Geometric Quantifier Elimination Heuristics for Automatically Generating Octagonal and Max-plus Invariants*

Deepak Kapur[1], Zhihai Zhang[2], Matthias Horbach[1], Hengjun Zhao[3], Qi Lu[1], and ThanhVu Nguyen[1]

[1] Department of Computer Science, University of New Mexico, Albuquerque, NM, USA
[2] School of Mathematical Sciences, Peking University, Beijing, China
[3] Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract. Geometric heuristics for the quantifier elimination approach presented by Kapur (2004) are investigated to automatically derive loop invariants expressing weakly relational numerical properties (such as $l \leq x \leq h$ or $l \leq \pm x \pm y \leq h$) for imperative programs. Such properties have been successfully used to analyze commercial software consisting of hundreds of thousands of lines of code (using for example, the Astrée tool based on abstract interpretation framework proposed by Cousot and his group). The main attraction of the proposed approach is its much lower complexity in contrast to the abstract interpretation approach ($O(n^2)$ in contrast to $O(n^4)$, where n is the number of variables) with the ability to still generate invariants of comparable strength. This approach has been generalized to consider disjunctive invariants of the similar form, expressed using maximum function (such as $\max(x+a, y+b, z+c, d) \leq \max(x+e, y+f, z+g, h)$), thus enabling automatic generation of a subclass of disjunctive invariants for imperative programs as well.

1 Introduction

In [23, 22], Kapur proposed an approach based on quantifier elimination for generating program invariants in general and loop invariants in particular. Depending upon the formulas of interest to serve as invariants at a program location, parametric formulas are identified such that when those parameters are fully instantiated, the results are the desired invariants. As an example, if a goal is to discover linear inequalities as the invariants, then the corresponding parametric form is $a \cdot x + b \cdot y + c \cdot z \leq d$, where x, y, z are program variables and a, b, c, d are parameters. Notice that the parametrized form is not a linear inequality because of

* Partially supported by NSF grants CCF-0729097 and CNS-0905222, by a fellowship from the Postdoc Program of the German Academic Exchange Service (DAAD), and by EXACTA and the China Scholarship Council.

presence of terms like $a \cdot x, b \cdot y, c \cdot z$, but rather a nonlinear (quadratic) inequality. If the goal is to find quadratic inequalities (such as ellipsoid inequalities), then the associated parametric form is still a nonlinear inequality, albeit of degree 3. Once a parametric form for invariants of interest is identified, then Kapur's approach involves generating verification conditions using the parametrized formulas for each distinct program path and eliminating program variables from the verification conditions using quantifier elimination to produce constraints on parameters. For any given parameter value satisfying the resulting constraints, the verification conditions instantiated with these parameter values then become valid, implying that the corresponding instantiation of the parametrized invariant is indeed a program invariant.

The main contribution of the paper is the development of an efficient geometric local heuristic for a restricted version of quantifier elimination over a subset of parametrized linear formulas so as to generate octagonal invariants and max-plus invariants for programs. The quantifier elimination problem of interest is of the form

$$\forall x_1, \ldots, x_n \; \Phi(p_1, \ldots, p_m, x_1, \ldots, x_n),$$

where $p_1, \ldots p_m$ are parameters, $x_1, \ldots, x_n$ are program variables, and Φ is a verification condition, a quantifier-free formula over $p_1, \ldots, p_m$ and $x_1, \ldots, x_n$. The objective is to generate a nontrivial quantifier-free formula over $p_1, \ldots, p_m$ that implies $\forall x_1, \ldots, x_n \; \Phi$. We focus on two types of parametric formulas: (i) a parametric formula obtained using a conjunction of atomic formulas of the form $l \leq x$, $x \leq h$, $l \leq x+y$, $x+y \leq h$, $l \leq x-y$, and $x-y \leq h$, where x, y are program variables and l, h are parameters, and (ii) a parametric formula obtained using a limited form of disjunctions of conjunctions of formulas of the form $l \leq x$, $x \leq h$, $l \leq x - y$, and $x - y \leq h$, which are equivalent to a pure conjunction of formulas of the form $\max(x_1 + a_1, x_2 + a_2, a_0) \geq \max(x_1 + b_1, x_2 + b_2, \ldots, x_k + b_k, b_0)$, where a_i, b_j are parameters. Such invariants have been found to be very effective in detecting bugs in commercial software for flight control and related embedded systems for memory violation [1] and numerical errors using tools such as Astrée [9].

Given that quantifier elimination is in general computationally a highly expensive operation (either undecidable or doubly exponential) and furthermore, outputs generated by complete quantifier elimination algorithms are huge, we address both of these problems by exploring a local incomplete quantifier-elimination geometric heuristic which considers formulas with constant number of variables (typically two variables) as well as which is geometrically based resulting in manageable outputs by focusing on relevant cases. As shown later in the paper, this quantifier elimination heuristic results in generating program invariants of strength and quality comparable to those obtained using the methods based on the abstract interpretation framework as discussed in [25], but with a much lower asymptotic complexity—$O(n^2)$ in the number of program variables in comparison to $O(n^4)$ for algorithms based on the abstract interpretation approach.

A fascinating aspect of our approach is that for octagonal invariants, since the parametric form is fixed and determined by the number of program variables, it is possible to develop local heuristics focusing on quantifier elimination to a pair of distinct variables. By analyzing different kinds of assignment statement, we have developed an approach for quantifier elimination using table look-ups based on the presence (or absence) of parameter-free atomic formulas (corresponding to various sides of octagons) appearing in a program path. Using these tables, it is possible to identify how these atomic formulas in a program path restrict octagonal invariants for various kinds of assignments on program variables. Parameter constraints generated by the quantifier-elimination heuristic can also be decomposed into subsets of constraints on at most four parameters, resulting in very efficient algorithms for a family of invariants of different quality and even generating the strongest possible invariants.

In our analysis, a formula over an arbitrary number of variables is decomposed into subformulas on a fixed number of variables. As a result, efficient heuristics can be designed exploiting the structure of these subformulas. This paper reports a few such heuristics we have developed; many more are still possible and are being explored.

A major advantage of the proposed approach is that it is highly parallelizable, which is especially good for scalability, since most of its steps can be done in parallel:

- Analysis for different program paths can be done in parallel.
- Table look-up for each of the tests to generate constraints on parameters can be done in parallel.
- Parameter constraints can be analyzed in parallel by decomposing them into blocks of constraints on a fixed number of variables.
- Generation of the strongest invariant after computing maximum lower bounds and minimum upper bounds on parameters can also be derived in parallel.

The sequential bottleneck in the analysis is the derivation of implicit tests from the tests appearing in a program path.

The paper is organized as follows: In the next subsection, we briefly review related work on the generation of octagonal and max plus invariants. This is followed by a high level comparison of the quantifier elimination approach and fixed point approaches for generating program invariants, with a particular focus on the abstract interpretation approach. We discuss the strength and quality of invariants generated by these approaches. Section 2 focuses on octagonal invariants. Section 3 reports our preliminary investigations for generating disjunctive invariants expressed using a conjunction of max-plus constraints. Section 4 briefly discusses future work.

1.1 Related Work

Quantifier elimination approaches for static program analysis have been investigated in many different ways, particularly for generating linear inequalities based

on Farkas's lemma [26], using linear constraints and skeletons [15, 16], program synthesis [14, 30, 28], termination of programs using linear and nonlinear ranking functions [7, 32, 31], as well as analysis of hybrid systems [29, 21].

The most popular approach for automatically generating invariants is using the abstract interpretation framework pioneered by Cousot and Cousot [8]. This research direction has resulted in very powerful tools, Astrée and its descendants, which have been used in finding bugs in commercial software, and a related set of experimental freely available tools (including Interproc [20], the tool we use in this paper for comparative purposes because it was designed for similar programs). One of the main reasons for the success of the abstract interpretation approach on real large numerical software is a collection of efficient algorithms designed for various operations for different abstract domains. Two Ph.D. theses by Miné [25] and Allamigeon [1] are the closest to the results presented in this paper. Miné's thesis focused on *weakly relational numerical* abstract domains where program invariants are specified using octagonal constraints (or a subset of octagonal constraints). Allamigeon's thesis considered max-plus invariants. As pointed out by Miné, the use of linear inequalities as an abstract domain proposed in [10] does not scale because of the exponential complexity of algorithms needed to perform abstract domain operations on convex polyhedra including conversion back and forth between their representation as a conjunction of linear inequalities and the generator (frame) representation.

Octagonal constraints (also called unit two variable per inequality or UTVPI constraints) have been extensively investigated. Octagonal constraints are also interesting to study from a complexity perspective and are a good compromise between interval constraints and linear constraints. Linear constraint analysis over the rationals ($\mathbb{Q}$) and reals ($\mathbb{R}$), while of polynomial complexity, has been found in practice to be inefficient and slow, especially when the number of variables grows [25, 9], since it must be used repeatedly in an abstract interpretation framework. Often, we are however interested in cases when program variables take integer values bound by computer arithmetic. If program variables are restricted to take integer values (which is especially the case for expressions serving as array indices and memory references), then octagonal constraints are among the most expressive fragments of linear (Presburger) arithmetic over the integers with a polynomial time complexity. It is well known that extending linear constraints to have three variables even with unit coefficients (i.e., ranging over $\{-1, 0, 1\}$) makes checking their satisfiability over the integers NP-complete [19, 27]; similarly, restricting linear arithmetic constraints to be just over two variables, but allowing non-unit integer coefficients of the variables also leads to the satisfiability check over the integers being NP-complete. The Floyd-Warshall algorithm, which is typically used to analyze the difference bound matrices representation of octagonal constraints (see [25]), must be extended for computing integral closure if variables are over the integers; see [3], where an $O(n^3)$ algorithm for computing the tight closure of octagonal constraints over the integers is presented that exploits integrality of constraints.

Max-plus constraints, i.e. constraints of the form $\max(x_1+a_1, \ldots, x_n+a_n, c) \leq \max(x_1+b_1, \ldots, x_n+b_n, d)$, have been investigated in [4, 6], and are an active area of research in combinatorics. They were first used for program analysis by Allamigeon et al. [2], who realized their value as an abstract domain that can express certain nonconvex sets, the so-called max-plus polyhedra, without the need for heuristics on how to manage the number of disjunctive components. Like classical convex polyhedra, (bounded) max-plus polyhedra can be equivalently represented by a set of constraints or by a set of extremal points. As in the classical case, the conversion between these representations is notoriously expensive. E.g. finding extremal points, which are usually called generators, is exponential in the number of constraints. Even a single inequation in n dimensions can give rise to quadratically many generators. The algorithms by Allamigeon et al. work purely on generators instead of constraints. In our quantifier-based approach, we will also restrict our attention to max-plus polyhedra represented by sets of generators.

1.2 On the Quality of Invariants Generated using Quantifier Elimination

We briefly compare the quantifier elimination approach for generating inductive invariants as proposed in [23, 22] to other approaches based on fixed point algorithms, in particular the abstract interpretation framework pioneered by [8, 10, 25, 1].

An invariant at any location in a program captures a superset of the states reached (also called reachable states) whenever program control passes through that particular location. The strongest possible invariant at any location is thus simply a disjunction specifying that the state at that location is one of these reachable states. If a location is visited finitely many times, then this disjunction is a formula as long as a single reachable state can be precisely characterized by a formula in a first-order theory that is expressive enough (provided the set of initial states of the program can be described in such a way). However, if a location is visited infinitely often in the case of a nonterminating program, then this set of infinitely many states must be specified by a finite formula in a richer language with interpreted function symbols; this may often be an approximation in the sense that the set of states by which such a formula is satisfied is typically a superset of the reachable states at the location. For every program path through the location, the effect of the statements on that path preserve the reachable set of states at the location. Given a set S of states, a formula ϕ is the strongest in a theory (such as Presburger arithmetic, Tarski's theory of real closed fields, the theory of polynomial equalities over an algebraically closed field, etc.) specifying S iff there is no other formula γ not equivalent in the theory to ϕ such that (i) $\gamma \implies \phi$ and (ii) γ is satisfied by every state in the set. Typically, formulas used for specifying states are quantifier-free as quantified formulas are difficult to analyze.

In the abstract interpretation approach, concrete states are abstracted to abstract states using an abstraction function and abstract states are specified

using the elements of an abstract domain, which is a lattice. In this context, each program variable, instead of taking a concrete value, may take an abstract value [8]. Concrete states and abstract states in their setting are related using a Galois connection. Examples of abstract values commonly used are the values of a variable being in an interval (zonal constraints), or in addition to being an interval, sum and difference of two different variables is also in an interval (octagonal constraint) [25], or more complex constraints on the values of variables including max-plus constraints [1] and linear constraints [10]; other domains have also been explored. An invariant is expressed as an abstract element or a set thereof and is computed by a terminating fixed point computation using a suitably defined widening operator (and narrowing operators); many heuristics have been proposed to improve the quality of the invariants computed using this approach [11, 5].

Elements in an abstract domain typically can be represented as a conjunction of atomic formulas over a suitable theory. For example, conjunctions of interval constraints, octagonal constraints, and max plus constraints can all be written in a small fragment of quantifier-free Presburger arithmetic, whereas a general conjunction of linear constraints uses full quantifier-free Presburger arithmetic (but without disjunction and negation).

As stated above, the quantifier elimination approach for generating invariants at a program location hypothesizes invariants as formulas of a certain form, which can be parametrized. The intuition behind this approach is that for some parameter values, the resulting instantiated formula characterizes a superset of reachable states at the program location. In other words, when parameters in such a formula are fully instantiated, the resulting formula is in (a fragment of) the language used for writing the formulas, even though the parametrized formula may be in a richer language. In the case of octagonal constraints, a parametrized formula is a conjunction of atomic formulas of the form $l \leq e$ and $e \leq h$, where l, h are parameters, and e is a variable, the difference of two variables or the sum of two variables; in this case, both the parametrized formula as well as its instantiation are in Presburger arithmetic (a parametrized atomic formula is expressed using at most three variables whereas a nonparametrized atomic formula is expressed using at most two variables). However, for linear constraints, a parametrized formula is not in the language of Presburger arithmetic since the parametrized formula could be $ax + by + cz \leq d$, where a, b, c, d are parameters, but its instantiations are in Presburger arithmetic.

Verification conditions corresponding to paths through the given program location are then generated and program variables are eliminated from the verification conditions, giving rise to constraints on parameters. If there is an invariant of the hypothesized shape associated with the program location, then the quantifier elimination approach would find such an invariant assuming that the method for quantifier elimination is complete. Furthermore, if all the solutions of constraints on parameters resulting from a complete quantifier elimination method can be finitely described, then this approach produces the strongest invariant of such shape, implying that it is stronger than the invariant generated

by any other approach including the abstract interpretation framework. Even if the quantifier elimination method is incomplete (but sound), then its result would lead to constraints on parameters such that all parameter values that satisfy these constraints on parameters, will result in an invariant. In this sense, the quantifier elimination approach is the most general method for computing invariants of programs.

2 Octagonal Invariants

2.1 Overview

In this section, we propose a method based on quantifier elimination for automatically generating program invariants which are conjunctions of octagonal constraints over the integers as atomic formulas. Such an atomic formula is a lower bound on a program variable x, an upper bound on a program variable x, a lower bound on $x + y$, where x, y are program variables, an upper bound on $x + y$, and similarly, a lower bound on $x - y$ or an upper bound on $x - y$. The reader would notice that a negation of any of the above atomic formulas over the integers can also be easily expressed. An expression x, $x + y$ or $x - y$ need not have a lower bound or an upper bound; to allow such possibilities, it is convenient to extend $\mathbb{Z}$, the domain of integers, to include both $-\infty$ and $+\infty$ with the usual semantics (see the Appendix about how various arithmetic operations and ordering relations are extended to consider $-\infty$ and $+\infty$). This also ensures that a trivial invariant will always be generated by the proposed method, in which these expressions have $-\infty$ as the lower bound and $+\infty$ as the upper bound, much like the trivial invariant `true`.

Let us assume that invariants (which we will have to compute) are associated with sufficiently many program locations (usually it suffices to associate an invariant with every loop and the entry and exit of every procedure/method). Verification conditions are then generated for every possible program path among pairs of such invariants. In the case of nested loops, invariants must be associated with every loop.

Assuming n program variables $x_1, \cdots x_n$ appearing along a program path, an invariant $I(X)$ expressed as a conjunction of octagonal constraints is of the form:

$$I(X) = \wedge_{1 \leq i < j \leq n} octa(x_i, x_j),$$

where X is $\{x_1, \ldots, x_n\}$, $octa(x_i, x_j)$ is a conjunction of atomic formulas discussed above and expressed using program variables $x_i, x_j, i \neq j$. A typical verification condition $\Phi(X)$ corresponding to a program path in a loop expressed using such invariants is:

$$(I(X) \wedge T(X)) \implies I(X'),$$

where X' contains the new values of the variables in X after all the assignments along the path, and $T(X)$ is a conjunction of all the loop tests and branch

conditions along the branch. Without any loss of generality we can and will assume that $T(X)$ is a conjunction of atomic formulas, as otherwise the verification condition can be split into a conjunction of several verification conditions, with each being considered separately. As an example, if a loop condition is $T(X) = T^1(X) \vee T^2(X)$, then the above subformula can be split into:

$$(I(X) \wedge T^1(X)) \implies I(X') \wedge$$
$$(I(X) \wedge T^2(X)) \implies I(X') \ .$$

It is also assumed in the analysis below that all branches indeed participate in determining the program behavior, i.e. there is no dead branch which is never executed for the initial states under consideration. Considering dead branches can unnecessarily weaken the invariants generated using the quantifier elimination approach by imposing unnecessary constraints on parameters.[4]

Assume a different parametrized loop invariant at the entry of every loop (and every function and procedure, if any, in a program). Given the fixed structure of octagonal constraints, this is relatively easy. The formula $octa(x_i, x_j)$ can be fully parametrized with 8 parameters, one parameter each for lower bound and upper bound for each of the two variables x_i, x_j, for the sum expression $x_i + x_j$ and the difference expression $x_i - x_j$. It is of the following form:

$octa(x_i, x_j) \triangleq (l_1 \leq x_i - x_j \leq u_1 \wedge l_2 \leq x_i + x_j \leq u_2 \wedge l_3 \leq x_i \leq u_3 \wedge l_4 \leq x_j \leq u_4)$.[5]

So there are $\frac{n \cdot (n-1)}{2}$ pairs of variables, and there are total $2n \cdot (n-1) + 2n = 2n^2$ parameters for each loop invariant assuming all the variables are needed to specify the strongest possible loop invariant. We also have the constraints that $l_i \leq u_i, i = 1, 2, 3, 4$.

For generating invariants, all program paths must be considered. The initial state of a program, expressed by a precondition, as well as other initialization assignments to program variables, may impose additional constraints on parameters.

To ensure that the verification condition generated from any program path also has the same types of atomic formulas, it is assumed that tests (for a loop as

[4] This is a weakness of the quantifier elimination approaches in contrast to other approaches where dead code gets automatically omitted in the analysis. Incomplete but fast dead code detectors are however a standard component of the static analysis performed in state of the art integrated program development environment including ECLIPSE (JAVA/C^{++}) and Microsoft Visual Studio, and can be switched on in the GNU Compiler Collection.

In our current implementation of the quantifier elimination approach, many dead branches are detected during the generation of the verification conditions, which tend to have trivially false antecedents for inexecutable paths.

[5] As the reader would have noticed, the closure of these constraints imposes a relationship among various parameters; for instance, lower and upper bounds on x_i, x_j can be deduced from the lower and upper bounds on $x_i - x_j$ and $x_i + x_j$. However, the most generic octagon still requires 8 parameters.

well as in a conditional statement) are of the same form. And assignment statements are of the form x := x+A, x := −x+A, and x := A, where A is a constant. Otherwise, tests and assignments must be approximated:

- An unsupported assignment is approximated to take an unknown value, i.e. for the variable x being assigned and any other variable y: $-\infty \leq x \leq +\infty$, $-\infty \leq x - y \leq +\infty$ and $-\infty \leq x + y \leq +\infty$.
- An unsupported test of a loop can be approximated to be both `true` and `false`, i.e. the loop can be arbitrarily continued or left after each iteration.
- An unsupported test of a conditional can also be approximated to be both `true` and `false`, i.e. both branches can always be executed.

2.2 Program Analysis using Octagonal Invariants

We now present our method for generating program invariants with octagonal constraints.

0. Associate a parametrized octagonal invariant with every loop entry as well as with the entry and exit of every function/procedure.
1. Within a function/procedure, for every program path, generate a verification condition from program invariants at every loop entry. This can be done by standard methods like the computation of weakest preconditions for each path using Hoare logic [18].
2. If the resulting verification condition cannot be expressed such that all atomic formulas are octagonal constraints and all assignments are of one of the supported forms, approximate them as detailed above (Section 2.1). This approximation is standard in program analysis. In this paper, we assume for simplicity that we do not have to perform any approximations for tests or assignments.
3. Eliminate the quantifiers from each verification condition. This results in a set of constraints on the parameters of the involved invariants. To keep the quantifier elimination procedure fast (quadratic in the number of program variables), the verification condition is approximated using a geometric heuristic (Section 2.5).
4. Take the union of all the constraint sets thus generated. Every parameter value that satisfies the constraints leads to an invariant. To accommodate program variables having no lower or upper bounds, parameters are allowed to have $-\infty$ and $+\infty$ as possible values. Because of this, an octagonal invariant is always generated, with the trivial invariant being the one where $-\infty$ and $+\infty$ serve as lower and upper bounds for every arithmetic expression $(x, x + y, x - y)$.

The remainder of this section is mainly devoted to the development of an efficient way to perform the quantifier elimination in Step 3: In Subsection 2.3, we will discuss ways to make the tests appearing in the verification condition leaner or richer, Subsections 2.4 and 2.5 contain an explanation of the rationale behind

our method, and in Sections 2.6–2.8, we will show how to perform the actual quantifier elimination efficiently using a series of simple table look-ups.

Our approach does not involve any direct fixed point computation. The analysis is done only once for every program branch, in contrast to the abstract interpretation approach which requires the analysis to be done multiple times, depending upon the nature of the widening operator used for a particular abstract domain to ensure the termination of the fixed point computation. Furthermore, much like traditional Floyd-Hoare analysis, derivation of invariants is done without making any assumption about the termination of programs, which is handled separately. As illustrated below, our approach can derive invariants of nonterminating programs as well and can thus be effective for nonterminating reactive programs as well.

2.3 Trivially Redundant and Implicit Conditions

The formula $T(X)$, which is a conjunction of test conditions along a program path, can contain trivially redundant constraints and it can imply additional constraints (including their unsatisfiability). By trivially redundant constraints, we mean multiple constraints on the lower bound (upper bound) of the same expression (such as $5 \leq x - y$ as well as $4 \leq x - y$) out of which the respective greatest lower bound (the respective least upper bound) only needs to be retained; such trivially redundant constraints can be removed in $O(m)$, where m is the number of such constraints. It can be shown easily that if any trivially redundant constraints were retained and subsequently used to generate parameter constraints from the tables discussed in the later subsections, such parameter constraints will also be trivially redundant, without affecting the loop invariants generated. Henceforth, we will always assume that trivially redundant constraints are removed, such that e.g. a set of octagonal constraints between two variables contains at most 8 constraints.

Since our method for quantifier elimination is driven by parameter-free constraints in a verification condition, it is useful to derive implicit constraints from $T(X)$. Checking for satisfiability as well as deriving additional constraints can require in the worst case $O(n^3)$ steps [3] due to the use of the cubic Floyd-Warshall normalization algorithm, where n is the number of program variables appearing in these constraints. As an illustration,

$$x + y \geq 1,\ z - y \geq 2,\ z \leq 1$$

gives $-y \geq 1$ from $z - y \geq 2 \wedge z \leq 1$, which with $x + y \geq 1$ results in $x \geq 2$.

By localizing the derivation of additional constraints by considering each pair of variables, constant time is needed; this implies that for $O(n^2)$ pairs of variables, the derivation of implicit constraints requires $O(n^2)$ steps. This preprocessing is performed for each pair of variables in random order, since when performed sequentially, the order can affect the output of the results. In the above illustration, picking y, z led to an additional constraint on y which with interaction with constraints on x, y led to an additional constraint on x. If instead the pair x, y had been picked first, then the additional implicit constraint

on x would have been missed by this localized closure. Heuristics can be developed to come up with a good order such that the localized closure produces results which are a good approximation of the global closure. As will be shown below, such localized closure of constraints will keep the complexity of the loop invariant generation method quadratic in the number of program variables.

2.4 Localized Quantifier Elimination

Our goal is to efficiently generate sufficient conditions on parameters so that the verification condition $\Phi(X)$ is satisfied by all parameter values satisfying these conditions (the soundness condition). Of course, it is most desirable to generate as strong an approximation as possible to the quantifier-free formula on parameters equivalent to $\forall X\ \Phi(X)$. Note that Φ is a conjunction of clauses of the form $I\wedge$ Let $\phi_{i,j}$ be the subformula of Φ expressed only using program variables x_i, x_j. E.g. if

$$\Phi = (x_1 \leq u^1 \wedge x_2 \leq u^2 \wedge x_2 \leq u^3) \implies (x_1 \leq 1 + u^1 \wedge x_2 \leq -l^2 \wedge x_2 \leq u^3)\ ,$$

then

$$\phi_{1,2} = (x_1 \leq u^1 \wedge x_2 \leq u^2) \implies (x_1 \leq 1 + u^1 \wedge x_2 \leq -l^2)\ .$$

Given the structure of Φ, it is easy to see that

$$[\wedge_{1\leq i\neq j\leq n}(\forall x_i, x_j\ \phi_{i,j})] \equiv [\forall X\ \Phi(X)]\ .$$

The following theorem enables us to factor quantifier elimination of $\forall X\ \Phi(X)$ by considering subformulas $\forall x_i, x_j\ \phi_{i,j}$ corresponding to a single pair of distinct variables in Φ, generating sufficient conditions on parameters for the subformula (soundness requirement on quantifier elimination heuristic), and then doing a conjunction of such conditions on parameters for every subformula on every possible pairs of variables. The result is then a sufficient condition for the verification condition $\forall X\ \Phi(X)$.

Theorem 1. *Let $pc_{i,j}$ be a quantifier-free formula on parameters in $\phi_{i,j}$ such that for every possible parameter assignment σ, if σ satisfies $pc_{i,j}$, then σ satisfies $\forall x_i, x_j\ \phi_{i,j}$. Then any parameter assignment that satisfies $\wedge_{1\leq i<j\leq n} pc_{i,j}$ also satisfies $\forall X\ \Phi(X)$.*

Proof. The proof follows from $\wedge_{1\leq i<j\leq n}(\forall x_i, x_j\ \phi(x_i, x_j))$ being equivalent to $\forall X\ \Phi(X)$. □

The above theorem enables localizing quantifier elimination from a formula of arbitrary size to a formula of fixed size: the size of $\forall x_i, x_j\ \phi(x_i, x_j)$ is determined by the parameter-free part, which is a conjunction of tests along a program path; other than this subformula, the hypothesis is a conjunction of 8 atomic formulas with parameters and the conclusion is also a conjunction of 8 atomic formulas with parameters. For such a formula, quantifier elimination can be performed in constant time. In contrast, the worst case complexity of a complete quantifier

elimination for linear constraints is exponential in the number of quantifiers alternations and doubly exponential in the number of quantified variables [24]. Below, we will focus on the subformula $\phi_{i,j}$ in the verification condition $\Phi(X)$ and discuss quantifier elimination of x_i, x_j from $\phi_{i,j}$. To make the presentation free of subscripts, we will replace x_i, x_j by x, y, and henceforth call this verification condition $\phi(x, y)$.

2.5 Geometric View of Quantifier Elimination

The subformula $\phi(x, y)$ is of the form

$$(octa(x, y) \wedge T(x, y)) \Rightarrow octa(x', y') \ ,$$

where x' and y' are the values of x and y after all the assignments on a program path have been executed and $T(x, y)$ is the conjunction of all the atomic formulas obtained by the localized closure of parameter-free octagonal constraints on x, y, resulting from the loop tests and branch conditions in the program path (after being appropriately modified due to assignment statements between any two tests).

As discussed above, the parametrized invariant $octa(x, y)$ specifies an octagon with 8 sides corresponding to each of the atomic constraints in $octa(x, y)$. In order to ensure that the verification condition $\phi(x, y)$ consists only of octagonal constraints, there can only be four different possibilities about the cumulative effect of all assignments of x, y along a program path (the fourth possibility is covered by the case 3 below by switching x and y).

Possibility 1 x := −x+A and y := −y+B,
Possibility 2 x := x+A and y := y+B,
Possibility 3 x := −x+A and y := y+B,

where A, B are constants.[6] Each of these possibilities gives rise to a transformed octagon $I(x', y')$; both the original (white) octagon $I(x, y)$ and the transformed (shaded) octagon $I(x', y')$ are depicted in Figures 1–3 on the following pages, corresponding to the three possibilities. The subformula $\phi(x, y)$ is:

$$\begin{aligned} &\big(l_1 \leq x - y \leq u_1 \wedge l_2 \leq x + y \leq u_2 \wedge l_3 \leq x \leq u_3 \wedge l_4 \leq y \leq u_4\big) \wedge T(x, y) \\ &\quad \Longrightarrow \ \big(l_1 \leq x' - y' \leq u_1 \wedge l_2 \leq x' + y' \leq u_2 \wedge l_3 \leq x' \leq u_3 \wedge l_4 \leq y' \leq u_4\big) \, , \end{aligned}$$

with $l_1, u_1, l_2, u_2, l_3, u_3, l_4, u_4$ as parameters. The goal is to eliminate program variables x, y from $\phi(x, y)$ and efficiently generate the strongest possible approximation of an equivalent quantifier-free formula on the parameters l_1, l_2, l_3, l_4 and u_1, u_2, u_3, u_4 to $\forall x, y \ \phi(x, y)$. The main requirement on the result is that it is a sound underapproximation in the sense that we may miss some valid invariants, but the method does not yield any formulas that are invalid invariants.

[6] Constant assignments of the form x := A or y := B are handled similarly. Their discussion is omitted due to lack of space.

It should be first observed that the hypothesis in $\phi(x, y)$ is a conjunction of a parametrized octagon $octa(x, y)$ and a (partial) concrete octagon $T(x, y)$; thus it corresponds to the intersection of these octagons. The conclusion $octa(x', y')$ is also a parametrized octagon; it is a displaced version of the octagon in the hypothesis. Constraints on parameters $l_1, l_2, l_3, l_4, u_1, u_2, u_3, u_4$ that ensure that the intersection octagon $octa(x, y) \wedge T(x, y)$ being contained in $octa(x', y')$ is a good approximation of a quantifier-free formula equivalent to $\forall x, y\ \phi(x, y)$.

In each subsection corresponding to one of the possibilities, using local geometric analysis, we consider one by one, every side of the concrete octagon (and hence every possible lower bound and upper bound on $x - y, x + y, x, y$ in T) to see how it rules out the portion of the parametrized octagon $octa(x, y)$ not included in $octa(x', y')$. This is ensured in two parts. Conditions on A, B must be identified for each of the above three possibilities such that there is an overlap between the transformed octagon and the original octagon; often, this overlap can be ensured by making a few of the sides to be at $-\infty$ (or $+\infty$) (the case of when all sides have to be unbounded, leads to a trivial invariant, similar to the invariant `true` for any loop).

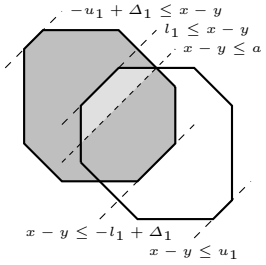
Depending upon the presence or absence of a side in the concrete octagon defined by T (i.e., a bound in T), parameter values are constrained by A or B. As an example, in Figure 1, if there is no upper bound constraint on $x - y$ in T, then $u_1 \leq (-l_1 + \Delta_1)$ (where $\Delta_1 = A - B$) ensures that the $x - y$ side of the original octagon is contained in the corresponding inverted side in the displaced octagon; we can also see this from $(x - y \leq u_1) \Rightarrow (l_1 \leq -x + y + \Delta_1)$ which is equivalent to $(x - y \leq u_1) \Rightarrow (x - y \leq -l_1 + \Delta_1)$. In the presence of an upper bound constraint of the form $x - y \leq a$ in T, both $a \leq u_1$ and $a \leq (-l_1 + \Delta_1)$ can be used to prune the original octagon.

These constraints on parameter values are derived below once and for all, and a table is constructed corresponding to each of the above three possibilities (see tables in Figures 1, 2, 3). For each possible bound, there is a table entry depending upon whether that bound is present or absent in T.[7] To generate a quantifier-free formula pc for the above verification condition $\phi(x, y)$, it suffices to take a conjunction of all table entries corresponding to the absence or presence of the bounds for $x, y, x - y$, and $x + y$ in T, where each table entry specifies a constraint on a parameter.

For example, when the signs of both variables are reversed in an assignment x := −x+A, y := −y+B (Figure 1) and a constraint $x \leq 5$ is present, the constraint $e \leq A - l_3$ is generated. When the signs of both variables are reversed in an assignment x := x+A, y := y+B (Figure 2) and no constraint of the form $x \leq e$ is present, either the constraint $u_3 = +\infty$ is generated (if $A > 0$) or no constraint is generated (if $A \not> 0$).

Below we show the derivation for the entries of the table in Figure 1. The tables in Figures 2 and 3 can be constructed analogously. For possibility 1, the assignment is x := −x+A, y := −y+B; for possibility 2, x := x+A, y := y+B; for possibility 3, x := −x+A, y := y+B. As should be evident from the table entries,

[7] This is why implicit constraints from T become relevant.



constraint	present	absent
$x - y \leq a$	$a \leq \Delta_1 - l_1$	$u_1 \leq \Delta_1 - l_1$
$x - y \geq b$	$\Delta_1 - u_1 \leq b$	$\Delta_1 - u_1 \leq l_1$
$x + y \leq c$	$c \leq \Delta_2 - l_2$	$u_2 \leq \Delta_2 - l_2$
$x + y \geq d$	$\Delta_2 - u_2 \leq d$	$\Delta_2 - u_2 \leq l_2$
$x \leq e$	$e \leq A - l_3$	$u_3 \leq A - l_3$
$x \geq f$	$A - u_3 \leq f$	$A - u_3 \leq l_3$
$y \leq g$	$g \leq B - l_4$	$u_4 \leq B - l_4$
$y \geq h$	$B - u_4 \leq h$	$B - u_4 \leq l_4$

Fig. 1. Signs of x and y are reversed: Constraints on Parameters

given two constraints on an expression (x, $x + y$, $x - y$) such that one of them is trivially redundant (e.g., $x - y \leq 4$ and $x - y \leq 10$), the corresponding entry in a table to the trivially redundant constraint is also redundant.

2.6 Reversal of The Signs of Both Variables

The verification condition for this case is:

$$(I(x, y) \wedge T(x, y)) \Rightarrow I(-x + A, -y + B),$$

which is

$$\begin{pmatrix} l_1 \leq x - y \leq u_1 \wedge l_2 \leq x + y \leq u_2 \\ \wedge\, l_3 \leq x \leq u_3 \wedge l_4 \leq y \leq u_4 \end{pmatrix} \wedge T(x, y)$$
$$\Rightarrow \begin{pmatrix} \Delta_1 - u_1 \leq x - y \leq \Delta_1 - l_1 \wedge \Delta_2 - u_2 \leq x + y \leq \Delta_2 - l_2 \\ \wedge\, A - u_3 \leq x \leq A - l_3 \wedge B - u_4 \leq y \leq B - l_4 \end{pmatrix},$$

where $\Delta_1 = A - B$ and $\Delta_2 = A + B$.

For $x - y$, the original octagon has l_1 and u_1 as lower and upper bound, respectively, whereas the transformed octagon has $-u_1 + \Delta_1$ and $-l_1 + \Delta_1$ as the lower and upper bounds. In the absence of any bounds on $x - y$ in T, $u_1 \leq \Delta_1 - l_1$ and $\Delta_1 - u_1 \leq l_1$ have to hold for the transformed octagon to include the original octagonal.

A concrete upper bound a on $x - y$ in T however changes the constraint on the parameter l_1: For the transformed octagon to include the original octagonal, $\Delta_1 - l_1$ has to be greater than or equal to one of the upper bounds u_1 and a of the original, corresponding to a constraint $a \leq u_1 \vee a \leq \Delta_1 - l_1$. Using such disjunctive constraints would directly lead to a combinatorial explosion of the analysis. Instead, we decided to only admit $a \leq \Delta_1 - l_1$ to the table, a safe over-approximation which reflects that in practice, tests the programmer specifies are actually relevant to the semantics of the program. Similarly, a concrete lower bound b on $x - y$ in T changes the constraint for u_1 to $\Delta_1 - u_1 \leq b$.

A similar analysis can be done for $x + y$: the hypothesis has $l_2 \leq x + y \leq u_2$ and the conclusion includes: $\Delta_2 - u_2 \leq x + y \leq \Delta_2 - l_2$. In the absence of

any bound on $x + y$ in $T(x, y)$, the parameter constraints $\Delta_2 - u_2 \leq l_2$ and $u_2 \leq \Delta_2 - l_2$ will ensure that the corresponding side of the transformed octagon includes that of the original octagon. If T contains a concrete upper bound c on $x + y$, then $c \leq \Delta_2 - l_2$. If T has a concrete lower bound d, then $\Delta_2 - u_2 \leq d$ ensures that the transformed octagon includes the original octagon.

We will omit the analysis leading to entries corresponding to the presence (or absence) of concrete lower and upper bounds on program variables x, y, as it is essentially the same. The above analysis is presented in the table in Figure 1. Depending upon a program and cases, looking up the table generates constraints on parameters in constant time.

The following lemma states that the above quantifier elimination method is sound.

Lemma 2. *Given a test T in the subformula $\phi(x, y)$ in the verification condition, let pc be the conjunction of the parameter constraints corresponding to the presence (or absence) of each type of concrete constraint in T. For every assignment of parameter values satisfying pc, substitution of these values for the parameters makes $\forall x, y\ \phi(x, y)$ valid.*

Proof. Follows directly from the above computations. □

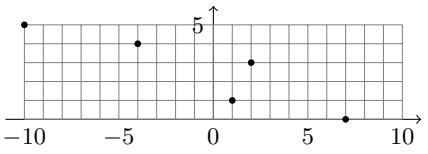
It thus follows:

Theorem 3. *Any assignment of parameter values satisfying pc in the above lemma generates $octa(x, y)$ as the subformula on program variables x, y serving as the invariant for the associated program path.*

We illustrate how the above table can be used to generate invariants for a program.

Example 4. Consider the following program:

```
x := 2; y := 3;
while (x+y ≥ 0) do
  if (y ≥ 2) then
    y := −y+4; x := −x+3;
  else
    x := −x−3; y := −y+5;
```



The actual state space for this program is depicted on the right. There are two branches in the loop body. For the first branch, the verification condition is:

$$\begin{aligned}&\left(l_1 \leq x - y \leq u_1 \wedge l_2 \leq x + y \leq u_2 \wedge l_3 \leq x \leq u_3 \wedge l_4 \leq y \leq u_4\right) \wedge T_1(x, y)\\ &\quad \Rightarrow \begin{pmatrix} -1 - u_1 \leq x - y \leq -1 - l_1 \wedge 7 - u_2 \leq x + y \leq 7 - l_2 \\ \wedge\, 3 - u_3 \leq x \leq 3 - l_3 \wedge 4 - u_4 \leq y \leq 4 - l_4 \end{pmatrix},\end{aligned}$$

where $A = 3$, $B = 4$, $\Delta_1 = -1$, $\Delta_2 = 7$, and $T_1(x, y) = x + y \geq 0 \wedge y \geq 2$. We approximate this verification condition by separating the components containing

common parameters:

$$\begin{aligned}
((l_1 \le x - y \le u_1) \wedge (x + y \ge 0 \wedge y \ge 2)) &\implies (-1 - u_1 \le x - y \le -1 - l_1)\\
((l_2 \le x + y \le u_2) \wedge (x + y \ge 0 \wedge y \ge 2)) &\implies (7 - u_2 \le x + y \le 7 - l_2)\\
((l_3 \le x \le u_3) \wedge (x + y \ge 0 \wedge y \ge 2)) &\implies (3 - u_3 \le x \le 3 - l_3)\\
((l_4 \le y \le u_4) \wedge (x + y \ge 0 \wedge y \ge 2)) &\implies (4 - u_4 \le y \le 4 - l_4)
\end{aligned}$$

The following parameter constraints are generated for branch 1:

1. Given no concrete lower or upper bounds on $x - y$ in $T_1(x, y)$, the table in Figure 1 generates the parameter constraints $u_1 \le -1 - l_1$ and $-1 - u_1 \le l_1$.
2. Since $x + y \ge 0$ in $T_1(x, y)$, the table entry corresponding to it is $7 - u_2 \le 0$. Since there is no concrete upper bound on $x + y$ specified in T_1, there is an additional constraint: $u_2 \le 7 - l_2$.
3. Similarly, absence of concrete upper or lower bounds on x in $T_1(x, y)$ gives $l_3 + u_3 = 3$.
4. For the concrete lower bound $y \ge 2$ and absence of any concrete upper bound on y in $T_1(x, y)$ the table entries are $4 - u_4 \le 2$ and $u_4 \le 4 - l_4$.

Similarly, for the second branch, the verification condition is:

$$\begin{aligned}
&\big(l_1 \le x - y \le u_1 \wedge l_2 \le x + y \le u_2 \wedge l_3 \le x \le u_3 \wedge l_4 \le y \le u_4\big) \wedge T_2(x, y)\\
&\quad \Rightarrow \begin{pmatrix} -u_1 - 8 \le x - y \le -l_1 - 8 \wedge -u_2 + 2 \le x + y \le -l_2 + 2 \\ \wedge\, -u_3 - 3 \le x \le -l_3 - 3 \wedge -u_4 + 5 \le y \le -l_4 + 5 \end{pmatrix},
\end{aligned}$$

Here $\Delta_1 = -8$ and $\Delta_2 = 2$. The tests along the branch are $(y + x) \ge 0 \wedge y \le 1$. Their closure is

$$T_2(x, y) = x + y \ge 0 \wedge y \le 1 \wedge x \ge -1 \wedge x - y \ge -2\ ,$$

because $x + y \ge 0 \wedge y \le 1 \implies x \ge -1 \wedge x - y \ge -2$. The following parameter constraints are generated for branch 2:

1. The only concrete constraint on $x - y \ge -2$ in $T_2(x, y)$ generates the parameter constraints $-8 - u_1 \le -2$ and $u_1 \le -8 - l_1$.
2. Similarly, the only constraint on $x + y \ge 0$ in $T_2(x, y)$ produces the parameter constraints $2 - u_2 \le 0$ and $u_2 \le 2 - l_2$.
3. Similarly, $x \ge -1$ in $T_2(x, y)$ generates $-3 - u_3 \le -1 \wedge u_3 \le -3 - l_3$.
4. Finally, the constraint $y \le 1$ in $T_2(x, y)$ generates the parameter constraints $-u_4 + 5 \le l_4$ and $1 \le -l_4 + 5$.

At the initial entry of the loop, $x = 2, y = 3$, which generates additional constraints on the parameters, like $l_3 \le 2 \le u_3$. Using these and the constraints computed from the two branches, the following parameter constraints are generated (after throwing away trivially redundant constraints on parameters):

$$\begin{aligned}
&l_1 \le -1 \wedge u_1 \ge -1 \wedge l_1 + u_1 \ge -1 \wedge l_1 + u_1 \le -8\\
&\wedge\, l_2 \le 5 \wedge u_2 \ge 7 \wedge l_2 + u_2 \le 2\\
&\wedge\, l_3 \le 2 \wedge u_3 \ge 2 \wedge l_3 + u_3 = 3 \wedge l_3 + u_3 \le -3\\
&\wedge\, l_4 \le 3 \wedge u_4 \ge 3 \wedge l_4 + u_4 \le 4 \wedge l_4 + u_4 \ge 5.
\end{aligned}$$

Any values of l_i, u_is that satisfy the above constraints result in an invariant for the loop. However, our goal is to generate the strongest possible invariant.

It is easy to see that the above constraints can be decomposed into disjoint subsets of constraints: (i) constraints on l_1, u_1, (ii) constraints on l_2, u_2, (iii) constraints on l_3, u_3, and finally, (iv) constraints on l_4, u_4, implying that each disjoint subset can be analyzed by itself. This is so because the table entries only relate parameters $l_i, u_i, 1 \leq i \leq 4$. This structure of parameter constraints is exploited later to check for satisfiability of parameter constraints to generate invariants including the strongest possible invariant.

Consider all the inequalities about l_1 and u_1, as an illustration:

$$l_1 \leq -1 \wedge u_1 \geq -1 \wedge l_1 + u_1 = -1 \wedge l_1 + u_1 \leq -8.$$

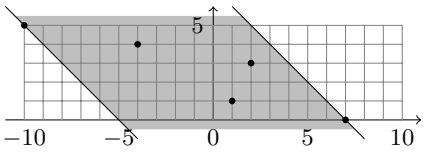
As the reader would notice there are no integers that satisfy the above constraints, since $-1 > -8$. But recall that we extended numbers to include $-\infty$ and $+\infty$ to account for no lower bounds and no upper bounds, and that both $\mathrm{x}(+\infty)+(-\infty) \leq a$ and $(+\infty)+(-\infty) \geq a$ hold for every integer a. Taking those observations into account, we find that $l_1 = -\infty$ and $u_1 = +\infty$ is a solution of this system. Similarly, $l_3 = l_4 = -\infty$ and $u_3 = u_4 = +\infty$ is obtained.

Finally, all the inequalities relating to l_2 and u_2 are as follows

$$l_2 \leq 5 \wedge u_2 \geq 7 \wedge l_2 + u_2 \leq 2.$$

Since we want to get as strong an invariant as possible, l_2 should be as large as possible and u_2 should be as small as possible. Hence, $l_2 = -5 \wedge u_2 = 7$, giving us the invariant

$$-5 \leq x + y \leq 7.^{8}$$



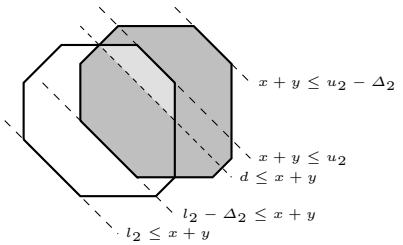
The reader should notice that we did not make use of the constraints $x \geq -1 \wedge x - y \geq -2$ obtained by closing the path constraints. So for this example, computing the closure did not change the constraints on parameters.

2.7 The Signs of Both Variables Remain Invariant

An analysis similar to the one discussed in the previous subsection is also done for the case when the effect of assignments on a pair of variables is of the form x := x+A and y := y+B. The verification condition $\phi(x, y)$ for this case is:

$$\begin{aligned}\big(l_1 \leq x - y \leq u_1 \wedge l_2 \leq x + y \leq u_2 \wedge l_3 \leq x \leq u_3 \wedge l_4 \leq y \leq u_4\big) \wedge T(x, y)\\ \Rightarrow \begin{pmatrix} l_1 - \Delta_1 \leq x - y \leq u_1 - \Delta_1 \wedge l_2 - \Delta_2 \leq x + y \leq u_2 - \Delta_2 \\ \wedge\, l_3 - A \leq x \leq u_3 - A \wedge l_4 - B \leq y \leq u_4 - B \end{pmatrix},\end{aligned}$$

[8] The Interproc tool produces the same invariant.



constraint	present	absent	side condition
$x - y \leq a$	$u_1 \geq a + \Delta_1$	$u_1 = +\infty$	$\Delta_1 > 0$
$x - y \geq b$	$l_1 \leq b + \Delta_1$	$l_1 = -\infty$	$\Delta_1 < 0$
$x + y \leq c$	$u_2 \geq c + \Delta_2$	$u_2 = +\infty$	$\Delta_2 > 0$
$x + y \geq d$	$l_2 \leq d + \Delta_2$	$l_2 = -\infty$	$\Delta_2 < 0$
$x \leq e$	$u_3 \geq e + A$	$u_3 = +\infty$	$A > 0$
$x \geq f$	$l_3 \leq f + A$	$l_3 = -\infty$	$A < 0$
$y \leq g$	$u_4 \geq g + B$	$u_4 = +\infty$	$B > 0$
$y \geq h$	$l_4 \leq h + B$	$l_4 = -\infty$	$B < 0$

Fig. 2. Signs of x and y do not change: Constraints on Parameters

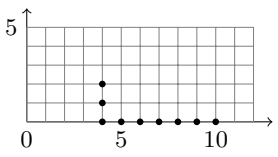
where $\Delta_1 = A - B$ and $\Delta_2 = A + B$. Figure 2 contains the transformed octagon (shaded) and the original octagon (white) corresponding to the octagonal invariant, and along with concrete lower and upper bounds on $x + y$ possibly appearing in T.

There are two main differences between this table and the previous table. Firstly, each table entry imposes constraints on a single parameter corresponding to an octagonal side. Secondly, for some sides of octagons, the original octagon does not have to be pruned. For instance, in this case when $\Delta_1 \leq 0$, an upper bound on $x - y$ does not make any difference; so there is no corresponding entry in the table.

We would like to point out similarities with the abstract interpretation approach, particularly how the widening operator is implicit in the quantifier elimination heuristic. Consider an entry in the table when $A > 0$. The condition $A > 0$ implies that if the path is executed several times, then x keeps increasing arbitrarily. In the absence of any upper bound on x in a test in the path, the value of x has no upper bound and u_3 has to be $+\infty$, similar to the widening operator of abstract interpretation. Similarly, if $\Delta_1 < 0$, then along that path, the value of $x - y$ keeps decreasing indefinitely unless again if there is a lower bound imposed by a test in the program path; the table entries correspond to both of these situations. We again illustrate the use of the above table to analyze the program below.

Example 5. Consider the following program:

```
x := 10; y := 0;
while (x−y ≥ 3) do
  if (x ≥ 5) then
    x := x−1;
  else
    y := y+1;
```



The state space for this program is depicted on the right. For initial values and the execution of the loop body, the following verification conditions are generated:

– Initial values: $l_1 \leq 10 \leq u_1 \wedge l_2 \leq 10 \leq u_2 \wedge l_3 \leq 10 \leq u_3 \wedge l_4 \leq 0 \leq u_4$.
– Branch 1:

$$\begin{aligned}&\big(l_1 \leq x-y \leq u_1 \wedge l_2 \leq x+y \leq u_2 \wedge l_3 \leq x \leq u_3 \wedge l_4 \leq y \leq u_4\big) \wedge T_1(x,y)\\ &\quad\Rightarrow \begin{pmatrix} l_1+1 \leq x-y \leq u_1+1 \wedge l_2+1 \leq x+y \leq u_2+1 \\ \wedge\, l_3+1 \leq x \leq u_3+1 \wedge l_4 \leq y \leq u_4 \end{pmatrix},\end{aligned}$$

where $\Delta_1 = -1, \Delta_2 = -1$ and $T_1(x,y) = x-y \geq 3 \wedge x \geq 5$.
– Branch 2:

$$\begin{aligned}&\big(l_1 \leq x-y \leq u_1 \wedge l_2 \leq x+y \leq u_2 \wedge l_3 \leq x \leq u_3 \wedge l_4 \leq y \leq u_4\big) \wedge T_2(x,y)\\ &\quad\Rightarrow \begin{pmatrix} l_1+1 \leq x-y \leq u_1+1 \wedge l_2-1 \leq x+y \leq u_2-1 \\ \wedge\, l_3 \leq x \leq u_3 \wedge l_4-1 \leq y \leq u_4-1 \end{pmatrix},\end{aligned}$$

where $\Delta_1 = -1$, $\Delta_2 = 1$ and $T_2(x,y) = x-y \geq 3 \wedge x \leq 4 \wedge y \leq 1 \wedge x+y \leq 5$.[9]

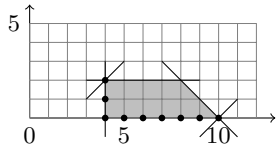
By combining the information from the initial condition with the constraints generated by the table in Figure 2, we get the following constraints on the parameters:

$$l_1 \leq 2 \wedge u_1 \geq 10 \wedge l_2 = -\infty \wedge u_2 \geq 10 \wedge l_3 \leq 4 \wedge u_3 \geq 10 \wedge l_4 \leq 0 \wedge u_4 \geq 2\ .$$

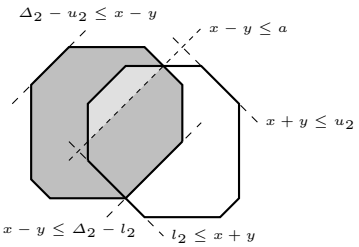
To generate the strongest possible invariant, the l_is should be as large as possible, and u_is should be as small as possible. The parameter values $l_1 = 2, u_1 = 10$, $l_2 = -\infty$, $u_2 = 10, l_3 = 4, u_3 = 10, l_4 = 0$ and $u_4 = 2$ satisfy these requirements, giving the following invariant:[10]

$$2 \leq x-y \leq 10 \wedge x+y \leq 10 \wedge 4 \leq x \leq 10 \wedge 0 \leq y \leq 2\ .$$

As the reader would notice, the above formula is not closed: we can easily obtain a lower bound of 4 on $x+y$ from lower bounds of x and y. This is despite the fact that parameter constraints above are closed. This suggests that even though the invariant thus generated is the strongest, its representation is not closed.



Note that computing the closure of the test conditions in the second branch strengthens the invariant generated using the proposed method. If the closure had not been computed, the implicit constraint $y \leq 1$ would not then generate the parameter constraint $u_4 \geq 2$. This example illustrates that computing the closure of test conditions (even locally) can help in generating a stronger invariant.



constraint	present	absent	side condition
$x - y \le a$	$a \le \Delta_2 - l_2$	$u_1 \le \Delta_2 - l_2$	–
$x - y \ge b$	$\Delta_2 - u_2 \le b$	$\Delta_2 - u_2 \le l_1$	–
$x + y \le c$	$c \le \Delta_1 - l_1$	$u_2 \le \Delta_1 - l_1$	–
$x + y \ge d$	$\Delta_1 - u_1 \le d$	$\Delta_1 - u_1 \le l_2$	–
$x \le e$	$e \le A - l_3$	$u_3 \le A - l_3$	–
$x \ge f$	$A - u_3 \le f$	$A - u_3 \le l_3$	–
$y \le g$	$u_4 \ge g + B$	$u_4 = +\infty$	$B > 0$
$y \ge h$	$l_4 \le h + B$	$l_4 = -\infty$	$B < 0$

Fig. 3. Sign of only x is reversed in assignment: Constraints on Parameters.

2.8 The Sign of Exactly One Variable is Reversed

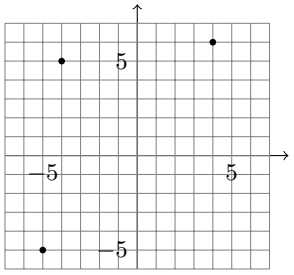
As in the previous two subsections, a similar analysis can be done to investigate the effect of assignments of the form x := −x+A and y := y+B, where the sign of exactly one variable changes. The parametric verification condition in this case is:

$$\begin{aligned}\big(l_1 \le x - y \le u_1 \wedge l_2 \le x + y \le u_2 \wedge l_3 \le x \le u_3 \wedge l_4 \le y \le u_4\big) \wedge T(x,y)\\ \Rightarrow \begin{pmatrix}\Delta_1 - u_1 \le x + y \le \Delta_1 - l_1 \wedge \Delta_2 - u_2 \le x - y \le \Delta_2 - l_2\\ \wedge\, A - u_3 \le x \le A - l_3 \wedge l_4 - B \le y \le u_4 - B\end{pmatrix},\end{aligned}$$

where $\Delta_1 = A - B$ and $\Delta_2 = A + B$. The associated table of generated constraints is given in Figure 3. The use of this table is once again illustrated using an example below.

Example 6. Let us consider the following loop:

```
x := 4; y := 6;
while (x+y ≥ 0)
  if (y ≥ 6) then
    x := −x; y := y−1;
  else
    x := x−1; y := −y;
```



The state space for this program is again depicted on the right. For initial values and the execution of the loop body, the following verification conditions are generated:

– Initial condition:

$$l_1 \le -2 \le u_1 \wedge l_2 \le 10 \le u_2 \wedge l_3 \le 4 \le u_3 \wedge l_4 \le 6 \le u_4\ .$$

[9] The closure of $(x - y \ge 3 \wedge x \le 4)$ gives $(y \le 1 \wedge x + y \le 5)$.
[10] Interproc outputs $2 \le x - y \le 10 \wedge 4 \le x + y \le 10 \wedge 4 \le x \le 10 \wedge 0 \le y \le \frac{7}{2}$ [sic].

– Branch 1:

$$\begin{pmatrix} l_1 \le x - y \le u_1 \wedge l_2 \le x + y \le u_2 \\ \wedge\, l_3 \le x \le u_3 \wedge l_4 \le y \le u_4 \end{pmatrix} \wedge T_1(x, y)$$
$$\Rightarrow \begin{pmatrix} 1 - u_1 \le x + y \le 1 - l_1 \wedge -1 - u_2 \le x - y \le -1 - l_2 \\ \wedge\, -u_3 \le x \le -l_3 \wedge 1 + l_4 \le y \le 1 + u_4 \end{pmatrix},$$

where $\Delta_1 = 1$, $\Delta_2 = -1$ and $T_1(x, y) = (x + y \ge 0 \wedge y \ge 6)$.

– Branch 2:

$$\begin{pmatrix} -u_1 \le y - x \le -l_1 \wedge l_2 \le y + x \le u_2 \\ \wedge\, l_4 \le y \le u_4 \wedge l_3 \le x \le u_3 \end{pmatrix} \wedge T_2(x, y)$$
$$\Rightarrow \begin{pmatrix} l_1 + 1 \le y + x \le u_1 + 1 \wedge -1 - u_2 \le y - x \le -1 - l_2 \\ \wedge\, 1 + l_3 \le x \le 1 + u_3 \wedge -u_4 \le y \le -l_4 \end{pmatrix},$$

where $T_2(x, y) = (x + y \ge 0 \wedge y \le 5 \wedge x \ge -5 \wedge x - y \ge -10)$.[11]

From the table, we get the following parameter constraints:

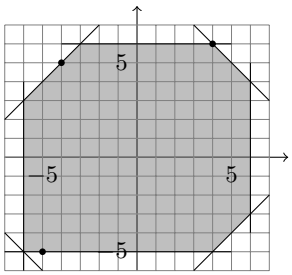
$$\begin{aligned} & l_1 \le -2 \wedge u_2 \ge 10 \wedge -1 \le l_1 + u_2 \le 1 \\ & \wedge\, l_2 \le -11 \wedge u_1 \ge 1 \wedge l_2 + u_1 \le -1 \wedge -1 \le u_2 - u_1 \le 1. \\ & \wedge\, l_3 \le -6 \wedge u_3 \ge 4 \wedge u_3 + l_3 = 0 \\ & \wedge\, l_4 \le -5 \wedge u_4 \ge 6 \wedge u_4 + l_4 \ge 0 \end{aligned}$$

Making the l_is as large as possible, and u_is as small as possible, we get:

$$l_1 = -9, u_1 = 9, l_2 = -11, u_2 = 10, l_3 = -6, u_3 = 6, l_4 = -5, u_4 = 6.$$

The corresponding invariant is:[12]

$$-9 \le x - y \le 9 \wedge -11 \le x + y \le 10 \wedge -6 \le x \le 6 \wedge -5 \le y \le 6\ .$$



If the closure of the test conditions for the second branch is not computed, then the invariant generated is weaker since the parameter constraint due to the fact that the implicit constraint $x \ge -5$ is omitted.

[11] This is because the closure of the constraint $(x + y \ge 0 \wedge y \le 5)$ also includes $x \ge -5$ and $x - y \ge -10$.

[12] The Interproc tool's output is $y \ge -5$.

2.9 The Effects of Computing the Closure of the Tests and Choosing Different Table Entries

The examples in the previous subsections illustrated how implicit constraints derived from the test conditions in a program path can improve the strength of invariants by generating additional constraints on the parameters. However, that is not always the case. The following example is a slight adaptation of Example 6.

Example 7. Let us consider the following loop:

```
x := 4; y := 6;
while (x+y ≥ 0)
  if (y ≥ 6) then
    x := −x; y := y−1;
  else
    x := −x; y := −y;
```

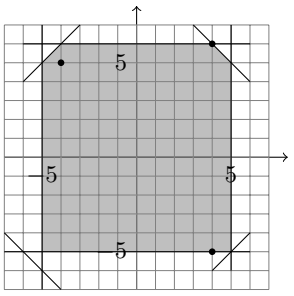
The parameter constraints after computing the closure $x \geq -5 \wedge x - y \geq -10$ of $x + y \geq 0 \wedge y \leq 5$ in the second branch lead to the computation of the following strongest invariant:

$$-10 \leq x - y \leq 10 \wedge -11 \leq x + y \leq 10 \wedge -5 \leq x \leq 5 \wedge -5 \leq y \leq 6 \ .$$

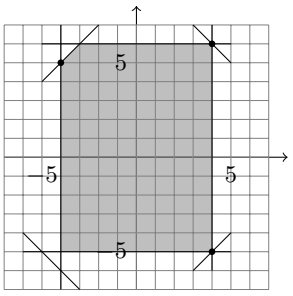
In contrast, without generating implicit constraints, the strongest invariant generated is:

$$-9 \leq x - y \leq 9 \wedge -10 \leq x + y \leq 10 \wedge -4 \leq x \leq 4 \wedge -5 \leq y \leq 6 \ .$$

The reader would note that $-10 \leq x + y$ is not closed; normalization would transform this into $-9 \leq x + y$.



invariant with closure



invariant without closure

The optimal bound, based on the states, would be $-1 \leq x + y$. This effect stems from the way we approximate the verification condition: To derive, say, the upper bound for x in the presence of a sign-reversing assignment, we look at the part of a verification condition involving x (and no other variable). Depending on whether or not a test on x is present on the path, the condition may have one of two forms:

$$\begin{aligned} x \leq u_3 \quad &\Longrightarrow x \leq A - l_3 \text{ or} \\ x \leq u_3 \wedge x \leq e \ &\Longrightarrow x \leq A - l_3 \ . \end{aligned}$$

The former implication is equivalent to $u_3 \leq A - l_3$, which is the corresponding table entry, and the latter is equivalent to $u_3 \leq A - l_3 \vee e \leq A - l_3$. Since the introduction of disjunctions in this way would adversely affect the complexity, our decision was to assume that the test $x \leq e$ carries more semantics and is most likely the more restrictive of the two, and thus to choose $e \leq A - l_3$ as the table entry. In the previous example, this choice did not lead to the optimal result. The effect of choosing the right table entry is even more evident in the following trivial example:

Example 8. Consider the following program:

```
x := 0;
while (x ≤ 10)
  x := 1−x;
```

The value of x jumps back and forth between 0 and 1 in each iteration and never comes close to violating the loop condition. The optimal octagonal invariant, which in one dimension is an interval, is $0 \leq x \leq 1$.

With the table entry $u_3 \leq A - l_3$ for the upper bound of x, we derive exactly this invariant. Our heuristic that the loop condition should impact the invariant is not optimal for this loop. So with the table entry $e \leq A - l_3$, we instead derive $-9 \leq x \leq 10$, which is exactly what also Interproc returns as the loop's invariant.

Below we discuss how the structure of parameter constraints generated by the above method can be exploited to develop fast methods to checking their satisfiability, generating an assignment to parameters to result in an invariant, as well as generating the strongest invariant.

The role of derived implicit constraints obtained from the closure operation and their relationship to different possible table entries needs further investigation. Conversely, it might also be useful to investigate dropping constraints which are otherwise implied by the remaining constraints.

2.10 Programs with Multiple Loops

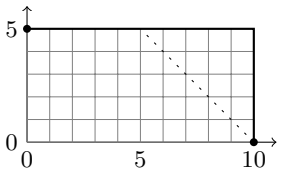
The proposed approach extends to programs containing several loops, both nested and sequential. The verification conditions for such programs will often relate two invariants associated with different loops, i.e. they take a shape like

$$(I(x, y) \wedge T(x, y)) \implies J(x', y') \ ,$$

where I and J are the two invariants that are involved. Following our quantifier elimination approach, the verification conditions can as before be approximated by constraints on the parameters using similar tables to the ones presented in Figures 1–3, which now contain constraints on the parameters of both invariants. Due to lack of space, we restrict ourselves to illustrating this extension on an example.

Example 9. Consider the following nonterminating program that contains two nested loops.

```
x := 0; y := 5;
while (true) do
  if (x < 10 and y = 5) then
    x := x+1;
  else if (x = 10 and y > 0) then
    y := y−1;
  else
    while (y < 5) do
      x := x−1; y := y+1;
```



In the depiction of this program's state space, states reachable at the outer loop are indicated by the continuous line and states reachable at the inner loop are indicated by the dotted line.

Let $I(x, y)$ be the invariant associated with the outer loop with parameters $l_1, l_2, l_3, l_4, u_1, u_2, u_3, u_4$. Similarly, let $J(x, y)$ be the invariant associated with the inner loop with parameters $l'_1, l'_2, l'_3, l'_4, u'_1, u'_2, u'_3, u'_4$. Besides verification conditions of the form discussed above, there are two verification conditions relating $I(x, y)$ to $J(x, y)$ and vice versa.

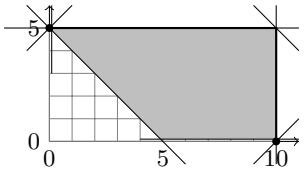
(i) $(J(x, y) \wedge \neg(y < 5)) \implies I(x, y)$, and
(ii) $(I(x, y) \wedge \neg(x < 10 \wedge y = 5) \wedge \neg(x = 10 \wedge y > 0)) \implies J(x, y)$.

These verification conditions relate parameters of the two loop invariants. From condition (i), for example, we can derive the constraints $l'_1 \leq l_1$ and $u_1 \leq u'_1$ and similarly for other parameters; because of the test $\neg(y < 5)$, or equivalently $\neg(y \leq 4)$, we get $l_4 \leq 5$. The loop invariants generated by the proposed approach are

$$I(x, y) = J(x, y) = \begin{pmatrix} x - y \leq 10 \wedge 5 \leq x + y \leq 15 \\ \wedge\, x \leq 10 \wedge 0 \leq y \leq 5 \end{pmatrix},$$

or after closure:

$$I(x, y) = J(x, y) = \begin{pmatrix} -5 \leq x - y \leq 10 \wedge 5 \leq x + y \leq 15 \\ \wedge\, 0 \leq x \leq 10 \wedge 0 \leq y \leq 5 \end{pmatrix}.$$



This is indeed the strongest octagonal invariant for this program.

2.11 Strongest Invariants and Complexity

As the reader would have noticed, every entry in each of the three tables is also an octagonal constraint on at most two parameters. Constraints on parameters imposed due to initialization are also octagonal. Further, there are constraints $l_i \leq u_i$ relating a lower bound with the associated upper bound. Most importantly, even though there are $2n^2$ parameters in a parametrized invariant, parameter interaction is localized. Constraints can be decomposed into disjoint parts based on parameters appearing in them, so that parameters appearing in one subset of constraints do not overlap with parameters appearing in other subsets of constraints. Consequently, a large formula expressed using many parameters can be decomposed into smaller subformulas expressed using a few parameters. And each of these subformulas can be processed separately.

We illustrate this using the parameter constraints generated in Example 6:

$$\begin{aligned} & l_1 \leq -2 \wedge u_1 \geq 1 \wedge l_2 \leq -11 \wedge u_2 \geq 10 \wedge l_3 \leq -6 \wedge u_3 \geq 4 \\ & \wedge\, l_4 \leq -5 \wedge u_4 \geq 6 \wedge -1 \leq l_1 + u_2 \leq 1 \wedge l_2 + u_1 \leq -1 \\ & \wedge\, u_3 + l_3 = 0 \wedge -1 \leq u_2 - u_1 \leq 1 \wedge u_4 + l_4 \geq 0. \end{aligned}$$

The above constraints can be separated into three disjoint parts $S_1 \wedge S_2 \wedge S_3$, where

$$\begin{aligned} S_1 = \; & (l_1 \leq -2 \wedge u_1 \geq 1 \wedge l_2 \leq -11 \wedge u_2 \geq 10 \wedge -1 \leq l_1 + u_2 \leq 1 \\ & \quad \wedge\, l_2 + u_1 \leq -1 \wedge -1 \leq u_2 - u_1 \leq 1)\ , \\ S_2 = \; & (l_3 \leq -6 \wedge u_3 \geq -4 \wedge u_3 + l_3 = 0)\ , \\ S_3 = \; & (l_4 \leq -5 \wedge u_4 \geq 6 \wedge u_4 + l_4 \geq 0)\ . \end{aligned}$$

Each of these subsystems can be solved separately, reducing the complexity of parameter constraint solving to $O(m)$, where m is the number of constraints, since the number of parameters is constant (2 or 4). The closure of S_1 gives

$$\begin{aligned} & l_1 \leq -2 \wedge -u_1 \leq -9 \wedge l_2 \leq -11 \wedge -u_2 \leq -10 \wedge -l_1 - u_2 \leq 1 \\ & \wedge\, l_1 + u_2 \geq 1 \wedge l_2 + u_1 \leq -1 \wedge u_2 - u_1 \leq 1 \wedge l_2 + u_2 \leq 0 \wedge l_2 - l_1 \leq 1\ . \end{aligned}$$

The maximum values of l_1 and l_2 are, respectively, $-9, -11$; the minimum values of u_1 and u_2 are 9 and 10, respectively. For S_2, its closure adds $l_3 \leq -6$, $-u_3 \leq -6$, and $l_3 + u_3 = 0$. The maximum value of l_3 is -6, the minimum value of u_3 is 6. For S_3, its closure is itself; the maximum value of l_4 is -5; the minimum value of u_4 is 6. These parameter values give the optimal invariant:

$$-9 \leq x - y \leq 9 \wedge -11 \leq x + y \leq 10 \wedge -6 \leq x \leq 6 \wedge -5 \leq y \leq 6.$$

As was stated above, the Interproc tool computes a much weaker invariant.

With these considerations in mind, it can be shown that program analysis using our geometric heuristic for quantifier elimination for octagons takes quadratic time in the number of program variables:

Theorem 10. *A constraint description of octagonal loop invariants for a program can be automatically derived using the geometric quantifier elimination heuristics proposed above in $O(k \cdot n^2 + p)$ steps, where n is the number of program variables, k is the number of program paths, and p is the size of the program's abstract syntax tree.*

Proof. There are two steps involved in automatically deriving an octagonal invariant, and the complexity of each step is analyzed below.

First, for every program path all tests performed must be collected as well as the cumulative effect of assignment statements on program variables must be computed to generate a verification condition. This can be done in $O(p)$ steps using a standard Hoare-style backward analysis.

For every pair of distinct program variables, there are at most 8 types of atomic formulas appearing as tests in a given program path (after removing redundant tests which again can be performed globally in $O(p)$ steps); the local closure of these tests is performed in time $O(n^2)$ for each path. From this, the verification condition for each path can be constructed in constant time. The generation of parameter constraints involves one table look-up for each lower and upper bound of the expressions $x, x + y, x - y$, where x and y are program variables. As mentioned before, there are $O(n^2)$ tests to consider for each path (present or absent) and thus also $O(n^2)$ table look-ups, each of which takes constant time. Given all k program paths, $O(k \cdot n^2)$ parameter constraints are generated in $O(k \cdot n^2)$ steps. □

The result of the presented algorithm does not consist of an explicit list of invariants for the program's loops, but it implicitly describes the invariants by constraints on their parameters. Each of the usually infinitely many instantiations of the parameters satisfying the constraints describes one list of invariants. Among these, we can effectively extract a unique *strongest* list of invariants that implies all other described invariant lists and that is (globally) closed.

Lemma 11. *From the set of constraints for a program with a single loop, computed by our algorithm and described by constraints on the invariants' parameters, the unique strongest one can be automatically derived in $O(k \cdot n^2)$ steps, where n is the number of program variables and k is the number of program paths.*

Proof. Using the algorithm by Bagnara et al. [3], it is possible to generate the maximum value of each lower bound parameter l_i ($-\infty$ if none exists) as well as the minimum value of each upper bound parameter u_j (similarly $+\infty$ if none exists) as follows:

For every l_i that has a maximum value in $\mathbb{Z}$, there is an invariant in which l_i takes that maximum value allowed by the constraints; similarly, for every u_j that has a minimum value in $\mathbb{Z}$, there is also an invariant in which u_j takes the minimum value allowed by the constraints (as otherwise, it would be possible to further tighten the associated octagon using the closure operation). If there is no invariant where l_i or u_i has a value in $\mathbb{Z}$, they are always $-\infty$ or ∞,

respectively. The conjunction of any two octagonal invariants thus generated is also an octagonal invariant. It thus follows that the invariant in which l_i's have the maximum of all their lower bounds and u_j's have the minimum of all their upper bounds gives an octagonal invariant that implies all other invariants described by the constraints.

The key to computing these optimal bounds efficiently lies in decomposing the parameter constraints into disjoint subsets of constant size that do not interact with each other:

From the above tables, it is clear that for any particular program variable x, its lower bound l and upper bound u can be related by constraints, but these parameters do not appear in relation with other parameters. This has also been discussed at the beginning of the current subsection. Thus for a particular program variable x, all constraints on l and u can be analyzed separately without having to consider other constraints. Their satisfiability as well as lower and upper bounds on these parameters can be computed in linear time the number of constraints relating l and u, which is at most 2 per path, i.e. $O(k)$: Since there are only a constant number of parameters in these constraints (namely 2), the algorithm for tight integral closure in [3] takes time proportional to the number of constraints on these two parameters. There are n variables, so computing their optimal lower and upper bounds takes $O(k \cdot n)$.

Similarly, for each variable pair (x_i, x_j) the parameters expressing lower and upper bounds for $x_i - x_j$ and $x_i + x_j$ appear together in table entries, and they do not appear in relation with other parameters. So each of those parameter sets of constant size 4 can be analyzed independently. There are $O(n^2)$ variable pairs, so analogously to above, computing their optimal lower and upper bounds takes $O(k \cdot n^2)$.

Overall, the strongest invariant can be computed from a conjunction of parameter constraints in $O(k \cdot n^2)$ steps. By construction of the algorithm by Bagnara et al., every program state satisfying this invariant satisfies all invariants described by the initial parameter constraints. □

Theorem 12. *From the set of constraints for a program with no nesting of loops, computed by our algorithm and described by constraints on the invariants' parameters, the unique strongest one can be automatically derived in $O(k \cdot n^2)$ steps, where n is the number of program variables and k is the number of program paths.*

Proof. There are no nested loops, so the loops can be partially ordered by their order of execution. For each verification condition $(I(X) \wedge T(X)) \implies J(X')$ involving two different parameterized loop invariants I and J, the loop for I is then executed before the one for J.

This ordering is obviously well-founded. We can thus examine the loops one at a time, inductively assuming that strongest invariants have already been computed for all loops that are executed before the one under consideration. When we analyze each loop, we can without loss of generality assume that all constraints include only parameters for the current loop: Optimal parameters for

the previous loops have already been instantiated and constraints for following loops do not impose any upper bounds on parameters l_i or lower bounds for parameters u_i of the current loop. We can also safely assume that during the generation phase, the constraints have already been assembled following the execution order, so we do not need to sort the constraints now to access the ones for the current loop.

Then the result follows directly from Lemma 11, since the global number of program paths corresponds to the sum of paths that are relevant for the individual loops. □

Our complexity analysis contains a parameter whose significance we have mostly ignored so far: The number k of program paths. In principle, it is easy to construct a program with 2^p program paths, where p is the size of the program, by simply making every statement a conditional. This would mean that our analysis would not scale at all.

In practice, however, a program has a far less paths; see also [12, 17] where techniques for managing program paths and associated verification conditions are discussed. Since our analysis is based on computing invariants that are satisfied at certain points in the program, the number of paths can also be kept in check by introducing invariants at additional program locations. These invariants provide a convenient way to trade off precision for complexity.

Example 13. Consider the following program stub:

```
x=0; y=0;
while (true) do
  if ... then ... ;
  if ... then ... ;
  // add invariant J here
  if ... then ... ;
  if ... then ... ;
```

To compute the invariant I for this loop, $2^4 = 16$ paths inside the loop have to be analyzed, depending on which of the 4 conditionals are used and which are not. If we introduce an additional invariant J after the second conditional, there are only 2^2 paths leading from I to J and another 2^2 paths from J to I, or 8 paths in total.

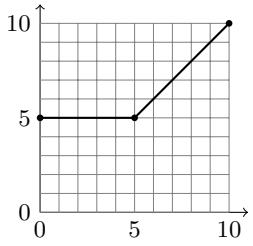
In general, adding an invariant after every conditional to directly merge the control flows would lead to an analysis resembling an abstract interpretation approach. As a side effect, such additional invariants allow for a convenient way to check assertions: Just add invariants at the locations of all assertions, analyze the program and check whether the computed invariants entail the corresponding assertions.

3 Invariant Generation with Max-Plus Polyhedra

In many cases where programs contain loops with several execution paths, purely conjunctive invariants can only inadequately model the program semantics.

Example 14. The strongest invariant at the loop entry point in the program

```
x := 0; y := 5;
while (x < 10) do
  if (x < 5) then
    x := x+1;
  else
    x := x+1; y := y+1;
```



is $(0 \leq x \leq 5 \wedge y = 5) \vee (5 \leq x \leq 10 \wedge x = y)$. This invariant is depicted on the right. It consists of two lines and is clearly not convex. Hence it cannot be expressed as a conjunction of linear constraints, including octagonal constraints or even general polyhedra.

One way of allowing more expressive loop invariants including disjunctive invariants is to use formulas defining so-called *max-plus polyhedra*, as proposed by Allamigeon [1, 6]. Such a formula allows disjunctions of a subset of octagonal constraints, particularly of constraints of the form $l_i \leq x_i \leq u_i$ and $l_{i,j} \leq x_i - x_j \leq u_{i,j}$. Atomic formulas of the form $a \leq x_i + x_j \leq b$ are not allowed in a strict max-plus setting. More specifically, the allowed disjunctions are those that can be written as

$$\max(x_1 + a_1, \ldots, x_n + a_n, c) \leq \max(x_1 + b_1, \ldots, x_n + b_n, d) \ ,$$

where a_i, b_i, c, d are integers or $-\infty$. For example,

$$\max(x + 0, y - \infty, -\infty) \leq \max(x - \infty, y + b, d)$$

would represent the disjunction $x \leq d \vee x - y \leq b$.

For a reader who is mainly interested in the high-level concepts, it suffices to note that sets of linear constraints formed in this way not with the standard addition and multiplication operators, but instead with max and +, can also be regarded as describing convex state spaces, which only have a different shape than for the standard operators.

Formally, the basis of max-plus polyhedra is formed by the *max-plus semiring* $(\mathbb{Z}_{\max}, \oplus, \otimes)$. The elements of $\mathbb{Z}_{\max}$ are those of $\mathbb{Z}$ and $-\infty$, and the semiring addition $\oplus$ and multiplication $\otimes$ are given by $x \oplus y = \max(x, y)$ and $x \otimes y = x + y$, with additive unit $\mathbb{0} = -\infty$ and multiplicative unit $\mathbb{1} = 0$. The usual order on $\mathbb{Z}$ extends to $\mathbb{Z}_{\max}$ by making $-\infty$ the least element. A *max-plus polyhedron* is a subset of $\mathbb{Z}_{\max}^n$ satisfying a finite set of linear max-plus inequalities of the form

$$(a_1 \otimes x_1) \oplus \cdots \oplus (a_n \otimes x_n) \oplus c \leq (b_1 \otimes x_1) \oplus \cdots \oplus (b_n \otimes x_n) \oplus d \ ,$$

where $a_i, b_i, c, d \in \mathbb{Z}_{\max}$.

Max-plus convex sets were introduced by Zimmermann [33], a general introduction can be found for example in Gaubert and Katz [13]. They are more expressive than Difference Bound Matrices: On the one hand, every invariant that can be represented by Difference Bound Matrices can also be represented by a max-plus polyhedron. To convert a Difference Bound Matrix into a max-plus polyhedron, each constraint $x_i - x_j \leq c$ is simply written as the equivalent max-plus constraint $0 \otimes x_i \leq c \otimes x_j$. On the other hand, a max-plus polyhedron can in general not be represented by a single Difference Bound Matrix. For example, the invariant from example 14 is a max-plus polyhedron but it cannot be represented by a Difference Bound Matrix, nor by any other other form of conjunctions of linear constraints.

Max-plus polyhedra are convex sets (with respect to the max-plus semiring). As such, they can be represented in multiple ways, just like traditional polyhedra. Of particular interest is the representation as the convex hull of a finite set of generator points (analogous to a frame representation of a convex polyhedron; cf. Allamigeon et al. [1] for details on how to convert between the representations). If $p_1, \ldots, p_n$ are points, we denote the max-plus polyhedron that is generated by these points as their convex hull by $\mathrm{co}(\{p_1, \ldots, p_n\})$. In contrast to standard polyhedra, the generator representation is arguably more intuitive for humans than the constraint representation for max-plus polyhedra (due to constraints like $\max(x, y+1, 2) \leq \max(x+2, y)$), and we will utilize it throughout most of this article. To keep the presentation even more intuitive, we will also restrict ourselves mainly to bounded polyhedra in $\mathbb{Z}^2_{\max}$, i.e. to programs with two variables whose range during program execution is bounded. As is customary when working with max-plus polyhedra, we write points in $\mathbb{Z}^2_{\max}$ as column vectors, i.e. the point with x-coordinate 0 and y-coordinate 5 is written as $\binom{0}{5}$.

Example 15. Consider the max-plus polyhedron with generators $\binom{0}{5}$ and $\binom{10}{10}$: As the convex hull of these two points, this max-plus polyhedron contains all points of the form $\alpha \otimes \binom{0}{5} \oplus \beta \otimes \binom{10}{10}$ with $\mathbb{0} \leq \alpha, \beta$ and $\alpha \oplus \beta = \mathbb{1}$. Expressed in standard arithmetic, these are the points of the form $\binom{\max(\alpha+0, \beta+10)}{\max(\alpha+5, \beta+10)}$ where $\max(\alpha, \beta) = 0$. It is easy to verify that this max-plus polyhedron is exactly the invariant from Example 14. A constraint representation of this max-plus polyhedron is

$$\begin{array}{ll} 0 \leq x \leq 10 & 5 \leq y \leq 10 \\ -5 \leq x - y \leq 0 & 0 \otimes y \leq (0 \otimes x) \oplus 5 \text{ (i.e. } y \leq \max(x, 5)) \ . \end{array}$$

This representation is of course harder to understand for the human reader than the one from Example 14. However, it is a pure (max-plus) conjunction, which is the key to it's fully automatic treatment.

To better understand the possible shapes of invariants, let us consider the case of a bounded two-dimensional max-plus polyhedron generated by two points $\binom{a_1}{b_1}$ and $\binom{a_2}{b_2}$. The possible shapes of such a max-plus polyhedron are depicted in Figure 4, where we assume, without loss of generality, $a_1 \leq a_2$.

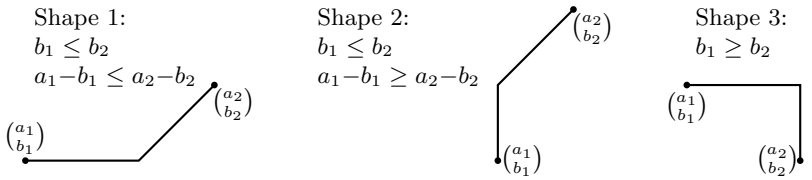
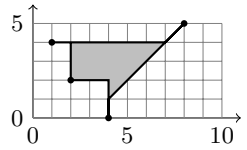


Fig. 4. Possible shapes of bounded two-dimensional max-plus polyhedra



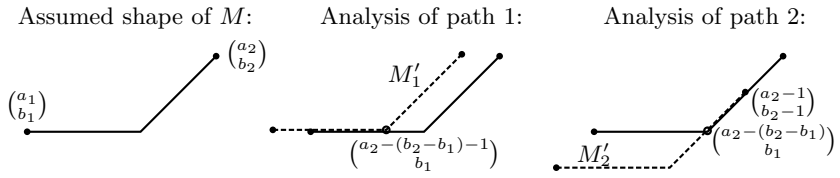
In general, also in higher dimensions, two generators are always connected by lines that run parallel, perpendicular, or at a 45 degree angle with all coordinate axes. A polynomial with multiple generators will consist of all these connections as well as the area that is surrounded by the connections. Such an example with four generators (marked by dots) can be seen to the right. Knowledge about these shapes will be used in the next section to develop a quantifier elimination heuristics based on table lookups, comparable to the one for octagonal constraints.

3.1 Invariant Generation

Similar to the procedure for octagonal constraints described in the previous section, we derive loop invariants described by max-plus polyhedra as follows: We start with a parametrized invariant given as a max-plus polyhedron M with a predetermined number of parametrized generators. Each path through the loop gives rise to a verification condition $M \cap \Gamma \subseteq M'$, where Γ is the polyhedron defined by the branch and loop tests along the path and M' arises from M as usual by the translation induced by the executed assignments. This verification condition can equivalently be expressed as $(M \setminus M') \cap \Gamma = \emptyset$. We then look up a representation of $M \setminus M'$ from a precomputed table that describes $M \setminus M'$ as a set of max-plus polyhedra and express the requirement that Γ cuts off all of $M \setminus M'$ to convert the verification condition into a set of ground constraints on the parameters of M. The verification condition for the loop entry is also encoded by such constraints.

Every solution of the constraints then corresponds to a valid loop invariant. As a heuristics to select a strong invariant, we choose a solution of the constraints that minimizes the distance between the generators: We maximize a_1 and b_1 and minimize a_2 and b_2, just as we maximized the l_i and minimized the u_i before. We will show in Theorem 18 that for two generators, this results in an optimal solution.

Example 16. To find a loop invariant for Example 14, we nondeterministically guess that the invariant is a max-plus polyhedron $M = \mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2}{b_2}\})$ of shape 1, as depicted below, and look for suitable instantiations of the parameters a_i and b_i. The loop contains two execution paths, depending on whether or not the condition x<5 is satisfied.



The loop invariant M must be satisfied when the loop is entered. We guess that the initial value is situated on the horizontal branch of M. This results in the constraints $a_1 \leq 0 \leq a_2 - (b_2 - b_1)$ and $b_1 = 5$.

Furthermore, the loop invariant must be maintained during the execution of the loop. Consider the first path with the assignment x := x+1. The verification condition for this branch is

$$\tbinom{x}{y} \in \mathrm{co}(\{\tbinom{a_1}{b_1}, \tbinom{a_2}{b_2}\}) \wedge (x < 10 \wedge x < 5) \implies \tbinom{x+1}{y} \in \mathrm{co}(\{\tbinom{a_1}{b_1}, \tbinom{a_2}{b_2}\})$$

or equivalently

$$\tbinom{x}{y} \in \mathrm{co}(\{\tbinom{a_1}{b_1}, \tbinom{a_2}{b_2}\}) \wedge (x < 10 \wedge x < 5) \implies \tbinom{x}{y} \in \mathrm{co}(\{\tbinom{a_1-1}{b_1}, \tbinom{a_2-1}{b_2}\}) \ .$$

So if M_1' is the max-plus polyhedron generated by $\binom{a_1-1}{b_1}$ and $\binom{a_2-1}{b_2}$, this verification condition can be rewritten, following the considerations above, as

$$(M \setminus M_1') \cap \{x \mid x < 10 \wedge x < 5\} = \emptyset \ .$$

The relation between M and M_1' is depicted in the middle of the above figure. For M to be an invariant, it is necessary that every point in $M \setminus M_1'$, i.e. every point in M right of $\binom{a_2-(b_2-b_1)-1}{b_1}$, is cut off by the constraints $x < 10 \wedge x < 5$ (or equivalently $x \leq 4$) along the path. This directly implies $a_2 - (b_2 - b_1) - 1 \geq 4$.

For the second path with the assignment x := x+1; y := y+1, let M_2' be the max-plus polyhedron generated by $\binom{a_1-1}{b_1-1}$ and $\binom{a_2-1}{b_2-1}$. For M to be an invariant, it is necessary that every point in $M \setminus M_2'$, i.e. every point in M left of $\binom{a_2-(b_2-b_1)}{b_1}$ or right of $\binom{a_2-1}{b_2-1}$, is cut off by the constraints $x < 10 \wedge x \geq 5$ (or equivalently $x \leq 9 \wedge x \geq 5$) along the path. This implies $a_2 - (b_2 - b_1) \leq 5$ and $a_2 - 1 \geq 9$.[13]

All these constraints can be combined to $a_1 \leq 0$, $b_1 = 5$, $a_2 \geq 10$, and $a_2 - b_2 = 5$. Every instantiation of the parameters that satisfies these constraints leads to a valid loop invariant. To find the strongest invariant, we maximize a_1 and b_1 and minimize a_2 and b_2. This yields the max-plus polyhedron generated by the two points $\binom{0}{5}$ and $\binom{10}{10}$.

Had we guessed shape 2 or 3 for M, the resulting set of constraints would have been unsatisfiable, indicating that the loop does not have an invariant that is expressible as one of these shapes.

In general, multiple tests in Γ may be used to cut off a component of $M \setminus M'$. Similar to our approach to quantifier elimination for octagons, we approximate

[13] The two other possibilities $a_1 \geq 10$ and $a_2 < 5$ that would result in all of M being cut off would later contradict the loop entry condition.

assignment statements	max-plus polyhedra to be cut off
$A = 0, B = 0$	–
$A > 0, B = 0,\ A \leq \Delta_a - \Delta_b$	$\mathrm{co}(\{\binom{a_2-\Delta_b-A+1}{b_1}, \binom{a_2}{b_2}\})$
$A > 0, B = A,\ A \leq \Delta_b$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2-\Delta_b-1}{b_1}\})$, if $\Delta_a \neq \Delta_b$ $\mathrm{co}(\{\binom{a_2-A+1}{b_2-A+1}, \binom{a_2}{b_2}\})$
$A > B, B > 0, \Delta \leq \Delta_a - \Delta_b, B \leq \Delta_b$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2-\Delta_b-\Delta-1}{b_1}\})$, if $\Delta_b + \Delta \neq \Delta_a$ $\mathrm{co}(\{\binom{a_2}{b_2}, \binom{a_2-\Delta_b-\Delta+1}{b_1}\})$
$A < 0, B = 0,\ \lvert A\rvert \leq \Delta_a - \Delta_b$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_1-A-1}{b_1}\})$ $\mathrm{co}(\{\binom{a_2-\Delta_b+1}{b_1+1}, \binom{a_2}{b_2}\})$, if $\Delta_b \neq 0$
$A < 0, B = A,\ \lvert A\rvert \leq \Delta_b$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2-\Delta_b-A-1}{b_1-A-1}\})$
$A < B, B < 0, \Delta \leq \Delta_a - \Delta_b, \lvert B\rvert < \Delta_b$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2-\Delta_b-B-1}{b_1-B-1}\})$ $\mathrm{co}(\{\binom{a_2}{b_2}, \binom{a_2-\Delta_b-B+1}{b_1-B+1}\})$, if $B + \Delta_b \neq 0$
all other cases	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2}{b_2}\})$

Fig. 5. Max-plus polyhedra to be cut off for shape 1

the verification conditions by assuming instead that only one of the tests in Γ may be used for an individual component. Due to this approximation, exactly which parts of a polyhedron have to be cut off by the constraints along a given path can be precomputed, just like for octagonal constraints. This gives rise to a finite table and reduces the analysis of each path in a loop to a table look-up. Such tables for the three shapes from Figure 4 are presented in Figures 5–7. The tables link assignments x := x+A; y := y+B along a path with the respective max-plus polyhedra that must be cut off. Throughout the tables, we use the abbreviations $\Delta_a = a_2 - a_1$, $\Delta_b = b_2 - b_1$, and $\Delta = A - B$.

Theorem 17. *The loop verification condition for an invariant of shape 1–3 and a path in the loop with assignments x := x+A; y := y+B and path condition Γ is equivalent to the condition that $\Gamma \cap M = \emptyset$ for each max-plus polyhedron M given in the respective entry of Figures 5–7.*

Proof. The proofs of correctness for all of the various table entries are very similar. As a typical representative, we show how to derive the entries for shape 1 in the case $A = B > 0$. Let M be the assumed (parametrized) invariant and let M^A result from translating M by $-A$ along both axes. Furthermore, let Γ be the set of all points that satisfy the constraints along the given path.

To obtain an invariant, M has to be chosen such that every point in $M \cap \Gamma$ is also in M^A. Considering a generic point $p = \binom{\alpha_1+a_1}{\alpha_1+b_1} \oplus \binom{\alpha_2+a_2}{\alpha_2+b_2}$ with $\alpha_1 \oplus \alpha_2 = \mathbb{1} = 0$ in M, this means that the parameters a_i, b_i must be such that if $p \in \Gamma$, then p can be written as $p = \binom{\beta_1+a_1-A}{\beta_1+b_1-A} \oplus \binom{\beta_2+a_2-A}{\beta_2+b_2-A}$ with $\beta_1 \oplus \beta_2 = \mathbb{1} = 0$.

assignment statements	max-plus polyhedra to be cut off
$A=0, B=0$	–
$A=0, B>0,\ B \le \Delta_b - \Delta_a$	$\mathrm{co}(\{\binom{a_1}{b_2-\Delta_a-B+1}, \binom{a_2}{b_2}\})$
$A=0, B<0,\ \lvert B\rvert \le \Delta_b - \Delta_a$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_1}{b_1-B-1}\})$ $\mathrm{co}(\{\binom{a_1+1}{b_2-\Delta_a+1}, \binom{a_2}{b_2}\})$, if $a_1 \neq a_2$
$A>0, B=A,\ A \le \Delta_a$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_1}{b_2-\Delta_a-1}\})$, if $\Delta_a \neq \Delta_b$ $\mathrm{co}(\{\binom{a_2}{b_2}, \binom{a_2-A+1}{b_2-A+1}\})$
$A>0, B>A,\ A \le \Delta_a, \Delta \ge \Delta_b - \Delta_a$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_1}{b_2-\Delta_a+\Delta-1}\})$ $\mathrm{co}(\{\binom{a_2}{b_2}, \binom{a_1}{b_2-\Delta_a+\Delta+1}\})$, if $\Delta_a - \Delta \neq \Delta_b$
$A<0, B=A,\ \lvert A\rvert \le \Delta_a$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_1-A-1}{b_2-\Delta_a-A-1}\})$
$A<0, B<A,\ \lvert A\rvert \le \Delta_a, \Delta \le \Delta_b - \Delta_a$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_1-A-1}{b_2-\Delta_a-A-1}\})$ $\mathrm{co}(\{\binom{a_2}{b_2}, \binom{a_1-A+1}{b_2-\Delta_a-A+1}\})$, if $\Delta_b + A \neq 0$
all other cases	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2}{b_2}\})$

Fig. 6. Max-plus polyhedra to be cut off for shape 2

assignment statements	max-plus polyhedra to be cut off
$A=0, B=0$	–
$A=0, B>0,\ B \le -\Delta_b$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2}{b_1-B+1}\})$
$A=0, B<0,\ B \ge \Delta_b$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2-1}{b_1}\})$, if $a_1 \neq a_2$ $\mathrm{co}(\{\binom{a_2}{b_2}, \binom{a_2}{b_2-B-1}\})$
$A>0, B=0,\ A \le \Delta_a$	$\mathrm{co}(\{\binom{a_2}{b_2}, \binom{a_2-A+1}{b_1}\})$
$A>0, B<0,\ A \le \Delta_a, B \ge \Delta_b$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2-A-1}{b_1}\})$, if $A \neq \Delta_a$ $\mathrm{co}(\{\binom{a_2}{b_2}, \binom{a_2-A+1}{b_1}\})$
$A<0, B=0,\ \lvert A\rvert \le \Delta_a$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_1-A-1}{b_1}\})$ $\mathrm{co}(\{\binom{a_2}{b_2}, \binom{a_2}{b_1-1}\})$, if $b_1 \neq b_2$.
$A<0, B>0,\ \lvert A\rvert \le \Delta_a, B \le -\Delta_b$	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2}{b_1-B+1}\})$ $\mathrm{co}(\{\binom{a_2}{b_2}, \binom{a_2}{b_1-B-1}\})$, if $\Delta_b + B \neq 0$.
all other cases	$\mathrm{co}(\{\binom{a_1}{b_1}, \binom{a_2}{b_2}\})$

Fig. 7. Max-plus polyhedra to be cut off for shape 3

– Case $-(b_2 - b_1) \leq \alpha_2 \leq -A$. Then $\alpha_1 = 0$. Choose $\beta_1 = 0, \beta_2 = \alpha_2 + A$ to show that $p \in M^A$.

$$(\beta_1 + a_1 - A) \oplus (\beta_2 + a_2 - A) = (a_1 - A) \oplus (\alpha_2 + a_2) = \alpha_2 + a_2$$
(because $\alpha_2 + a_2 \geq b_1 - b_2 + a_2 \geq a_1 > a_1 - A$ due to the shape.)
$$(\beta_1 + b_1 - A) \oplus (\beta_2 + b_2 - A) = (b_1 - A) \oplus (\alpha_2 + b_2) = \alpha_2 + b_2$$
(because $\alpha_2 + b_2 \geq b_1 > b_1 - A$)

So points that fall into this case are in the invariant and do not restrict the choice of the parameters.

– Case $\alpha_2 > -A$. If we assume $p \in M^A$, then we have the following contradiction:

$$b_2 - A \overset{\alpha_2 > -A}{<} \alpha_2 + b_2 \overset{\text{def } \oplus}{\leq} (\alpha_1 + b_1) \oplus (\alpha_2 + b_2)$$
$$\overset{p \in M^A}{=} (\beta_1 + b_1 - A) \oplus (\beta_2 + b_2 - A) \leq (b_1 - A) \oplus (b_2 - A) \overset{b_1 \leq b_2}{\leq} b_2 - A.$$

So $p \notin M^A$, which implies $p \notin \Gamma$.

– Case $\alpha_2 < -(b_2 - b_1)$. Then $\alpha_1 = 0$. Again assume $p \in M^A$. A look at the y-coordinate reveals:

$$b_1 = b_1 + (\overbrace{\alpha_1}^{=0} + \overbrace{b_1 - b_1}^{=0}) \oplus (\overbrace{\alpha_2 + b_2 - b_1}^{<0}))$$
$$= (\alpha_1 + b_1) \oplus (\alpha_2 + b_2) \overset{p \in M^A}{=} (\beta_1 + b_1 - A) \oplus (\beta_2 + b_2 - A)$$

Because $\beta_1 + b_1 - A < b_1$, this means that $b_1 = \beta_2 + b_2 - A$, or $\beta_2 = b_1 - b_2 + A$. So $\beta_2 = -(b_2 - b_1) - A < 0$, i.e. $\beta_1 = 0$ because of $\beta_1 \oplus \beta_2 = 0$. For the x-coordinate, this leads to the following contradiction:

$$a_1 \overset{\text{def } \oplus}{\leq} a_1 \oplus (\alpha_2 + a_2) = (\overbrace{\alpha_1}^{=0} + a_1) \oplus (\alpha_2 + a_2)$$
$$\overset{p \in M^A}{=} (\beta_1 + a_1 - A) \oplus (\beta_2 + a_2 - A) = (\overbrace{a_1 - A}^{<a_1}) \oplus (b_1 - b_2 + a_2) \overset{\text{shape}}{<} a_1$$

This means that the polyhedron has to degenerate to a line or $p \notin \Gamma$.

Overall, we have the following result:

– Γ must cut off the top right part of the polyhedron (case 2).
– If $b_1 - b_2 + A > 0$, Γ must cut off the left or lower arm. Otherwise either $a_1 = b_1 - b_2 + a_2$, i.e. the whole polyhedron consists only of one arm, or Γ must cut off the left or lower arm (case 3). □

As a heuristic to reduce the computational cost, we usually assume that each component of $M \setminus M'$ is cut off by a single constraint.

Note that Γ can contain any path constraint that the employed constraint solver can handle. In particular, Γ is not restricted to a conjunction of constraints

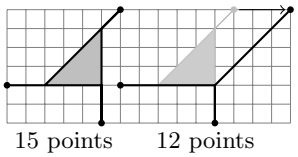
of the form $\pm x_i \leq c$ or $x_i \pm x_j \leq c$. E.g. in Example 14 and 16 the loop condition 3x+2y < 50 would have led to a similar analysis.

For two generators, our heuristics of minimizing the distance between the generators leads to an optimal invariant.

Theorem 18. *Among a set of bounded max-plus polyhedra with two generators* $\binom{a_1}{b_1}$ *and* $\binom{a_2}{b_2}$*, the state spaces described by those max-plus polyhedra that minimize* $|a_2 - a_1| + |b_2 - b_1|$ *are minimal.*

Proof. Since $|a_2 - a_1|$ and $|b_2 - b_1|$ are (discrete) nonnegative integers, the minimum is assumed, even if we consider an infinite set of bounded max-plus polyhedra.

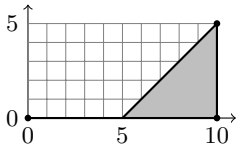
A polyhedron of shape 1 contains $(|a_2 - a_1| - |b_2 - b_1|) + |b_2 - b_1| + 1 = |a_2 - a_1| + 1$ points (over $\mathbb{Z}$), independently of $|b_2 - b_1|$. Analogously, a polyhedron of shape 2 contains $(|b_2 - b_1| - |a_2 - a_1|) + |a_2 - a_1| + 1 = |b_2 - b_1| + 1$ points, independently of $|a_2 - a_1|$, and one of shape 3 contains $|a_2 - a_1| + |b_2 - b_1| + 1$ points. So whatever the shape of the polyhedron, the described state space is minimal whenever $|a_2 - a_1| + |b_2 - b_1|$ is minimal. □



In the case of more than two generators, the situation is more complicated, and indeed increasing the Euclidean distance between two generators may result in a smaller polyhedron. In practice, however, the constraints imposed by the invariant generation process usually circumvent this situation. The reason is that, for example in the depicted situation, generators of two invariants can only differ by translations along the one-dimensional components leading to them. So the leftmost generator can only be moved horizontally, the lower one only vertically and the top one only along the diagonal. If generators are moved in a different way, the result cannot be an invariant any more.

Example 19. As a simple example whose invariant cannot be expressed with two generators, consider the following program:

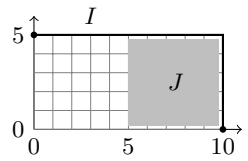
```
x := 0; y := 0;
while (true) do
  if (y = 0 and x < 10) then
    x := x+1;
  else if (x ≥ 10 and y < 5) then
    y := y+1;
  else x := x−1; y := y−1;
```



The minimal max-plus invariant for this loop is generated by the points $\binom{0}{0}$, $\binom{10}{0}$, and $\binom{10}{5}$, that can be found with an analysis similar to the one shown above. Note that the program is not terminating, and that the max-plus invariant consists of two components of different codimension, namely the line $0 \leq x \leq 5 \wedge y = 0$ and the triangle $x \leq 10 \wedge y \geq 0 \wedge x - y \geq 5$. The true optimal invariant of the program is the boundary of the one computed.

Example 20. Let us look again at the program from Example 9, which is also nonterminating and contains a nested loop:

```
x := 0; y := 5;
while (true) do
  if (x < 10 and y = 5) then
    x := x+1;
  else if (x = 10 and y > 0) then
    y := y−1;
  else
    while (y < 5) do
      x := x−1; y := y+1;
```



The minimal max-plus invariant $J(x, y)$ for the inner loop is the polyhedron generated by $\binom{5}{0}$, $\binom{5}{5}$, and $\binom{10}{0}$, i.e. the square $5 \leq x \leq 10 \wedge 0 \leq y \leq 5$. The resulting minimal max-plus invariant I for the outer loop is generated by $\binom{0}{5}$ and $\binom{10}{0}$, i.e. it is the union of two (one-dimensional) lines. This invariant is also the true optimal invariant of the loop.

In contrast, the octagonal invariant derived in Example 9 is a (two-dimensional) polygon. Indeed, every invariant for the outer loop that is expressed as a conjunction of traditional linear constraints is convex in the classical sense and therefore must contain at least the whole convex hull of I.

4 Concluding Remarks and Future Work

We have presented a new approach for investigating quantifier elimination of a subclass of formulas in Presburger arithmetic that exploits the structure of verification conditions generated from programs as well as the (lack of) interaction among different variables appearing in a verification condition. This approach has led to a geometric local method which is of quadratic complexity, much lower than the complexity of invariant generation using other approaches. Furthermore, the approach is amenable to development of additional heuristics that can additionally exploit the structure of verification conditions. The invariants generated by the proposed approach are of comparable strength to the invariants generated by Miné's method. To further improve the heuristics and assess the mentioned scalability issues arising from large path numbers, we are implementing the proposed approach and will experiment with it on a large class of benchmarks.

As should be evident from the discussion in the previous section, the work on max plus invariants is somewhat preliminary and still under progress. Of course, we are implementing the derivation of max plus constraints as well and, more importantly, investigate heuristics to make the approach scalable. While the method can easily be extended to more generators to specify more and more expressive disjunctive invariants, the complexity of quantifier elimination heuristics increases with the number of generators, due to the large number of possible shapes. We are currently exploring an alternative approach that uses a representation of max plus invariants by constraints instead of generators.

References

1. X. Allamigeon. *Static analysis of memory manipulations by abstract interpretation– Algorithmics of tropical polyhedra, and application to abstract interpretation.* PhD thesis, PhD thesis, Ecole Polytechnique, Palaiseau, France, 2009.
2. Xavier Allamigeon, Stphane Gaubert, and Goubault. Inferring min and max invariants using max-plus polyhedra. In Mara Alpuente and Germn Vidal, editors, *Static Analysis*, volume 5079 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2008.
3. R. Bagnara, P. M. Hill, and E. Zaffanella. An improved tight closure algorithm for integer octagonal constraints. In F. Logozzo, D. Peled, and L. Zuck, editors, *Verification, Model Checking and Abstract Interpretation: Proceedings of the 9th International Conference (VMCAI 2008)*, volume 4905 of *Lecture Notes in Computer Science*, pages 8–21, San Francisco, USA, 2008. Springer-Verlag, Berlin.
4. M. Bezem, R. Nieuwenhuis, and E. Rodriguez-Carbonell. Hard problems in max-algebra, control theory, hypergraphs and other areas. *Information Processing Letters*, 110(4):133–138, 2010.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. *The Essence of Computation*, pages 85–108, 2002.
6. Peter Butkovič. *Max-Linear Systems: Theory and Algorithms.* Springer Monographs in Mathematics. Springer, 2010.
7. Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking abstractions. In *ESOP*, pages 148–162, 2008.
8. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
9. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyzer. *Programming Languages and Systems*, pages 140–140, 2005.
10. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.
11. Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2:511–547, 1992.
12. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205, 2001.
13. Stephane Gaubert and Ricardo Katz. Max-plus convex geometry. In Renate A. Schmidt, editor, *RelMiCS*, volume 4136 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2006.
14. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 62–73. ACM, 2011.
15. Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292, 2008.
16. Sumit Gulwani and Ashish Tiwari. Constraint-based approach for analysis of hybrid systems. In *CAV*, pages 190–203, 2008.

17. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 232–244. ACM, 2004.
18. Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
19. J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Beyond finite domains. In *Principles and Practice of Constraint Programming*, pages 86–94. Springer, 1994.
20. B. Jeannet, M. Argoud, and G. Lalire. The Interproc interprocedural analyzer.
21. Susmit Jha, Sanjit A. Seshia, and Ashish Tiwari. Synthesis of optimal switching logic for hybrid systems. In *EMSOFT'11*, pages 107–116. ACM, 2011.
22. D. Kapur. A quantifier-elimination based heuristic for automatically generating inductive assertions for programs. *Journal of Systems Science and Complexity*, 19(3):307–330, 2006.
23. Deepak Kapur. Automatically generating loop invariants using quantifier elimination—preliminary report. Technical report, University of New Mexico, Albuquerque, NM, USA, 2004.
24. R. Loos and V. Weispfenning. Applying linear quantifier elimination. *Computer Journal*, 1993.
25. A. Miné. Weakly relational numerical abstract domains. *These de doctorat en informatique, École polytechnique, Palaiseau, France*, 2004.
26. S. Sankaranarayanan, H.B. Sipma, and Z. Manna. Non-linear Loop Invariant Generation using Gröbner Bases. *Symp. on Principles of Programming Languages*, 2004.
27. H. Sheini and K. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In *Theory and Applications of Satisfiability Testing*, pages 68–81. Springer, 2005.
28. Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.
29. Thomas Sturm and Ashish Tiwari. Verification and synthesis using real quantifier elimination. In *ISSAC'11*, pages 329–336. ACM, 2011.
30. Ankur Taly, Sumit Gulwani, and Ashish Tiwari. Synthesizing switching logic using constraint solving. *STTT*, 13(6):519–535, 2011.
31. Bican Xia and Zhihai Zhang. Termination of linear programs with nonlinear constraints. *J. Symb. Comput.*, 45(11):1234–1249, 2010.
32. Lu Yang, Chaochen Zhou, Naijun Zhan, and Bican Xia. Recent advances in program verification through computer algebra. *Frontiers of Computer Science in China*, 4(1):1–16, 2010.
33. Karel Zimmermann. A general separation theorem in extremal algebras. *Ekonomia Matematyczna Obzory*, 13:179–201, 1977.

A Arithmetic of Infinity

It is convenient to introduce extend $\mathbb{Z}$ with $-\infty$ and $+\infty$, to stand for no lower bound and no higher bound, respectively, for an arithmetic expression. Below, arithmetic operators and ordering relations used in this paper are extended to work on $-\infty$ and $+\infty$.

Addition

$$\begin{aligned}
&\forall a\ a \neq +\infty \implies a + (-\infty) = -\infty \\
&\forall b\ b \neq -\infty \implies b + (+\infty) = +\infty \\
&(+\infty) + (-\infty) \text{ is undefined}
\end{aligned}$$

Negation and Subtraction

$$\begin{aligned}
&-(-\infty) = +\infty,\ -(+\infty) = -\infty \\
&\forall a, b\ a - b = a + (-b)
\end{aligned}$$

Orderings

$$\begin{aligned}
&\forall a\ (a \neq +\infty) \implies (a < +\infty) \\
&\forall a\ (a \neq -\infty) \implies (a > -\infty) \\
&\forall a\ (a \in \mathbb{Z}) \implies ((+\infty) + (-\infty) > a) \\
&\forall a\ (a \in \mathbb{Z}) \implies ((+\infty) + (-\infty) < a)
\end{aligned}$$

In particular, the relations $\leq$ and $>$ are *not* complementary when infinities are involved, and neither are $\geq$ and $<$.

Moreover, the equivalence $a + b \leq a + c \iff b \leq c$ that is regularly used to simplify formulas over $\mathbb{Z}$ is *not* valid, for example $+\infty + 1 \leq +\infty + 0$, but $1 \not\leq 0$.