A Note on Karr's Algorithm

Markus Müller-Olm[1]* and Helmut Seidl[2]

[1] FernUniversität Hagen, FB Informatik, LG PI 5, Universitätsstr. 1, 58097 Hagen, Germany
mmo@ls5.informatik.uni-dortmund.de
[2] TU München, Informatik, I2, 85748 München, Germany
seidl@informatik.tu-muenchen.de

Abstract. We give a simple formulation of Karr's algorithm for computing all affine relationships in affine programs. This simplified algorithm runs in time $\mathcal{O}(nk^3)$ where n is the program size and k is the number of program variables assuming unit cost for arithmetic operations. This improves upon the original formulation by a factor of k. Moreover, our re-formulation avoids exponential growth of the lengths of intermediately occurring numbers (in binary representation) and uses less complicated elementary operations. We also describe a generalization that determines all polynomial relations up to degree d in time $\mathcal{O}(nk^{3d})$.

1 Introduction

In 1976, Michael Karr came up with an ingenious algorithm that computes for each program point in a flow graph a vector space of affine relations that hold among the program variables whenever control reaches the program point [6].[1] His algorithm is an iterative fixpoint algorithm that propagates affine spaces through the flow graph and computes for each program point u an affine space that over-approximates the set of run-time states that occur at u, i.e., contains all those run-time states. Hence, affine relationships valid for all states of the computed affine space are also valid for all possible run-time states. Karr represents affine spaces by kernels of affine transformations, i.e., as sets of solutions of linear equation systems. From this representation the affine relations valid for all states in a given affine space can be read off easily.

Finding valid affine relations has many applications. Many classical data flow analysis problems can be conceived as problems about affine relations such as *definite equalities among variables* like $\mathbf{x} = \mathbf{y}$ and *constant propagation*. More general affine relations (such as $2\mathbf{x} + 3\mathbf{y} = 0$) found by automatic analysis routines can also be used as valid assertions in program verification. Leroux uses affine relations for the analysis of counter systems [7]. More applications are discussed in [6, 11].

In recent related work [4, 8, 11] a number of difficulties associated with Karr's algorithm have been observed. Firstly, Karr's algorithm uses quite complicated operations like the transfer function for ("non-invertible") assignments and the union of affine spaces. Secondly, due to the complexity of these operations a straightforward implementation of Karr's algorithm performs $\mathcal{O}(nk^4)$ arithmetic operations in the worst-case

* On leave from Universität Dortmund.

[1] An *affine relation* is a property of the form $a_0 + \sum_{i=1}^{k} a_i\mathbf{x}_i = 0$, where $\mathbf{x}_1, \ldots, \mathbf{x}_k$ are program variables and $a_0, \ldots, a_k$ are elements of the underlying field of values.

J. Díaz et.al. (Eds.): ICALP 2004, LNCS 3142, pp. 1016–1027, 2004.

(where n is the size of the flow graph and k is the number of program variables) and it is not obvious to improve upon this complexity by using standard tricks like semi-naïve fixpoint iteration. Thirdly, the algorithm can lead to exponentially large numbers.

The main contribution of this paper is an extremely simple formulation of Karr's algorithm which solves all three above problems. By using a different representation of affine spaces – we represent an affine space A of dimension l by $l+1$ affine independent points of A – the union operation and the transfer functions become virtually trivial; by using semi-naïve iteration, the complexity goes down to $\mathcal{O}(nk^3)$; and the involved numbers remain of polynomial length. We also show how to generalize our version of Karr's algorithm to determine *polynomial relations*, i.e., properties of the form $p = 0$, where p is a multi-variate polynomial in the program variables $\mathbf{x}_i$.

In this paper we study *affine programs* [11] which differ from ordinary programs in that they have non-deterministic (instead of conditional) branching, and contain only assignments where the right-hand sides either are affine expressions like in $\mathbf{x}_3 := \mathbf{x}_1 - 3\mathbf{x}_2 + 7$ or equal "?" denoting an unknown value. Clearly, our analysis can be applied to arbitrary programs by ignoring the conditions at branchings and simulating input operations and non-affine right-hand sides in assignments through assignments of unknown values. As a byproduct of our considerations we show that Karr's algorithm is *precise* for affine programs, i.e., computes not just some but *all* valid affine relations. While this is kind of folklore knowledge in the field, it has (up to our knowledge) not been formally stated and proved before. Similarly, we show that our extension determines all valid polynomial relations up to a given degree in an affine program.

Related Work. Karr's algorithm has been generalized in different directions. A prominent generalization is the use of polyhedra instead of affine spaces for approximation of sets of program states; the classic reference is Cousot's and Halbwachs' paper [3]. Polyhedra allow us to determine also valid affine *in*equalities like $3\mathbf{x}_1 + 5\mathbf{x}_2 \leq 7\mathbf{x}_3$. Since the lattice of polyhedra has infinite height, widening must be used to ensure termination of the analysis (see [1] for a recent discussion) – making it unsuitable for precise analyses. Like Karr's original algorithm, analyses using polyhedra suffer from the problem of potentially large numbers.

More recently, we have described an analysis that determines all valid polynomial relations of bounded degree in *polynomial programs* [10, 9] with techniques from computable algebra. (In polynomial programs deterministic assignments with polynomial right hand side as well as polynomial disequality guards are allowed.) However, while we can show termination of the analysis we do not know an upper complexity bound.

Gulwani and Necula [4] present a probabilistic analysis for finding affine relations that with a (small) probability yields non-valid affine relations. Unlike the algorithms described so far, however, their algorithm assumes that variables take values in the finite field $\mathbb{Z}_p = \mathbb{Z}/(p\mathbb{Z})$ of natural numbers modulo p, where p is a (large) prime number, instead of natural or rational numbers. This assumption is introduced for two reasons. Firstly, it is needed for the estimation of the error probability. Secondly, it avoids problems with exponentially large numbers. In comparison our version of Karr's algorithm guarantees to yield only valid affine relations and to use only polynomially large numbers despite of working with rational numbers.

Like Karr's algorithm the analyses described so far are intraprocedural algorithms, i.e., they do not treat procedures. Precise *interprocedural* algorithms for affine programs that compute all valid affine or polynomial relations of bounded degree, respectively, are presented in [11]. While these algorithms run in polynomial time, they are asymptotically slower than Karr's, even if we specialize them to the intraprocedural case.

2 Affine Programs

We use a similar notation as in [11]. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ be the set of variables the program operates on and let $\mathbf{x}$ denote the vector of variables $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)$. We assume that the variables take values in $\mathbb{Q}$, the field of rational numbers. Then a *state* assigning values to the variables is conveniently modeled by a k-dimensional vector $x = (x_1, \ldots, x_k) \in \mathbb{Q}^k$; x_i is the value assigned to variable $\mathbf{x}_i$. Note that we distinguish variables and their values by using a different font.

For the moment, we assume that the basic statements in the program are *affine assignments* of the form $\mathbf{x}_j := t_0 + \sum_{i=1}^k t_i \mathbf{x}_i$ (with $t_i \in \mathbb{Q}$ for $i = 0, \ldots, k$ and $\mathbf{x}_j \in \mathbf{X}$) and that branching is non-deterministic. We show in Sect. 4 how to extend the basic algorithm to non-deterministic assignments $x_i :=?$ and discuss guards in Sect. 7. Let Stmt be the set of affine assignments. Each affine assignment $s \equiv \mathbf{x}_j := t_0 + \sum_{i=1}^k t_i \mathbf{x}_i$ induces a transformation, $[\![s]\!]$, on the program state given by $[\![s]\!]x = (x_1, \ldots, x_{j-1}, t_0 + \sum_{i=1}^k t_i x_i, x_{j+1}, \ldots, x_k)$. It is easy to see that $[\![s]\!]$ is an affine transformation, i.e., it can be written in the form $[\![s]\!]x = Ax + b$ for a matrix $A \in \mathbb{Q}^{k \times k}$ and a vector $b \in \mathbb{Q}^k$.

An *affine program* is given by a *control flow graph* $G = (N, E, \mathsf{st})$ that consists of: a set N of *program points*; a set of edges $E \subseteq N \times \mathsf{Stmt} \times N$; and a special *entry (or start) point* $\mathsf{st} \in N$.

As common in flow analysis, we use the program's collecting semantics [2] as a reference point for judging the soundness and completeness of Karr's algorithm. The collecting semantics assigns to each program point $u \in N$ the set of all those states that occur at u in some execution of the program. It can be characterized as the least solution of the following constraint system, V, on sets of states, i.e., subsets of $\mathbb{Q}^k$:

$$
\begin{array}{ll}
[\mathrm{V1}] & \mathbf{V}[\mathsf{st}] \supseteq \mathbb{Q}^k \\
[\mathrm{V2}] & \mathbf{V}[v] \supseteq f_s(\mathbf{V}[u])\,, \ \text{for each } (u, s, v) \in E\,,
\end{array}
$$

where the transfer functions f_s are defined by $f_s(X) = \{[\![s]\!]x \mid x \in X\}$. We denote the components of the least solution of the constraint system V (which exists by Knaster-Tarski fixpoint theorem) by $V[v]$, $v \in N$.

3 The Algorithm

The *affine hull* of a subset $G \subseteq \mathbb{Q}^k$ is the set

$$
\mathsf{aff}(G) = \{\sum_{j=0}^{m} \lambda_j x_j \mid m \geq 0, x_j \in G, \lambda_j \in \mathbb{Q}, \sum_{j=0}^{m} \lambda_j = 1\}\,.
$$

In particular, $\mathsf{aff}(G) = G$ whenever G contains at most one element. Whenever $X = \mathsf{aff}(G)$ for some G, we call X an *affine space* and G a set of *generators* for X. If G is a minimal set with $X = \mathsf{aff}(G)$ we call G an *affine basis* of X. The goal of our algorithm is easily stated in terms of the collecting semantics: compute for each program point u the affine hull of the collecting semantics for u, $\mathsf{aff}(V[u])$.

Obviously, aff is a *closure operator*, i.e., it is monotonic and we have, $\mathsf{aff}(X) \supseteq X$ and $\mathsf{aff}(\mathsf{aff}(X)) = \mathsf{aff}(X)$ for all $X \subseteq \mathbb{Q}^k$. It is well-known in abstract interpretation, that the image of a closure operator on a complete lattice is a complete lattice as well (cf., e.g., [2]). By definition, the image of aff consists of the affine subspaces of $\mathbb{Q}^k$. Let us denote this complete lattice by $(\mathbb{D}, \sqsubseteq) = (\{X \subseteq \mathbb{Q}^k \mid X = \mathsf{aff}(X)\}, \subseteq)$. The least element of $\mathbb{D}$ is $\emptyset$ and its greatest element is $\mathbb{Q}^k$. It is well-known that affine spaces are closed under intersection but not under union. Correspondingly, the meet and join operations of the lattice $\mathbb{D}$ are given by the following equations: $\sqcap\mathcal{X} = \cap\mathcal{X}$ and $\sqcup\mathcal{X} = \mathsf{aff}(\cup\mathcal{X})$ for $\mathcal{X} \subseteq \mathbb{D}$. In particular, we have:

Lemma 1. *For all sets $\mathcal{X} \subseteq 2^{\mathbb{Q}^k}$ of subsets of states,* $\mathsf{aff}(\cup\mathcal{X}) = \sqcup\{\mathsf{aff}(X) \mid X \in \mathcal{X}\}$.

The height of $\mathbb{D}$ is $k+1$ as in any strictly increasing chain $A_0 \subset A_1 \subset \ldots$ the dimensions must strictly increase: $\dim(A_0) < \dim(A_1) < \ldots$. Here, the dimension of $\emptyset$ is -1, and the dimension of a non-empty affine space X is the dimension of the linear space $L = \{x - x_0 \mid x_0, x \in X\}$. Thus, the dimensions are bounded by -1 from below and by k from above. (It is easy to construct a strictly increasing chain of length $k+1$.)

Recall that every statement s defines an affine transformation $[\![s]\!]$. Therefore:

Lemma 2. *For all statements s and $X \subseteq \mathbb{Q}^k$,* $\mathsf{aff}(f_s(X)) = f_s(\mathsf{aff}(X))$.

Let $V^\sharp$ be the following constraint system obtained from V by replacing "$\supseteq$" with "$\sqsupseteq$", i.e., switching from the complete lattice of subsets of states to the lattice of affine spaces.

$$
\begin{array}{ll}
[\mathrm{V1}^\sharp] & \mathbf{V}^\sharp[\mathsf{st}] \sqsupseteq \mathbb{Q}^k \\
[\mathrm{V2}^\sharp] & \mathbf{V}^\sharp[v] \ \sqsupseteq f_s(\mathbf{V}^\sharp[u])\,, \ \text{for each } (u,s,v) \in E\,.
\end{array}
$$

We denote the components of the least solution of $V^\sharp$ over the domain $(\mathbb{D}, \sqsubseteq)$ by $V^\sharp[v]$, $v \in N$. This solution again exists by Knaster-Tarski fixpoint theorem. Lemmas 1 and 2 together with the fact that $\mathsf{aff}(\mathbb{Q}^k) = \mathbb{Q}^k$ imply by standard argumentation from abstract interpretation that the least solution of the abstract constraint system $V^\sharp$ is the precise abstraction of the least solution of the concrete constraint system V, i.e.:

Lemma 3. *For all program points v,* $V^\sharp[v] = \mathsf{aff}(V[v])$.

In order to obtain an effective algorithm we must choose a finitary representation of affine spaces. As mentioned, Karr represents affine spaces by kernels of affine transformation. Instead, we represent an affine space $X \subseteq \mathbb{Q}^k$ by an affine basis of X. This enables us to use semi-naïve fixpoint iteration for computing the solution of constraint system $V^\sharp$. A corresponding algorithm is given in Fig. 1. The algorithm uses an array G indexed by the program points $u \in N$ to store the sets of vectors to become generating sets for $V^\sharp[u]$. Moreover, it uses a workset W in which it holds pairs of the form

```
forall (v ∈ N)  G[v] = ∅;
G[st] = {0, e_1, ..., e_k};
W = {(st, 0), (st, e_1), ..., (st, e_k)};
while (W ≠ ∅) {
    (u, x) = Extract(W);
    forall (s, v with (u, s, v) ∈ E) {
        t = [[s]]x;
        if (t ∉ aff(G[v])) {
            G[v] = G[v] ∪ {t};
            W = W ∪ {(v, t)};
        }
    }
```

Fig. 1: The base algorithm.

$(u, x) \in N \times \mathbb{Q}^k$; each pair (u, x) stored in W records that vector x has still to be propagated from program point u. We write $\mathbf{0}$ for the zero vector and $e_1, \ldots, e_k$ for the standard basis of the vector space $\mathbb{Q}^k$. The function $\mathit{Extract}(W)$ returns an arbitrary element of W and removes it from W.

The idea of semi-naïve fixpoint iteration is to propagate just "increments" instead of full abstract values via the edges of the flow graph. Thus it avoids full re-computation of the transfer functions for new abstract values. In our case a full abstract value is an affine subspace of $\mathbb{Q}^k$ and an "increment" amounts to a new affine independent vector x that is added to a generating set stored for some program point u. The propagation of x via an outgoing edge (u, s, v) is done by applying the concrete semantics of statement s, $[\![s]\!]$, to the vector x, and adding the result to the set of generators stored for the target program point of this edge, v, if it is not already in the affine hull of $G[v]$. Intuitively, this is sufficient because, by Lemma 2, $G' = \{[\![s]\!]x \mid x \in G\}$ is a generating set for $f_s(X)$ if $X = \mathsf{aff}(G)$. Sect. 3.1 contains a more formal correctness argument.

3.1 Correctness

We claim that the algorithm in Fig. 1 computes sets of generators for the affine spaces $V^\sharp[v]$. The proof of this claim is based on two invariants of the **while**-loop:

I1: for all $v \in N$, $G[v] \subseteq V[v]$, and for all $(u, x) \in W$, $x \in V[u]$.
I2: for all $(u, s, v) \in E$, $\mathsf{aff}(G[v] \cup \{[\![s]\!]x \mid (u, x) \in W\}) \sqsupseteq f_s(\mathsf{aff}(G[u]))$.

Both invariants can be easily verified by inspection of the initialization code and body of the **while**-loop. We thus obtain:

Theorem 1. *a) The above algorithm terminates after at most $nk + n$ iterations of the loop (where $n = |N|$ and k is the number of variables).*
b) For all $v \in N$, we have $\mathsf{aff}(G_{\mathrm{fin}}[v]) = V^\sharp[v]$, where $G_{\mathrm{fin}}[v]$ is the value of $G[v]$ upon termination of the algorithm.

Proof. **a)** In each iteration of the loop an entry is extracted from the workset W until the workset is empty. Therefore, the number of loop iterations equals the number of elements that are put to the workset. We observe that a new pair (u, x) is put to the workset only when the affine space $\mathsf{aff}(G[u])$ has been enlarged. In summary, this is also true for the initialization of G and W. Since each strictly ascending chain of affine spaces has length at most $k+1$, we conclude that for every program point u, there are at most $(k+1)$ insertions into W. Since there are at most n program points, the algorithm terminates after at most $n \cdot (k+1)$ iterations of the **while**-loop.

b) In order to show the inclusion $\mathsf{aff}(G_{\text{fin}}[v]) \sqsubseteq V^\sharp[v]$ we note that the loop invariant **I1** implies in particular that $G_{\text{fin}}[v] \subseteq V[v]$ for each $v \in N$. Hence, $\mathsf{aff}(G_{\text{fin}}[v]) \sqsubseteq \mathsf{aff}(V[v]) = V^\sharp[v]$ for each $v \in N$.

In order to prove the reverse inclusion, $\mathsf{aff}(G_{\text{fin}}[v]) \sqsupseteq V^\sharp[v]$, we observe that the invariant **I2** implies that upon termination when the workset W is empty, we have

$$\mathsf{aff}(G_{\text{fin}}[v]) \sqsupseteq f_s(\mathsf{aff}(G_{\text{fin}}[u]))$$

for all $(u, s, v) \in E$. We also have $\mathsf{aff}(G_{\text{fin}}[\mathsf{st}]) \sqsupseteq \mathsf{aff}(\{\mathbf{0}, e_1, \ldots, e_k\}) = \mathbb{Q}^k$ because the elements $\mathbf{0}, e_1, \ldots, e_k$ assigned to $G[\mathsf{st}]$ by the initialization are never removed. Hence the family of values $(\mathsf{aff}(G_{\text{fin}}[v]))_{v \in N}$ satisfies all the constraints of the constraint system $V^\sharp$. As the values $V^\sharp[v]$ are the components of the *least* solution of $V^\sharp$, this implies $\mathsf{aff}(G_{\text{fin}}[v]) \sqsupseteq V^\sharp[v]$ for all $v \in N$. $\square$

3.2 Complexity

In order to reason about the complexity of the algorithm, we consider a uniform cost measure, i.e., we count each arithmetic operation for 1. Moreover, we assume that the affine assignments at control flow edges are of *constant size*, meaning that all occurring coefficients are of constant size, and that each assignment s may contain only a constant number of variables with non-zero coefficients. Note that this assumption does not impose any restriction on the expressiveness of programs since more complicated assignments can easily be simulated by sequences of simpler ones. As a consequence, the size of the control flow graph, $n = |N| + |E|$, can be considered as a fair measure of the size of the input to the analysis algorithm.

Taking a closer look at the algorithm, we notice that each iteration of the **while**-loop consists in processing one pair (u, x) by inspecting each outgoing edge (u, s, v) of u. Thus, its time complexity is proportional to $1 + \mathsf{out}(u) \cdot C$ where $\mathsf{out}(u)$ is the out-degree of u and C is the complexity of checking whether a vector t is contained in $\mathsf{aff}(G[v])$ for some program point v. Since the sum $\sum_{u \in N} \mathsf{out}(u)$ equals the number of edges of the control flow graph, the complexity of the algorithm is proportional to

$$(k+1) \cdot \sum_{u \in N} (1 + \mathsf{out}(u) \cdot C) \;\leq\; (k+1) \cdot (n + n \cdot C) \;=\; (k+1) \cdot n \cdot (C+1)\,.$$

It remains to determine the complexity C of testing whether a vector t is contained in the affine hull of $G[v]$ for some program point v. If $G[v]$ is empty, the test will always return false. Otherwise, $G[v]$ consists of vectors $x_0, \ldots, x_m$, $0 \leq m \leq k$. Then $t \in \mathsf{aff}(G[v])$ iff the vector $t - x_0$ is contained in the *linear* vector space generated from $B = \{x_1 - x_0, \ldots, x_m - x_0\}$. This can be decided by means of Gaussian elimination – resulting in an $\mathcal{O}(k^3)$ upper bound on the complexity C of the element test.

We can do better, though. The key idea is to avoid repeated Gaussian elimination on larger and larger subsets of vectors. Instead, we maintain for v with $G[v] \neq \emptyset$ a *diagonal* basis $B' = \{x'_1, \ldots, x'_m\}$ spanning the same linear vector space as B. This means: if l_i is the index of the first non-zero component of x'_i for $i = 1, \ldots, m$, then the l_i'th component of all other basis vectors x'_j, $j \neq i$ is zero. *Reduction* of a vector

$x = t - x_0$ w.r.t. the diagonal basis B' then amounts to successively subtracting suitable multiples of the vectors x'_i from x in order to make the l_i'th components of x zero. Let x' denote the vector obtained by reduction of $t - x_0$. Then $x' = \mathbf{0}$ iff $t - x_0$ is contained in L or, equivalently, $t \in \mathsf{aff}(\{x_0, \ldots, x_m\})$. If $x' \neq \mathbf{0}$, the algorithm inserts t into the set $G[v]$. Therefore, we must extend B' to a diagonal basis for $\mathbf{Span}(B \cup \{t - x_0\})$ in this case. Indeed, this is very simple: we only need to subtract suitable multiples of x' from the vectors $x'_1, \ldots, x'_m$ in order to make the l'th component of these vectors zero, where l is the index of the first non-zero component of x'. Afterwards, we add x' to the set consisting of the resulting vectors. In summary, we have replaced a full Gaussian elimination for each test $t \in \mathsf{aff}(G[u])$ by the reduction of $t - x_0$ possibly followed by the reduction of the vectors in B' by x'. Subtraction of a multiple of one x'_i from t and of a multiple of x' from x'_i uses $\mathcal{O}(k)$ operations. Since $m \leq k$, reduction of $t - x_0$ as well as reduction of B' can thus be done in time $\mathcal{O}(k^2)$. Therefore we obtain:

Theorem 2. *The affine hulls $V^\sharp[u] = \mathsf{aff}(V[u])$ of the sets of program states reaching $u, u \in N$, can be computed in time $\mathcal{O}(nk^3)$ where n is the size of the program and k the number of program variables.*

Moreover this computation performs arithmetic operations only on numbers upto bit length $\mathcal{O}(nk^2)$.

Proof. It only remains to estimate the lengths of numbers used by the algorithm. First, we observe that the algorithm performs at most $n \cdot (k+1)$ evaluations of assignment statements s. Each assignment may increase the maximal absolute value of entries of a vector x at most by a constant factor $d > 0$. Therefore, the absolute values of entries of all vectors in $G_{\text{fin}}[u], u \in N$, are bounded by $d^{n \cdot (k+1)}$. Now for each set $G_{\text{fin}}[u] = \{x_0, \ldots, x_m\}$ with $m > 0$, the algorithm successively applies reduction to construct a diagonal basis for the vectors $x_j - x_0, j = 1, \ldots, m$. Altogether these reduction steps perform one Gaussian elimination on all m vectors. It is well-known that Gaussian elimination introduces rational numbers whose numerators and denumerators are determinants of minors of the original coefficient matrix [12, Problem 11.5.3]. In our application, the original entries have absolute values at most $2 \cdot d^{n \cdot (k+1)}$. At most k-fold products therefore have absolute values at most $2^k \cdot d^{n \cdot (k+1)k}$. Finally, determinants are at most $(k!)$-fold sums of such products. Therefore, their absolute values are bounded by $k! \cdot 2^k \cdot d^{nk(k+1)} = 2^{\mathcal{O}(n \cdot k^2)}$ – which completes the proof. □

4 Non-Deterministic Assignments

Let us now extend affine programs as defined in Section 2 with non-deterministic assignments $\mathbf{x}_i := ?$. Such assignments are necessary to model input routines returning unknown values or variable assignments whose right-hand sides are not affine expressions. The semantics of such a statement may update $\mathbf{x}_i$ in the current state with any possible value. Therefore, the transfer function $f_{\mathbf{x}_i := ?}$ is given by $f_{\mathbf{x}_i := ?}(X) = \bigcup\{f_{\mathbf{x}_i := c}(X) \mid c \in \mathbb{Q}\}$. Unfortunately, this is not a finitary definition no matter whether X is an affine space or not. Fortunately, we have:

Lemma 4. $f_{x_i:=?}(\mathsf{aff}(G)) = (f_{x_i:=0}(\mathsf{aff}(G))) \sqcup (f_{x_i:=1}(\mathsf{aff}(G)))$.

Thus for affine X, the infinite union in the definition of $f_{x_i:=?}$ can be simplified to the least upper bound of two affine spaces. Lemma 4 implies that we can treat unknown assignments in flow graphs by replacing each edge (u, s, v) that is annotated with an unknown assignment, $s \equiv x_i :=?$, by the two edges $(u, x_i := 0, v)$ and $(u, x_i := 1, v)$ labeled by affine assignments prior to the analysis.

5 Affine Relations

An equation $a_0 + a_1\mathbf{x}_1 + \ldots + a_k\mathbf{x}_k = 0$ is called an *affine relation*. Clearly, such a relation can be uniquely represented by its coefficient vector $a = (a_0, \ldots, a_k) \in \mathbb{Q}^{k+1}$. The affine relation a is *valid* for set $X \subseteq \mathbb{Q}^k$ iff a is satisfied by all $x \in X$, i.e.,

$$a_0 + \sum_{i=1}^{k} a_i \cdot x_i = 0 \qquad \text{for all} \quad (x_1, \ldots, x_k) \in X \,.$$

Accordingly, the relation a is *valid* at a program point u iff it is valid for the set $V[u]$ of all program states reaching u. The key objective, now of Karr's algorithm was *not* to determine (an approximation of) the collecting semantics of the program but to determine, for every program point u, the set $V^{\top}[u]$ of all affine relations valid at u. Here we show that this task is easy — once we have computed the affine hull $V^{\sharp}[u]$ of the sets of program states reaching u. First we recall from linear algebra that the set:

$$A(X) = \{a \in \mathbb{Q}^{k+1} \mid a \text{ is valid for } X\}$$

is a *linear* vector space. Moreover, we have for every affine relation a:

Lemma 5. *For every $X \subseteq \mathbb{Q}^k$, a is valid for X iff a is valid for* $\mathsf{aff}(X)$.

Thus, given a set $\{x_0, \ldots, x_m\}$ of vectors generating $\mathsf{aff}(X)$, we can determine the set $A(X)$ as the set of solutions of the linear equation system:

$$\mathbf{a}_0 + \mathbf{a}_1 \cdot x_{i1} + \ldots + \mathbf{a}_k \cdot x_{ik} \;=\; 0 \qquad i = 0, \ldots, m$$

if $x_i = (x_{i1}, \ldots, x_{ik})$. Determining a basis for the vector space of solutions can again be done, e.g., by Gaussian elimination. Thus, we obtain:

Theorem 3. *Assume p is an affine program of size n with k program variables. Then the sets of all relations valid at program points u can be computed in time $\mathcal{O}(nk^3)$.*

The computation requires algebraic operations only for integers of lengths bounded by $\mathcal{O}(nk^2)$.

Recall, moreover, that our algorithm not only provides us, for every program point u, with a finite set of generators of $\mathsf{aff}(V[u])$. Whenever $\mathsf{aff}(V[u]) \neq \emptyset$, it also returns a pair (x_0, B) where x_0 is an element of $V[u]$ and B is a diagonal basis of a linear vector space L such that $x \in \mathsf{aff}(V[u])$ iff $x = x_0 + x'$ for some $x' \in L$.

Lemma 6. *Assume a non-empty affine space X is given by a vector $x_0 \in X$ together with a basis B for the linear vector space $L = \{x - x_0 \mid x \in X\}$. Then the set of affine relations valid for X is the set of all solutions of the equation system:*

$$\begin{aligned} \mathbf{a}_0 + \mathbf{a}_1 \cdot x_{01} + \ldots + \mathbf{a}_k \cdot x_{0k} &= 0 \\ \mathbf{a}_1 \cdot x'_{i1} + \ldots + \mathbf{a}_k \cdot x'_{ik} &= 0 \quad \textit{for } i = 1, \ldots, m\,, \end{aligned}$$

where $x_0 = (x_{01}, \ldots, x_{0k})$ and $B = \{x'_1, \ldots, x'_m\}$ with $x'_i = (x'_{i1}, \ldots, x'_{ik})$.

Moreover, if the basis B is already in diagonal form, we directly can read off a basis for $A(X)$. From a practical point of view, we therefore can be even more efficient and avoid the extra post-processing round of Gaussian elimination.

6 Polynomial Relations

In [11], an interprocedural algorithm is presented which not only computes, for every program point u of an affine program, the set of valid affine relations but the set of all *polynomial relations* of degree at most d in time $\mathcal{O}(nk^{8d})$. Here we show how our version of Karr's algorithm can be extended to compute polynomial relations intraprocedurally much faster.

A *polynomial relation* is an equation $p = 0$ for a polynomial $p \in \mathbb{Q}^{\leq d}[\mathbf{X}]$, i.e., a polynomial in the unknowns $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ with coefficients from $\mathbb{Q}$ and degree bounded by d. Recall that any such polynomial can be represented as its coefficient vector $a = (a_I)_{I \in \mathcal{I}_d}$ where the index set $\mathcal{I}_d$ is given by

$$\mathcal{I}_d = \{(i_1, \ldots, i_k) \mid i_1 + \ldots + i_k \leq d\}\,.$$

Recall that $|\mathcal{I}_d| = \binom{k+d}{d}$. The polynomial relation $p = 0$ is *valid* for a set $X \subseteq \mathbb{Q}^k$ iff p is satisfied by all $x \in X$, i.e., $p[x/\mathbf{x}] = 0$ for all $(x_1, \ldots, x_k) \in X$. Accordingly, the relation $p = 0$ is *valid at a program point* u iff it is valid for the set $V[u]$ of all program states reaching u. Our goal is to determine, for every program point u, the set of all polynomial relations of degree up to d valid at u. Note that the set:

$$P_d(X) = \{p \in \mathbb{Q}^{\leq d}[\mathbf{X}] \mid p \text{ is valid for } X\}$$

is still a *linear* vector space of dimension less or equal $\binom{k+d}{d} = \mathcal{O}(k^d)$. This vector space, however, can no longer be determined from the *affine hull* of X.

As a simple example consider the two flow graphs in Fig. 2. In G_1, we have $V[1] = V^\sharp[1] = \{(x_1, x_2) \in \mathbb{Q}^2 \mid x_1 = x_2\}$. In G_2, we have $V[5] = \{(0,0), (1,1)\}$. Hence $V^\sharp[5] = \mathsf{aff}(V[5]) = \{(x_1, x_2) \in \mathbb{Q}^2 \mid x_1 = x_2\} = V^\sharp[1]$. It is easy to see, however, that at node 5 the polynomial relation $x_1^2 - x_2 = 0$ holds for all run-time states in contrast to node 1.

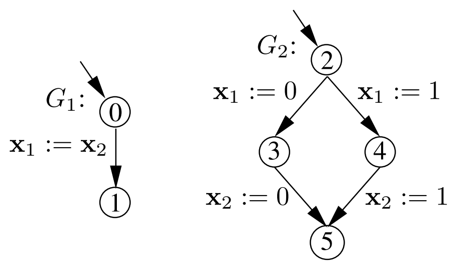

Fig. 2: Polynomial relations and affine hull.

Therefore, we define the *polynomial hull* $\mathsf{pol}_d(X)$. We do this in two steps. For a vector $x = (x_1, \ldots, x_k) \in \mathbb{Q}^k$, we define its *polynomial extension* $\eta_d(x) = (x^I)_{I \in \mathcal{I}_d}$ of degree d by: $x^{(i_1,\ldots,i_k)} = x_1^{i_1} \cdot \ldots \cdot x_k^{i_k}$, where, in particular, $x^{(0,\ldots,0)} = 1$. Thus, the polynomial extension of x has exactly $\binom{k+d}{d}$ components. Let $\eta_d(X) = \{\eta_d(x) \mid x \in X\}$. We call a vector x *polynomially implied* (up to degree d) by $X \subseteq \mathbb{Q}^k$ iff $\eta_d(x) \in \mathbf{Span}(\eta_d(X))$, i.e., iff the polynomial extension $\eta_d(x)$ is contained in the *linear hull* of the polynomial extensions of the vectors in X. The *polynomial hull* of degree d, $\mathsf{pol}_d(X)$, then consists of all vectors which are polynomially implied by X :

$$\mathsf{pol}_d(X) = \{x \in \mathbb{Q}^k \mid \eta_d(x) \in \mathbf{Span}(\eta_d(X))\} .$$

It is easily verified that the polynomial hull of X of degree 1 coincides with the affine hull of X. Moreover, we show for every polynomial p of degree at most d:

Lemma 7. *For every $X \subseteq \mathbb{Q}^k$, $p = 0$ is valid for X iff $p = 0$ is valid for $\mathsf{pol}_d(X)$.*

Thus, given a set $\{x_0, \ldots, x_m\}$ of vectors whose extensions $\eta_d(x_i) = (z_{iI})_{I \in \mathcal{I}_d}$ generate the linear vector space $\mathbf{Span}(\eta_d(X))$, we can determine the set $P_d(X)$ as the set of solutions of the linear equation system:

$$\sum_{I \in \mathcal{I}_d} \mathbf{a}_I \cdot z_{iI} = 0 \quad , \qquad i = 0, \ldots, m$$

Determining a basis for the vector space of solutions can again be done, e.g., by Gaussian elimination — now with $\mathcal{O}(k^d)$ variables. Thus, in order to compute the sets $P_d(V[u])$, we modify our base fixpoint algorithm to compute, instead of a finite generating set of $\mathsf{aff}(V[u])$, a finite set $G_d[u]$ generating the polynomial hull of $V[u]$. It is easily verified that pol_d is again a closure operator. Also Lemma 2 remains valid for the polynomial hull, i.e., $\mathsf{pol}_d(f_s(X)) = f_s(\mathsf{pol}_d(X))$ for all statements s and $X \subseteq \mathbb{Q}^k$. A suitable set of vectors that represents $\mathbb{Q}^k$ up to pol_d is given by the following lemma:

Lemma 8. $\mathsf{pol}_d(\mathcal{I}_d) = \mathbb{Q}^k$.

Sketch of proof. The vector space spanned by $\eta_d(\mathbb{Q}^k)$ is contained in the vector space $\mathbb{Q}^{d'}$ for $d' = \binom{d+k}{d}$. It trivially subsumes the span of $\eta_d(\mathcal{I}_d)$, i.e., $\mathbf{Span}(\eta_d(\mathcal{I}_d)) \subseteq \mathbf{Span}(\eta_d(\mathbb{Q}^k)) \subseteq \mathbb{Q}^{d'}$. We prove by induction on $k + d$ that, for all $p \in \mathbb{Q}^{\leq d}[X]$: $p(x) = 0$ for all $x \in \mathcal{I}_d$ implies $p \equiv \mathbf{0}$. From this we conclude that the set of polynomial extensions $\eta_d(x), x \in \mathcal{I}_d$, is in fact linearly independent. Therefore, their span, $\mathbf{Span}(\eta_d(\mathcal{I}_d))$, has dimension d' and thus equals $\mathbb{Q}^{d'}$. This implies $\mathsf{pol}_d(\mathcal{I}_d) = \mathbb{Q}^k$. $\square$

By arguing similarly to Sect. 3, we obtain an algorithm that computes a finite generating set of $\mathsf{pol}_d(V[u])$ by modifying the algorithm in Fig. 1 as follows. We replace the test "$t \notin \mathsf{aff}(G[u])$" with "$t \notin \mathsf{pol}_d(G[u])$" and the initialization of $G[\mathsf{st}]$ and W with

$$G[\mathsf{st}] = \mathcal{I}_d\,; \;\; W = \{(\mathsf{st}, I) \mid I \in \mathcal{I}_d\}\,;$$

In order to avoid replicated Gaussian elimination, we may maintain a diagonal basis B_d for the current vector space $\mathbf{Span}(\eta_d(G_d[u]))$. This simplifies the element test for every newly encountered $x \in \mathbb{Q}^k$ to the reduction of the extension $\eta_d(x)$ of x w.r.t. B_d possibly followed by reduction of the vectors in B_d with the reduced vector. We obtain:

Theorem 4. *Assume* p *is an affine program of size* n *with* k *program variables. Then the sets of all polynomial relations of degree at most* d *which are valid at program points* u *can be computed in time* $\mathcal{O}(nk^{3d})$.

The computation requires algebraic operations only for integers of lengths bounded by $\mathcal{O}(nk^{2d})$.

Similarly to [11] we can treat non-deterministic assignments $x_i :=?$ by replacing each edge $(u, x_i :=?, v)$ by $d+1$ edges $(u, x_i := l, v)$ for $l = 0, \ldots, d$. Note that the complexity of the resulting intraprocedural algorithm improves upon the complexity of our interprocedural algorithm in [11] by a factor of k^{5d}.

7 Positive Guards

In this paper we restricted attention to affine programs for which we have shown our algorithms to be precise. In Karr's paper, one can also find a non-trivial treatment of branching nodes with affine guards. The main idea is to intersect in the "true" branch the propagated affine space with the hyperplane described by the guard. While this leads to more precise results than ignoring guards totally, it is not a complete treatment of positive affine guards. Indeed, as we show next it is undecidable to decide in affine programs with positive affine guards (or even with only equality guards) whether a given affine relation holds at a program point or not. This implies that a complete algorithmic treatment of positive affine guards is impossible.

We exhibit a reduction of the Post correspondence problem (PCP) inspired by Hecht [5, 8]. A Post correspondence system is a finite set of pairs $(u_1, v_1), \ldots, (u_m, v_m)$ with $u_i, v_i \in \{0,1\}^*$. The correspondence system has a solution, if and only if there is a non-empty sequence $i_1, \ldots, i_n$ such that $u_{i_1} \cdot \ldots \cdot u_{i_n} = v_{i_1} \cdot \ldots \cdot v_{i_n}$. From a given Post correspondence system we construct an affine program with an equality guard as indicated in Fig. 3. We write $|u|$ for the length of a string $u \in \{0,1\}^*$ and $\langle u \rangle_2$ for the number represented by u in standard binary number representation. The variables $\mathbf{x}$ and $\mathbf{y}$ hold binary numbers that represent strings in $\{0,1\}^*$. For each pair $(u_i, v_i) \in S$ there is an edge from program point 1 to 2 that appends the strings u_i and v_i to x and y, respectively, by appropriate affine computations. The program can loop back from program point 2 to 1 by a skip-edge. The initialization of x and y with 1 avoids a problem with leading zeros. It is not hard to see that there is an execution in which $\mathbf{x} = \mathbf{y}$ is true at program point 2 if and only if the Post correspondence system admits a solution. Only in this case the path from program point 2 via 3 to 4 can be executed. We conclude that the affine relation $\mathbf{z} = 0$ is valid at program point 4 if and only if the given Post correspondence system S does not admit a solution.

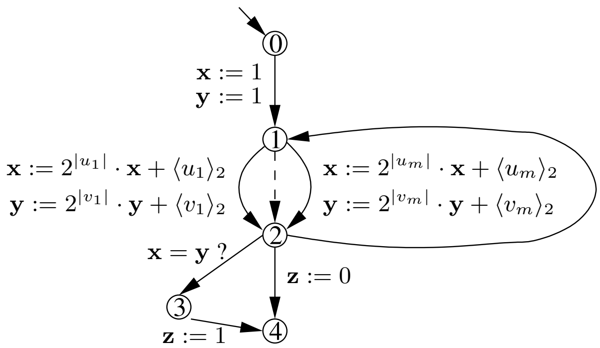

Fig. 3: A PCP reduction with affine guards.

8 Discussion and Perspective

We have presented a variant of Karr's algorithm for computing valid affine relationships among the variables in a program that has a better worst-case complexity than Karr's original formulation, avoids exponentially large numbers, and is easy to implement. We also showed how to generalize this algorithm to determine polynomial relationships.

Instrumental for our results is that we represent affine spaces by affine bases instead of kernels of affine transformations. Ironically, Karr discards a closely related representation early in his paper [6, p. 135] by remarking that the number of valid affine relationships typically will be small and hence the dimension of the affine spaces will be large, such that many basis vector but few relations are required for representation. This leads to the question whether our representation can compete with Karr's as far as memory consumption is concerned. Clearly, we need more memory for representing an affine space A of high dimension, if we store all the vectors in an affine basis $\{x_0, \ldots, x_m\}$ of A explicitly. Fortunately, instead of storing the affine basis, it suffices to store one vector, x_0, together with the diagonal basis of $\mathbf{Span}(\{x_1 - x_0, \ldots, x_m - x_0\})$ that is computed for the membership tests. The other vectors $x_1, \ldots, x_m$ need not be stored because they are neither needed for the membership tests nor for extraction of the final result. The vectors in the diagonal basis, however, can be stored sparsely such that only the non-zero components (together with their index) are stored. Then we need for representing an affine space of dimension m, $0 \leq m \leq k$, at most $k + m + m(k - m)$ entries compared to at most $2k - 2m + m(k - m)$ in a (sparse) representation by affine relations. Surprisingly, the maximal difference is just $2k$. Insights into the practical behavior of these two representations require experiments for real-world programs which we leave for future work.

References

1. R. Bagnara, P. Hill, E. Ricci, and E. Zaffanella. Precise Widening Operators for Convex Polyhedra. In *10th Int. Static Analysis Symp. (SAS)*, 337–354. LNCS 2694, Springer, 2003.
2. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL*, 1977.
3. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *5th POPL*, 84–97, 1978.
4. S. Gulwani and G. Necula. Discovering Affine Equalities Using Random Interpretation. In *30th POPL*, 74–84, 2003.
5. M. S. Hecht. *Flow analysis of computer programs*. Elsevier North-Holland, 1977.
6. M. Karr. Affine Relationships Among Variables of a Program. *Acta Inf.*, 6:133–151, 1976.
7. J. Leroux. *Algorithmique de la Vérification des Systèmes à Compteurs: Approximation et Accélération*. PhD thesis, Ecole Normale Supérieure de Cachan, 2003.
8. M. Müller-Olm and O. Rüthing. The Complexity of Constant Propagation. In *10th European Symposium on Programming (ESOP)*, 190–205. LNCS 2028, Springer, 2001.
9. M. Müller-Olm and H. Seidl. Computing Polynomial Program Invariants. Submitted, 2003.
10. M. Müller-Olm and H. Seidl. Polynomial Constants are Decidable. In *9th Static Analysis Symposium (SAS)*, 4–19. LNCS 2477, Springer, 2002.
11. M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *31st POPL*, 330–341, 2004.
12. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.