5].

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

i) the result is still sound, i.e., all values of variables possible at the junction node are accounted for,

ii) the procedure for computing invariants terminates; and

iii) the method is powerful enough to generate useful invariants.

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

Definition 1. *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 $\mathbf{V}(I \nabla J) \supseteq \mathbf{V}(I \cap J)$ as we do not wish to miss any states).*
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 decreasing chain.*

These two properties take care of the conditions $i)$ and $ii)$ mentioned earlier. As regards $iii)$, in Sections 6 and 7 we will give evidence that our choice of the widening operator is quite powerful.

Definition 2. *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*

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

where $GB(I, \succ)$ stands for a Gröbner basis of an ideal I with respect to the term ordering $\succ$.

Theorem 1. *The operator ∇_d is a widening.*

Proof. It is easy to see from the following relation that given two ideals of variety $I, J \subseteq \mathbb{K}[\bar{x}]$, then $I \nabla_d J \subseteq I \cap J$:

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

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

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 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 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 $i \geq 1$ there exists $S \subseteq \mathbb{K}_d[\bar{x}]$ such that $I_i = \mathbf{IV}(S)$ (namely, $S = GB(I_{i-1} \cap J_i, \succ) \cap \mathbb{K}_d[\bar{x}]$). Then

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

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

Applying the Widening. Let us apply the widening to our running example for $d = 2$. Assume that $I_2 = \langle x_1, x_2 \rangle$, $I_3 = \langle x_1 - x_2^2, x_1^2 - 6x_2x_1 + 11x_1 - 6x_2 \rangle$ and $I_6 = \langle x_1 - x_2^2, x_1^2 - 10x_1x_2 + 35x_1 - 50x_2 + 24 \rangle$. Taking the graded term ordering $\succ$=**grevlex**$(x_1 > x_2)$,

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

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

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

Example 1. Here we give the first iteration of the forward propagation algorithm on our running example for $d = 2$. Due to lack of space, we cannot provide the full trace; for more details about the algorithm as well as the trace of the algorithm, please consult [19].

The calculations are done using $\succ$=**grevlex**$(x_1 > x_2)$. By definition, $\forall j : 0 \le j \le 7$, $I_j^{(0)} = \langle 1 \rangle$. Assuming nothing about the values of variables at the entry point, $I_0^{(1)} = \langle 0 \rangle$. After the assignments $x_1 := 0$ and $x_2 := 0$ (which are not invertible), respectively

$I_1^{(1)} = (\langle x_1 \rangle + \langle 0 \rangle) \cap \mathbb{K}[\bar{x}] = \langle x_1 \rangle$,
$I_2^{(1)} = (\langle x_2 \rangle + \langle x_1 \rangle) \cap \mathbb{K}[\bar{x}] = \langle x_1, x_2 \rangle$.

When dealing with the loop header, 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, taking the *false* output path of the loop test we add the condition $x_2 - x_3$:

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

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

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

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

Then, $\forall i : 0 \le i \le 7$, $I_i^{(6)} = I_i^{(5)}$. In this case, the widening operator accomplishes its function and the algorithm stabilizes in 6 iterations, yielding the loop invariant $\{x_1 = x_2^2\}$.

6 Completeness

We show in this section that, under certain assumptions on the semantics of program constructs, the method is complete for finding polynomial invariants up to the degree d, where d is the parameter in the widening. We simplify the semantics as given in Section 5 as follows: firstly, conditions in test nodes are considered to be *true*[2]; further, all assignments are assumed to be linear (i.e., of the form $x_i := p(\bar{x})$, with p a polynomial of degree 1).

The ideal-theoretic semantics of program constructs is used to associate a system of equations $\bar{I} = F(\bar{I})$ to a program, where the unknowns are the invariant ideals and F is an expression using sum, intersection and quotient of ideals and elimination of variables. The least of the solutions to this fix point equation with respect to $\supseteq$ can be shown to yield the optimal invariants; but, in general, it cannot be computed in a finite number of steps by applying forward propagation. The above proposed widening approximates the intersection of ideals when handling loop junction nodes with a loss of completeness of the method. The following theorem, however, shows that the widening is fine enough so as to keep all those polynomials of degree $\leq d$ of any fix point (for a proof, see [19]).

Theorem 2. *Let $\bar{I}^*$ be a fix point 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 procedure using ∇_d instead of intersection at loop junction nodes. Then $\forall i \in \mathbb{N}$ and $\forall a$ program point, $I_a^* \cap \mathbb{K}_d[\bar{x}] \subseteq I_a^{(i)}$.*

In particular, $\bar{I}^*$ may be the least fix point 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 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 fix point are retained. For that we use the key property of the widening: the approximation includes all polynomials of degree $\leq d$ of the intersection; in other words, given I, J ideals, $I \cap J \cap \mathbb{K}_d[\bar{x}] \subseteq I \nabla_d J$.

7 Examples

We have implemented a modified version of the method presented above using the algebraic geometry tool Macaulay2 ([12]) on a Pentium 4 2.5 Ghz. processor with 512 Mb of memory. In the implementation, the semantics of programs as given in Section 5 has been simplified: in order to speed up the algorithm, the $\mathbf{IV}(\cdot)$ computations are not performed (although in practice for all our examples, some of which are shown in this section, we found the expected invariants without losing any information); and the only boolean conditions considered are polynomial equalities. Various paths in conditional statements are incrementally selected. And, for Gröbner basis computations, coefficients are considered over a finite field (with a large prime).

Since we are interested in determining nonlinear invariants, we start with the value of d to be 2. If that does not work, then we increment the value.

[2] As stated earlier, the method can deal with polynomial dis/equalities in test nodes.

Example 2. In order to compare the techniques, the first example has been extracted from [22]. 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, $\{xu + yv = ab\}$, computed in 1.96 sec. (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, we have $x = y \wedge gcd(x, y) = gcd(a, b) \wedge xu + yv = ab$, which implies $u + v = lcm(a, b)$.

Example 3. 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 [17]), using a division program extracted from [3]. Notice that it has several levels of nested loops and 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 d, D: integer end var
        (d, D):=(1, b);
        while c ≥ 2D do  (d, D):=(2d, 2D); end while
        (c, k):=(c − D, k + d);
    end while
    (a, b, p, q, r, s):=(b, c, q, p − qk, s, r − sk);
end while
```

We get the following invariants in 9.34 sec. 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: $\{px+ry = a \wedge qx+sy = b \wedge kb+c = a \wedge db = D \wedge Dk+dc = da\}$.

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 4. The following example is a version of a program in [17] that tries to find a divisor d of a natural number N using a parameter D:

```
var N, D, d, r, t, q: integer end var
(d, r, t, q):=(D, N mod D, N mod (D − 2), 4(N div (D − 2) − N div D));
while d ≤ ⌊√N⌋ ∧ r ≠ 0 do
    if 2r − t + q < 0 then
```

```
        (d,r,t,q):=(d+2, 2r−t+q+d+2, r, q+4);
    else if 0 ≤ 2r−t+q < d+2 then
        (d,r,t):=(d+2, 2r−t+q, r);
    else if d+2 ≤ 2r−t+q < 2d+4 then
        (d,r,t,q):=(d+2, 2r−t+q−d−2, r, q−4);
    else
        (d,r,t,q):=(d+2, 2r−t+q−2d−4, r, q−8);
    end if
end while
```

This is the most nontrivial program we have attempted. With $d = 2$, after 7.86 sec. we do not get any invariant; with $d = 3$, the invariant $\{d(dq - 4r + 4t - 2q) + 8r = 8N\}$ is generated in 48.82 sec. Even though we abstract the tests to be *true*, this is a strong enough polynomial invariant; together with other non-polynomial invariants, it can be used to prove that on termination, if $r = 0$ then d is a divisor of N.

Other Examples. The following table summarizes the results obtained after applying our techniques to other examples [3]. There is a row for each program; the columns provide the following information:

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

Table 1. Table of examples

PROGRAM	COMPUTING	SOURCE	d	VAR	IF	LOOP	DEPTH	INV	TIME
cohencu	cube	[3]	3	5	0	1	1	4	2.45
dershowitz	real division	[9]	2	7	1	1	1	3	1.71
divbin	integer division	[13]	2	5	1	2	1	2-1	1.91
euclidex1	Bezout's coefs	[17]	2	10	0	2	2	3-4	7.15
euclidex2	Bezout's coefs	$(*)$	2	8	1	1	1	5	3.69
fermat	divisor	[2]	2	5	0	3	2	1-1-1	1.55
prod4br	product	$(*)$	3	6	3	1	1	1	8.49
freire1	integer sqrt	[11]	2	3	0	1	1	1	0.75
hard	integer division	[22]	2	6	1	2	1	3-3	2.19
lcm2	lcm	[10]	2	6	1	1	1	1	2.03
readers	simulation	[22]	2	6	3	1	1	2	4.15

[3] These examples are available at `www.lsi.upc.es/~erodri`

8 Conclusions

We have presented an approach based on abstract interpretation for generating polynomial invariants of imperative programs. The techniques have been implemented using the algebraic geometry tool Macaulay2 [12]. The implementation has successfully computed invariants for many nontrivial programs. Its performance is very good as evident from the above table.

In the proposed method, the semantics of statements is given using ideal-theoretic operations; this is a novel idea in contrast to the axiomatic semantics or denotational semantics typically given for program constructs. Obviously, only certain kinds of statements can be considered this way; in particular, restrictions on tests in conditionals and loops, as well as on assignments, must be imposed. However, using the approach discussed in [15], where an ideal-theoretic interpretation of first-order predicate calculus is presented, it might be possible to give an algebraic semantics of arbitrary programming constructs using ideal-theoretic operations. This needs further investigation.

Another issue for further research is the widening operator for the semantics of loop junctions. The widening here presented, which retains polynomials of degree less than or equal to a certain a priori bound, works 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 [21], 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 will be easy to integrate it with the techniques for generating invariant linear inequalities discussed in [6]. Such an integration will result in an effective powerful method for generating loop invariants expressed as a combination of linear inequalities and polynomial equations, thus handling a large class of programs. In contrast, we do not see how this is feasible with the recent approaches presented in [22]. The use of the abstract interpretation framework is also likely to open doors for extending our approach to consider programs manipulating complex data structures including arrays, records and recursive data structures.

Acknowledgements. The authors would like to thank R. Clarisó, R. Nieuwenhuis, A. Oliveras and the anonymous referees for their help and advice.

References

1. François 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.
2. David M. Bressoud. *Factorization and Primality Testing.* Springer-Verlag, 1989.
3. Edward Cohen. *Programming in the 1990s.* Springer-Verlag, 1990.
4. M. A. Colón, S. Sankaranarayanan, and H.B. Sipma. Linear Invariant Generation Using Non-Linear Constraint Solving. In *Computer-Aided Verification (CAV*

2003), volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer-Verlag, 2003.

5. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.
6. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, 1978.
7. D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer-Verlag, 1998.
8. J.H. Davenport, Y.Siret, and E.Tournier. *Computer Algebra: Systems and Algorithms for Algebraic Computation*. Academic Press, 1988.
9. N. Dershowitz and Z. Manna. Inference rules for program annotation. In *Proceedings of the 3rd International Conference on Software Engineering*, pages 158–167, 1978.
10. E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
11. Pedro Freire. `www.pedrofreire.com/crea2_en.htm?`
12. Daniel R. Grayson and Michael E. Stillman. Macaulay 2, a Software System for Research in Algebraic Geometry. Available at `http://www.math.uiuc.edu/Macaulay2/`.
13. A. Kaldewaij. *Programming. The Derivation of Algorithms*. Prentice-Hall, 1990.
14. D. Kapur. A Refutational Approach to Geometry Theorem Proving. *Artificial Intelligence*, 37:61–93, 1988.
15. D. Kapur and P. Narendran. An equational approach to theorem proving in first-order predicate calculus. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 1146–1153, August 1985.
16. M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
17. D. E. Knuth. *The Art of Computer Programming. Volume 2, Seminumerical Algorithms*. Addison-Wesley, 1969.
18. M. Müller-Olm and H. Seidl. Computing Interprocedurally Valid Relations in Affine Programs. In *ACM SIGPLAN Principles of Programming Languages (POPL 2004)*, pages 330–341, 2004.
19. E. Rodríguez-Carbonell and D. Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. (extended version) `www.lsi.upc.es/~erodri`.
20. E. Rodríguez-Carbonell and D. Kapur. Program Verification Using Automatic Generation of Polynomial Invariants. `www.lsi.upc.es/~erodri`.
21. E. Rodríguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *International Symposium on Symbolic and Algebraic Computation 2004 (ISSAC04)*, 2004.
22. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear Loop Invariant Generation Using Gröbner Bases. In *ACM SIGPLAN Principles of Programming Languages (POPL 2004)*, pages 318–329, 2004.