Precise Interprocedural Analysis through Linear Algebra

Markus Müller-Olm [*]
FernUniversität Hagen, LG Praktische Informatik 5
58084 Hagen, Germany
mmo@ls5.cs.uni-dortmund.de

Helmut Seidl
TU München, Lehrstuhl für Informatik II
80333 München, Germany
seidl@informatik.tu-muenchen.de

Abstract

We apply linear algebra techniques to precise interprocedural dataflow analysis. Specifically, we describe analyses that determine for each program point identities that are valid among the program variables whenever control reaches that program point. Our analyses fully interpret assignment statements with affine expressions on the right hand side while considering other assignments as non-deterministic and ignoring conditions at branches. Under this abstraction, the analysis computes the set of all affine relations *and, more generally, all* polynomial relations of bounded degree *precisely. The running time of our algorithms is linear in the program size and polynomial in the number of occurring variables. We also show how to deal with affine preconditions and local variables and indicate how to handle parameters and return values of procedures.*

Categories and Subject Descriptors: F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; D.3.3 [Programming Languages]: Language Constructs and Features—*procedures, functions, and subroutines*; D.3.4 [Programming Languages]: Processors—*compilers*; D.3.4 [Programming Languages]: Processors—*optimization*

General Terms: algorithms, theory, verification.

Keywords: interprocedural analysis, linear algebra, weakest precondition, affine relation, polynomial relation.

1 Introduction

The field of program analysis is concerned with designing algorithms that compute information about the dynamic behavior of programs by a static analysis. Such information is useful in many circumstances. Important application areas are optimizing compilers and validation or verification of programs. An often used simplification is to work with *intraprocedural* or *context-insensitive* analyses. Intraprocedural analyses treat bodies of single procedures (or methods) in isolation, while context-insensitive analyses assume conservatively that a procedure called at one call site may return to any other call site of this procedure. The context-insensitive approach, though interprocedural, still is limited in the quality of the computed information: if only weak information about one particular calling context of a procedure or method is available, this may affect all other calling contexts. The design of *context-sensitive* interprocedural analyses that mirror the actual call/return behavior of programs is generally deemed to be challenging. Here, if we speak about interprocedural analyses without further qualification, we always will mean context-sensitive ones.

In this paper, we show how linear algebra techniques can be used for interprocedural flow analysis. Our specific goal is to determine for each program point *affine* and, more generally, *polynomial relations* that are valid among the program variables whenever control reaches that program point. An affine relation is a condition of the form $a_0 + \sum_{i+1}^{n} a_i \mathbf{x}_i = 0$, where $a_0, \ldots, a_n$ are constants and $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are program variables; a polynomial relation is a condition of the form $p(\mathbf{x}_1, \ldots, \mathbf{x}_n) = 0$, where p is a multi-variate polynomial in $\mathbf{x}_1, \ldots, \mathbf{x}_n$. We call an analysis *precise* (w.r.t. a given class of programs) if it computes for every program point u of a program all valid relations of the given form which are valid along every feasible program path reaching u. (A program path is called *feasible* if it mirrors the actual call/return behavior of procedures.)

Looking for valid affine and polynomial relations is a rather general question with many applications. First of all, many classical data flow analysis problems can be seen as problems about affine and polynomials relations. Some examples are: finding *definite equalities among variables* like $\mathbf{x} = \mathbf{y}$; *constant propagation*, i.e. detecting variables or expressions with a constant value at run-time; *discovery of symbolic constants* like $\mathbf{x} = 5\mathbf{y} + 2$ or even $\mathbf{x} = \mathbf{y}\mathbf{z}^2 + 42$; *detection of complex common sub-expressions* where even expressions are sought which are syntactically different but have the same value at run-time; and *discovery of loop induction variables*. Karr [10] also discusses applications in connection with parallelization of do-loops.

Affine and polynomial relations found by an automatic analysis routine are also useful in program verification contexts, as they provide non-trivial valid assertions about the program. In particular, certain loop invariants can be discovered in this way fully automatically. As affine and, even more, polynomial relations express quite complex relationships among variables, the discovered assertions

[*]On leave from Universität Dortmund, FB 4, LS 5, 44221 Dortmund, Germany.

may form the backbone of the program proof and thus significantly simplify the verification task.

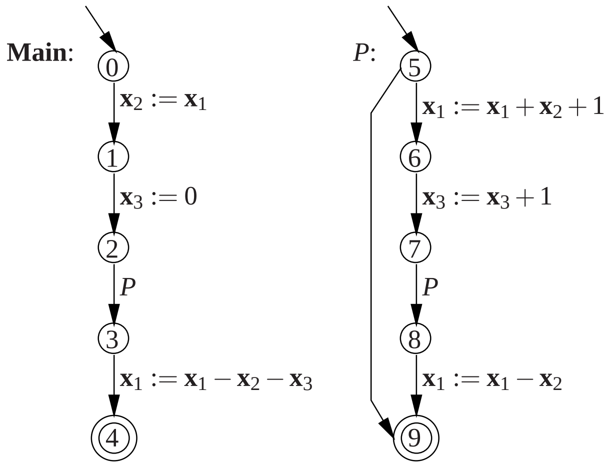
In this paper we consider *affine programs* for which our analysis will be precise, i.e., compute not some but *all* affine relations which are valid at a program point. We then extend this analysis to compute all valid polynomial relations up to a given degree d and to take affine preconditions into account completely. Affine programs differ from ordinary programs over integers in that they have non-deterministic (instead of conditional) branching, and contain only assignments where the right-hand sides either equal "?" denoting an unknown value, or are affine expressions such as in $\mathbf{x}_3 := \mathbf{x}_1 - 3\mathbf{x}_2 + 7$. Clearly, in practice our analyses can also be applied to arbitrary programs simply by ignoring the conditions at branchings and simulating input operations and non-affine right-hand sides in assignments through assignments of unknown values.

To use linear algebra for program analysis is not a new idea. In his seminal paper [10], Karr presents an intraprocedural analysis that determines all intraprocedurally valid affine relations in an affine program. However, the potential of linear algebra has never been exploited fully. We extend Karr's work in three respects. Firstly, we describe a *precise interprocedural* analysis of affine programs. Secondly, we extend our algorithm to an algorithm that computes all interprocedurally valid *polynomial relations* of degree bounded by some fixed d. Thirdly, we show how to treat *local variables* and indicate how to handle *parameters and result values* of procedures. Our base algorithm as well as the extended algorithms run in time linear in the program size and polynomial in the number of program variables.

The key observation onto which our algorithms are based is that the weakest precondition of an affine or polynomial relation a along a single run of an affine program can be determined by means of a linear transformation applied to a. The set of all linear transformations of a vector space again forms a vector space and we can compute for each program point u the finite-dimensional subspace generated by the linear transformations induced by the program runs reaching u. A relation a turns out to be valid at u if and only if the subspace of linear transformations computed for u transforms a into $\mathbf{0}$ (or a relation implied by the precondition, respectively). This implies that the set of all valid relations can be computed as the set of solutions of a linear equation system. Thus, finite-dimensional subspaces of linear transformation describe the effect of procedure calls precisely enough.

The program in Figure 1 illustrates the kind of properties our analyses can handle. It consists of two procedures **Main** and P. After memorizing the (unknown) initial value of variable $\mathbf{x}_1$ in variable $\mathbf{x}_2$ and initializing $\mathbf{x}_3$ by zero, **Main** calls P. Procedure P can either terminate without changing any variable or call itself recursively. In the latter case, it increments $\mathbf{x}_1$ by $\mathbf{x}_2+1$ and $\mathbf{x}_3$ by 1 before the recursive call and decrements $\mathbf{x}_1$ by $\mathbf{x}_2$ afterwards. Therefore, the total effect of each instance of P with a recursive call is to increment both $\mathbf{x}_1$ and $\mathbf{x}_3$ by one. Thus, upon termination of the call to P in **Main** (i.e., at program point 3), $\mathbf{x}_3$ holds the number of recursive calls of P and $\mathbf{x}_1$ the value $\mathbf{x}_2+\mathbf{x}_3$. Consequently, the final assignment in **Main** always assigns zero to $\mathbf{x}_1$. More formally, this amounts to saying that the *affine relation* $\mathbf{x}_1 - \mathbf{x}_2 - \mathbf{x}_3 = 0$ is valid at program point 3 and that the affine relation $\mathbf{x}_1 = 0$ is valid at program point 4.

Another interesting relationship between the variables holds whenever P is called. As mentioned, variable $\mathbf{x}_3$ counts the number of recursive calls, and, thus, how often $\mathbf{x}_1$ has been incremented by



Main:
0
$\mathbf{x}_2 := \mathbf{x}_1$
1
$\mathbf{x}_3 := 0$
2
P
3
$\mathbf{x}_1 := \mathbf{x}_1 - \mathbf{x}_2 - \mathbf{x}_3$
4
P:
5
$\mathbf{x}_1 := \mathbf{x}_1 + \mathbf{x}_2 + 1$
6
$\mathbf{x}_3 := \mathbf{x}_3 + 1$
7
P
8
$\mathbf{x}_1 := \mathbf{x}_1 - \mathbf{x}_2$
9

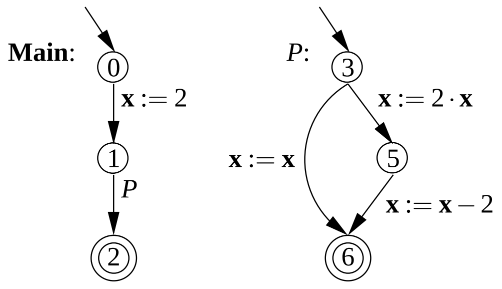
Figure 1. An example program.

$\mathbf{x}_2+1$. Consequently, at any call to P variable $\mathbf{x}_1$ holds the value $\mathbf{x}_2+\mathbf{x}_3(\mathbf{x}_2+1) = \mathbf{x}_2\mathbf{x}_3+\mathbf{x}_2+\mathbf{x}_3$. This amounts to saying that the *polynomial relation* (of degree 2) $\mathbf{x}_2\mathbf{x}_3 - \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 = 0$ is valid at program points 2, 5 and 7.

Related Work

Unlike our algorithm, Karr's intraprocedural algorithm [10] works with a forward propagation of affine spaces and uses quite complicated subroutines to deal with join points and assignments $\mathbf{x}_j := t$ where the affine right-hand side depends on the variable $\mathbf{x}_j$ on the left-hand side. Similar to our approach, it abstracts non-affine assignments and general guards. Due to the forward propagation strategy, however, it is able to handle *positive affine guards* precisely. In [13] we observe that, in absence of affine guards, *checking* a given affine relation for validity at a program point can be performed by a simpler *backward propagating* algorithm which in turn is generalized to a backward propagating algorithm for checking arbitrary polynomial relations for polynomial programs (where polynomial right hand side of assignments are interpreted) in [15, 14]. In a recent paper, Gulwani and Necula [8] present a probabilistic analysis for finding affine relationships. Their algorithm is also just intraprocedurally applicable but asymptotically faster than Karr's—at the price of a (small) probability of yielding non-valid affine relations.

A generalization of these approaches to the interprocedural case is not obvious. The functional approach of Sharir/Pnueli [20, 11] to designing interprocedural data flow analyses is limited to finite lattices of data flow informations. Accordingly, it has successfully been applied to the detection of *copy constants* [17]. In copy constant detection only assignments of the form $\mathbf{x} := a$ are treated exactly where a is a constant or a variable. The lattice of affine spaces, however, is clearly infinite. The call string approach of [20], on the other hand, is applicable to more general lattices but abstracts the call/return behavior of procedures. Thus, it does not lead to *precise* interprocedural analyses. In more recent work on precise interprocedural analysis, Horwitz et al. propose a polynomial-time algorithm for detecting *linear constants* [9] interprocedurally. In linear constant detection only those affine assignments are interpreted whose right-hand sides contain at most *one occurrence* of a variable. We strictly improve on these results as our analyses treat *all* affine assignments exactly and determine more general properties.



Main:
0
$\mathbf{x} := 2$
1
P
2
P:
3
$\mathbf{x} := 2 \cdot \mathbf{x}$
$\mathbf{x} := \mathbf{x}$
5
$\mathbf{x} := \mathbf{x} - 2$
6

Figure 2. Another example program.

A generalization of Karr's algorithm in another direction is the use of polyhedra instead of affine spaces for approximately representing sets of program states; the classic reference is Cousot's and Halbwachs' paper [6]. Polyhedra allow to determine besides affine equalities also affine inequalities like $3\mathbf{x}_1 + 5\mathbf{x}_2 \leq 7\mathbf{x}_3$. Since the lattice of polyhedra has infinite height, widenings must be used to ensure termination of the analysis (see [2] for a recent discussion)—making it unsuitable for precise analyses. Sets of affine inequalities, however, allow to relate the values of variables before and after a procedure call (a *relational analysis* in the terminology of Cousot)—thus naturally allowing for an interprocedural generalization. A relational analysis, however, that uses affine spaces or polyhedra for approximating the relational semantics of procedures is not precise enough to detect all valid affine relations in a program with procedures. For a simple example see Figure 2. The true relational semantics of procedure P is described by the formula $\mathbf{x} = \mathbf{x}_0 \vee \mathbf{x} = 2 \cdot \mathbf{x}_0 - 2)$ where $\mathbf{x}_0$ represents the initial and $\mathbf{x}$ the final value of the variable. The best approximation of this relation by an affine space or polyhedron is described by the formula true. It is obvious that this approximation of P's semantics is too weak to detect that the affine relation $\mathbf{x} = 2$ is valid at program point 2 in procedure **Main**.

The paper is organized as follows. In Section 2, we formally introduce the programs to be analyzed together with their semantics. In Section 3, we introduce affine relations, their weakest preconditions along a program run and explain our algorithm for this special case. In Section 4, we generalize our approach to deal with arbitrary polynomial relations of bounded degree. In Section 5, we extend our approach to procedures with local variables and in Section 6 we show how to take into account affine preconditions completely.

2 Affine Programs

We model programs by systems of non-deterministic flow graphs that can recursively call each other as in Figure 1. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ be the set of (global) variables the program operates on. We use $\mathbf{x}$ to denote the column vector[1] of variables $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)^t$. We assume that the variables take values in a fixed field $\mathbb{F}$. In practice, $\mathbb{F}$ is the field of rational numbers. Then a *state* assigning values to the variables is conveniently modeled by a k-dimensional (column) vector $x = (x_1, \ldots, x_k)^t \in \mathbb{F}^k$; 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 a state x, a variable $\mathbf{x}_i$ and a value $c \in \mathbb{F}$, we write $x[\mathbf{x}_i \mapsto c]$ for the state $(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_k)^t$.

We assume that the basic statements in the program are either *affine assignments* of the form $\mathbf{x}_j := t_0 + \sum_{i=1}^k t_i \mathbf{x}_i$ (with $t_i \in \mathbb{F}$ for $i = 0, \ldots, k$ and $\mathbf{x}_j \in \mathbf{X}$) or *non-deterministic assignments* of the form $\mathbf{x}_j :=?$ (with $\mathbf{x}_j \in \mathbf{X}$). Assignments $\mathbf{x}_j := \mathbf{x}_j$ have no effect onto the program state. They are also called skip statements and omitted in pictures. Non-deterministic assignments $\mathbf{x}_j :=?$ represent a safe abstraction of statements in a source program our analysis cannot handle precisely, for example of assignments $\mathbf{x}_j := t$ with non-affine expressions t or of read statements $\mathsf{read}(\mathbf{x}_j)$. Let Stmt be the set of basic statements.

[1]The superscript "t" denotes the *transpose* operation which mirrors a matrix at the main diagonal and changes a row vector into a column vector (and vice versa).

A *program* comprises a finite set Proc of *procedure names* that contains a distinguished procedure **Main**. Execution starts with a call to **Main**. Each procedure name $p \in \mathsf{Proc}$ is associated with a *control flow graph* $G_p = (N_p, E_p, A_p, e_p, r_p)$ that consists of:

- a set N_p of *program points*;
- a set of edges $E_p \subseteq N_p \times N_p$;
- a mapping $A_p : E_p \to \mathsf{Stmt} \cup \mathsf{Proc}$ that annotates each edge with a basic statement of the form described above or a procedure call;
- a special *entry (or start) point* $e_p \in N_p$; and
- a special *return point* $r_p \in N_p$.

We assume that the program points of different procedures are disjoint: $N_p \cap N_q = \emptyset$ for $p \neq q$. This can always be enforced by renaming program points.

We write N for $\bigcup_{p \in \mathsf{Proc}} N_p$, E for $\bigcup_{p \in \mathsf{Proc}} E_p$, and A for $\bigcup_{p \in \mathsf{Proc}} A_p$. We agree that $\mathsf{Base} = \{e \mid A(e) \in \mathsf{Stmt}\}$ is the set of base edges and $\mathsf{Call}_p = \{e \mid A(e) \equiv p\}$ is the set of edges that call procedure p.

The core part of our algorithm can be understood as a precise abstract interpretation of a constraint system characterizing the program executions that reach program points. We represent program executions or *runs* by sequences of affine assignments. Formally, a *run* r is a finite sequence

$$r \equiv s_1; \ldots; s_m$$

of assignments s_i of the form $\mathbf{x}_j := t$ where $\mathbf{x}_j \in \mathbf{X}$ and $t \equiv t_0 + \sum_{i=1}^k t_i \mathbf{x}_i$ for some $t_0, \ldots, t_k \in \mathbb{F}$. We write Runs for the set of runs. The set of runs *reaching program point* $u \in N$ can be characterized as the least solution of a system of subset constraints on run sets (see, e.g., [19] for a similar approach for explicitly parallel programs). We start by defining the program executions of base edges e in isolation. If e is annotated by an affine assignment, i.e., $A(e) \equiv \mathbf{x}_j := t$, it gives rise to a single execution: $\mathbf{S}(e) = \{\mathbf{x}_j := t\}$. The effect of base edges e annotated by a non-deterministic assignment $\mathbf{x}_j :=?$ is captured by all runs that assign some value from $\mathbb{F}$ to $\mathbf{x}_j$:

$$\mathbf{S}(e) = \{\mathbf{x}_j := c \mid c \in \mathbb{F}\}.$$

Thus, we capture the effect of non-deterministic assignments by collecting *all constant* assignments. Next, we characterize *same-level runs*. Same-level runs of procedures capture complete runs of procedures in isolation. As auxiliary sets we consider same-level runs of program nodes, i.e., those runs that reach a program point u in a procedure p from a call to p on same-level, i.e., after all procedures called by p have terminated. The same-level runs of procedures and program nodes are the smallest solution of the constraint system $\mathbf{S}$:

$$
\begin{array}{lll}
\text{[S1]} & \mathbf{S}(q) \supseteq \mathbf{S}(r_q) & \\
\text{[S2]} & \mathbf{S}(e_q) \supseteq \{\varepsilon\} & \\
\text{[S3]} & \mathbf{S}(v) \supseteq \mathbf{S}(u)\,;\mathbf{S}(e) & \text{if } e=(u,v)\in \mathsf{Base} \\
\text{[S4]} & \mathbf{S}(v) \supseteq \mathbf{S}(u)\,;\mathbf{S}(p) & \text{if } e=(u,v)\in \mathsf{Call}_p
\end{array}
$$

where "ε" denotes the empty run, and the operator ";" denotes concatenation of run sets. By [S1], the set of same-level runs of a procedure q comprises all same-level runs reaching the return point of q. By [S2], the set of same-level runs of the entry point of a procedure contains the empty run. By [S3] and [S4], a same-level run for a program point v is obtained by considering an ingoing edge $e=(u,v)$. In both cases, we concatenate a same-level run reaching u with a run corresponding to the edge. If e is a base edge, we concatenate with an edge from $\mathbf{S}(e)$. If e is a call to a procedure p, we take a same-level run of p.

Next, we characterize the runs that reach program points. They are the smallest solution of the constraint system $\mathbf{R}$:

$$
\begin{array}{llll}
\text{[R1]} & \mathbf{R}(\mathbf{Main}) \supseteq \{\varepsilon\} & & \\
\text{[R2]} & \mathbf{R}(p) & \supseteq \mathbf{R}(u) & \text{if } (u,\_)\in \mathsf{Call}_p \\
\text{[R3]} & \mathbf{R}(u) & \supseteq \mathbf{R}(p)\,;\mathbf{S}(u) & \text{if } u\in N_p
\end{array}
$$

By [R1], the procedure **Main** is reachable by the empty path. By [R2], every procedure p is reachable by a path reaching a call of p. By [R3], we obtain a run reaching a program point u in some procedure p, by composing a run reaching p with a same-level run reaching u.

So far, we have furnished procedural flow graphs with a symbolic operational semantics only by describing the sets of sequences of assignments possibly reaching program points. Each of these runs gives rise to a transformation of the underlying program state $x\in\mathbb{F}^k$. Every assignment statement $\mathbf{x}_i := t$ induces a state transformation $[\![\mathbf{x}_j := t]\!] : \mathbb{F}^k \to \mathbb{F}^k$ given by

$$[\![\mathbf{x}_j := t]\!]x \;=\; x[\mathbf{x}_j \mapsto t(x)]\,,$$

where $t(x)$ is the value of term t in state x. This definition is inductively extended to runs: $[\![\varepsilon]\!] = \mathsf{Id}$, where Id is the identical mapping and $[\![ra]\!] = [\![a]\!]\circ[\![r]\!]$.

The state transformation of an affine assignment $\mathbf{x}_j := t_0 + \sum_{i=1}^k t_i\mathbf{x}_i$ is an affine transformation. Hence, it can be written in the form $[\![\mathbf{x}_j := t]\!]x = Ax+b$ with a matrix $A\in\mathbb{F}^{k\times k}$ and a (column) vector $b\in\mathbb{F}^k$. More specifically, A and b have the form indicated below:

$$A = \left(\begin{array}{c|c} I_{j-1} & 0 \\ \hline \multicolumn{2}{c}{t_1\ \dots\ t_k} \\ \hline 0 & I_{k-j} \end{array}\right) \qquad b = \left(\begin{array}{c} 0 \\ \hline t_0 \\ \hline 0 \end{array}\right) \tag{1}$$

Here, I_i is the unit matrix with i rows and columns and 0 denotes zero matrices and vectors of appropriate dimension. In b, t_0 appears as j-th component.

As a composition of affine transformations, the state transformer of a run is an affine transformation as well. For any run r, let $A_r\in\mathbb{F}^{k\times k}$ and $b_r\in\mathbb{F}^k$ be such that $[\![r]\!]x = A_r x + b_r$.

3 Affine Relations and Weakest Preconditions

An *affine relation* over a vector space $\mathbb{F}^k$ is an equation $a_0 + a_1\mathbf{x}_1 + \dots a_k\mathbf{x}_k = 0$ for some $a_i\in\mathbb{F}$. Geometrically, it can be viewed as a hyper-plane in the k-dimensional vector space $\mathbb{F}^k$. Such a relation can be represented as a polynomial of degree at most 1 (namely, the left-hand side) or, equivalently, as a column vector $a=(a_0,\dots,a_k)^{\mathrm{t}}$. In particular, the set of all affine relations forms an $\mathbb{F}$-vector space which is isomorphic to $\mathbb{F}^{k+1}$. The vector $y\in\mathbb{F}^k$ *satisfies* the affine relation a iff $a_0 + a'\cdot y = 0$ where $a' = (a_1,\dots,a_k)^{\mathrm{t}}$ and "$\cdot$" denotes scalar product. We write $y\models a$ to denote this fact. Geometrically, this means that the point y is an element of the hyper-plane described by a.

The affine relation a is *valid* after a single run r iff $[\![r]\!]x \models a$ for all $x\in\mathbb{F}^k$, i.e., iff $a_0 + a'\cdot[\![r]\!]x = 0$ for all $x\in\mathbb{F}^k$; x represents the unknown initial state. Thus, $a_0 + a'\cdot[\![r]\!]\mathbf{x} = 0$ is the *weakest precondition* for validity of the affine relation a after run r. We have

$$
\begin{array}{ll}
 & a_0 + a'\cdot[\![r]\!]\mathbf{x} = 0 \\
\text{iff} & \text{[Choice of } A_r \text{ and } b_r\text{]} \\
 & a_0 + a'\cdot(A_r\mathbf{x} + b_r) = 0 \\
\text{iff} & \text{[Linearity, rearrangement]} \\
 & (a_0 + a'\cdot b_r) + a'\cdot A_r\mathbf{x} = 0 \\
\text{iff} & \text{[Law } x\cdot Ay = A^{\mathrm{t}}x\cdot y \text{ from linear algebra]} \\
 & (a_0 + a'\cdot b_r) + (A_r^{\mathrm{t}}a')\cdot\mathbf{x} = 0
\end{array}
$$

From this characterization we see that the weakest precondition is again an affine relation. Even better: The mapping that assigns to each affine relation its weakest precondition before run r is the linear map described by the following $(k+1)\times(k+1)$ matrix W_r:

$$W_r = \left(\begin{array}{c|c} 1 & b_r^{\mathrm{t}} \\ \hline 0 & A_r^{\mathrm{t}} \end{array}\right) \tag{2}$$

In particular, we have proved that for every $x\in\mathbb{F}^k$:

$$[\![r]\!]x \models a \quad\text{iff}\quad x\models W_r\,a\,. \tag{3}$$

Thus, the matrix W_r provides us with a finite description of the weakest precondition transformer for affine relations of a single program execution r.

Note that the only affine relation which is true for *all program states* is the relation $\mathbf{0} = (0,\dots,0)^{\mathrm{t}}$. Thus, the affine relation a is valid after run r iff $W_r\,a = \mathbf{0}$, because the initial state is arbitrary. Accordingly, the affine relation a is valid at a program point u, iff it is valid after all runs $r\in\mathbf{R}(u)$. Summarizing, we have:

LEMMA 1. *The affine relation $a\in\mathbb{F}^{k+1}$ is valid at program point u iff $W_r a = \mathbf{0}$ for all $r\in\mathbf{R}(u)$.*

Thus, the set $W = \{W_r \mid r\in\mathbf{R}(u)\}$ gives us a handle to solve the validity problem for affine relations. The problem is that we do not know how to represent W in a finitary way—let alone how to compute it. In this place, we recall from linear algebra that the set of $(k+1)\times(k+1)$ matrices again forms an $\mathbb{F}$-vector space. The dimension of this vector space equals $(k+1)^2$. We observe:

LEMMA 2. *Let M denote a set of $n\times n$ matrices.*

a) For every $W\in M$, the set $\{a \mid Wa = \mathbf{0}\}$ forms a subspace of $\mathbb{F}^n$.

b) As an intersection of vector spaces, the set $\{a \mid \forall W\in M : Wa = \mathbf{0}\}$ forms a subspace of $\mathbb{F}^n$.

c) *For every $a \in \mathbb{F}^n$, the following three statements are equivalent:*

   – *$Wa = \mathbf{0}$ for all $W \in M$;*

   – *$Wa = \mathbf{0}$ for all $W \in$ **Span**(M).*

   – *$Wa = \mathbf{0}$ for all W in a basis of* **Span**(M).

Here, **Span**(M) denotes the vector space generated by the elements in M, i.e., the vector space of all linear combinations of elements in M. We conclude that we can work with **Span**(W), i.e., the subspace of $\mathbb{F}^{(k+1)\times(k+1)}$ generated by W without losing interesting information. As a subspace of the vector space $\mathbb{F}^{(k+1)\times(k+1)}$ of dimension $(k+1)^2$, **Span**(W) can be described by a basis of at most $(k+1)^2$ matrices. Indeed, due to the special form of the matrices W_r—in the first column all but the first entry are zero—**Span**(W) can have at most dimension k^2+k+1.

Based on these observations, we can determine the set of all affine relations at program point u from a basis of $\mathbf{Span}(\{W_r \mid r \in \mathbf{R}(u)\})$ and estimate the complexity of the resulting algorithm. For simplicity we use here and in the following unit cost measure for arithmetic operations.

THEOREM 1. *Assume we are given a basis B for the set* $\mathbf{Span}(\{W_r \mid r \in \mathbf{R}(u)\})$. *Then we have:*

a) *Affine relation $a \in \mathbb{F}^{k+1}$ is valid at program point u iff $Wa = \mathbf{0}$ for all $W \in B$.*

b) *A basis for the subspace of all affine relations valid at program point u can be computed in time $O(k^5)$.*

PROOF. Statement a) follows directly from Lemma 1 and Lemma 2,c).

For seeing b), consider that by a) the affine relation a is valid at u iff a is a solution of all the equations

$$\sum_{j=0}^{k} w_{ij}\mathbf{a}_j = 0$$

for each matrix $W = (w_{ij}) \in B$ and $i = 0,\ldots,k$.

The basis B contains at most $O(k^2)$ matrices each of which contributes $k+1$ equations. Thus, we must determine, the solution of an equation system with $O(k^3)$ equations over $k+1$ variables. This can be done, e.g. by Gaussian elimination, in time $O(k^5)$. □

So we are left with the task to compute, for every program point u, (a basis of) $\mathbf{Span}(\{W_r \mid r \in \mathbf{R}(u)\})$. This subspace of $\mathbb{F}^{(k+1)\times(k+1)}$ can be seen as an abstraction of the set $\mathbf{R}(u)$ of program executions reaching u. We are going to compute it by an abstract interpretation of the constraint system for $\mathbf{R}(u)$ from Section 2. Recall that the set of subspaces of a finite-dimensional $\mathbb{F}$-vector space V forms a complete lattice (w.r.t. the ordering set inclusion) where the least element is given by the 0-dimensional vector space consisting of the 0-vector only. The least upper bound of two spaces V_1, V_2 is given by:

$$\begin{array}{rcl} V_1 \sqcup V_2 & = & \mathbf{Span}(V_1 \cup V_2) \\ & = & \{v_1 + v_2 \mid v_i \in V_i\}. \end{array}$$

We denote the complete lattice of subspaces of V by **Sub**(V). The height of **Sub**(V), i.e., the maximal length of a strictly increasing chain, equals the dimension of V.

The desired abstraction of run sets is described by the mapping $\alpha : 2^{\mathsf{Runs}} \to \mathbf{Sub}(\mathbb{F}^{(k+1)\times(k+1)})$:

$$\alpha(R) = \mathbf{Span}(\{W_r \mid r \in R\}).$$

Thus, we have:

$$\begin{array}{rcl} \alpha(\emptyset) & = & \mathbf{Span}(\emptyset) \;=\; \{\mathbf{0}\} \\ \alpha(\{r\}) & = & \mathbf{Span}(\{W_r\}) \end{array}$$

for a single run r. By Equation (2) we get for the empty run,

$$\alpha(\{\varepsilon\}) \quad = \quad \mathbf{Span}(\{I_{k+1}\})$$

because $A_\varepsilon = I_k$ and $b_\varepsilon = \mathbf{0}$.

The mapping α is monotonic (w.r.t. subset ordering on sets of runs and subspaces.) Also it is not hard to see that it commutes with arbitrary unions.

In order to solve the constraint system for the run sets $\mathbf{R}(u)$ over abstract domain $\mathbf{Sub}(\mathbb{F}^{(k+1)\times(k+1)})$, we need adequate abstract versions of the operators and constants in this constraint system. In particular, we need an abstract version of the concatenation of run sets. For $M_1, M_2 \subseteq \mathbb{F}^{(k+1)\times(k+1)}$, we define:

$$M_1 \circ M_2 = \mathbf{Span}(\{A_1A_2 \mid A_i \in M_i\}).$$

First of all, we observe:

LEMMA 3. *For all sets of matrices M_1, M_2,*

$$\mathbf{Span}(M_1) \circ \mathbf{Span}(M_2) = M_1 \circ M_2\,.$$

PROOF. Observe first that $\mathbf{Span}(M_i) \supseteq M_i$ and therefore,

$$\mathbf{Span}(M_1) \circ \mathbf{Span}(M_2) \supseteq M_1 \circ M_2$$

by monotonicity of "$\circ$".

For the reverse inclusion, consider arbitrary elements $B_i = \sum_j \lambda_j^{(i)} \cdot A_j^{(i)}$ in $\mathbf{Span}(M_i)$ for suitable $A_j^{(i)} \in M_i$. Then

$$B_1B_2 = \sum_m \sum_j \lambda_m^{(1)} \lambda_j^{(2)} \cdot A_m^{(1)} A_j^{(2)}$$

by linearity of matrix multiplication. Since each $A_m^{(1)}A_j^{(2)}$ is contained in $M_1 \circ M_2$, B_1B_2 is contained in $M_1 \circ M_2$ as well. Therefore, also the inclusion "$\subseteq$" follows. □

Accordingly, a generating system for $M_1 \circ M_2$ can be computed from generating systems G_1, G_2 for M_1 and M_2 by multiplying each matrix in G_1 with each matrix in G_2.

Secondly, we observe that "$\circ$" precisely abstracts the concatenation of run sets:

LEMMA 4. *Let $R_1, R_2 \subseteq$* Runs. *Then*

$$\alpha(R_1) \circ \alpha(R_2) = \alpha(R_1\,;R_2)\,.$$

PROOF. Consider the auxiliary map W mapping run sets to sets of matrices by:

$$W(R) = \{W_r \mid r \in R\}\,.$$

Then we have $\alpha(R) = \mathbf{Span}(W(R))$. We observe:

$$\{A_1A_2 \mid A_i \in W(R_i)\} = W(R_1\,;R_2)\,.$$

This suffices as the span construction commutes with composition by Lemma 3. $\square$

Let us now turn attention to the abstraction of base edges. Let us first consider a base edge $e \in \mathsf{Base}$ annotated by an affine assignment, i.e., $A(e) \equiv x_j := t$ where $t \equiv t_0 + \sum_{i=1}^{n} t_i \mathbf{x}_i$. Then $\mathbf{S}(e) = \{\mathbf{x}_j := t\}$. By (1) and (2), the corresponding abstract transformer is given by

$$\begin{array}{rcl} \alpha(\mathbf{S}(e)) & = & \alpha(\{\mathbf{x}_j := t\}) \\ & = & \mathbf{Span}\left(\left\{\left(\begin{array}{c|c|c} I_j & t_0 & 0 \\ \hline & \vdots & \\ \hline 0 & t_k & I_{k-j} \end{array}\right)\right\}\right) \end{array}$$

Informally, the weakest precondition for an affine relation $a \in \mathbb{F}^{k+1}$ is computed by substituting t into $\mathbf{x}_j$ of the corresponding affine combination.

Next, consider a base edge $e \in \mathsf{Base}$ annotated with $\mathbf{x}_j :=?$. In this case, $\mathbf{S}(e) = \{\mathbf{x}_j := c \mid c \in \mathbb{F}\}$—implying that we have to abstract an *infinite* set of runs if the field $\mathbb{F}$ is infinite. Clearly, the abstraction of this set again can be finitely represented. We obtain this representation by selecting two different values from $\mathbb{F}$, e.g., 0 and 1. We find:

LEMMA 5.

$$\begin{array}{rcl} \alpha(\mathbf{S}(e)) & = & \alpha(\{\mathbf{x}_j := c \mid c \in \mathbb{F}\}) \\ & = & \mathbf{Span}(\{T_0, T_1\})\,, \end{array}$$

where $T_c = W_{\mathbf{x}_j := c}$ is the matrix obtained from I_{k+1} by replacing the $j+1$-th column with $(c, 0, \ldots, 0)^{\mathrm{t}}$.

PROOF. Only the second equation requires a proof. From Equations (1) and (2) we get $\alpha(\{\mathbf{x}_j := c \mid c \in \mathbb{F}\}) = \mathbf{Span}(\{T_c \mid c \in \mathbb{F}\})$. We verify: $T_c = (1-c)\cdot T_0 + c \cdot T_1$. Hence, $T_c \in \mathbf{Span}(\{T_0, T_1\})$ and $\mathbf{Span}(\{T_c \mid c \in \mathbb{F}\}) = \mathbf{Span}(\{T_0, T_1\})$. $\square$

From the constraint systems $\mathbf{S}$ and $\mathbf{R}$ for run sets, we construct now the constraint systems $\mathbf{S}_\alpha$ and $\mathbf{R}_\alpha$ by application of α. The variables in the new constraint systems take subspaces of $\mathbb{F}^{(k+1)\times(k+1)}$ as values. We apply α to the occurring constant sets $\{\varepsilon\}$ and $\mathbf{S}(e)$ and replace the concatenation operator ";" with "$\circ$":

$$\begin{array}{lll} \mathbf{S}_\alpha(q) & \supseteq \mathbf{S}_\alpha(r_q) & \\ \mathbf{S}_\alpha(e_q) & \supseteq \mathbf{Span}(\{\mathsf{Id}\}) & \\ \mathbf{S}_\alpha(v) & \supseteq \mathbf{S}_\alpha(u) \circ \alpha(\mathbf{S}(e)) & \text{if } e = (u,v) \in \mathsf{Base} \\ \mathbf{S}_\alpha(v) & \supseteq \mathbf{S}_\alpha(u) \circ \mathbf{S}_\alpha(p) & \text{if } e = (u,v) \in \mathsf{Call}_p \\ \mathbf{R}_\alpha(\mathbf{Main}) & \supseteq \mathbf{Span}(\{\mathsf{Id}\}) & \\ \mathbf{R}_\alpha(p) & \supseteq \mathbf{R}_\alpha(u) & \text{if } (u, \_) \in \mathsf{Call}_p \\ \mathbf{R}_\alpha(u) & \supseteq \mathbf{R}_\alpha(p) \circ \mathbf{S}_\alpha(u) & \text{if } u \in N_p \end{array}$$

The resulting constraint system can be solved by computing on bases. For estimating the complexity of the resulting algorithm, we assume that the basic statements in the given program have size $O(1)$. Thus, we measure the size n of the given program by $|N| + |E|$. Note that program nodes typically have bounded outdegree, such that typically $|N| + |E| = O(|N|)$.

THEOREM 2. *For every program of size n with k variables the following holds:*

a) The values:
$\mathbf{Span}(\{W_r \mid r \in \mathbf{S}(u)\})$, $u \in N$,
$\mathbf{Span}(\{W_r \mid r \in \mathbf{S}(p)\})$, $p \in \mathsf{Proc}$,
$\mathbf{Span}(\{W_r \mid r \in \mathbf{R}(u)\})$, $p \in \mathsf{Proc}$, and
$\mathbf{Span}(\{W_r \mid r \in \mathbf{R}(u)\})$, $u \in N$,
are the least solutions of the constraint systems $\mathbf{S}_\alpha$ and $\mathbf{R}_\alpha$, respectively.

b) These values can be computed in time $O(p \cdot k^8)$.

c) The sets of all valid affine relations at program point u, $u \in N$, can be computed in time $O(n \cdot k^8)$.

PROOF. Statement a) amounts to saying that the least solution of constraint systems $\mathbf{S}_\alpha$ and $\mathbf{R}_\alpha$ is obtained from the least solution of $\mathbf{S}$ and $\mathbf{R}$ by applying the abstraction α. This follows from the Transfer Lemma known in fixpoint theory (see, e.g., [1, 4]), which can be applied since α commutes with arbitrary unions, the concatenation operator is precisely abstracted by the operator $\circ$ (Lemma 4), and the constant run sets $\{\varepsilon\}$ and $\mathbf{S}(e)$ are replaced by their abstractions $\alpha(\{\varepsilon\}) = \mathbf{Span}(\{\mathsf{Id}\})$ and $\alpha(\mathbf{S}(e))$, respectively.

For b) we show that the least solution of the abstracted constraint systems can be computed in time $O(n \cdot k^8)$. For that, recall that the lattice of all subspaces of $\mathbb{F}^{(k+1)\times(k+1)}$ has height $(k+1)^2$. Thus, a worklist-based fixpoint algorithm will evaluate at most $O(n \cdot k^2)$ constraints. Each constraint evaluation consists of multiplying two sets of at most $(k+1)^2$ matrices. The necessary $(k+1)^4$ matrix multiplications can be executed in time $O(k^7)$. Finally, we must compute a basis for the span of the resulting $(k+1)^4$ matrices. By Gaussian elimination, this can be done in time $O(k^8)$. Altogether, we obtain an upper complexity bound of $O(n \cdot k^2 \cdot k^8) = O(n \cdot k^{10})$. A better running time can be obtained if we use a semi-naive fixpoint iteration strategy [16, 3, 7]. The idea here is that when the value of a fixpoint variable changes, we do not propagate the complete new value to all uses of the variable in right-hand sides of constraints but just the increment, i.e., in our case the new matrices extending the current basis (instead of the complete new basis). The total time spent with a constraint then sums up to $O(k^8)$ which overall results in the desired complexity $O(n \cdot k^8)$.

Finally, for c) we recall that we know from Theorem 1 that, from bases of $\mathbf{Span}(\{W_r \mid r \in \mathbf{R}(u)\})$ for all program points u, we can compute the sets of all valid affine relations within the stated complexity bounds. $\square$

Let us consider the example program from Figure 1 for illustration. Due to lack of space, we cannot describe the fixpoint iteration in detail or give the full result. However, we report and discuss some characteristic values. The fixpoint iteration for $\mathbf{S}_\alpha$ stabilizes after 3 iterations. We obtain: $\mathbf{S}_\alpha(P) = \mathbf{S}_\alpha(9) = \mathbf{Span}(\{I_4, W_1\})$ and $\mathbf{S}_\alpha(3) = \mathbf{Span}(\{W_1, W_2\})$, where W_1, W_2 are the matrices

$$W_1 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad W_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Also, $\mathbf{S}_\alpha(\mathbf{Main}) = \mathbf{S}_\alpha(4) = \mathbf{Span}(\{W_3, W_4\})$, where

$$W_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad W_4 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

As there are no recursive calls to **Main**, reaching runs and same-level runs coincide for the program points of **Main**. Consequently, we have, $\mathbf{R}_\alpha(3) = \mathbf{S}_\alpha(3) = \mathbf{Span}(\{W_1, W_2\})$. Hence, at program point 3 just the affine relations $a = (a_0, \ldots, a_k)^t$ with $W_1 a = 0$ and $W_2 a = 0$ are valid which reduces to the requirements $a_0 = 0$ and $a_2 = a_3 = -a_1$. Therefore, just the affine relations of the form $a_1\mathbf{x}_1 - a_1\mathbf{x}_2 - a_1\mathbf{x}_3 = 0$ are valid at program point 3, in particular, $\mathbf{x}_1 - \mathbf{x}_2 - \mathbf{x}_3 = 0$ which confirms our informal reasoning from the introduction.

For program point 4 we have $\mathbf{R}_\alpha(4) = \mathbf{S}_\alpha(4) = \mathbf{Span}(\{W_3, W_4\})$. Here, the requirements $W_3 a = 0$ and $W_4 a = 0$ reduce to $a_0 = a_2 = a_3 = 0$. Thus, just the affine relations of the form $a_1\mathbf{x}_1 = 0$ are valid at program point 4, in particular, $\mathbf{x}_1 = 0$. Again this confirms our informal reasoning that $\mathbf{x}_1$ is a constant of value zero.

The computation of $\mathbf{R}_\alpha$ for the program points of P stabilizes again after 3 iterations. For the program point 7 just before the recursive call to P, we obtain $\mathbf{R}_\alpha(7) = \mathbf{Span}(\{W_5, W_6\})$, where

$$W_5 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad W_6 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Here the conditions $W_5 a = 0$ and $W_6 a = 0$ for valid affine relations translate into $a_0 = a_1 = a_2 = a_3 = 0$. Interestingly, this implies that no non-trivial affine relation is valid at every call to P.

In order to find out about validity of the polynomial relation $\mathbf{x}_2\mathbf{x}_3 - \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 = 0$ at program point 7, hinted upon in the introduction, we must generalize our analysis to polynomial relations which is the topic of the next section.

4 Polynomial Relations of Bounded Degree

Polynomial relations are much more expressive than affine relations. In particular, they are closed under disjunction: $p = 0 \vee q = 0$ holds if and only if $pq = 0$. For example, the relation:

$$(\mathbf{x}_1 - 1) \cdot (\mathbf{x}_1 - \mathbf{x}_2) = 0$$

represents the *disjunction* of the two affine relations:

$$\mathbf{x}_1 - 1 = 0 \;\vee\; \mathbf{x}_1 - \mathbf{x}_2 = 0\,.$$

Also, the property whether a variable $\mathbf{x}_j$ has a value in a given finite set $\{c_1, \ldots, c_r\} \subseteq \mathbb{F}$ with r elements can be expressed by a polynomial relation:

$$(\mathbf{x}_j - c_1) \cdot \ldots \cdot (\mathbf{x}_j - c_r) = 0\,.$$

Formally, a *polynomial relation* over a vector space $\mathbb{F}^k$ is an equation $p = 0$ where p is a polynomial over the unknowns $\mathbf{X}$, i.e., $p \in \mathbb{F}[\mathbf{X}]$. The vector $y \in \mathbb{F}^k$ *satisfies* the polynomial relation $p = 0$, $y \models p$ for short, iff $p[y/\mathbf{x}] = 0$ where $[y/\mathbf{x}]$ denotes the substitution of the values y_i for the variables $\mathbf{x}_i$.

The set of polynomials $\mathbb{F}[\mathbf{X}]$ forms an $\mathbb{F}$-vector space. However, as the dimension of this vector space is infinite, we cannot effectively compute with bases. One way out is to restrict attention to polynomials of bounded degree. The *degree* of a polynomial p (or the polynomial relation $p = 0$) is the maximal sum $j_1 + \ldots + j_k$ of exponents of a monomial $a\mathbf{x}_1^{j_1} \ldots \mathbf{x}_k^{j_k}$ occurring in p. We denote the set of polynomials of degree at most d by $\mathbb{F}_{\leq d}[\mathbf{X}]$.

$\mathbb{F}_{\leq d}[\mathbf{X}]$ is an $\mathbb{F}$-vector space of dimension $\binom{k+d}{d} = O((k+d)^d)$: obviously, the monomials $\mathbf{x}_1^{j_1} \ldots \mathbf{x}_k^{j_k} \in \mathbb{F}_{\leq d}[\mathbf{X}]$ with coefficient 1 form a basis of $\mathbb{F}_{\leq d}[\mathbf{X}]$ and we prove momentarily by induction that there are $\binom{k+d}{d}$ such monomials. For $d = 0$ or $k = 0$ there is just the single monomial $\mathbf{x}_1^0 \ldots \mathbf{x}_k^0$ or 1, respectively, and indeed $\binom{k}{0} = \binom{d}{d} = 1$. So assume $d > 0$ or $k > 0$. By induction hypothesis there are $\binom{k+d-1}{d-1}$ monomials of degree less than d. The monomials with degree d over k variables are obtained from the $\binom{k-1+d}{d}$ monomials $\mathbf{x}_1^{j_1} \ldots \mathbf{x}_{k-1}^{j_{k-1}}$ of degree at most d over the first $k-1$ variables by multiplying with $x_k^{l_k}$ where $l_k = d - \sum_{i=1}^{k-1} l_i$. Altogether, there are thus $\binom{k+d-1}{d-1} + \binom{k-1+d}{d} = \binom{k+d}{d}$ monomials with coefficient 1 of degree at most d.

The polynomial relation $p = 0$ is valid after a single run r iff for all $x \in \mathbb{F}^k$, $p[[\![r]\!]x/\mathbf{x}] = 0$ or, equivalently, $p[(A_r x + b_r)/\mathbf{x}] = 0$ where A_r, b_r are defined as in Section 2. Thus, $p[(A_r\mathbf{x} + b_r)/\mathbf{x}] = 0$ is the *weakest precondition* for validity of $p = 0$ after run r. We observe:

LEMMA 6. *1. The polynomial $p[(A_r\mathbf{x} + b_r)/\mathbf{x}]$ is again of degree at most d.*

2. The mapping $W_r^{(d)}$ which maps polynomials p of degree at most d to $p[(A_r\mathbf{x} + b_r)/\mathbf{x}]$ is linear.

PROOF. For a proof of the first statement, it suffices to consider a run $r \equiv \mathbf{x}_i := t$, $t \equiv t_0 + \sum_{m=1}^{k} t_m\mathbf{x}_m$, of a single assignment and a single monomial $p \equiv \mathbf{x}_1^{j_1} \ldots \mathbf{x}_k^{j_k}$. Then

$$\begin{aligned} p[(A_r\mathbf{x} + b_r)/\mathbf{x}] &= p[t/\mathbf{x}_i] \\ &= \textstyle\sum_{\kappa_0 + \ldots + \kappa_k = j_i} \binom{j_i}{\kappa_0, \ldots, \kappa_k} \cdot t_0^{\kappa_0} \ldots t_k^{\kappa_k} \cdot \\ &\qquad \mathbf{x}_1^{j_1 + \kappa_1} \ldots \mathbf{x}_{i-1}^{j_{i-1} + \kappa_{i-1}} \mathbf{x}_i^{\kappa_i} \mathbf{x}_{i+1}^{j_{i+1} + \kappa_{i+1}} \ldots \mathbf{x}_k^{j_k + \kappa_k}, \end{aligned}$$

where the $\binom{j_i}{\kappa_0, \ldots, \kappa_k}$ are the multinomial coefficients for the j_i-th power of a sum on $k+1$ summands. Since in each monomial of the result, $\kappa_0 + \ldots + \kappa_k = j_i$, the degree of $p[t/\mathbf{x}_i]$ is bounded by $j_1 + \ldots + j_k$, i.e., the degree of p.

The second assertion follows since substitution commutes with sums and constant multiples. $\square$

The only polynomial relation which is true for *all program states* is the zero relation $0 = 0$. As for affine relations, we conclude that the polynomial relation $p = 0$ is valid after run r iff $W_r^{(d)} p = \mathbf{0}$ (where $\mathbf{0}$ denotes the zero polynomial). Summarizing, we have:

LEMMA 7. *The polynomial relation p of degree at most d is valid at program point u iff $W_r^{(d)} p = \mathbf{0}$ for all $r \in \mathbf{R}(u)$.*

Now we can proceed analogously to Section 3. By applying Lemma 2, we can safely replace the set $\{W_r^{(d)} \mid r \in \mathbf{R}(u)\}$ with its span. The resulting subspace of linear mappings can be described by a basis of at most $O((k+d)^{2d})$ matrices. The entries of these matrices are now indexed by pairs of tuples $J = (j_1, \ldots, j_k)$,

$\sum_{i=1}^{k} j_i \leq d$. Let $\mathcal{I}$ denote the set of all such tuples. We determine the set of *all* valid polynomial relations at program point u for polynomials of degree at most d as follows:

THEOREM 3. *Assume we are given a basis B for the set $\mathbf{Span}(\{W_r^{(d)} \mid r \in \mathbf{R}(u)\})$. Then we have:*

a) The polynomial relation $p = 0$ of degree at most d is valid at program point u iff $W\,p = \mathbf{0}$ for all $W \in B$.

b) A basis of the subspace of all polynomial relations of degree at most d valid at program point u can be computed in time $O((k+d)^{5d})$.

PROOF. Statement a) follows directly from Lemma 2 and Lemma 7.

For the proof of b), note that by a) the polynomial relation $p = 0$ is valid at u iff $p \equiv \sum_{J=(j_1,\ldots,j_k)\in\mathcal{I}} a_J\,\mathbf{x}_1^{j_1}\ldots\mathbf{x}_k^{j_k}$, where the $a_J, J \in \mathcal{I}$, are a solution of the equation:

$$\sum_{J\in\mathcal{I}} w_{IJ}\,\mathbf{a}_J = 0$$

for every matrix $W = (w_{IJ}) \in B$ and every $I \in \mathcal{I}$. The basis B may contain at most $O((k+d)^{2d})$ matrices each of which contributes $O((k+d)^d)$ equations. Thus, we have to compute the solution of an equation system with $O((k+d)^{3d})$ equations over $O((k+d)^d)$ variables. This can be done in time $O((k+d)^{5d})$. $\square$

By Theorem 3, it suffices to compute, for every program point u, the span of the set of all precondition transformers $W_r^{(d)}$, $r \in \mathbf{R}(u)$. We do so by abstracting the run sets to subspaces of linear transformations now of polynomials of degree at most d. The abstraction is thus given by:

$$\alpha^{(d)}(R) = \mathbf{Span}(\{W_r^{(d)} \mid r \in R\})\,.$$

As in the case of affine relations, we have:

$$\begin{array}{rcl} \alpha^{(d)}(\emptyset) & = & \mathbf{Span}(\emptyset) \;=\; \{\mathbf{0}\} \\ \alpha^{(d)}(\{r\}) & = & \mathbf{Span}(\{W_r^{(d)}\}) \end{array}$$

for a single run r. In particular,

$$\alpha^{(d)}(\{\varepsilon\}) \;=\; \mathbf{Span}(\{I_{\mathcal{I}}\})\,,$$

where $I_{\mathcal{I}}$ is the diagonal matrix describing the identity. The mapping $\alpha^{(d)}$ is again monotonic (w.r.t. subset ordering on sets of runs and subspaces) and commutes with arbitrary unions. Also, Lemma 4 analogously holds for $\alpha^{(d)}$. Therefore, the desired values can be computed by abstracting the constraint systems for same-level and reaching run sets. In order to obtain an effective algorithm, it remains to derive explicit abstractions for the effects of base edges.

For a definite assignment $\mathbf{x}_j := t$, this is obviously possible. It remains to consider a base edge $e \in \mathsf{Base}$ annotated by $\mathbf{x}_j :=?$ with $\mathbf{S}(e) = \{\mathbf{x}_j := c \mid c \in \mathbb{F}\}$.

In case $\mathbb{F}$ contains less than $d+1$ elements, the set $\mathbf{S}(e)$ is also finite and we simply may enumerate it. More interesting is the case when $\mathbb{F}$ has at least $d+1$ elements, e.g., because $\mathbb{F}$ has characteristic 0.

Each polynomial $p \in \mathbb{F}_{\leq d}[X]$ can be written as $p \equiv \sum_{i=0}^{d} p_i \cdot \mathbf{x}_j^i$ for polynomials p_i not containing $\mathbf{x}_j$. The coefficient polynomials p_i have at most degree d and are uniquely determined by p. For $0 \leq i \leq d$ let C_i be the mapping on $\mathbb{F}_{\leq d}[X]$ that maps each p to its i-th coefficient polynomial, i.e., $C_i(p) = p_i$. It is not hard to see that C_i is a linear map and hence can be represented by a matrix. We find that $\alpha^{(d)}(\mathbf{S}(e))$ can be finitely represented by $C_0,\ldots,C_d$:

LEMMA 8. *If $\mathbb{F}$ has more than d elements, then*

$$\begin{array}{rcl} \alpha^{(d)}(\mathbf{S}(e)) & = & \alpha^{(d)}(\{\mathbf{x}_j := c \mid c \in \mathbb{F}\}) \\ & = & \mathbf{Span}(\{C_l \mid l = 0,\ldots,d\})\,. \end{array}$$

PROOF. From the definitions we have $\alpha^{(d)}(\mathbf{S}(e)) = \alpha^{(d)}(\{\mathbf{x}_j := c \mid c \in \mathbb{F}\}) = \mathbf{Span}(\{W_{x_j:=c}^{(d)} \mid c \in \mathbb{F}\})$. It remains to show that this span equals $\mathbf{Span}(\{C_l \mid l = 0,\ldots,d\})$. For this we show:

1. $W_{x_j:=c}^{(d)} \in \mathbf{Span}(\{C_l \mid l = 0,\ldots,d\})$ for all $c \in \mathbb{F}$.

2. $C_l \in \mathbf{Span}(\{W_{x_j:=c}^{(d)} \mid c \in \mathbb{F}\})$ for $l = 0,\ldots,d$.

To 1: For arbitrary $p \in \mathbb{F}_{\leq d}[X]$ we have

$$W_{x_j:=c}^{(d)}(p) \;=\; p[c/\mathbf{x}_j] \;=\; \sum_{i=0}^{d} C_i(p)c^i\,.$$

This means $W_{x_j:=c}^{(d)} = \sum_{i=0}^{d} c^i C_i$, which implies 1.

To 2: Since the cardinality of $\mathbb{F}$ is at least $d+1$, we can find $d+1$ distinct elements $c_0,\ldots,c_d \in \mathbb{F}$. Defining matrix A by

$$A \;=\; \begin{pmatrix} 1 & c_0 & \ldots & c_0^d \\ 1 & c_1 & \ldots & c_1^d \\ \vdots & \vdots & \ddots & \vdots \\ 1 & c_d & \ldots & c_d^d \end{pmatrix}$$

it is not hard to see, that

$$A\begin{pmatrix} C_0(p) \\ \vdots \\ C_d(p) \end{pmatrix} = \begin{pmatrix} p[c_0/\mathbf{x}_j] \\ \vdots \\ p[c_d/\mathbf{x}_j] \end{pmatrix}$$

The determinant of A is an instance of what is known as Vandermonde's determinant and has the value $\prod_{0\leq i<l\leq d}(c_l - c_i)$. As all c_i are distinct, the determinant is different from 0. Therefore, matrix A is invertible and for the inverse matrix $A^{-1} = (b_{il})$, we have

$$\begin{pmatrix} C_0(p) \\ \vdots \\ C_d(p) \end{pmatrix} \;=\; A^{-1}\begin{pmatrix} p[c_0/\mathbf{x}_j] \\ \vdots \\ p[c_d/\mathbf{x}_j] \end{pmatrix}$$

Thus, $C_i(p) = \sum_{l=0}^{d} b_{il}\,p[c_l/\mathbf{x}_j] = \sum_{l=0}^{d} b_{il} W_{x_j:=c_l}^{(d)}(p)$. This shows $C_i = \sum_{l=0}^{d} b_{il} W_{x_j:=c_l}^{(d)}$, which implies 2. $\square$

Analogously to the last section, we construct constraint systems $\mathbf{S}_{\alpha^{(d)}}, \mathbf{R}_{\alpha^{(d)}}$ which are obtained from the constraint systems $\mathbf{S}$ and $\mathbf{R}$ by applying $\alpha^{(d)}$. We conclude:

THEOREM 4. *For every program of size n with k variables the following holds:*

a) The values:
$\mathbf{Span}(\{W_r^{(d)} \mid r \in \mathbf{S}(u)\})$, $u \in N$,
$\mathbf{Span}(\{W_r^{(d)} \mid r \in \mathbf{S}(p)\})$, $p \in \mathsf{Proc}$,

$\mathbf{Span}(\{W_r^{(d)} \mid r \in \mathbf{R}(p)\})$, $p \in \mathsf{Proc}$, and
$\mathbf{Span}(\{W_r^{(d)} \mid r \in \mathbf{R}(u)\})$, $u \in N$,
are the least solutions of the constraint systems $\mathbf{S}_{\alpha^{(d)}}$ and $\mathbf{R}_{\alpha^{(d)}}$, respectively.

b) *The sets of all valid polynomial relations of degree at most d at program point u, $u \in N$, can be computed in time $O(n \cdot (k+d)^{8d})$.*

Consider again the example program from the introduction. Since it uses three program variables, the vector-space of polynomials of degree at most 2 has dimension $\binom{3+2}{2} = 10$. Assume we have ordered the index tuples of monomials lexicographically as follows:

$$(0,0,0) \prec (0,0,1) \prec \ldots \prec (1,1,0) \prec (2,0,0).$$

Then the pre-condition transformer, e.g., of the assignment $\mathbf{x}_1 := \mathbf{x}_1 + \mathbf{x}_2 + 1$ is given by the matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

We refrain from describing the details of the fixpoint iteration. The least fixpoint computation for analyzing the valid quadratic relations at the entry of procedure p stabilizes after three iterations with three matrices. The rows of these matrices span a vector space of dimension 9 and have the (coefficients of) the relation $\mathbf{x}_2\mathbf{x}_3 - \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 = 0$ as their only non-trivial solution (up to constant multiples, of course). Again, this confirms our informal reasoning from the introduction.

5 Local Variables

So far we have considered programs which operate on global variables only. In this section, we explain how our techniques can be extended to work on procedures with global and local variables.

For notational convenience, we assume that all procedures have the same set $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ of variables where the first k are global and the remaining $m-k$ are local. For describing program executions, it now no longer suffices to consider execution *paths*. Instead, we have to take the proper nesting of calls into account. Therefore, *same-level runs s* and *reaching runs r* are now finite sequences of (unranked) trees b and, possibly, enter:

$$\begin{array}{lcll} st & ::= & \mathbf{x}_j := t \mid \mathsf{call}\langle s\rangle & \\ s & ::= & st_1; \ldots; st_n & (n \geq 0) \\ rt & ::= & \mathbf{x}_j := t \mid \mathsf{call}\langle s\rangle \mid \mathsf{enter} & \\ r & ::= & rt_1; \ldots; rt_n & (n \geq 0) \end{array}$$

Trees represent base actions or complete executions of procedures. Same-level runs represent sequences of such completed executions, while reaching runs may enter a procedure—without ever leaving it again.

The set of runs *reaching program point* $u \in N$ can again be characterized as the least solution of a system of subset constraints on run sets. If e is annotated by an affine assignment, i.e., $A(e) \equiv \mathbf{x}_j := t$, we again define: $\mathbf{S}'(e) = \{\mathbf{x}_j := t\}$. Similarly for $A(e) \equiv \mathbf{x}_j :=?$,

$$\mathbf{S}'(e) = \{\mathbf{x}_j := c \mid c \in \mathbb{F}\}.$$

The same-level runs of procedures and program nodes are the smallest solution of the following constraint system $\mathbf{S}'$:

$$\begin{array}{lll} [\mathrm{S}'1] & \mathbf{S}'(q) \supseteq \mathbf{S}'(r_q) & \\ [\mathrm{S}'2] & \mathbf{S}'(e_q) \supseteq \{\varepsilon\} & \\ [\mathrm{S}'3] & \mathbf{S}'(v) \supseteq \mathbf{S}'(u); \mathbf{S}'(e) & \text{if } e = (u,v) \in \mathsf{Base} \\ [\mathrm{S}'4] & \mathbf{S}'(v) \supseteq \mathbf{S}'(u); \mathsf{call}\langle \mathbf{S}'(p)\rangle & \text{if } e = (u,v) \in \mathsf{Call}_p \end{array}$$

Note that, for convenience, the application of the constructor call to all sequences of a set S is denoted by $\mathsf{call}\langle S\rangle$. Constraints [S′1], [S′2] and [S′3] are as in Section 2. The new constraint [S′4] deals with calls. If the ingoing edge $e = (u,v)$ is a call to a procedure p, we concatenate a same-level run reaching u with a tree constructed from a same-level run of p by applying the constructor call.

For characterizing the runs that reach program points and procedures, we construct the constraint system $\mathbf{R}'$:

$$\begin{array}{llll} [\mathrm{R}'1] & \mathbf{R}'(\mathbf{Main}) \supseteq \{\varepsilon\} & & \\ [\mathrm{R}'2] & \mathbf{R}'(p) & \supseteq \mathbf{R}'(u), & \text{if } (u,\_) \in \mathsf{Call}_p \\ [\mathrm{R}'3] & \mathbf{R}'(u) & \supseteq \mathbf{R}'(p); \{\mathsf{enter}\}; \mathbf{S}'(u), & \text{if } u \in N_p \end{array}$$

Constraints [R′1] and [R′2] are as in Section 3. The only modification occurs in [R′3] where an enter is inserted between the run reaching the current procedure p and the same-level run inside p.

Each of these runs gives rise to a transformation of the underlying program state $x \in \mathbb{F}^m$. Here, we just explain how the transformations of enter and $\mathsf{call}\langle s\rangle$ are obtained. The transformation $[\![\mathsf{enter}]\!]$ passes the values of the globals $\mathbf{x}_j$ ($j = 1, \ldots, k$) and sets the locals $\mathbf{x}_j$, $j > k$, to 0.[2] Thus,

$$[\![\mathsf{enter}]\!] = [\![\mathbf{x}_{k+1} := 0; \ldots; \mathbf{x}_m := 0]\!] = \left(\begin{array}{c|c} I_k & 0 \\ \hline 0 & 0 \end{array}\right).$$

Let us denote this $m \times m$ matrix by E'.

The transformation $[\![\mathsf{call}\langle s\rangle]\!]$ is more complicated. Like $[\![\mathsf{enter}]\!]$, it must pass the values of the globals into the execution of the called procedure and initialize its local variables. In addition, it must return the values of the globals to the calling context and restore the values of the local variables. Given that $[\![s]\!]\,x = A_s x + b_s$ as in Section 2, we define:

$$\begin{array}{rcl} [\![\mathsf{call}\langle s\rangle]\!]\,x & = & E'([\![s]\!](E'x)) + T'x \\ & = & (E'A_sE' + T')x + E'b_s, \end{array}$$

where T' is the $m \times m$ matrix $\left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & I_{m-k} \end{array}\right)$. The outermost application of E' in the first summand prohibits propagation of the called procedure's local variables and the second summand, $T'x$ bypasses the values of the local variables of the calling context. The above calculation shows that $[\![\mathsf{call}\langle s\rangle]\!]$ is an affine transformation as well.

[2]By convention, local variables are initialized by 0. Other conventions could easily be modeled as well. Uninitialized local variables as in C, for instance, can be handled by adding $x_j :=?$ statements for $j = k+1, \ldots, m$ at the beginning of each procedure body.

We want to determine for every (reaching or same-level) run the transformation which produces the weakest precondition. For simplicity, we construct the weakest precondition transformer only for affine relations. The weakest precondition transformer for enter is given by:

$$W_{\mathsf{enter}} = W_{\mathbf{x}_{k+1}:=0;\ldots;\mathbf{x}_m:=0} = \left(\begin{array}{c|c} I_{k+1} & 0 \\ \hline 0 & 0 \end{array}\right).$$

Let E denote this matrix. To obtain analogous results as in Section 3, we determine the weakest precondition transformation of $\mathsf{call}\langle s\rangle$. We define an operator $\Box : \mathbb{F}^{(m+1)\times(k+1)} \to \mathbb{F}^{(m+1)\times(k+1)}$ on $(m+1)\times(m+1)$ matrices by:

$$\Box(W) = EWE + w\cdot T$$

where w is the element in the left upper corner of W, and T is the $(m+1)\times(m+1)$ matrix $\left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & I_{m-k} \end{array}\right)$.

The operator $\Box$ returns a linear transformation and is itself linear. This implies that $\Box$ maps subspaces of $\mathbb{F}^{(k+1)\times(k+1)}$ to subspaces of $\mathbb{F}^{(k+1)\times(k+1)}$ and, considered as a mapping on subspaces, commutes with arbitrary least upper bounds. Moreover, we have:

LEMMA 9. *Let W_s denote the precondition transformer for s. Then for an affine relation $a \in \mathbb{F}^m$ and a program state $x \in \mathbb{F}^m$, $[\![\mathsf{call}\langle s\rangle]\!]x \models a$ iff $x \models \Box(W_s)\,a$.*

Thus, $W_{\mathsf{call}\langle s\rangle} = \Box(W_s)$ is the weakest precondition transformer for $\mathsf{call}\langle s\rangle$. In order to furnish the same approach as for global variables, we define the abstraction function α for sets R of (same-level or reaching) runs by:

$$\alpha(R) = \mathbf{Span}(\{W_r \mid r \in R\})$$

In particular, $\alpha(\{\mathsf{enter}\}) = \mathbf{Span}(\{E\})$.

Analogously to Lemma 4, we find:

LEMMA 10. *For every set S of same-level runs,*

$$\alpha(\mathsf{call}\langle S\rangle) = \alpha(\{\mathsf{call}\langle s\rangle \mid s \in S\}) = \Box(\alpha(S)).$$

Finally, we construct constraint systems $\mathbf{S}'_\alpha$ and $\mathbf{R}'_\alpha$ from $\mathbf{S}'$ and $\mathbf{R}'$ by applying α where concatenation is replaced with "$\circ$" and the constructor call is replaced with "$\Box$". Then we obtain our main theorem for programs with local variables:

THEOREM 5. *For a program of size n with m global and local variables the following holds:*

a) The values:
$\mathbf{Span}(\{W_s \mid s \in \mathbf{S}'(u)\})$, $u \in N$,
$\mathbf{Span}(\{W_s \mid s \in \mathbf{S}'(p)\})$, $p \in \mathsf{Proc}$,
$\mathbf{Span}(\{W_s \mid s \in \mathbf{R}'(p)\})$, $p \in \mathsf{Proc}$, and
$\mathbf{Span}(\{W_r \mid r \in \mathbf{R}'(u)\})$, $u \in N$,
are the least solutions of the constraint systems $\mathbf{S}'_\alpha$ and $\mathbf{R}'_\alpha$, respectively.

b) The sets of all valid affine relations at program point u, $u \in N$, can be computed in time $O(n\cdot m^8)$.

Our technique can be adapted to procedures with parameters. Value parameters, for instance, can be simulated via a *scratch pad* of globals through which the actual parameters are communicated from the caller to the callee. Return values can be treated similarly.

6 Affine Preconditions

The analyses considered so far assume that we have no knowledge whatsoever about the initial state in which the program is started. However, in a verification context we are often in a more lucky situation when we are given a precondition that constrains potential initial states. Of course, if less initial states are possible more relations may be valid at the nodes of a program and an analyses that ignores the precondition may be overly pessimistic. In this section we extend the analyses of Section 3 and Section 4 to take into account *affine preconditions* completely. The analyses of this section thus compute for each program point of an affine program the space of all those affine or polynomial relations that are valid whenever the program is started in a state satisfying a given affine precondition.

Assume given a finite set $Pre \subseteq F^{k+1}$ of affine relations, representing the affine precondition. We say that Pre is *satisfiable* if there is an $x \in \mathbb{F}^k$ such that $x \models h$ for all $h \in Pre$. If Pre is not satisfiable, all relations are valid at all program points under precondition Pre. As we can check whether Pre is satisfiable or not with the aid of Gaussian elimination, we can detect this trivial case. Thus, we assume without loss of generality that Pre is satisfiable in the following. In this case, the set of states satisfying Pre, $\mathbf{Sat}(Pre) = \{x \in \mathbb{F}^k \mid x \models h, h \in Pre\}$, is an affine subspace of $\mathbb{F}^k$ and can be represented in the form $\mathbf{Sat}(Pre) = x_0 + L$, where $x_0 \in \mathbf{Sat}(Pre)$ and L is a (linear) subspace of $\mathbb{F}^k$. Assume that $x_1,\ldots,x_l$ with $l \leq k$ is a basis of L. Then we have

$$\mathbf{Sat}(Pre) = \{x_0 + \sum_{r=1}^{l} \lambda_r x_r \mid \lambda_1,\ldots,\lambda_l \in \mathbb{F}\}. \qquad (4)$$

Vectors $x_0,\ldots,x_l \in \mathbb{F}^k$ with this property can be computed from Pre with standard techniques from linear algebra.

Obviously, an affine relation a is valid at a program point u under precondition Pre, iff its weakest precondition for each program path r reaching u is valid for all $x \in \mathbf{Sat}(Pre)$, i.e., if $x \models W_r\,a$ for all $r \in \mathbf{R}(u)$, $x \in \mathbf{Sat}(Pre)$. By the characterization of $\mathbf{Sat}(Pre)$ in Equation 4, we thus have:

LEMMA 11. *The affine relation $a \in \mathbb{F}^{k+1}$ is valid at program point u under precondition Pre iff $x_0 + \sum_{r=1}^{l} \lambda_r x_r \models W_r\,a$ for all $\lambda_1,\ldots,\lambda_l \in \mathbb{F}$, $r \in \mathbf{R}(u)$.*

By arguing analogously to Section 3 we can equivalently require this property for all matrices W in a basis of $\mathbf{Span}\{W_r \mid r \in \mathbf{R}(u)\}$. Thus, we obtain the following generalization of Theorem 1:

THEOREM 6. *Assume we are given a basis B for the set $\mathbf{Span}(\{W_r \mid r \in \mathbf{R}(u)\})$. Then we have:*

a) Affine relation $a \in \mathbb{F}^{k+1}$ is valid at program point u under precondition Pre iff $x_0 + \sum_{r=1}^{l} \lambda_r x_r \models Wa$ for all $\lambda_1,\ldots,\lambda_l \in \mathbb{F}$, $W \in B$.

b) A basis for the subspace of all affine relations valid at program point u under precondition Pre can be computed in time $O(k^5)$.

PROOF. As a) has already been justified we prove only b). By a) an affine relation a is valid at u if and only if for all $W \in B$:

$$x_0 + \sum_{r=1}^{l} \lambda_r x_r \models Wa \qquad \text{for all } \lambda_1,\ldots,\lambda_l \in \mathbb{F}. \qquad (5)$$

By unfolding the definition of "$\models$" and writing $W = (w_{ij})$ and $x_i = (x_{i1}, \ldots, x_{ik})^t$ for $i = 0, \ldots, l$, Formula (5) means that

$$\sum_{j=0}^{k} a_j (w_{0j} + \sum_{i=1}^{k} x_{0i} w_{ij}) + \sum_{r=1}^{l} \lambda_r \sum_{j=0}^{k} a_j \sum_{i=1}^{k} x_{ri} w_{ij} = 0$$

for all $\lambda_1, \ldots, \lambda_l \in \mathbb{F}$. This is an affine equation in the λ_r whose coefficients are affine combinations of the a_j. It is valid for all λ_r if and only if all these combinations are 0. Therefore, an affine relation a is valid at u under precondition *Pre* if and only if it satisfies the equations:

$$\sum_{j=0}^{k} \mathbf{a}_j (w_{0j} + \sum_{i=1}^{k} x_{0i} w_{ij}) = 0$$

and

$$\sum_{j=0}^{k} \mathbf{a}_j \sum_{i=1}^{k} x_{ri} w_{ij} = 0 \quad \text{for } r = 1, \ldots, l$$

for all $W = (w_{ij}) \in B$. We can hence compute the subspace of all valid affine relations by setting up and solving the linear equation system consisting of all these equations.

Let us estimate the complexity of this procedure. Each matrix $W \in B$ contributes $l + 1 = O(k)$ equations and there are at most $O(k^2)$ matrices in B. Hence the equation system has $O(k^3)$ equations. It is not hard to see, that the coefficients of each equation can be computed in time $O(k^2)$. Hence the equation system can be set up in time $O(k^5)$. As a linear equation system with $O(k^3)$ equations in the $k+1$ variables $\mathbf{a}_0, \ldots, \mathbf{a}_k$ it can be solved, e.g., by Gaussian elimination in time $O(k^5)$. $\square$

From Theorem 2 we know that we can compute a basis of $\mathbf{Span}(\{W_r \mid r \in \mathbf{R}(u)\})$ in time $O(n \cdot k^8)$. Together with Theorem 6 this implies:

COROLLARY 1. *The sets of all valid affine relations at program point u, $u \in N$, under precondition Pre can be computed in time $O(n \cdot k^8)$.*

This approach for treating affine preconditions can straightforwardly be generalized to the setting of Section 4. Here a polynomial relation of degree at most d turns out to be valid at a program point u under precondition *Pre* if and only if for all W in a basis of $\mathbf{Span}(\{W_r^{(d)} \mid r \in \mathbf{R}(u)\})$:

$$x_0 + \sum_{r=1}^{l} \lambda_r x_r \models W p \qquad \text{for all } \lambda_1, \ldots, \lambda_l \in \mathbb{F}. \tag{6}$$

This time, this translates to a *polynomial* equation (of degree at most d) in the λ_r whose coefficients are affine combinations of the coefficients of p. Again all these affine combinations must equal 0 which gives rise to a linear equation system that we can set up and solve.

COROLLARY 2. *The sets of all valid polynomial relations of degree at most d at program point u, $u \in N$, under precondition Pre can be computed in time $O(n \cdot (k+d)^{8d})$.*

7 Conclusion

We have presented an interprocedural analysis that determines for each program point of an affine program the set of all valid affine relations. We generalized the algorithm to infer all *polynomial* relations of bounded degree and showed that our methods work also in presence of local variables and parameter passing by value and result. We also generalized our analyses to take affine preconditions into account.

All our analyses run in polynomial time. More precisely, they are linear in the program size and polynomial of a higher degree in the number of variables. It remains for future work to find out whether this theoretical complexity bound is prohibitive to apply the analysis in practice or in how far heuristic methods are necessary to identify promising but sufficiently small sets of variables to be included in the analysis.

Instrumental for our approach is that we can capture the effect of procedures as weakest precondition transformers for affine and polynomial relations completely by subspaces of linear maps. This provides us with a kind of abstract "higher-order denotation" of procedures that we can compute in polynomial time and use at any call site. Similar in spirit is "relational analysis" of recursive procedures as proposed by Cousot [5, 6] and the "functional approach" to interprocedural analysis [20, 11]. While these approaches rely on relations or functions, we capture the effects of procedures by finitely representable *sets* of functions.

Our results improve on the analysis of linear constants by Horwitz et al. [9, 17] and, upto the treatment of positive affine guards, also on the results obtained by Karr [10]. In a recent paper, Reps, Schwoon, and Jha [18] use a library for reachability analysis of weighted pushdown systems for interprocedural dataflow analysis. They report that G. Balakrishnan has created a prototype implementation of our interprocedural analysis for affine relations based on a preliminary version of the current paper.

The results of this paper are still not strong enough to deal with positive affine guards as Karr's approach. Also, they do not generalize our *intraprocedural* analysis in [15, 14] where we succeed in checking the validity of arbitrary polynomial relations for *polynomial* programs—even in presence of negative polynomial guards. It remains as challenging open problems whether or not precise interprocedural treatments of positive guards or precise interprocedural analysis of polynomial programs are possible.

Acknowledgments

We would like to thank the anonymous POPL reviewers for their remarks that helped to improve our original submission. The first author would also like to thank the members of IFIP Working Group 2.2 for their constructive comments on a talk about a preliminary version of this paper.

8 References

[1] K. R. Apt and G. D. Plotkin. Countable Nondeterminism and Random Assignment. *Journal of the ACM*, 33(4):724–767, 1986.

[2] R. Bagnara, P. Hill, E. Ricci, and E. Zaffanella. Precise Widening Operators for Convex Polyhedra. In *10th Int. Static Analysis Symposium (SAS)*, pages 337–354. LNCS 2694, Springer-Verlag, 2003.

[3] I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Programming (JLP)*, 4(3):259–262, 1987.

[4] P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Elec-*

tronic Notes in Theoretical Computer Science, 6, 1997. URL: `www.elsevier.nl/locate/entcs/volume6.html`.

[5] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Recursive Procedures. In E. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1977.

[6] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *5th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 84–97, 1978.

[7] C. Fecht and H. Seidl. Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems. *Nordic Journal of Computing (NJC)*, 5(4):304–329, 1998.

[8] S. Gulwani and G. Necula. Discovering Affine Equalities Using Random Interpretation. In *30th Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 74–84, 2003.

[9] S. Horwitz, T. Reps, and M. Sagiv. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science (TCS)*, 167(1&2):131–170, 1996.

[10] M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[11] J. Knoop and B. Steffen. The Interprocedural Coincidence Theorem. In *Compiler Construction (CC)*, pages 125–140. LNCS 541, Springer-Verlag, 1992.

[12] S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice Hall, Engelwood Cliffs, New Jersey, 1981.

[13] M. Müller-Olm and O. Rüthing. The Complexity of Constant Propagation. In *10th European Symposium on Programming (ESOP)*, pages 190–205. LNCS 2028, Springer-Verlag, 2001.

[14] M. Müller-Olm and H. Seidl. Computing Polynomial Program Invariants. Submitted for publication.

[15] M. Müller-Olm and H. Seidl. Polynomial Constants are Decidable. In *9th Static Analysis Symposium (SAS)*, pages 4–19. LNCS 2477, Springer-Verlag, 2002.

[16] B. Paige and S. Koenig. Finite Differencing of Computable Expressions. *ACM Trans. Prog. Lang. and Syst.*, 4(3):402–454, 1982.

[17] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 49–61. ACM Press, 1995.

[18] T. Reps, S. Schwoon, and S. Jha. Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis. In *Int. Static Analysis Symposium (SAS)*, pages 189–213. LNCS 2694, Springer-Verlag, 2003.

[19] H. Seidl and B. Steffen. Constraint-Based Inter-Procedural Analysis of Parallel Programs. *Nordic Journal of Computing (NJC)*, 7(4):375–400, 2000.

[20] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In [12], chapter 7, pages 189–233.