Interprocedurally Analysing Linear Inequality Relations

Helmut Seidl, Andrea Flexeder and Michael Petter

Technische Universität München, Boltzmannstrasse 3, 85748 Garching, Germany,
{seidl, flexeder, petter}@cs.tum.edu,
WWW home page: http://www2.cs.tum.edu/~{seidl,flexeder,petter}

Abstract. In this paper we present an alternative approach to interprocedurally inferring linear inequality relations. We propose an abstraction of the effects of procedures through *convex sets* of transition matrices. In the absence of conditional branching, this abstraction can be characterised precisely by means of the least solution of a constraint system. In order to handle conditionals, we introduce auxiliary variables and postpone checking them until after the procedure calls. In order to obtain an effective analysis, we approximate convex sets by means of *polyhedra*. Since our implementation of function composition uses the frame representation of polyhedra, we rely on the subclass of *simplices* to obtain an efficient implementation. We show that for this abstraction the basic operations can be implemented in polynomial time. First practical experiments indicate that the resulting analysis is quite efficient and provides reasonably precise results.

1 Introduction

In [5], Cousot and Halbwachs present an *intraprocedural* analysis of linear inequalities based on an abstraction of the collecting semantics [4] by means of convex polyhedra. They draw upon both the frame and the constraint representation of polyhedra to perform the subsumption test and widening [5] on polyhedra. More precise widening strategies on convex polyhedra are provided in [1]. Based on this approach an *interprocedural* analysis can be obtained by relating input and output states of a procedure call by means of linear inequalities. This leads to convex *transition invariants* on program variables before and after the procedure call.

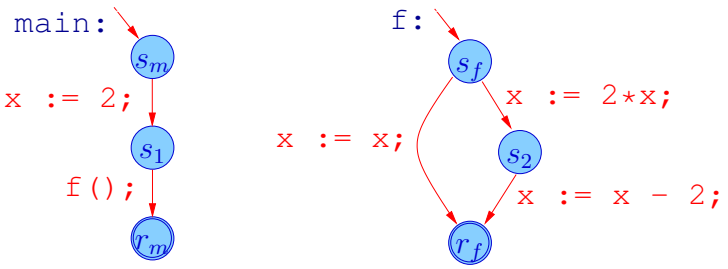


Fig. 1. An example program for transition invariants

In the example in figure 1 (from [8]), the procedure call `f()` at program state 2 can be described by the transition invariant $x = x' \vee x = 2 \cdot x' - 2$. The approximation of this invariant by polyhedra leads to a complete loss of information. Although transition invariants work in several practical cases (e.g., the `McCarthy91` function [6]), they seem too restrictive for a precise interprocedural analysis.

Instead, we propose an interprocedural alternative to transition invariants. Our approach is based on convex sets of *transition matrices* to capture the effects of procedures. For our intraprocedural reachability analysis, the program states are abstracted by convex sets of vectors, describing the values of the program variables. The transformation of program states is described by linear *transition matrices*, similar to [8]. The key idea of our approach is to compute a finite representation for the effect of procedures [9] (i.e. convex sets of transition matrices) which can be embedded into the reachability analysis. In the absence of conditionals, this abstraction can be characterised precisely by means of the least solution of a suitable constraint system. Since conditional branching cannot be represented by linear transformations, conditionals can obviously not be evaluated on convex sets of transition matrices. Therefore, we introduce *auxiliary variables* for each condition and postpone checking them until after the procedure call. In order to obtain an effective analysis, we follow the standard approach of approximating convex sets by means of *convex polyhedra* [5]. Our composition operation for polyhedra relies on the *frame representation* of polyhedra, represented by sets of points, rays and lines. In order to avoid the expensive continual conversion between the two polyhedral representation forms [5], we resort to the frame representation alone. Testing for *subsumption* as well as computing the *union* of two convex polyhedra is reduced to linear programming problems [5]. In order to infer the linear inequalities for a program point, the conversion to the constraint representation is deferred to the end of the computation of the procedure effects or the very end of the analysis. However, polyhedra tend to be complex. Consequently, operations on polyhedra are expensive [10]. This induces the demand for perhaps more efficient representations of convex sets. In [7], *octagons*, an efficient subclass of polyhedra, are introduced. Within this approach at most two program variables per inequality are allowed with restrictions on the coefficients, permitting only inequalities of the form $\pm x \pm y \leq c$. *Simon* et al. abandon all restrictions on the coefficients for the considered pair of occurring program variables in [13]. In contrast, *Clarisó* et al. propose in [3] to approximate polyhedra with *octahedra*. In contrast to the former approaches, they allow any number of program variables but the coefficients of the inequalities are restricted to ± 1 or 0. A quite general approach is introduced by *Sankaranarayanan* et al., who introduce generic inequality templates and solve systems of inequalities on the coefficients of the templates [11]. All these approaches consist in restricted classes of constraint systems, though their frame representation can easily become exponential.

This does not hold for *simplices*. Simplices are convex polyhedra which are restricted in the number of frame elements to at most n *linearly independent elements* and a base point, if n is the dimension of the underlying vector space. Thus, simplices form a subclass of polyhedra where the frame representation has approximately the same size as the constraint representation. This is the reason why we started to experiment with approximations of convex sets by means of simplices. Based on this approximation, we

achieve that subsumption testing reduces to solving $n+1$ systems of at most n linear equations. Thus, our subsumption test can be performed in polynomial time.

Our approach for interprocedurally identifying linear inequality relations among the variables of a program is subsequently described in detail. In section 2 we introduce control flow graphs, representing our inspected program class. Furthermore, the collecting semantics of our program class is described. In section 3 we turn to the abstraction of the concrete semantics based on convex sets. In addition, we present a method for interprocedurally dealing with conditionals. Effective approximations of convex sets are discussed in section 5, where we also present simplices and the basic operations on simplices. Finally, first experimental results and comparisons of the various approaches are reported in section 6.

2 The general set-up

This section introduces the programs to be analysed together with their collecting semantics. We assume that a program is represented by a finite set of disjoint *control flow graphs* $\mathbf{G}$, as illustrated in figure 1.

Each graph $G_f \in \mathbf{G}$ corresponds to a *procedure* f from a finite set $Proc$ of procedures. Each control flow graph $G_f \in \mathbf{G}$ consists of:

- a finite set N_f of *program points* of the procedure f,
- a finite set $E_f \subseteq (N_f \times Label \times N_f)$ of labeled control-flow *edges*,
- the start point $s_f \in N_f$ for the procedure f, as well as the
- return point $r_f \in N_f$ of f.

Labels at control-flow edges either are linear assignments (e.g. $\mathtt{x}_1 \mathtt{:=} \mathtt{x}_2 \mathtt{+5}$), procedure calls (e.g. $\mathtt{g()}$), non-deterministic assignments ($\mathtt{x}_1 \mathtt{:=\ ?}$) or linear conditions (e.g. $\mathtt{x}_1 \mathtt{-x}_2 \mathtt{-3\ \geq\ 0}$). For simplicity, we only consider conditionals of the form $t \geq 0$.

We suppose that the program operates on the n global program variables $\mathbf{x}_1, \ldots, \mathbf{x}_n$. We assume the variables to take values on an ordered field $\mathbb{F}$. In the following we consider the field $\mathbb{Q}$. Then a program state can be modelled by a $(n+1)$-dimensional column *vector* $x = (1, x_1, \ldots, x_n)^T \in \{1\} \times \mathbb{Q}^n$. Each component x_i, $i > 0$, of the vector x represents the value assigned to the program variable $\mathbf{x}_i$. Note that we use an extra 0-th component 1. This extra component allows modelling the semantic effects of affine assignments through linear transformations, e.g. as considered in [8].

The set of all state vectors attained at a program point through program execution forms the *collecting semantics* of the program at this program point. Every assignment $\mathbf{x}_i := t$ of a linear term $t = t_0 + \sum_{j=1}^{n} t_j \cdot \mathbf{x}_j$; causes a *linear transformation* $[\![\mathbf{x}_i := t]\!] : 2^{\mathbb{Q}^{n+1}} \to 2^{\mathbb{Q}^{n+1}}$ on the underlying set of program states. Its effect onto a single program state can be described by multiplication of x with the following matrix:

$$[\![\mathbf{x}_i := t_0 + \sum_{j=1}^{n} t_j \cdot \mathbf{x}_j;]\!] \;=\; \left(\begin{array}{c|c} \mathbf{I}_i & \mathbf{0} \\ \hline t_0 \;\ldots\; t_n & \\ \hline \mathbf{0} & \mathbf{I}_{n-i} \end{array}\right)$$

with $\mathbf{I}_i : (i \times i)$-dimensional identity matrix in $\mathbb{Q}^{i \times i}$. As we consider extended program states, the matrix of this definition is from $\mathbb{Q}^{(n+1)^2}$. We only consider matrices where

the entry at position $(0, 0)$ is equal to 1 and the remaining entries in the 0th row are all 0.

For the beginning, we assume that the program does not contain conditional branching, i.e. edges labeled with inequalities. Since linear transformations are closed under composition, we realise that the effects of procedures can be represented by *sets* of linear transformations of the extended program state. These sets of transformations can be characterised by the least solution of the following constraint system $\mathbf{T}$:

$$
\begin{array}{lll}
[\mathrm{T0}]\ \mathbf{T}(s_f) \supseteq \{\mathbf{Id}\} & & \\
[\mathrm{T1}]\ \mathbf{T}(v) \supseteq \{[\![\mathbf{x}_i := t]\!]\} \circ \mathbf{T}(u) & & \text{for } (u,\ \mathtt{x_i} := \mathtt{t};\ ,\ v) \in E_f \\
[\mathrm{T2}]\ \mathbf{T}(v) \supseteq \{[\![\mathbf{x}_i := c]\!] \mid c \in \mathbb{Q}\} \circ \mathbf{T}(u) & & \text{for } (u,\ \mathtt{x_i} := ?;\ ,\ v) \in E_f \\
[\mathrm{T3}]\ \mathbf{T}(v) \supseteq \mathbf{T}(r_g) \circ \mathbf{T}(u) & & \text{for } (u,\ \mathtt{g}();\ ,\ v) \in E_f
\end{array}
$$

Here, the operator $\circ$ denotes the element-wise function composition of two sets of transformations. Thus, the effect of a whole procedure is the effect accumulated at the return point of the procedure.

Constraint [T0] expresses that no initialisation of the program variables is performed at the start point s_f of any procedure $f \in Proc$. This results in the identity mapping $\mathbf{Id}$. [T1] describes the accumulation of the effect of a linear assignment. It is obtained by the composition of the linear transformation corresponding to the assignment with the effect already accumulated for the start point u of the edge. If the edge is labeled with a non-deterministic assignment, each value $c \in \mathbb{Q}$ can be assigned to the program variable $\mathbf{x}_i$. This is described by the constraint [T2]. Constraint [T3] describes the handling of edges, which are labeled with a procedure call. We simply compose all transformations of the called procedure with the transformations accumulated before the procedure call. Since all right-hand sides in the constraint system $\mathbf{T}$ represent monotonic functions, a least solution for this system exists. We denote the components of this least solution by $\mathbf{T}(u)$ (u a program point).

Given the effects of procedures, we can characterise the sets of program states reaching program points by the least solution of the constraint system $\mathbf{A}$:

$$
\begin{array}{lll}
[\mathrm{A0}]\ \mathbf{A}(s_{main}) \supseteq \{1\} \times \mathbb{Q}^n & & \\
[\mathrm{A1}]\ \mathbf{A}(s_g) \supseteq \mathbf{A}(u) & & \text{if } (u,\ \mathtt{g}();\ ,\ \_) \text{ calls } g \in Proc \\
[\mathrm{A2}]\ \mathbf{A}(v) \supseteq [\![\mathbf{x}_i := t]\!]\ \mathbf{A}(u) & & \text{for } (u,\ \mathtt{x_i} := \mathtt{t};\ ,\ v) \in E_f \\
[\mathrm{A3}]\ \mathbf{A}(v) \supseteq \bigcup_{c \in \mathbb{Q}} [\![\mathbf{x}_i := c]\!]\ \mathbf{A}(u) & & \text{for } (u,\ \mathtt{x_i} := ?;\ ,\ v) \in E_f \\
[\mathrm{A4}]\ \mathbf{A}(v) \supseteq \mathbf{T}(r_g)\ \mathbf{A}(u) & & \text{for } (u,\ \mathtt{g}();\ ,\ v) \in E_f
\end{array}
$$

The first constraint [A0] expresses that program execution starts with a call to the specific procedure `main`. At this point, no assumptions on the set of program states can be made. Constraint [A1] describes that the start point of a procedure f is dependent of all the program points where f is called. Linear and non-deterministic assignments, as defined in [A2] and [A3], result in applying the transformation functions corresponding to the edges element-wise to the sets of vectors reaching the start point of the edge. The same does also hold for procedure calls where the effect of a call to f is given by the set of transformations $\mathbf{T}(r_f)$, provided by the least solution of the constraint system $\mathbf{T}$.

This is formalised in the constraint [A4], where the function application is performed element-wise to the sets of vectors. Again, the least solution for this constraint system exists according to the fixpoint theorem of Knaster-Tarski and we denote its components by $\mathbf{A}(u)$ (u a program point).

3 Convex abstraction

In order to interprocedurally infer linear inequality relations, we want to construct a precise abstraction for our collecting semantics. The abstraction should provide for every program point u (hopefully all) linear inequalities, which are valid for all program states reaching u. Geometrically, a linear inequality specifies a half space. The conjunctive combination of these half spaces results in a *convex set of vectors*. This may serve as a justification of an abstraction of the concrete semantics by means of convex sets, the *convex abstraction*.

Formally, let $\mathcal{C}(\mathbb{Q}^{n+1})$ denote the set of all convex subsets of vectors over $\mathbb{Q}^{n+1}$. On convex sets, the greatest lower bound $\sqcap$ is given by the set theoretical intersection, while the least upper bound $\sqcup$ is given by the convex hull of the set theoretical union: $\langle X_1\rangle \sqcup \langle X_2\rangle = \langle X_1 \cup X_2\rangle$ with $X_i \subseteq \{1\} \times \mathbb{Q}^n, i = 1, 2$. The set $\mathcal{C}(\mathbb{Q}^{n+1})$ of all convex subsets of vectors together with the subset relation $\subseteq$ as partial ordering relation (denoted by $\sqsubseteq$ here) forms a complete lattice.

Now, we define the abstraction $\alpha : 2^{\mathbb{Q}^{n+1}} \to \mathcal{C}(\mathbb{Q}^{n+1})$ by: $\alpha(X) = \langle X\rangle$ where $\langle X\rangle$ denotes the least convex set containing $X \subseteq \{1\} \times \mathbb{Q}^n$. The convex set $\langle X\rangle$ can be obtained from X by applying the convex hull operation to X:

$$\langle X\rangle \;=\; \left\{ \sum_{i=1}^{n} \lambda_i z_i \mid n \in \mathbb{N} \;\wedge\; 0 \le \lambda_i \;\wedge\; \sum_{i=1}^{n} \lambda_i = 1 \;\wedge\; z_i \in X \right\}$$

Clearly, α commutes with arbitrary unions and therefore is an abstraction.

Within our interprocedural approach the effect of assignments is modelled by a set of linear transformations. Each of these transformations can be represented by a matrix, similar to [8]. As a matrix can be seen as a $(n+1)^2$-dimensional vector, the abstraction α is also applicable to sets of matrices. Thus, the abstract effect $[\![\mathbf{x}_i := t]\!]^\sharp$ of a linear assignment $\mathbf{x}_i := t$ results in the convex hull of the single $(n+1)^2$ vector obtained from $[\![\mathbf{x}_i := t]\!]$. In the case of a non-deterministic assignment, all possible constant values of $\mathbb{Q}$ could be assigned to the program variable. This effect is described by the following convex set of transition matrices:

$$[\![\mathbf{x}_i :=?]\!]^\sharp \;=\; \left\langle \left\{ \left(\begin{array}{c|c} \mathbf{I}_i & \mathbf{0} \\ \hline \lambda\, 0\, \ldots\, 0 & \\ \hline \mathbf{0} & \mathbf{I}_{n-i} \end{array} \right) \mid \lambda \in \mathbb{Q} \right\} \right\rangle$$

In order to approximate the convex abstraction of the effect of procedure calls, we apply the abstraction to the constraint system $\mathbf{T}$. The resulting system $\mathbf{T}^\sharp$ is given by:

$$\begin{array}{lll}
[\mathrm{T0}^\sharp]\ \mathbf{T}^\sharp(s_f) \sqsupseteq \{\mathbf{I}\} & & \mathbf{I} \in \mathbb{Q}^{(n+1)^2} \\
[\mathrm{T1}^\sharp]\ \mathbf{T}^\sharp(v) \;\sqsupseteq [\![\mathbf{x}_i := t]\!]^\sharp \circ^\sharp \mathbf{T}^\sharp(u) & & \text{for } (u,\ \mathtt{x_i := t;}\ ,\ v) \in E_f \\
[\mathrm{T2}^\sharp]\ \mathbf{T}^\sharp(v) \;\sqsupseteq [\![\mathbf{x}_i :=?]\!]^\sharp \circ^\sharp \mathbf{T}^\sharp(u) & & \text{for } (u,\ \mathtt{x_i :=?;}\ ,\ v) \in E_f \\
[\mathrm{T3}^\sharp]\ \mathbf{T}^\sharp(v) \;\sqsupseteq \mathbf{T}(r_g)^\sharp \circ^\sharp \mathbf{T}^\sharp(u) & & \text{for } (u,\ \mathtt{g();}\ ,\ v) \in E_f
\end{array}$$

In the abstraction, we have used the convex composition $\circ^\sharp$ on convex sets of linear transformations, which is defined by an element-wise matrix-multiplication composed with the convex hull operation:

$$\langle \mathbf{C}_1 \rangle \circ^\sharp \langle \mathbf{C}_2 \rangle = \langle C_1 C_2 \mid C_i \in \mathbf{C}_i \rangle \;\text{ with }\; \mathbf{C}_i \subseteq \mathbb{Q}^{(n+1)^2}$$

The least solution of $\mathbf{T}^\sharp$ provides an abstract effect of a procedure represented as a convex set of transformation matrices. We only consider those matrices where the entry at position $(0,0)$ is equal to 1 and the remaining entries at the 0th row are all 0. We denote the components of this least solution by $\mathbf{T}^\sharp(u)$ (u a program point).

Accordingly, we can describe the reachability analysis in the convex abstraction by the constraint system $\mathbf{A}^\sharp$ obtained from the concrete constraint system $\mathbf{A}$ by applying the abstraction α:

$$\begin{array}{lll}
[\mathrm{A0}^\sharp]\ \mathbf{A}^\sharp(s_{main}) \sqsupseteq \{1\} \times \mathbb{Q}^n & & \\
[\mathrm{A1}^\sharp]\ \mathbf{A}^\sharp(s_g) \sqsupseteq \mathbf{A}^\sharp(u) & & \text{if } (u,\ \mathtt{g}();\ ,\ _) \text{ calls } g \in Proc \\
[\mathrm{A2}^\sharp]\ \mathbf{A}^\sharp(v) \sqsupseteq [\![\mathbf{x}_i := t]\!]^\sharp \cdot^\sharp \mathbf{A}^\sharp(u) & & \text{for } (u,\ \mathtt{x_i} := \mathtt{t};\ ,\ v) \in E_f \\
[\mathrm{A3}^\sharp]\ \mathbf{A}^\sharp(v) \sqsupseteq [\![\mathbf{x}_i := ?]\!]^\sharp \cdot^\sharp \mathbf{A}^\sharp(u) & & \text{for } (u,\ \mathtt{x_i} := ?;\ ,\ v) \in E_f \\
[\mathrm{A4}^\sharp]\ \mathbf{A}^\sharp(v) \sqsupseteq \mathbf{T}^\sharp(r_g) \cdot^\sharp \mathbf{A}^\sharp(u) & & \text{for } (u,\ \mathtt{g}();\ ,\ v) \in E_f
\end{array}$$

Analogously to the abstract composition operator $\circ^\sharp$, the abstract application operator $\cdot^\sharp$ is defined by element-wise application composed with the convex hull operation. The least solution of the system $\mathbf{A}^\sharp$ again exists and provides us with a convex set of vectors for every program point u. For convenience, we denote the components of this least solution by $\mathbf{A}^\sharp(u)$ (u a program point).

First we want to show the safety and precision of the convex abstraction. For this purpose we verify that the abstraction commutes with function application and composition of the linear transformations.

Proposition 1. *For every set of vectors $X \subseteq \{1\} \times \mathbb{Q}^n$ and all sets of transformation matrices $\mathbf{C}, \mathbf{C}_1, \mathbf{C}_2 \subseteq \mathbb{Q}^{(n+1)^2}$, the following equalities hold:*

1. $\langle \{ Cx \mid\ x \in X\ ,\ C \in \mathbf{C}\ \} \rangle = \langle \{ Cx \mid\ x \in \langle X \rangle\ ,\ C \in \langle \mathbf{C} \rangle\ \} \rangle$
2. $\langle \{ C_1 C_2 \mid\ C_i \in \mathbf{C}_i\ \} \rangle = \langle \{ C_1 C_2 \mid\ C_i \in \langle \mathbf{C}_i \rangle\ \} \rangle$

For the constraint systems $\mathbf{A}^\sharp$ and $\mathbf{T}^\sharp$ we therefore obtain from proposition 1 with the fixpoint transfer lemma:

Theorem 1. *For every program point u and every procedure f of the program with return point r_f, the following holds:*

1. $\mathbf{A}^\sharp(u) = \alpha(\mathbf{A}(u)) = \langle \mathbf{A}(u) \rangle$
2. $\mathbf{T}^\sharp(r_f) = \alpha(\mathbf{T}(r_f)) = \langle \mathbf{T}(r_f) \rangle$

This theorem means that the smallest fixpoints of the constraint systems $\mathbf{T}^\sharp$ and $\mathbf{A}^\sharp$ precisely characterise the convex abstraction α applied to the smallest fixpoints of the constraint systems $\mathbf{T}$ and $\mathbf{A}$ for the collecting semantics.

In general, the least solutions of the abstract constraint systems will not be reached after finitely many fixpoint iterations. In order to arrive at practical algorithms for computing safe (over-) approximations of the least solutions of these constraint systems, we therefore must rely on effective representations of convex sets together with effective abstract *composition* and *application* operations as well as effective implementations of *subsumption* and *union*. In order to speed up fixpoint iteration, a *widening* operator must be provided.

By now, we have specified the convex abstraction and verified its *correctness and precision*. However, our abstraction of the effects of procedures only works for non-deterministic branching, i.e. in the absence of inequality guards. Linear inequality analysis is not yet very significant without the handling of conditionals. The next section therefore provides a technique to enhance the base framework to handle linear inequality guards.

4 Linear Inequality Guards

Clearly, the reachability analysis naturally can be enhanced to deal with linear inequality guards $(\mathbf{b} \geq 0)$. As in [5], the effect of such a guard is interpreted as the intersection with the corresponding half-space of state vectors, which satisfy the guard:

$$[\![\mathbf{b} \geq 0]\!]\, X \;=\; \{x \in X \mid \mathbf{b}x \geq 0\}$$

where for $\mathbf{b} = b_0 + b_1\mathbf{x}_1 + \ldots + b_n\mathbf{x}_n$ and $x = (1, x_1, \ldots, x_n)^T$,

$$\mathbf{b}x = b_0 + b_1x_1 + \ldots + b_nx_n$$

When analysing programs with conditional branching, the effects of procedures can no longer be described by sets of linear transformations. Since the constraint system $\mathbf{T}$ only speaks about linear transformations, conditionals cannot easily be integrated into our concrete semantics. The constraint system $\mathbf{A}$ for the reachbility analysis, however, can be extended to conditionals by introducing the following constraint:

$$[\mathrm{A5}]\ \mathbf{A}(v) \supseteq \{x \in \mathbf{A}(u) \mid \mathbf{b}x \geq 0\} \ \text{ for } (u,\ (\mathbf{b} \geq 0),\ v) \in E_f$$

Within the convex abstraction, the idea for interprocedurally handling conditionals therefore is to *postpone* their evaluation during the computation of procedure effects until the reachability analysis. Up to this time, we suggest to store the value of each condition in an *auxiliary variable*, which then can be checked for non-negativity.

Thus, we extend the original semantics by introducing new "program variables ", one for each guard. Assuming that the guards are numbered $n+1, \ldots, n+g$, the auxiliary variables are denoted by $\mathbf{x}_{n+1} \ldots \mathbf{x}_{n+g}$. This leads to an extension of every program state by g extra components. All the auxiliary variables are initially set to 0. During the effect computation we replace the jth conditional $(\mathbf{b} \geq 0)$ with the assignment $\mathbf{x}_{n+j} := \mathbf{b}$. As the value of each condition is just stored in an auxiliary variable, conditionals now can be treated within our effect computation. Accordingly, we modify the constraint system $\mathbf{T}^\sharp$ as follows:

$$[\mathrm{T4}^\sharp]\ \mathbf{T}^\sharp(v) \sqsupseteq [\![\mathbf{x}_{n+j} := \mathbf{b}]\!]^\sharp \circ^\sharp \mathbf{T}^\sharp(u) \ \text{ for } (u,\ (\mathbf{b} \geq 0),\ v) \in E_f$$

where $(\mathbf{b} \geq 0)$ denotes the jth conditional.

Clearly, every feasible program execution path of the original program will also be a feasible execution path of the transformed program — but not necessarily vice versa. Thus, our postponed evaluation of guards introduces a safe over-approximation of the concrete semantics. Due to the extension of every program state the constraint $[\mathrm{A0}^\sharp]$ must be adapted to handle conditionals:

$$[\mathrm{A0}^\sharp]\ \mathbf{A}^\sharp(s_{main}) \sqsupseteq 1 \times \mathbb{Q}^n \times 0^g$$

This constraint shows that at the start point of procedure $main$ every program state is possible, in which all auxiliary variables are 0.

There are two natural choices for scheduling the evaluation of the postponed guards $(\mathbf{x}_{n+j} \geq 0)$ during the reachability analysis. The first alternative is to schedule their evaluation *directly after* each procedure call. Then the constraint system $\mathbf{A}^\sharp$ is modified as follows:

$$[\mathrm{A4}^\sharp]\ \mathbf{A}^\sharp(v) \sqsupseteq \mathbf{T}^\sharp(r_g) \circ^\sharp \mathbf{A}^\sharp(u) \cap \{(1, x_1, \ldots, x_{n+g}) \mid x_{n+j} \geq 0 \text{ for all } j\} \quad \text{for } (u,\ g(),\ v) \in E_f$$

$$[\mathrm{A5}^\sharp]\ \mathbf{A}^\sharp(v) \sqsupseteq \{x \in \mathbf{A}^\sharp(u) \mid \mathbf{b}x \geq 0\}\ \text{ for } (u,\ (\mathbf{b} \geq 0),\ v) \in E_f$$

The modified constraint $[\mathrm{A4}^\sharp]$ describes the postponed evaluation of guards after each procedure call, whereas the additional constraint $[\mathrm{A5}^\sharp]$ illustrates the direct evaluation when a conditional has been visited.

As a second alternative, we may postpone the evaluation of guards even during the reachability analysis — in order to perform a *single* check for every program point u just before the valid linear inequalities for u are inferred. To this end, we use the original constraints $[\mathrm{A4}^\sharp]$. Furthermore, we replace the constraints $[\mathrm{A5}^\sharp]$ with corresponding assignments to the auxiliary variables:

$$[\mathrm{A5}^\sharp]\ \mathbf{A}^\sharp(v) \sqsupseteq [\![\mathbf{x}_{n+j} := \mathbf{b}]\!]^\sharp(\mathbf{A}^\sharp(u))\ \text{ for } (u,\ (\mathbf{b} \geq 0),\ v) \in E_f$$

where $(\mathbf{b} \geq 0)$ denotes the jth conditional. Finally, we introduce extra unknowns $\mathbf{A}^\sharp(u)'$ for each program point u which are meant to receive the final analysis results. For these, we introduce the extra constraints:

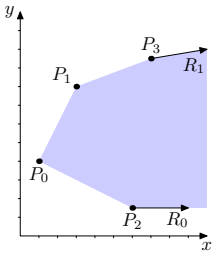
$$[\mathrm{A}']\ \mathbf{A}^\sharp(u)' \sqsupseteq \mathbf{A}^\sharp(u) \cap \{(1, x_1, \ldots, x_{n+g}) \mid x_{n+j} \geq 0 \text{ for all } j\}\ \text{ for } u\ \in N_f$$

The latter alternative may lose more precision in comparision to the immediate evaluation, because more execution paths are admitted. A first comparison between the two alternatives is shown in section 6. In case of an analysis over integer variables, however, all of the second analysis can be performed within the field $\mathbb{Q}$ — up to the final condition evaluation. Thus, we even obtain a tight integer solution already if the final round of intersections is performed by an ILP solver.

5 Representing convex sets

So far, we have introduced a framework for an interprocedural analysis for inferring linear inequalities. In order to arrive at practical analysis algorithms, it remains to choose suitable effective representations for convex sets, which support the necessary operations as well as a widening operation to enforce termination of the fixpoint iteration.

Convex Polyhedra



For this purpose we focus on the subset of $\mathcal{C}(\mathbb{Q}^{n+1})$ of *convex polyhedra* [5], denoted by $\mathcal{P}$. For our approach, we find it convenient to use the *frame representation* of polyhedra. This means that a polyhedron $\mathbf{F}$ is represented as a triple $\mathbf{F} = \langle \mathbf{P}, \mathbf{R}, \mathbf{L} \rangle$ where $\mathbf{P}$ denotes a finite set of points, $\mathbf{R}$ is a finite set of rays and $\mathbf{L}$ is a finite set of lines. The figure on the left-hand side illustrates a polyhedron in $\mathbb{Q}^2$, which consists of a point set and a ray set, forming the polyhedron $\langle \{P_0, P_1, P_2, P_3\}, \{R_0, R_1\}, \emptyset \rangle$.
Every element of $\mathbf{R}$, respectively $\mathbf{L}$, is a vector, which can be considered as the difference of two points in the considered vector space. As mentioned in 2, we use projective space in our vectors. Thus, the extra 0-th component of a vector is always 1 for points and 0 for rays or lines. The set of points, represented by $\langle \mathbf{P}, \mathbf{R}, \mathbf{L} \rangle$, is given by:

$$[\![\langle \mathbf{P}, \mathbf{R}, \mathbf{L} \rangle]\!] = \{ \sum_{i=0}^{q} \lambda_i P_i + \sum_{i=0}^{r} \mu_i R_i + \sum_{i=0}^{s} \eta_i L_i \mid q, r, s \geq 0 \wedge \lambda_i, \mu_i \geq 0 \wedge \sum_i \lambda_i = 1 \}$$

with $\mathbf{P} = \{P_0, \ldots, P_q\},\ \mathbf{R} = \{R_0, \ldots, R_r\},\ \mathbf{L} = \{L_0, \ldots, L_s\}$. In order to use polyhedra as effective representation of convex sets of transition matrices in the constraint system $\mathbf{T}^\sharp$, we must provide algorithms for composition, union, widening as well as an effective test for subsumption on polyhedra. We introduce the polyhedral composition $\circ^{\mathcal{P}}$ as an abstraction of $\circ^{\#}$, in order to easily express the composition on the frame representation of polyhedra.

Composition. Let $\mathbf{F}_i = \langle \mathbf{P}_i, \mathbf{R}_i, \mathbf{L}_i \rangle$, $i = 1, 2$, denote the frame representation of two polyhedra of transition matrices. The polyhedral composition $\mathbf{F} = \mathbf{F}_1 \circ^{\mathcal{P}} \mathbf{F}_2$ results in the frame $\mathbf{F}$ defined by the triple $\langle \mathbf{P}, \mathbf{R}, \mathbf{L} \rangle$, where

$$\begin{aligned}
\mathbf{P} &= \{\mathbf{P}_1 \circ \mathbf{P}_2\} \\
\mathbf{R} &= \{\mathbf{P}_1 \circ \mathbf{R}_2 \ \cup\ \mathbf{R}_1 \circ \mathbf{R}_2 \ \cup\ \mathbf{R}_1 \circ \mathbf{P}_2\} \\
\mathbf{L} &= \{\mathbf{L}_1 \circ \mathbf{P}_2 \ \cup\ \mathbf{L}_1 \circ \mathbf{R}_2 \ \cup\ \mathbf{L}_1 \circ \mathbf{L}_2 \ \cup\ \mathbf{P}_1 \circ \mathbf{L}_2 \ \cup\ \mathbf{R}_1 \circ \mathbf{L}_2\}
\end{aligned}$$

Here, $\circ$ denotes the element-wise multiplication of two sets of matrices.
By construction we obtain:

Proposition 2. *The result of the polyhedral composition is a superset of the convex composition:* $[\![\mathbf{F}_1 \circ^{\mathcal{P}} \mathbf{F}_2]\!] \supseteq [\![\mathbf{F}_1]\!] \circ^{\sharp} [\![\mathbf{F}_2]\!]$

The other direction $\sqsubseteq$ is not necessarily valid in presence of rays and lines. If the frame consists of points only, the polyhedral composition $\circ^{\mathcal{P}}$ is equivalent to the convex composition $\circ^{\sharp}$.

Widening. In order to compute effectively some (hopefully non-trivial) solution of the constraint system $\mathbf{T}^\sharp$ by means of convex polyhedra, we should avoid infinite ascending chains during fixpoint iteration. This can be achieved by the use of widening for polyhedra, e.g. the *standard widening* introduced by Cousot and Halbwachs [5]. Here, we rely on those more precise widening strategies of Bagnara et.al. [1], which are restricted to the frame representation of a convex polyhedron.

Union and Subsumption. In every step of the fixpoint iteration we must check if the next polyhedron $\mathbf{F}$ for a constraint variable is already subsumed by the old value $\mathbf{F}'$, i.e. whether $[\![\mathbf{F}]\!] \sqsubseteq [\![\mathbf{F}']\!]$. This subsumption test can be implemented by successively testing for all frame elements of polyhedron $\mathbf{F}$ whether they can be represented by the elements of the polyhedron $\mathbf{F}'$ or not. Subsumption testing, thus, reduces to checking the feasibility of a *linear program* [12]. Union for two polyhedra (following referred to as *polyhedral union*) on the other hand is implemented readily using set theoretical union on each of the three components of the frame representation. Subsequent subsumption testing may be used to remove redundant elements from the result.

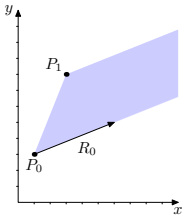
Linear Guards. According to the extended constraint system for the reachability analysis, as presented in section 4, both alternatives for evaluating conditionals can be applied to convex polyhedra. For performing intersections on polyhedra we apply the techniques from [5].

In practice, program analysis using polyhedra is quite expensive [10]. Thus, in recent approaches special subclasses of polyhedra have been proposed, e.g. octagons [7] or octahedra [3]. These subclasses rely on restricted forms of constraint systems to specify polyhedra, which then can be handled efficiently. Since the frame representation of these polyhedra can be easily exponential in the number of constraints, they cannot be applied here.

This is the reason why we will turn our attention to *simplices*, a particular subclass of polyhedra, whose frame representation has almost the same size as the constraint representation.

Simplices

The idea is to *restrict the number of frame elements* in the frame representation $\langle \mathbf{P}, \mathbf{R}, \mathbf{L} \rangle$ of a non-empty polyhedron to n frame elements and a base point $P_0 \in \mathbf{P}$, whereas the differences $P - P_0, P_0 \neq P \in \mathbf{P}$ together with the rays and lines are *all linearly independent*. In the following this fact is referred to as the linear independence of frame elements.



The figure on the left-hand side illustrates the simplex $\langle \{P_0, P_1\}, \{R_0\}, \emptyset \rangle \subseteq \mathbb{Q}^2$. Two-dimensional simplices may consist of at most 3 frame elements. Obviously, in this example the difference $P_1 - P_0$ is linearly independent from the ray R_0. For simplices, we need again an appropriate *subsumption test*, *union* as well as an effective *composition*. Furthermore, *widening* on simplices must be introduced to assure the linear independence of frame elements. Union and composition for simplices can be readily implemented by using the corresponding polyhedral operations and subsequently determining a preferably small simplex (referred to as *enclosing simplex*) which encloses the polyhedron.

Enclosing Simplex. Given a polyhedron $\mathbf{F}$, a simplex $\mathbf{S}$ is called *enclosing simplex* for $\mathbf{F}$ iff $[\![\mathbf{F}]\!] \sqsubseteq [\![\mathbf{S}]\!]$. This enclosing simplex is realised by successively building up the simplex. Starting with an empty simplex, which is successively widened with all the frame elements of the polyhedron $\mathbf{F}$.

Subsumption. As for polyhedra the subsumption test for simplices $[\![\mathbf{S}]\!] \sqsubseteq [\![\mathbf{S}']\!]$ is performed by successively checking the points, rays and lines of $\mathbf{S}$ whether they can be expressed through the points, rays and lines of $\mathbf{S}'$ or not. However, for simplices each such test can be performed through solving an appropriate *system of linear equations*. Because of the linear independence of frame elements, this system has a unique solution. In order to determine whether a point P, a ray R or a line L is subsumed by the simplex $\langle \mathbf{P}, \mathbf{R}, \mathbf{L} \rangle$, the corresponding system of linear equations has to be solved:

$$P = P_0 + \sum_{i=1}^{q} \lambda_i (P_i - P_0) + \sum_{i=0}^{r} \mu_i R_i + \sum_{i=0}^{s} \eta_i L_i \tag{1}$$

$$R = \sum_{i=0}^{r} \mu_i R_i + \sum_{i=1}^{s} \eta_i L_i \tag{2}$$

$$L = \sum_{i=0}^{s} \eta_i L_i \tag{3}$$

where $\sum_{i=1}^{q} \lambda_i \leq 1 \wedge \lambda_i, \mu_i \geq 0$ holds, $P_i \in \mathbf{P}$, $R_i \in \mathbf{R}$, $L_i \in \mathbf{L}$ and P_0 as base point. The complexity of solving such a system of linear equations is *cubic* in the number of frame elements. If the system of linear equations is feasible and the restrictions for the coefficients λ_i, μ_i hold, the point P, the ray R or the line L is considered as subsumed.

Composition. The composition of two simplices (referred to as simplicial composition) is reduced to the polyhedral composition $\circ^{\mathcal{P}}$ and subsequently determining the enclosing simplex.

Union. Union for two simplices $\mathbf{S}_1, \mathbf{S}_2$ (simplicial union) is implemented using the polyhedral union of the simplices and subsequently determining the enclosing simplex for this polyhedron. This can be efficiently realised by successively widening of simplex $\mathbf{S}_1$ with all the frame elements of $\mathbf{S}_2$.

Widening. Widening of a simplex $\mathbf{S}$ with a frame element E results in three distinct cases: First, if the frame element E is linearly independent of all the frame elements of $\mathbf{S}$, E can be directly added to the corresponding element set of $\mathbf{S}$. Secondly, if E is already subsumed, $\mathbf{S}$ does not have to be widened. In the third case the linearly dependent frame elements of $\mathbf{S}$ (i.e. their linear combination represents E) are widened according to one of the algorithms presented in figure 2, 3, 4.

```
1  widen(⟨P, R, L⟩, P) {
     choose some base point P_0 ∈ P;
     determine λ_i, μ_j with 1 ≤ i ≤ q, 0 ≤ j ≤ r
     P = P_0 + Σ_{i=1}^{q} λ_i(P_i − P_0) + Σ_{j=0}^{r} μ_j R_j + Σ_{i=0}^{s} η_i L_i
       for all j s.t. μ_j < 0: L ← L ∪ R_j; R ← R \ R_j;
6      for all i s.t. λ_i ≠ 0:
         if (λ_i < 0) {L ← L ∪ (P_0 − P_i); P ← P \ P_i;}
         if (λ_i > 1) {R ← R ∪ (P_i − P_0); P ← P \ P_i; }
       while (Σ_{j=1}^{q} λ_j > 1) {R ← R ∪ (P_q − P_0); P ← P \ P_q; q ← q − 1; }
       return ⟨P, R, L⟩;
11 }
```

Fig. 2. Widening of a simplex with a *point* P

When widening the simplex with a *point* P, the system of equations (1) has to be solved to determine the coefficients for the frame elements, who contribute to the linear

combination of P (v. line 4 of the algorithm in figure 2). The frame elements, more precisely the points and rays of the simplex, whose restrictions on the coefficients do not hold, have to be widened.

If the restriction of a ray R_j does not hold, i.e. $\mu_j < 0$, the ray R_j is removed from the ray set of the simplex and added to its line set, as line 5 of the algorithm in figure 2 demonstrates. Furthermore, if the restriction on the coefficient of a point P_i does not hold there are two cases: if $\lambda_i < 0$ then P_i is removed from the point set and the difference $P_0 - P_i$ is added to the line set, whereas if $\lambda_i > 1$ the difference $P_i - P_0$ is added to the ray set. Additionally, the restriction on the sum of the points' coefficients $\sum_{i=1}^{q} \lambda_i \leq 1$ must be preserved. As long as this restriction does not hold, the ray set is augmented with the differences $P_i - P_0$ (v. line 9 of the algorithm in figure 2). The resulting simplex subsumes P and does only consist of linearly independent frame elements. Note that the precision of the widening presented here strongly depends on the choice of the *base point* P_0, but can be implemented in such a way, the choice of P_0 becomes irrelevant for the precision of the resulting simplex.

```
   widen(⟨P, R, L⟩, R) {
     choose some base point P_0 ∈ P;
     determine λ_i, μ_j with 1 ≤ i ≤ q, 0 ≤ j ≤ r
4    R = Σ_{i=1}^{q} λ_i(P_i − P_0) + Σ_{j=0}^{r} μ_j R_j + Σ_{i=0}^{s} η_i L_i
       for all j s.t. μ_j < 0: L ← L ∪ R_j; R ← R \ R_j;
       for all i s.t. λ_i ≠ 0:
         if (λ_i < 0) {L ← L ∪ (P_0 − P_i); P ← P \ P_i;}
         if (λ_i > 0) {R ← R ∪ (P_i − P_0); P ← P \ P_i; }
9      return ⟨P, R, L⟩;
   }
```

Fig. 3. Widening of a simplex with a *ray* R

In the case of widening a simplex with a *ray* R, we determine the coefficients for the differences $P_i - P_0, P_0 \neq P_i \in \mathbf{P}$ and the rays $\mathbf{R}$ (v. line 4 of the algorithm in figure 3). Analogously to the algorithm widening with a point 2, all the points and rays, whose coefficients do not hold, are widened i.e. they are added to the ray set respectively line set (v. line 5/6 of the algorithm in figure 3).

```
   widen(⟨P, R, L⟩, L) {
     choose some base point P_0 ∈ P;
     determine λ_i, μ_j with 1 ≤ i ≤ q, 0 ≤ j ≤ r
     L = Σ_{i=1}^{q} λ_i(P_i − P_0) + Σ_{j=0}^{r} μ_j R_j + Σ_{i=0}^{s} η_i L_i
5    for all j s.t. μ_j ≠ 0 do L ← L ∪ R_j; R ← R \ R_j; od
     for all i s.t. λ_i ≠ 0 do L ← L ∪ (P_0 − P_i); P ← P \ P_i; od
     return ⟨P, R, L⟩;
   }
```

Fig. 4. Widening of a simplex with a *line* L

Considering widening a simplex with a *line* L, all the rays and point differences with non-zero coefficient, i.e. contributing to represent L, are widened to new lines, as described in detail in the algorithm in figure 4 . □

When using simplices, termination of the fixpoint algorithm over the constraint system $\mathbf{T}^\sharp$ need not be ensured by introducing additional widening. Since in $\mathbb{Q}^k$, $k = \mathcal{O}(n^2)$, a non-empty simplex can be enlarged at most $3k$-times, no infinite ascending chains may occur. Note, however, that due to the frequent computation of the enclosing

simplex, the fixpoint iteration over the constraint system $\mathbf{T}^\sharp$ based on simplices leads to a less precise approximation of convex sets than convex polyhedra.

Linear Guards. Since the class of simplices has been introduced in order to efficiently approximate convex polyhedra when computing the effects of procedures, it is not required to evaluate linear guards on simplices within our approach. Our reachability analysis relies on polyhedra, on which the conditions can be directly evaluated, v. section 4. When using simplices for the reachablity analysis, the evaluation of conditionals on simplices cannot be performed directly after each procedure call or when a condition is passed, because the result of an intersection is not necessarily again a simplex. Since the creation of an enclosing simplex after the condition evaluation will cause too much imprecision, checking the condition must be postponed until the end of the analysis. Thus, it is preferable to transform the simplex into a convex polyhedron and additionally perform the condition evaluation.

Further on, operations on simplices have a better runtime complexity than on polyhedra:

Theorem 2. *All the simplicial operations (subsumption, union, widening and composition) can be performed in a time, polynomial in the number of variables* n*.*

Proof. Assume that the simplices in question describe subsets of $\mathbb{Q}^k$ where $k = \mathcal{O}(n^2)$. The simplicial operations of *inclusion* testing and *widening* are reduced to solving a system of at most $k+1$ linear equations, which can be performed in $\mathcal{O}(k^3)$ for a simplex with $k+1$ frame elements. *Union* is reduced to $(k+1)$-times successive widening, *subsumption* to $(k+1)$-times inclusion tests. Thus, each operation can be performed in time $\mathcal{O}(k^4)$. The simplicial *composition* is given by the element-wise composition of the frame elements (i.e., $\mathcal{O}(k^2)$ matrix multiplications) and subsequently determining its enclosing simplex, leading to a total complexity of $\mathcal{O}(k^5)$. □

6 Preliminary experimental results

So far, we have introduced two different representations for convex sets – convex polyhedra and simplices. Even more, we have presented two alternatives for evaluating conditionals within the reachability analysis – directly after each procedure call or once at the end of the analysis. To get a general idea of the performance of these different options in practical application, we have examined the behaviour of our interprocedural approach on a collection of example programs. Here, we concentrate on three characteristic examples, *recursive add, array bounds* and *nested loops*.

The example program *recursive add* contains a procedure, that recursively calls itself, computing the addition of two numbers. Furthermore, in *array bounds* array bound checking, as done by Java programs, is simulated. Finally, we consider the iteration variables in the program *nested loops*, containing four nested `for`-loops. This program also covers the case that a loop is bounded by the iteration variable of an outer loop.

The analysis set-up consists of approximating convex sets either by convex polyhedra or simplices and trying either direct condition evaluation or a single evaluation at the end of the analysis. Our prototypical implementation is more complex than the theoretical analysis described in this paper, as it deals with local variables, passing of parameters and return values in procedures.

The following chart compares the effect analysis by means of convex polyhedra and with simplices for each example program:

Program	LOC	# Procedures	Increase in efficiency	Precision
recursive add	26	4	62 %	100 %
array bounds	25	2	97 %	100 %
nested loops	28	2	98 %	75 %

Table 1. Simplex compared to polyhedra

The *runtime* of the reachability analysis by means of convex polyhedra does not differ significantly from the reachability analysis by means of simplices. The effect analysis by means of simplices is, however, *dramatically faster* than the effect analysis by means of convex polyhedra, as the column "Increase in efficiency" of table 1 illustrates. Effect analysis with simplices has terminated in few seconds for all benchmarks.

Concerning the *precision* of the inferred inequalities, we find that both the approach via simplices and that via convex polyhedra have been able to infer the exact result for the recursive function in the case of *recursive add* and the dependence of the iteration variable from the variable upper bound for *array bounds*. Yet for the example program *nested loops* both approaches returned quite precise results. In this case, however, the analysis by means of simplices missed some lower loop bounds and thus did not reach the full accuracy of the analysis with polyhedra, cf. table 1.

Since the analysis using simplices is rather fast and the quality of the inferred inequalities is not too imprecise, we conclude that it might be a good compromise to rely on simplices for the effect analysis, and to resort to convex polyhedra or other approximations of convex polyhedra (e.g. octahedra from [3]) for the reachability analysis. Contrary to our theoretical expectations from section 4, no advantage could be observed of immediate condition evaluation over single evaluation at the very end of the analysis — but this may just be due to the perhaps not very representative selection of benchmark programs.

7 Conclusion

We have introduced a general framework for interprocedurally identifying linear inequality relations between the variables of a program for each program point. This can be achieved by representing the effects of procedures with convex sets of transition matrices. Within our approach we accumulate the single edge effects in order to describe the effect of a whole procedure. These procedure effects can be simply embedded into a reachability analysis by means of arbitrary approximations of convex polyhedra.

In the absence of conditional branching the convex abstraction can be characterised precisely by the least solution of a constraint system. In order to handle conditional branching within our framework, we propose to store the value of each conditional in an auxiliary variable during effect analysis and postpone the evaluation up to the reachability analysis. This postponement is safe, merely leading to an over-approximation.

In order to finitely represent and compute with convex sets, we approximate them by means of convex polyhedra. We resort to the frame representation of polyhedra, thus avoiding the expensive continual conversion between the two representations. The frame representations of convex polyhedra, on the other hand, can be exponentially

larger than their constraint representations. For this reason, we propose the subclass of *simplices* as an abstract domain Since for simplices, the number of frame elements is restricted, we obtain small representations for convex sets. Moreover, the basic operations on simplices can be performed in polynomial time. Thus, our effect analysis by means of simplices runs in polynomial time, more precisely, the analysis is linear in the program size and polynomial in the number of program variables and guards.

First practical experiments indicate that this approach is quite efficient and provides reasonably precise results. In contrast to convex transition invariants, our interprocedural analysis is able to yield the exact invariant $x = 2$ for program point r_m in figure 1. It remains for future work to examine the scalability of our approach for larger and more realistic benchmark programs. If, however, the complexity for larger programs prevents a practical application of our approach, *clustering* , as introduced in Astrée [2], could be included.

References

1. R. Bagnara, E. Zaffanella, P. M. Hill, and E. Ricci. Precise widening operators for convex polyhedra. In *10th International Static Analysis Symposium (SAS)*, pages 337–354, 2003.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 196–207, 2003.
3. R. Clarisó and J. Cortadella. The Octahedron abstract domain. In *11th International Static Analysis Symposium (SAS)*, pages 312–327, 2004.
4. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
5. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th Ann. ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–97, 1978.
6. Z. Manna and J. McCarthy. Properties of programs and partial function logic. *Machine Intelligence, 5:2737*, 1970.
7. A. Miné. The Octagon abstract domain. In *Analysis, Slicing, and Transformation (AST)*, pages 310–319, 2001.
8. M. Müller-Olm and H. Seidl. Program analysis through linear algebra. In *31th Ann. ACM Symposium on Principles of Programming Languages (POPL)*, 2004.
9. M. Müller-Olm and H. Seidl. A generic framework for interprocedural analysis of numerical properties. In *12th Static Analysis Symposium (SAS)*, pages 235–250, 2005.
10. S. Sankaranarayanan, M. Colon, H. Sipma, and Z. Manna. Efficient strongly relational polyhedral analysis. In *7th International Conference, Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2006.
11. S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint based linear relations analysis. In *11th International Static Analysis Symposium (SAS)*, pages 53–68, 2004.
12. A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., 1986.
13. A. Simon, A. King, and J. M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In *Logic Based Program Development and Transformation (LOPSTR)*, pages 71–89, 2002.