Computing Interprocedurally Valid Relations in Affine Programs

Markus Müller-Olm
Universität Dortmund, FB 4, LS V
44221 Dortmund, Germany
e-mail: mmo@ls5.cs.uni-dortmund.de

Helmut Seidl
Universität Trier, FB 4 — Informatik
54286 Trier, Germany
e-mail: seidl@uni-trier.de

Abstract

We consider an abstraction of programs which preserves affine assignments exactly while conservatively dealing with other assignments and ignoring conditions at branches. We present an interprocedural analysis of such abstracted programs which for every program point v determines the set of all affine relations *between program variables which are valid when reaching v. The runtime of this algorithm is linear in the program size and polynomial in the number of occurring variables. We extend this result to a polynomial-time algorithm which determines for every program point the set of all valid* polynomial relations *between program variables of bounded degree.*

1. Introduction

Due to fundamental undecidability reasons, it is impossible to design an algorithm which, for every program determines whether or not a variable has a specific value whenever the program reaches a program point. This negative result, though, does not preclude techniques which obtain useful information about programs in many practical cases. A key idea is to abstract the program to be analyzed to a program which has a simpler semantics and thus is amenable to an algorithmic solution. In our case, this abstraction consists in replacing conditional branching with non-deterministic choice, i.e., by ignoring conditions at branches. Moreover, while preserving certain "tractable" assignments $\mathbf{x}_j := t$, all others are replaced with nondeterministically assigning *any value*. One instance of this scheme has been considered in copy-constant propagation [5]. Here, the right-hand sides of the preserved assignments either consist of single constants or variables. Clearly, the necessary abstraction is severe and thus the quality of the resulting information quite limited. Therefore, various generalizations have been considered. In 1976, Karr proposes *affine* programs where arbitrary affine assignments such as $\mathbf{x}_3 := \mathbf{x}_1 - 3\mathbf{x}_2 + 7$ are treated exactly [7]. In generalization of these, Müller-Olm and Seidl consider *polynomial programs* where assignments with polynomial right-hand sides such as in $\mathbf{x}_3 := \mathbf{x}_1\mathbf{x}_2 - 2\mathbf{x}_3 + 3$ are allowed [15].

Given an abstracted program, the goal is to compute *precise information* about the values of variables which means that the analysis should not introduce any further loss of precision. Various types of information about the values of variables are of interest. In the simplest case, we aim at detecting variables which definitely have a fixed value (say, 42) when reaching a program point. Related analyses are known as *constant propagation*. Or, we want to determine definite equalities between variables. In his seminal paper [7], Karr proposes, as a generalization of both questions, to search for arbitrary *affine relations* among variables. Such detailed information has many applications. It can be used, e.g., for triggering optimizing program transformations such as aggressive common sub-expression elimination. It is also detailed enough to infer or test program invariants in the context of program verification.

Karr presents an algorithm which determines for every program point the set of *all* valid affine relations. His algorithm is forward-propagating and uses a quite complicated subroutine to deal with assignments $\mathbf{x}_j := t$ where the right-hand side depends on the variable $\mathbf{x}_j$ from the left-hand side. In [13], Müller-Olm et al. observe that *checking* a given affine relation for validity at a program point can be performed by a much simpler *backward propagating* algorithm which in turn is generalized to a backward propagating algorithm for checking arbitrary polynomial relations for polynomial programs in [15].

All these algorithms have been designed for *intraprocedural* analysis. Clearly, for copy constant propagation, the intraprocedural approach can be generalized to programs with procedures since the involved domain is finite [17]. Horwitz et al. also present a precise interprocedural constant propagation algorithm running in polynomial time if only assignments are treated exactly whose right-hand sides contain at most *one occurrence* of a variable [6].

Here, we present an algorithm which computes the set of all valid affine relations for arbitrary affine programs with

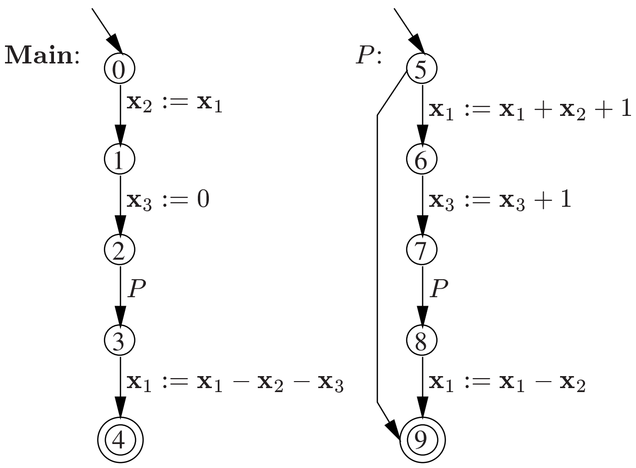

Figure 1. An example program.

procedures. Our algorithm is more general than Karr's as we allow procedures, and it is more general than those by Horwitz et al. since we consider arbitrary affine programs and infer more detailed information. Also, we extend our algorithm to compute for every program point of an affine program the set of all valid *polynomial relations* of degree bounded by some fixed d. The base algorithm as well as the extended algorithm runs in time linear in the program size and polynomial in the number of program variables.

The key observation onto which our algorithm is based is that the weakest precondition of an affine relation a along a single run of the 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. We succeed in describing the effect of procedure calls by means of a finite-dimensional sub-space of linear transformations. The affine relation a at a program point u then turns out to be valid iff it is transformed into $\mathbf{0}$ by the vector space generated by the set of all linear transformations corresponding to reaching program runs. We conclude that the set of all valid affine relations can be computed as the set of solutions of an appropriate linear equation system.

The program in figure 1 illustrates the kind of properties our analyses can handle. It consists of two procedures $\mathbf{Main}$ 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, $\mathbf{Main}$ 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 $\mathbf{Main}$ (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 $\mathbf{Main}$ 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 $\mathbf{x}_2 + 1$. 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.

Our 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.

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 $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)^{\mathrm{t}}$. We assume that the variables take values in a fixed field $\mathbb{F}$. In practice, $\mathbb{F}$ is the field of rational or real numbers. Then a *state* assigning values to the variables is conveniently modeled by a k-dimensional (column) vector $x = (x_1, \ldots, x_k)^{\mathrm{t}} \in \mathbb{F}^k$; 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 $(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_k)^{\mathrm{t}}$.

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.

A *program* comprises a finite set Proc of *procedure names* that contains a distinguished procedure $\mathbf{Main}$. Execution starts with a call to $\mathbf{Main}$. Each procedure name

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

$p \in \mathsf{Proc}$ 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 $A_p : E_p \rightarrow \mathsf{Stmt} \cup \mathsf{Proc}$ 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., [18] 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* refinements. 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 to 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 $\mathbf{S}$:

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

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 p comprises all same-level runs reaching the return point of p. 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 $\mathbf{R}$:

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

By [R1], the procedure $\mathbf{Main}$ u 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 from 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, however, 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 $[\![\mathbf{x}_j := t]\!] : \mathbb{F}^k \rightarrow \mathbb{F}^k$ 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:

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

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 $a_0 + a_1\mathbf{x}_1 + \ldots a_k\mathbf{x}_k = 0$ 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 $a = (a_0, \ldots, a_k)^{\mathrm{t}}$. 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 $a' = (a_1, \ldots, a_k)^{\mathrm{t}}$ 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 a.

The affine relation a is *valid* after a single run r 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

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

From this characterization we see, that the weakest precondition is again an affine relation. Even better: The mapping that assigns 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:

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

This matrix 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 $\mathbf{0} = (0, \ldots, 0)^{\mathrm{t}}$. Thus, the affine relation a is valid after run r iff $W_r\, a = \mathbf{0}$.

The states that potentially occur at a program point $u \in N$ are the states $[\![r]\!]x$ with $r \in \mathbf{R}(u)$ and $x \in \mathbb{F}^k$. Consequently, the affine relation a is valid at a program point u, iff for all $x \in \mathbb{F}^k$ and $r \in \mathbf{R}(u)$, $[\![r]\!]x \models a$ or, equivalently, $x \models W_r\, a$. Summarizing, we have:

Lemma 1 *Let $\mathcal{W} = \{W_r \mid r \in \mathbf{R}(u)\}$. Then the affine relation $a \in \mathbb{F}^{k+1}$ is valid at program point u iff $W a = \mathbf{0}$ for all $W \in \mathcal{W}$.*

Thus, the set $\mathcal{W}$ gives us a handle to solve the interprocedural validity problem of affine relations. The problem is that we do not know how to represent this set $\mathcal{W}$ 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.*

- *For every $W \in M$, the set $\{a \mid Wa = \mathbf{0}\}$ forms a sub-space of $\mathbb{F}^n$;*
- *As an intersection of vector spaces, the set $\{a \mid \forall W \in M: \; Wa = \mathbf{0}\}$ forms a sub-space of $\mathbb{F}^n$;*
- *For every $a \in \mathbb{F}^n$, the following two statements are equivalent:*
  - *$Wa = \mathbf{0}$ for all $W \in M$;*
  - *$Wa = \mathbf{0}$ for all $W \in \mathbf{Span}(M)$.*

Here, $\mathbf{Span}(M)$ 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 $\mathbf{Span}(\mathcal{W})$, i.e., the subspace of $\mathbb{F}^{(k+1)\times(k+1)}$ generated by $\mathcal{W}$ without loosing interesting information. As a sub-space of the vector space $\mathbb{F}^{(k+1)\times(k+1)}$ of dimension $(k+1)^2$, $\mathbf{Span}(\mathcal{W})$ 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—$\mathbf{Span}(\mathcal{W})$ can have at most dimension $k^2 + k + 1$. Based on this observation, we determine the set of *all* valid affine relations at program point u as follows:

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:*

1. *$a \in \mathbb{F}^{k+1}$ is valid at program point u iff $Wa = \mathbf{0}$ for all $W \in B$;*
2. *A basis for the sub-space of all affine relations valid at program point u can be computed in time $\mathcal{O}(k^5)$.*

By statement 1 of Theorem 1, the affine relation a is valid at u iff a is a solution of the equation system:

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

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

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

So, we are left with the task to compute, for every program point u, $\mathbf{Span}(\{W_r \mid r \in \mathbf{R}(u)\})$. Our goal therefore is to abstract sets of program executions to subspaces of $\mathbb{F}^{(k+1)\times(k+1)}$. 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. In particular, the least upper bound of two spaces V_1, V_2 is given by:

$$\begin{array}{rcl} V_1 \sqcup V_2 & = & \mathbf{Span}(V_1 \cup V_2) \\ & = & \{c_1 v_1 + c_2 v_2 \mid c_i \in \mathbb{F}, v_i \in V_i\} \end{array}$$

The height of this complete lattice, 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:

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

for a single run r. In particular by equation (2),

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

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.) Moreover, it commutes with arbitrary unions. It remains to show that the desired values can be computed by abstracting the constraint systems for same-level and reaching run sets. In particular, we need an abstract version of the concatenation of run sets. For $M_1, M_2 \subseteq \mathbb{F}^{(k+1)^2}$, we define:

$$M_1 \circ M_2 = \mathbf{Span}(\{A_1 A_2 \mid A_i \in M_i\})$$

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

$$B_1 B_2 = \sum_m \sum_j \lambda_m^{(1)} \lambda_j^{(2)} \cdot A_m^{(1)} A_j^{(2)}$$

by linearity of matrix multiplication. Since each $A_m^{(1)} A_j^{(2)}$ is contained in $M_1 \circ M_2$, $B_1 B_2$ is contained in $M_1 \circ M_2$ as well. Therefore, also the inclusion "$\subseteq$" follows. $\square$

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 $R_1, R_2 \subseteq \mathsf{Runs}$. Then*

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

Proof: Consider the auxiliary map $\mathcal{W}$ mapping run sets to sets of matrices by:

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

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

$$\{A_1 A_2 \mid A_i \in \mathcal{W}(R_i)\} = \mathcal{W}(R_1\,;\,R_2)$$

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

For the abstraction of base edges, we distinguish two cases. 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 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. 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

$$\begin{aligned}\alpha(\mathbf{S}(e)) &= \alpha(\{\mathbf{x}_j := c \mid c \in \mathbb{F}\}) \\ &= \alpha(\{\mathbf{x}_j := 0\ ,\ \mathbf{x}_j := 1\}) \\ &= \mathbf{Span}(\{T_0, T_1\})\end{aligned}$$

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: We verify: $T_c = (1-c)\cdot T_0 + c \cdot T_1$. Hence, $T_c \in \mathbf{Span}(\{T_0, T_1\})$. □

From the constraint systems $\mathbf{S}$ and $\mathbf{R}$ for run sets, we construct constraint systems $\mathbf{S}_\alpha$ and $\mathbf{R}_\alpha$ by application of α. The variables in the new constraint systems take sub-spaces of $(k+1)\times(k+1)$ matrices as values. Then we apply α to the occurring constant sets $\{\epsilon\}$ and $\mathbf{S}(e)$ and replace the concatenation operator ";" with "$\circ$". For the resulting constraint systems, we obtain our main theorem:

Theorem 2 *For every program of size p with k variables the following holds:*

1. *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.
2. *The sets of all valid affine relations at program point u, $u \in N$, can be computed in time $\mathcal{O}(p \cdot k^8)$.*

Proof: Since the mapping α commutes with arbitrary unions, assertion (1) follows from lemma 4 by the fundamental results in [9, 10]. We already have seen in theorem 1 that, given the sets of precondition transformers for all program points u, we can compute the sets of all valid affine relations within the stated complexity bounds. So, it remains to prove that the least solution of the abstracted constraint systems can be computed in time $\mathcal{O}(p \cdot k^8)$. For that, recall that the lattice of all sub-spaces of $\mathbb{F}^{(k+1)^2}$ has height $(k+1)^2$. Thus, a worklist based fixpoint algorithm will evaluate at most $\mathcal{O}(p \cdot k^2)$ 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 $\mathcal{O}(k^7)$. 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 $\mathcal{O}(k^8)$. Altogether, we obtain an upper complexity bound of $\mathcal{O}(p \cdot k^2 \cdot k^8) = \mathcal{O}(p \cdot k^{10})$. A better runtime can be obtained if we use a semi-naive fixpoint iteration strategy [16, 1, 4]. 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 $\mathcal{O}(k^8)$ which overall results in the desired runtime $\mathcal{O}(p \cdot k^8)$. □

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 $\mathbf{Main}$, reaching runs and same-level runs coincide for the program points of $\mathbf{Main}$. 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 $a = (a_0, \ldots, a_k)^{\mathrm{t}}$ 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 to $a_0 = a_1 = a_2 = a_3 = 0$. 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. For example, the relation:

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

represents the *disjunction* of 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 *degree* of the 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. The set of all polynomials in $\mathbb{F}[\mathbf{X}]$ of degree at most d forms an $\mathbb{F}$-vector space of dimension $\binom{k+d}{d} = \mathcal{O}((k+d)^d)$. The vector $y \in \mathbb{F}^k$ *satisfies* the polynomial relation $p = 0$ iff $p[y/\mathbf{x}] = 0$ where $[y/\mathbf{x}]$ denotes the substitution of the y_i into the variables $\mathbf{x}_i$.

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 $p[t/\mathbf{x}_j]$ 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. □

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 *Let $\mathcal{W}^{(d)} = \{W_r^{(d)} \mid r \in \mathbf{R}(u)\}$. Then the polynomial relation p of degree at most d is valid at program point u iff $Wp = \mathbf{0}$ for all $W \in \mathcal{W}^{(d)}$.*

Now it is clear how we proceed. By applying lemma 2, we can safely replace the set $\mathcal{W}^{(d)}$ with its span. The resulting subspace of linear mappings can be described by a basis of at most $\mathcal{O}((k+d)^{2d})$ matrices. Note that the entries of these matrices are now indexed by 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:*

1. *the polynomial relation $p = 0$ of degree at most d is valid at program point u iff $Wp = \mathbf{0}$ for all $W \in B$;*
2. *A basis of the subspace of all polynomial relations of degree at most d valid at program point u can be computed in time $\mathcal{O}((k+d)^{5d})$.*

By statement 1 of Theorem 3, 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 $\mathcal{O}((k+d)^{2d})$ matrices each of which contributes $\mathcal{O}((k+d)^d)$ equations. Thus, we have to compute the solution of an equation system with $\mathcal{O}((k+d)^{3d})$ equations over $\mathcal{O}((k+d)^d)$ variables — giving the desired complexity bounds.

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,

$$\alpha^{(d)}(\{\epsilon\}) \quad = \quad \mathbf{Span}(\{I_{\mathcal{I}}\})$$

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 $\alpha^{(d)}$ as well. 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 $\mathbf{S}(e) = \{\mathbf{x}_j := c \mid c \in \mathbb{F}\}$. Here it turns out that the abstraction can be finitely represented by picking $d+1$ different values from $\mathbb{F}$, e.g., 0 through d. We find:

Lemma 8

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

Proof: We verify that there exist $\lambda_0, \ldots, \lambda_d \in \mathbb{F}$ such that for every polynomial p of degree at most d and every $c \in \mathbb{F}$,

$$p[c/\mathbf{x}_j] = \sum_{l=0}^{d} \lambda_l \cdot p[l/\mathbf{x}_j]$$

For a proof, we note that p can be written as:

$$p \equiv \sum_{i=0}^{d} p_i \cdot \mathbf{x}_j^i$$

for polynomials p_i not containing $\mathbf{x}_j$. We define:

$$A \quad = \quad \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 1 & 1 & \ldots & 1 \\ \vdots & \vdots & \ldots & \vdots \\ 1 & d & \ldots & d^d \end{pmatrix}$$

The matrix A is invertible where for the inverse matrix $A^{-1} = (b_{il})$,

$$p_i = \sum_{l=0}^{d} b_{il} \cdot p[l/\mathbf{x}_j]$$

Thus,

$$\begin{array}{rcl} p[c/\mathbf{x}_j] & = & \displaystyle\sum_{i=0}^{d} \sum_{l=0}^{d} c^i \, b_{il} \cdot p[l/\mathbf{x}_j] \\ & = & \displaystyle\sum_{l=0}^{d} \sum_{i=0}^{d} c^i \, b_{il} \cdot p[l/\mathbf{x}_j] \end{array}$$

Accordingly, we choose: $\lambda_l = \sum_{i=0}^{d} c^i \, b_{il}$ independently of p. This implies the assertion. □

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 p with k variables the following holds:*

1. *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.

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

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:

$$\begin{array}{rcl} b & ::= & \mathbf{x}_j := t \;\mid\; \mathsf{call}\langle s\rangle \\ s & ::= & b_1; \ldots; b_n \qquad (n \geq 0) \\ b' & ::= & \mathbf{x}_j := t \;\mid\; \mathsf{call}\langle s\rangle \;\mid\; \mathsf{enter} \\ r & ::= & b'_1; \ldots; b'_n \qquad (n \geq 0) \end{array}$$

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 **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 **R'**:

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

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 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 $T' = \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & I_{m-k} \end{array}\right)$. In particular, $[\![\mathsf{call}\langle s\rangle]\!]$ is an affine transformation as well.

[2] This is the convention chosen in languages like Java. 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 $\Box : \mathbb{F}^{(m+1)^2} \to \mathbb{F}^{(m+1)^2}$ 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. We prove:

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,*

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

The operator "$\Box$" is not only monotonic on sub-spaces, but commutes with arbitrary least upper bounds. 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 p with m global and local variables the following holds:*

1. *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.

2. *The sets of all valid affine relations at program point u, $u \in N$, can be computed in time $\mathcal{O}(p \cdot m^8)$.*

Our technique can easily also handle 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. Conclusion

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

Our analyses are constructed in the spirit of "relational analysis" of recursive procedures of Cousot [2, 3] and the "functional approach" to interprocedural analysis, where the effect of procedures is captured by functions on dataflow informations [19, 8]. We succeed in capturing the effect of procedures as weakest precondition transformers for affine and polynomial relations by sub-spaces of linear maps. This allows us to apply linear algebra techniques.

Our results improve both on the results obtained by Karr [7] and Horwitz et al. [6, 17]. However, they do not generalize the *intraprocedural* analysis of Müller-Olm and Seidl in [15] where we succeed in checking the validity of arbitrary polynomial identities for *polynomial* programs. It remains as a challenging open problem whether or not precise interprocedural constant propagation for these is decidable.

One might also be tempted to generalize the methods to the interprocedural analysis of explicitly parallel programs. In case of atomic assignments (even in absence of synchronizations), however, already copy constant propagation is undecidable [14]. Amazingly enough, in case of *non-atomic* assignments or, equivalently, in presence of local variables, copy constant propagation has recently been shown decidable [12]. It is another challenging question whether this also holds for full affine constant propagation.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] C. Fischer and R. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1988.

[6] 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.

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

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

[9] U. Möncke and R. Wilhelm. Iterative Algorithms on Grammar Graphs. In H. Göttler, editor, *8th GI Conference on Graph-Theoretical Concepts in Computer Science*, pages 177–194. Hanser Verlag, 1982.

[10] U. Möncke and R. Wilhelm. Grammar Flow Analysis. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, pages 151–186. LNCS 545, Springer Verlag, 1991.

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

[12] M. Müller-Olm. Precise Interprocedural Dependence Analysis of Parallel Programs. Technical Report 08, University of Trier, 2002.

[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. On Optimal Slicing of Parallel Programs. In *33th ACM Symp. on Theory of Computing (STOC)*, pages 647–656. ACM Press, 2001.

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

[19] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In [11], chapter 7, pages 189–233.