Nonlinear Polynomials, Interpolants and Invariant Generation for System Analysis

Deepak Kapur

Department of Computer Science
University of New Mexico
Albuquerque, NM, USA

with Rodriguez-Carbonell, Zhihai Zhang, Hengjun Zhao, Stephan Falke, Naijun Zhan, Ting Gan, Bican Xia and others
(work in progress)

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.
- Quantifier Elimination Approach for Generating (Loop) Invariants - Review with examples.

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.
- Quantifier Elimination Approach for Generating (Loop) Invariants - Review with examples.
- Geometric and Local Quantifier Elimination Heuristic

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.
- Quantifier Elimination Approach for Generating (Loop) Invariants - Review with examples.
- Geometric and Local Quantifier Elimination Heuristic
  - Octagonal Invariants - simple Convex Linear Constraints

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.
- Quantifier Elimination Approach for Generating (Loop) Invariants - Review with examples.
- Geometric and Local Quantifier Elimination Heuristic
  - Octagonal Invariants - simple Convex Linear Constraints
  - $O(k * n^2)$ algorithm – strongest invariant, k:the number of program paths and n: program variables

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.
- Quantifier Elimination Approach for Generating (Loop) Invariants - Review with examples.
- Geometric and Local Quantifier Elimination Heuristic
  - Octagonal Invariants - simple Convex Linear Constraints
  - $O(k * n^2)$ algorithm – strongest invariant, k:the number of program paths and n: program variables
  - Disjunctive Linear Invariants – **max and min** constraints

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.
- Quantifier Elimination Approach for Generating (Loop) Invariants - Review with examples.
- Geometric and Local Quantifier Elimination Heuristic
  - Octagonal Invariants - simple Convex Linear Constraints
  - $O(k * n^2)$ algorithm – strongest invariant, k:the number of program paths and n: program variables
  - Disjunctive Linear Invariants – **max and min** constraints
- Termination analysis using templates

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.
- Quantifier Elimination Approach for Generating (Loop) Invariants - Review with examples.
- Geometric and Local Quantifier Elimination Heuristic
  - Octagonal Invariants - simple Convex Linear Constraints
  - $O(k * n^2)$ algorithm – strongest invariant, k:the number of program paths and n: program variables
  - Disjunctive Linear Invariants – **max and min** constraints
- Termination analysis using templates
- Interpolant generation using quantifier elimination

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.
- Quantifier Elimination Approach for Generating (Loop) Invariants - Review with examples.
- Geometric and Local Quantifier Elimination Heuristic
  - Octagonal Invariants - simple Convex Linear Constraints
  - $O(k * n^2)$ algorithm – strongest invariant, k:the number of program paths and n: program variables
  - Disjunctive Linear Invariants – **max and min** constraints
- Termination analysis using templates
- Interpolant generation using quantifier elimination
- Generalization of IC^3, particularly showing completeness.

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.
- Quantifier Elimination Approach for Generating (Loop) Invariants - Review with examples.
- Geometric and Local Quantifier Elimination Heuristic
  - Octagonal Invariants - simple Convex Linear Constraints
  - $O(k * n^2)$ algorithm – strongest invariant, k:the number of program paths and n: program variables
  - Disjunctive Linear Invariants – **max and min** constraints
- Termination analysis using templates
- Interpolant generation using quantifier elimination
- Generalization of IC^3, particularly showing completeness.
- Saturation based approach for generating inductive invariants – computing abductors

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.
- Quantifier Elimination Approach for Generating (Loop) Invariants - Review with examples.
- Geometric and Local Quantifier Elimination Heuristic
  - Octagonal Invariants - simple Convex Linear Constraints
  - $O(k * n^2)$ algorithm – strongest invariant, k:the number of program paths and n: program variables
  - Disjunctive Linear Invariants – **max and min** constraints
- Termination analysis using templates
- Interpolant generation using quantifier elimination
- Generalization of IC^3, particularly showing completeness.
- Saturation based approach for generating inductive invariants – computing abductors
- Complexity barriers – localization (exploiting structure of verification conditions) and geometric heuristics relating preconditions vs post conditions.

Outline

- Ideal-theoretic approach for generating nonlinear polynomial equalities as invariants.
- Quantifier Elimination Approach for Generating (Loop) Invariants - Review with examples.
- Geometric and Local Quantifier Elimination Heuristic
  - Octagonal Invariants - simple Convex Linear Constraints
  - $O(k * n^2)$ algorithm – strongest invariant, k:the number of program paths and n: program variables
  - Disjunctive Linear Invariants – **max and min** constraints
- Termination analysis using templates
- Interpolant generation using quantifier elimination
- Generalization of IC^3, particularly showing completeness.
- Saturation based approach for generating inductive invariants – computing abductors
- Complexity barriers – localization (exploiting structure of verification conditions) and geometric heuristics relating preconditions vs post conditions.
- Challenges for symbolic computation community.

Invariants and Program Verification

- Building reliable and safe software is critical because of its use everywhere, especially in critical applications.

Invariants and Program Verification

- Building reliable and safe software is critical because of its use everywhere, especially in critical applications.

- Static analysis of software plays an important role. Examples are type checking, array bound check, null pointer check, ensuring behavioral specification, etc.

Invariants and Program Verification

- Building reliable and safe software is critical because of its use everywhere, especially in critical applications.
- Static analysis of software plays an important role. Examples are type checking, array bound check, null pointer check, ensuring behavioral specification, etc.
- Automation and scalability are critical for success.

Invariants: Integer Square Root

Example

```
x := 1, y := 1, z := 0;
while (x <= N) {
  x := x + y + 2;
  y := y + 2;
  z := z + 1
}
return z
```

Invariants: Integer Square Root

Example

```
x := 1, y := 1, z := 0;
while (x <= N) {
  x := x + y + 2;
  y := y + 2;
  z := z + 1
}
return z
```

$x = (z+1)^2$ is a loop invariant

Invariants: Integer Square Root

Example

```
x := 1, y := 1, z := 0;
while (x <= N) {
  x := x + y + 2;
  y := y + 2;
  z := z + 1
}
return z
```

$x = (z+1)^2$ is a loop invariant

Explore methods that can generate (strong) loop invariants (useful program properties) automatically for a large class of programs

Two Approaches for Generating Loop Invariants Automatically

1. Ideal-Theoretic Methods

- properties of programs specified by a conjunction of polynomial equations.

Two Approaches for Generating Loop Invariants Automatically

1. Ideal-Theoretic Methods

- properties of programs specified by a conjunction of polynomial equations.
- associated with every program location is an invariant (radical) ideal.

Two Approaches for Generating Loop Invariants Automatically

1. Ideal-Theoretic Methods

- properties of programs specified by a conjunction of polynomial equations.
- associated with every program location is an invariant (radical) ideal.
- Semantics of program constructs as ideal theoretic (algebraic varieties) operations – implemented using Gröbner basis computations.

Two Approaches for Generating Loop Invariants Automatically

1. Ideal-Theoretic Methods

- properties of programs specified by a conjunction of polynomial equations.
- associated with every program location is an invariant (radical) ideal.
- Semantics of program constructs as ideal theoretic (algebraic varieties) operations – implemented using Gröbner basis computations.
- Existence of a finite basis ensured by Hilbert's basis condition.

Two Approaches for Generating Loop Invariants Automatically

1. Ideal-Theoretic Methods

- properties of programs specified by a conjunction of polynomial equations.
- associated with every program location is an invariant (radical) ideal.
- Semantics of program constructs as ideal theoretic (algebraic varieties) operations – implemented using Gröbner basis computations.
- Existence of a finite basis ensured by Hilbert's basis condition.
- approximations and fixed point computation to generate such ideals.

Two Approaches for Generating Loop Invariants Automatically

1. Ideal-Theoretic Methods

- properties of programs specified by a conjunction of polynomial equations.
- associated with every program location is an invariant (radical) ideal.
- Semantics of program constructs as ideal theoretic (algebraic varieties) operations – implemented using Gröbner basis computations.
- Existence of a finite basis ensured by Hilbert's basis condition.
- approximations and fixed point computation to generate such ideals.

Two Approaches for Generating Loop Invariants Automatically

1. Ideal-Theoretic Methods

- properties of programs specified by a conjunction of polynomial equations.
- associated with every program location is an invariant (radical) ideal.
- Semantics of program constructs as ideal theoretic (algebraic varieties) operations – implemented using Gröbner basis computations.
- Existence of a finite basis ensured by Hilbert's basis condition.
- approximations and fixed point computation to generate such ideals.

Papers with Enric Rodríguez-Carbonell in ISSAC (2004), SAS (2004), ICTAC (2004), Science of Programming (2007), Journal of Symbolic Computation (2007)

2. Quantifier-Elimination of Program Variables from Parameterized Formulas

2. Quantifier-Elimination of Program Variables from Parameterized Formulas

Papers in ACA-2004, Journal of Systems Sciences and Complexity-2006

2. Quantifier-Elimination of Program Variables from Parameterized Formulas

Papers in ACA-2004, Journal of Systems Sciences and Complexity-2006

Geometric and Local Heuristics for Quantifier Elimination for Automatically Generating Octagonal Invariants

Papers in TAMC (2012), McCuneMemorial (2013).

2. Quantifier-Elimination of Program Variables from Parameterized Formulas

Papers in ACA-2004, Journal of Systems Sciences and Complexity-2006

Geometric and Local Heuristics for Quantifier Elimination for Automatically Generating Octagonal Invariants

Papers in TAMC (2012), McCuneMemorial (2013).

Interplay of Computational Logic and Algebra

Generating Loop Invariant: Approach

- Guess/fix the shape of invariants of interest at various program locations with some parameters which need to be determined.

Generating Loop Invariant: Approach

- Guess/fix the shape of invariants of interest at various program locations with some parameters which need to be determined.
- Here is an illustration of generation of nonlinear invariants.
  $\mathbf{I}: \quad A\ x^2 + B\ y^2 + C\ z^2 + D\ xy + E\ xz + F\ yz + G\ x + H\ y + J\ z + K = 0.$

Generating Loop Invariant: Approach

- Guess/fix the shape of invariants of interest at various program locations with some parameters which need to be determined.
- Here is an illustration of generation of nonlinear invariants.
  $\mathbf{I}: \quad A\ x^2 + B\ y^2 + C\ z^2 + D\ xy + E\ xz + F\ yz + G\ x + H\ y + J\ z + K = 0.$
- Generate verification conditions using the hypothesized invariants from the code.

Generating Loop Invariant: Approach

- Guess/fix the shape of invariants of interest at various program locations with some parameters which need to be determined.
- Here is an illustration of generation of nonlinear invariants.
  I : $A\ x^2 + B\ y^2 + C\ z^2 + D\ xy + E\ xz + F\ yz + G\ x + H\ y + J\ z + K = 0.$
- Generate verification conditions using the hypothesized invariants from the code.
  - **VC1:** At first possible entry of the loop (from initialization):

    $$A + B + D + G + H + K = 0.$$

Generating Loop Invariant: Approach

- Guess/fix the shape of invariants of interest at various program locations with some parameters which need to be determined.
- Here is an illustration of generation of nonlinear invariants.
  I : $A\ x^2 + B\ y^2 + C\ z^2 + D\ xy + E\ xz + F\ yz + G\ x + H\ y + J\ z + K = 0.$
- Generate verification conditions using the hypothesized invariants from the code.
  - **VC1:** At first possible entry of the loop (from initialization):

    $$A + B + D + G + H + K = 0.$$
  - **VC2:** For every iteration of the loop body:

    $$(I(x, y, z) \wedge x \leq N) \Longrightarrow I(x + y + 2, y + 2, z + 1).$$

Generating Loop Invariant: Approach

- Guess/fix the shape of invariants of interest at various program locations with some parameters which need to be determined.
- Here is an illustration of generation of nonlinear invariants.
  I : $A\ x^2 + B\ y^2 + C\ z^2 + D\ xy + E\ xz + F\ yz + G\ x + H\ y + J\ z + K = 0.$
- Generate verification conditions using the hypothesized invariants from the code.
  - **VC1:** At first possible entry of the loop (from initialization):

    $$A + B + D + G + H + K = 0.$$
  - **VC2:** For every iteration of the loop body:

    $$(I(x, y, z) \wedge x \leq N) \Longrightarrow I(x + y + 2, y + 2, z + 1).$$
- Using quantifier elimination, find constraints on parameters $A, B, C, D, E, F, G, H, J, K$ which ensure that the verification conditions are valid for all possible program variables.

Quantifier Elimination from Verification Conditions

Considering **VC2:**

- $(\ A\,x^2 + B\ y^2 + C\ z^2 + D\ xy + E\ xz + F\ yz + G\ x + H\ y + J\ z + K = 0) \Longrightarrow (\ A\,(x+y+2)^2 + B\ (y+2)^2 + C\ (z+1)^2 + D\ (x+y+2)(y+2) + E\ (x+y+2)(z+1) + F\ (y+2)(z+1) + G\ (x+y+2) + H\ (y+2) + J\ (z+1) + K = 0)$

Quantifier Elimination from Verification Conditions

Considering **VC2:**

- $(\; A\, x^2 + B\; y^2 + C\; z^2 + D\; xy + E\; xz + F\; yz + G\; x + H\; y + J\; z + K = 0) \Longrightarrow (\; A\,(x+y+2)^2 + B\;(y+2)^2 + C\;(z+1)^2 + D\;(x+y+2)(y+2) + E\;(x+y+2)(z+1) + F\;(y+2)(z+1) + G\;(x+y+2) + H\;(y+2) + J\;(z+1) + K = 0)$
- Expanding the conclusion gives:
  $Ax^2 + (A + B + D)y^2 + cz^2 + (D + 2A)xy + Exz + (E + F)yz + (G + 4A + 2D + E)x + (H + 4A + 4B + 4D + E + F + G)y + (J + 2C + 2E + 2F)z + (4A + 4B + C + 4D + 2E + 2F + 2G + 2H + J + K) = 0$

Quantifier Elimination from Verification Conditions

Considering **VC2:**

- $(\ A\,x^2 + B\,y^2 + C\,z^2 + D\,xy + E\,xz + F\,yz + G\,x + H\,y + J\,z + K = 0) \Longrightarrow (\ A\,(x+y+2)^2 + B\,(y+2)^2 + C\,(z+1)^2 + D\,(x+y+2)(y+2) + E\,(x+y+2)(z+1) + F\,(y+2)(z+1) + G\,(x+y+2) + H\,(y+2) + J\,(z+1) + K = 0)$
- Expanding the conclusion gives:
  $Ax^2 + (A+B+D)y^2 + cz^2 + (D+2A)xy + Exz + (E+F)yz + (G+4A+2D+E)x + (H+4A+4B+4D+E+F+G)y + (J+2C+2E+2F)z + (4A+4B+C+4D+2E+2F+2G+2H+J+K) = 0$
- Simplifying using the hypothesis gives:
  $(A+D)y^2 + 2Axy + Eyz + (4A+2D+E)x + (4A+4B+4D+E+F+G)y + (2C+2E+2F)z + (4A+4B+C+4D+2E+2F+2G+2H+J) = 0$

Quantifier Elimination from Verification Conditions

Considering **VC2:**

- $(\; A\,x^2 + B\,y^2 + C\,z^2 + D\,xy + E\,xz + F\,yz + G\,x + H\,y + J\,z + K = 0) \Longrightarrow (\; A\,(x+y+2)^2 + B\,(y+2)^2 + C\,(z+1)^2 + D\,(x+y+2)(y+2) + E\,(x+y+2)(z+1) + F\,(y+2)(z+1) + G\,(x+y+2) + H\,(y+2) + J\,(z+1) + K = 0)$
- Expanding the conclusion gives: $Ax^2 + (A+B+D)y^2 + cz^2 + (D+2A)xy + Exz + (E+F)yz + (G+4A+2D+E)x + (H+4A+4B+4D+E+F+G)y + (J+2C+2E+2F)z + (4A+4B+C+4D+2E+2F+2G+2H+J+K) = 0$
- Simplifying using the hypothesis gives: $(A+D)y^2 + 2Axy + Eyz + (4A+2D+E)x + (4A+4B+4D+E+F+G)y + (2C+2E+2F)z + (4A+4B+C+4D+2E+2F+2G+2H+J) = 0$
- Since this should be 0 for all values of x, y, z: we have: $A+D=0$; $A=0$; $E=0$ which implies $D=0$; using these gives: $2C+2F=0$ which implies $C=-F$; using all these: $G=-4B-F$, $H=-G-K-B$ and $J=-2B-F+2K$.

Generating the Strongest Invariant

- Constraints on parameters are:

$$C = -F, \quad J = -2B - F + 2K, \quad G = -4B - F, \quad H = 3B + F - K.$$

Generating the Strongest Invariant

- Constraints on parameters are:

  $$C = -F, \quad J = -2B - F + 2K, \quad G = -4B - F, \quad H = 3B + F - K.$$

- Every value of parameters satisfying the above constraints leads to an invariant (including the trivial invariant **true** when all parameter values are 0).

Generating the Strongest Invariant

- Constraints on parameters are:

$$C = -F, \quad J = -2B - F + 2K, \quad G = -4B - F, \quad H = 3B + F - K.$$

- Every value of parameters satisfying the above constraints leads to an invariant (including the trivial invariant **true** when all parameter values are 0).
- 7 parameters and 4 equations, so 3 independent parameters, say B, F, K. Making every independent parameter 1 separately with other independent parameters being 0, derive values of dependent parameters.

Generating the Strongest Invariant

- Constraints on parameters are:

  $$C = -F, \quad J = -2B - F + 2K, \quad G = -4B - F, \quad H = 3B + F - K.$$

- Every value of parameters satisfying the above constraints leads to an invariant (including the trivial invariant **true** when all parameter values are 0).
- 7 parameters and 4 equations, so 3 independent parameters, say B, F, K. Making every independent parameter 1 separately with other independent parameters being 0, derive values of dependent parameters.
- $K = 1, H = -1, J = 2$ gives $-y + 2z + 1 = 0$.

Generating the Strongest Invariant

- Constraints on parameters are:

  $$C = -F, \quad J = -2B - F + 2K, \quad G = -4B - F, \quad H = 3B + F - K.$$

- Every value of parameters satisfying the above constraints leads to an invariant (including the trivial invariant **true** when all parameter values are 0).
- 7 parameters and 4 equations, so 3 independent parameters, say B, F, K. Making every independent parameter 1 separately with other independent parameters being 0, derive values of dependent parameters.
- $K = 1, H = -1, J = 2$ gives $-y + 2z + 1 = 0$.
- $F = 1, C = -1, J = -1, G = -1, H = 1$ gives $-z^2 + yz - x + y - z - 0.$

Generating the Strongest Invariant

- Constraints on parameters are:

  $$C = -F, \quad J = -2B - F + 2K, \quad G = -4B - F, \quad H = 3B + F - K.$$

- Every value of parameters satisfying the above constraints leads to an invariant (including the trivial invariant **true** when all parameter values are 0).
- 7 parameters and 4 equations, so 3 independent parameters, say B, F, K. Making every independent parameter 1 separately with other independent parameters being 0, derive values of dependent parameters.
- $K = 1, H = -1, J = 2$ gives $-y + 2z + 1 = 0$.
- $F = 1, C = -1, J = -1, G = -1, H = 1$ gives $-z^2 + yz - x + y - z - 0$.
- $B = 1, J = -2, G = -4, H = 3$ gives $y^2 - 4x + 3y - 2z = 0$.

Generating the Strongest Invariant

- Constraints on parameters are:

  $$C = -F, \quad J = -2B - F + 2K, \quad G = -4B - F, \quad H = 3B + F - K.$$

- Every value of parameters satisfying the above constraints leads to an invariant (including the trivial invariant **true** when all parameter values are 0).
- 7 parameters and 4 equations, so 3 independent parameters, say B, F, K. Making every independent parameter 1 separately with other independent parameters being 0, derive values of dependent parameters.
- $K = 1, H = -1, J = 2$ gives $-y + 2z + 1 = 0$.
- $F = 1, C = -1, J = -1, G = -1, H = 1$ gives $-z^2 + yz - x + y - z - 0$.
- $B = 1, J = -2, G = -4, H = 3$ gives $y^2 - 4x + 3y - 2z = 0$.
- The most general invariant describing all invariants of the above form is a conjunction of:

  $$y = 2z + 1; \quad z^2 - yz + z + x - y = 0 \quad y^2 - 2z - 4x + 3y = 0,$$

  from which $x = (z+1)^2$ follows.

Method for Automatically Generating Invariants by Quantifier Elimination

- Hypothesize assertions, which are parametrized formulas, at various points in a program.

Method for Automatically Generating Invariants by Quantifier Elimination

- Hypothesize assertions, which are parametrized formulas, at various points in a program.
  - Typically entry of every loop and entry and exit of every procedure suffice.

Method for Automatically Generating Invariants by Quantifier Elimination

- Hypothesize assertions, which are parametrized formulas, at various points in a program.
  - Typically entry of every loop and entry and exit of every procedure suffice.
  - Nested loops and procedure/function calls can be handled.

Method for Automatically Generating Invariants by Quantifier Elimination

- Hypothesize assertions, which are parametrized formulas, at various points in a program.
  - Typically entry of every loop and entry and exit of every procedure suffice.
  - Nested loops and procedure/function calls can be handled.
- Generate verification conditions for every path in the program (a path from an assertion to another assertion including itself).

Method for Automatically Generating Invariants by Quantifier Elimination

- Hypothesize assertions, which are parametrized formulas, at various points in a program.
  - Typically entry of every loop and entry and exit of every procedure suffice.
  - Nested loops and procedure/function calls can be handled.
- Generate verification conditions for every path in the program (a path from an assertion to another assertion including itself).
  - Depending upon the logical language chosen to write invariants, approximations of assignments and test conditions may be necessary.

Method for Automatically Generating Invariants by Quantifier Elimination

- Hypothesize assertions, which are parametrized formulas, at various points in a program.
  - Typically entry of every loop and entry and exit of every procedure suffice.
  - Nested loops and procedure/function calls can be handled.
- Generate verification conditions for every path in the program (a path from an assertion to another assertion including itself).
  - Depending upon the logical language chosen to write invariants, approximations of assignments and test conditions may be necessary.
- Find a formula expressed in terms of parameters eliminating all program variables (using quantifier elimination).

Quality of Invariants

Soundness and Completeness

- Every assignment of parameter values which make the formula true, gives an inductive invariant.

Soundness and Completeness

- Every assignment of parameter values which make the formula true, gives an inductive invariant.
  - If no parameter values can be found, then invariants of hypothesized forms may not exist. Invariants can be guaranteed not to exist if no approximations are made, while generating verification conditions.

Quality of Invariants

Soundness and Completeness

- Every assignment of parameter values which make the formula true, gives an inductive invariant.
  - If no parameter values can be found, then invariants of hypothesized forms may not exist. Invariants can be guaranteed not to exist if no approximations are made, while generating verification conditions.
- If all assignments making the formula true can be finitely described, invariants generated may be the strongest of the hypothesized form. Invariants generated are guaranteed to be the strongest if no approximations are made, while generating verification conditions.

How to Scale this Approach

- Quantifier Elimination Methods typically do not scale up due to high complexity even in this restricted case of $\exists\forall$.

How to Scale this Approach

- Quantifier Elimination Methods typically do not scale up due to high complexity even in this restricted case of $\exists\forall$.
  - Even for Presburger arithmetic, complexity is doubly exponential in the number of quantifier alternations and triply exponential in the number of quantified variables

How to Scale this Approach

- Quantifier Elimination Methods typically do not scale up due to high complexity even in this restricted case of $\exists\forall$.
  - Even for Presburger arithmetic, complexity is doubly exponential in the number of quantifier alternations and triply exponential in the number of quantified variables
  - Output is huge and difficult to decipher.

How to Scale this Approach

- Quantifier Elimination Methods typically do not scale up due to high complexity even in this restricted case of $\exists\forall$.
  - Even for Presburger arithmetic, complexity is doubly exponential in the number of quantifier alternations and triply exponential in the number of quantified variables
  - Output is huge and difficult to decipher.
  - In practice, they often do not work (i.e., run out of memory or hang).

How to Scale this Approach

- Quantifier Elimination Methods typically do not scale up due to high complexity even in this restricted case of $\exists\forall$.
  - Even for Presburger arithmetic, complexity is doubly exponential in the number of quantifier alternations and triply exponential in the number of quantified variables
  - Output is huge and difficult to decipher.
  - In practice, they often do not work (i.e., run out of memory or hang).
- Linear constraint solving on rationals and reals (polyhedral domain), while of polynomial complexity, has been found in practice to be inefficient and slow, especially when used repeatedly as in abstract interpretation approach [Miné]

Making QE based Method Practical

- Identify (atomic) formulas and program abstractions resulting in verification conditions with good shape and structure.

Making QE based Method Practical

- Identify (atomic) formulas and program abstractions resulting in verification conditions with good shape and structure.
- Develop QE heuristics which exploit *local* structure of formulas (e.g. two variables at a time) and geometry of state space defined by formulas.

Making QE based Method Practical

- Identify (atomic) formulas and program abstractions resulting in verification conditions with good shape and structure.
- Develop QE heuristics which exploit *local* structure of formulas (e.g. two variables at a time) and geometry of state space defined by formulas.
- Among many possibilities in a result after QE, identify those most likely to be useful.

Making QE based Method Practical

- Identify (atomic) formulas and program abstractions resulting in verification conditions with good shape and structure.
- Develop QE heuristics which exploit *local* structure of formulas (e.g. two variables at a time) and geometry of state space defined by formulas.
- Among many possibilities in a result after QE, identify those most likely to be useful.
- **Octagonal formulas** : $l \leq \pm x \pm y \leq h$, a highly restricted subset of linear constraints (at most two variables with coefficients from $\{-1, 0, 1\}$).

Making QE based Method Practical

- Identify (atomic) formulas and program abstractions resulting in verification conditions with good shape and structure.
- Develop QE heuristics which exploit *local* structure of formulas (e.g. two variables at a time) and geometry of state space defined by formulas.
- Among many possibilities in a result after QE, identify those most likely to be useful.
- **Octagonal formulas** : $l \leq \pm x \pm y \leq h$, a highly restricted subset of linear constraints (at most two variables with coefficients from $\{-1, 0, 1\}$).
  - This fragment is the most expressive fragment of linear arithmetic over the integers with a polynomial time decision procedure.

Making QE based Method Practical

- Identify (atomic) formulas and program abstractions resulting in verification conditions with good shape and structure.
- Develop QE heuristics which exploit *local* structure of formulas (e.g. two variables at a time) and geometry of state space defined by formulas.
- Among many possibilities in a result after QE, identify those most likely to be useful.
- **Octagonal formulas** : $l \leq \pm x \pm y \leq h$, a highly restricted subset of linear constraints (at most two variables with coefficients from $\{-1, 0, 1\}$).
  - This fragment is the most expressive fragment of linear arithmetic over the integers with a polynomial time decision procedure.
- **Max, Min formulas**: $max(\pm x - l, \pm y - h)$, expressing disjunction $((x - l \geq y - h \wedge x - l \geq 0) \ \vee \ (y - h \geq x - h \wedge y - h \geq 0))$.

Making QE based Method Practical

- Identify (atomic) formulas and program abstractions resulting in verification conditions with good shape and structure.
- Develop QE heuristics which exploit *local* structure of formulas (e.g. two variables at a time) and geometry of state space defined by formulas.
- Among many possibilities in a result after QE, identify those most likely to be useful.
- **Octagonal formulas** : $l \leq \pm x \pm y \leq h$, a highly restricted subset of linear constraints (at most two variables with coefficients from $\{-1, 0, 1\}$).
  - This fragment is the most expressive fragment of linear arithmetic over the integers with a polynomial time decision procedure.
- **Max, Min formulas**: $max(\pm x - l, \pm y - h)$, expressing disjunction $((x - l \geq y - h \wedge x - l \geq 0) \ \vee \ (y - h \geq x - h \wedge y - h \geq 0))$.
- Combination of Octagonal and Max formulas.

Octagonal Formulas

- Octagonal formulas over two variables have a fixed shape. Its parameterization can be given using 8 parameters.

Octagonal Formulas

- Octagonal formulas over two variables have a fixed shape. Its parameterization can be given using 8 parameters.
- Given n variables, the most general formula (after simplification) is of the following form
  $$\bigwedge_{i,j}(\quad Octa_{i,j}: \quad a_{i,j} \leq x_i - x_j \leq b_{i,j}, \quad c_{i,j} \leq x_i + x_j \leq d_{i,j} \\ e_i \leq x_i \leq f_i \quad g_j \leq x_j \leq h_j)$$
  for every pair of variables x_i, x_j, where $a_{i,j}$, $b_{i,j}$, $c_{i,j}$, $d_{i,j}$, e_i, f_i, g_j, h_j are parameters.

Octagonal Formulas

- Octagonal formulas over two variables have a fixed shape. Its parameterization can be given using 8 parameters.
- Given n variables, the most general formula (after simplification) is of the following form
  $$\bigwedge_{i,j}(\quad Octa_{i,j}: \quad a_{i,j} \leq x_i - x_j \leq b_{i,j}, \quad c_{i,j} \leq x_i + x_j \leq d_{i,j}$$
  $$e_i \leq x_i \leq f_i \quad g_j \leq x_j \leq h_j)$$
  for every pair of variables x_i, x_j, where $a_{i,j}$, $b_{i,j}$, $c_{i,j}$, $d_{i,j}$, e_i, f_i, g_j, h_j are parameters.
- Class of programs that can be analyzed are very restricted. Still using octagonal constraints (and other heuristics), ASTREE is able to successfully analyze hundreds of thousands of lines of code of numerical software for array bound check, memory faults, and related bugs.

Octagonal Formulas

- Octagonal formulas over two variables have a fixed shape. Its parameterization can be given using 8 parameters.
- Given n variables, the most general formula (after simplification) is of the following form
  $\bigwedge_{i,j}(\quad Octa_{i,j} : \quad a_{i,j} \leq x_i - x_j \leq b_{i,j}, \quad c_{i,j} \leq x_i + x_j \leq d_{i,j}$
  $\quad e_i \leq x_i \leq f_i \quad g_j \leq x_j \leq h_j)$
  for every pair of variables x_i, x_j, where $a_{i,j}$, $b_{i,j}$, $c_{i,j}$, $d_{i,j}$, e_i, f_i, g_j, h_j are parameters.
- Class of programs that can be analyzed are very restricted. Still using octagonal constraints (and other heuristics), ASTREE is able to successfully analyze hundreds of thousands of lines of code of numerical software for array bound check, memory faults, and related bugs.
  - Algorithms used in ASTREE are of $O(n^3)$ complexity (sometimes, $O(n^4)$), where n is the number of variables (Miné, 2003).

Octagonal Formulas

- Octagonal formulas over two variables have a fixed shape. Its parameterization can be given using 8 parameters.
- Given n variables, the most general formula (after simplification) is of the following form
  $$\bigwedge_{i,j}(\quad Octa_{i,j}: \quad a_{i,j} \leq x_i - x_j \leq b_{i,j}, \quad c_{i,j} \leq x_i + x_j \leq d_{i,j}$$
  $$e_i \leq x_i \leq f_i \quad g_j \leq x_j \leq h_j)$$
  for every pair of variables x_i, x_j, where $a_{i,j}$, $b_{i,j}$, $c_{i,j}$, $d_{i,j}$, e_i, f_i, g_j, h_j are parameters.
- Class of programs that can be analyzed are very restricted. Still using octagonal constraints (and other heuristics), ASTREE is able to successfully analyze hundreds of thousands of lines of code of numerical software for array bound check, memory faults, and related bugs.
  - Algorithms used in ASTREE are of $O(n^3)$ complexity (sometimes, $O(n^4)$), where n is the number of variables (Miné, 2003).
- **Goal:** Performance of QE heuristic should be at least as good.

A Simple Example

Example

```
 x := 4; y := 6;
while (x + y >= 0) do
 if (y >= 6) then { x := -x;  y := y - 1 }
    else { x := x - 1; y := -y }
endwhile
```

A Simple Example

Example

```
 x := 4; y := 6;
while (x + y >= 0) do
 if (y >= 6) then { x := -x;   y := y - 1 }
     else { x := x - 1; y := -y }
endwhile
```

VC0: $I(4, 6)$
VC1: $(I(x, y) \wedge (x + y) \geq 0 \wedge y \geq 6) \implies I(-x, y - 1)$.
VC2: $(I(x, y) \wedge (x + y) \geq 0 \wedge y < 6) \implies I(x - 1, -y)$.

Approach: Local QE Heuristics

- A program path is a sequence of assignment statements interspersed with tests. Its behavior may have to be approximated to generate the post condition in which both the hypothesis and the conclusion are each conjunctions of atomic octagonal formulas.

Approach: Local QE Heuristics

- A program path is a sequence of assignment statements interspersed with tests. Its behavior may have to be approximated to generate the post condition in which both the hypothesis and the conclusion are each conjunctions of atomic octagonal formulas.
- A verification condition is expressed using atomic formulas that are all octagonal constraints.

$$\bigwedge_{i,j} \left((Octa_{i,j} \wedge \alpha(x_i, x_j)) \Rightarrow Octa'_{i,j} \right),$$

along with additional parameter-free constraints $\alpha(x_i, x_j)$, of the same form in which lower and upper bounds are constants.

Approach: Local QE Heuristics

- A program path is a sequence of assignment statements interspersed with tests. Its behavior may have to be approximated to generate the post condition in which both the hypothesis and the conclusion are each conjunctions of atomic octagonal formulas.
- A verification condition is expressed using atomic formulas that are all octagonal constraints.

$$\bigwedge_{i,j} \left((Octa_{i,j} \wedge \alpha(x_i, x_j)) \Rightarrow Octa'_{i,j} \right),$$

  along with additional parameter-free constraints $\alpha(x_i, x_j)$, of the same form in which lower and upper bounds are constants.
- Analysis of a big conjunctive constraint on every possible pair of variables can be considered individually by considering the subformula on each distinct pair.

Geometric QE Heuristic

- Analyze how a general octagon gets transformed due to assignments. For each assignment case, a table is built showing the effect on the parameter values.

Geometric QE Heuristic

- Analyze how a general octagon gets transformed due to assignments. For each assignment case, a table is built showing the effect on the parameter values.
- Identify conditions under which the transformed octagon includes the portion of the original octagon satisfying tests along a program path. This is guided again locally for every side of the octagon,

Geometric QE Heuristic

- Analyze how a general octagon gets transformed due to assignments. For each assignment case, a table is built showing the effect on the parameter values.
- Identify conditions under which the transformed octagon includes the portion of the original octagon satisfying tests along a program path. This is guided again locally for every side of the octagon,
- In the case of many possibilities, the one likely to generate the most useful invariant is identified.

Geometric QE Heuristic

- Analyze how a general octagon gets transformed due to assignments. For each assignment case, a table is built showing the effect on the parameter values.
- Identify conditions under which the transformed octagon includes the portion of the original octagon satisfying tests along a program path. This is guided again locally for every side of the octagon,
- In the case of many possibilities, the one likely to generate the most useful invariant is identified.
- Quantifier elimination heuristics to generate constraints on lower and upper bounds by table look ups in $O(n^2)$ steps, where n is the number of program variables.

Table 3: Sign of exactly one variable is changed

$x := -x + A$
$y := y + B$

$$\Delta_1 = A - B, \quad \Delta_2 = A + B.$$

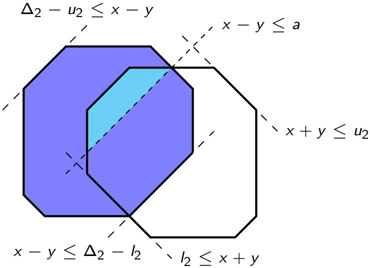
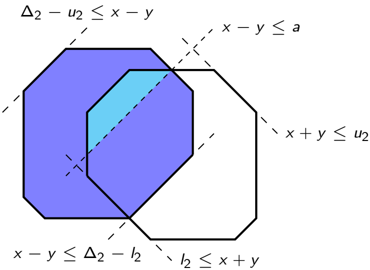


Table 3: Sign of exactly one variable is changed

$$x := -x + A$$
$$y := y + B$$

$$\Delta_1 = A - B, \qquad \Delta_2 = A + B.$$



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

A Simple Example

Example

```
 x := 4; y := 6;
while (x + y >= 0) do
 if (y >= 6) then { x := -x;  y := y - 1 }
    else { x := x - 1; y := -y }
endwhile
```

A Simple Example

Example

```
 x := 4; y := 6;
while (x + y >= 0) do
 if (y >= 6) then { x := -x;  y := y - 1 }
     else { x := x - 1; y := -y }
endwhile
```

VC0: $I(4, 6)$
VC1: $(I(x, y) \wedge (x + y) \geq 0 \wedge y \geq 6) \implies I(-x, y - 1)$.
VC2: $(I(x, y) \wedge (x + y) \geq 0 \wedge y < 6) \implies I(x - 1, -y)$.

Generating Constraints on Parameters

- **VC0:**
  $l_1 \leq -2 \leq u_1 \wedge l_2 \leq 10 \leq u_2 \wedge l_3 \leq 4 \leq u_3 \wedge l_4 \leq 6 \leq u_4.$

Generating Constraints on Parameters

- **VC0:**
  $l_1 \leq -2 \leq u_1 \wedge l_2 \leq 10 \leq u_2 \wedge l_3 \leq 4 \leq u_3 \wedge l_4 \leq 6 \leq u_4$.
- **VC1:** $x - y$: $-u_2 - 1 \leq l_1 \wedge u_1 \leq -l_2 - 1$.
  $x + y$: $-u_1 + 1 \leq 0 \wedge u_2 \leq -l_1 + 1$.
  x: $l_3 + u_3 = 0$.
  y: $l_4 \leq 5$.

Generating Constraints on Parameters

- **VC0:**
  $l_1 \le -2 \le u_1 \wedge l_2 \le 10 \le u_2 \wedge l_3 \le 4 \le u_3 \wedge l_4 \le 6 \le u_4$.
- **VC1:** $x - y$: $-u_2 - 1 \le l_1 \wedge u_1 \le -l_2 - 1$.
  $x + y$: $-u_1 + 1 \le 0 \wedge u_2 \le -l_1 + 1$.
  x: $l_3 + u_3 = 0$.
  y: $l_4 \le 5$.
- **VC2:** $x - y$: $-u_2 - 1 \le -u_1 \wedge 10 \le -l_2 - 1$.
  $x + y$: $l_1 + 1 \le 0 \wedge u_2 \le u_1 + 1$.
  x: $l_3 \le -6$.
  y: $-u_4 \le l_4 \wedge 5 \le -l_4$.

Generating Constraints on Parameters

- **VC0:**
  $l_1 \leq -2 \leq u_1 \wedge l_2 \leq 10 \leq u_2 \wedge l_3 \leq 4 \leq u_3 \wedge l_4 \leq 6 \leq u_4.$
- **VC1:** $x - y$: $-u_2 - 1 \leq l_1 \wedge u_1 \leq -l_2 - 1.$
  $x + y$: $-u_1 + 1 \leq 0 \wedge u_2 \leq -l_1 + 1.$
  x: $l_3 + u_3 = 0.$
  y: $l_4 \leq 5.$
- **VC2:** $x - y$: $-u_2 - 1 \leq -u_1 \wedge 10 \leq -l_2 - 1.$
  $x + y$: $l_1 + 1 \leq 0 \wedge u_2 \leq u_1 + 1.$
  x: $l_3 \leq -6.$
  y: $-u_4 \leq l_4 \wedge 5 \leq -l_4.$
- Make l_i's as large as possible and u_i's as small as possible:
  $l_1 = -10, u_1 = 9, l_2 = -11, u_2 = 10,$
  $l_3 = -6, u_3 = 6, l_4 = -5, u_4 = 6.$

Generating Constraints on Parameters

- **VC0:**
  $l_1 \le -2 \le u_1 \wedge l_2 \le 10 \le u_2 \wedge l_3 \le 4 \le u_3 \wedge l_4 \le 6 \le u_4$.
- **VC1:** $x - y$: $-u_2 - 1 \le l_1 \wedge u_1 \le -l_2 - 1$.
  $x + y$: $-u_1 + 1 \le 0 \wedge u_2 \le -l_1 + 1$.
  x: $l_3 + u_3 = 0$.
  y: $l_4 \le 5$.
- **VC2:** $x - y$: $-u_2 - 1 \le -u_1 \wedge 10 \le -l_2 - 1$.
  $x + y$: $l_1 + 1 \le 0 \wedge u_2 \le u_1 + 1$.
  x: $l_3 \le -6$.
  y: $-u_4 \le l_4 \wedge 5 \le -l_4$.
- Make l_i's as large as possible and u_i's as small as possible:
  $l_1 = -10, u_1 = 9, l_2 = -11, u_2 = 10,$
  $l_3 = -6, u_3 = 6, l_4 = -5, u_4 = 6$.
- The corresponding invariant is:
  $-10 \le x - y \le 9 \wedge -11 \le x + y \le 10$
  $\wedge -6 \le x \le 6 \wedge -5 \le y \le 6$.

Generating Invariants using Table Look-ups

- Parameter constraints corresponding to a specific program path are read from the corresponding entries in tables.

Generating Invariants using Table Look-ups

- Parameter constraints corresponding to a specific program path are read from the corresponding entries in tables.
- Accumulate all such constraints on parameter values. They are also **octagonal**.

Generating Invariants using Table Look-ups

- Parameter constraints corresponding to a specific program path are read from the corresponding entries in tables.
- Accumulate all such constraints on parameter values. They are also **octagonal**.
- Every parameter value that satisfies the parameter constraints leads to an invariant.

Generating Invariants using Table Look-ups

- Parameter constraints corresponding to a specific program path are read from the corresponding entries in tables.
- Accumulate all such constraints on parameter values. They are also **octagonal**.
- Every parameter value that satisfies the parameter constraints leads to an invariant.
- Maximum values of lower bounds and minimal values of upper bounds satisfying the parameter constraints gives the strongest invariants. Maximum and minimum values can be computed using Floyd-Warshall's algorithm.

Complexity and Parallelization

- Overall Complexity: $O(k * n^2)$:

Complexity and Parallelization

- Overall Complexity: $O(k * n^2)$:
  - For every pair of program variables, parametric constraint generation is constant time: 8 constraints, so 8 entries.

Complexity and Parallelization

- Overall Complexity: $O(k * n^2)$:
  - For every pair of program variables, parametric constraint generation is constant time: 8 constraints, so 8 entries.
  - Parametric constraints are decomposed based on parameters appearing in them: there are $O(n^2)$ such constraints on disjoint blocks of parameters of size ≤ 4.

Complexity and Parallelization

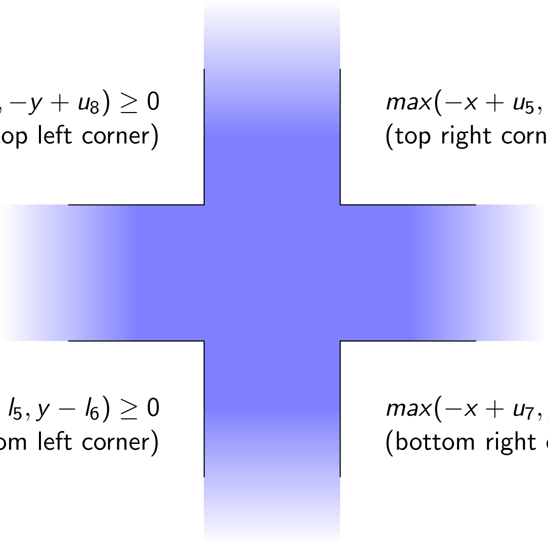
- Overall Complexity: $O(k * n^2)$:
  - For every pair of program variables, parametric constraint generation is constant time: 8 constraints, so 8 entries.
  - Parametric constraints are decomposed based on parameters appearing in them: there are $O(n^2)$ such constraints on disjoint blocks of parameters of size ≤ 4.
- Program paths can be analyzed in parallel. Parametric constraints can be processed in parallel.

Max Formulas

Pictorial representation of all possible cases of $max(\pm x + l, \pm y + h)$. Observe that every defined region is nonconvex.

$max(x - l_8, -y + u_8) \geq 0$
(top left corner)

$max(-x + u_5, -y + u_6) \geq 0$
(top right corner)



$max(x - l_5, y - l_6) \geq 0$
(bottom left corner)

$max(-x + u_7, y - l_7) \geq 0$
(bottom right corner)

Max Formulas

A typical template: octagonal formulas and max formulas.

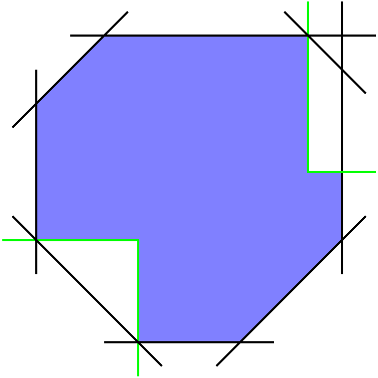
$$\begin{aligned}
l_1 \leq & \quad x - y & \leq u_1 \\
l_2 \leq & \quad x + y & \leq u_2 \\
l_3 \leq & \quad\;\; x & \leq u_3 \\
l_4 \leq & \quad\;\; y & \leq u_4
\end{aligned}$$

$$\begin{aligned}
max(\;\; x - \; l_5, \;\; y - \; l_6) &\geq 0 \\
max(\;\; x - \; l_8, -y + u_8) &\geq 0 \\
max(-x + u_7, \;\; y - \; l_7) &\geq 0 \\
max(-x + u_5, -y + u_6) &\geq 0
\end{aligned}$$

Max Formulas – some nonconvex regions

An octagon with two corners cut out.

A square that turns into 2 disconnected components.



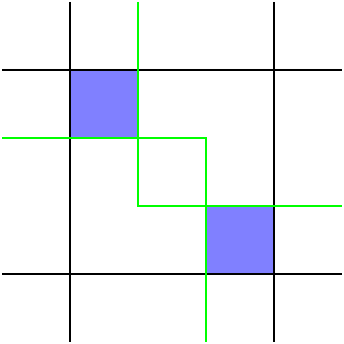


Table 6: Parametric Constraints for assignments with sign of one variable reversed

Assignments: $x := -x + A, y := y + B$

Bottom left and bottom right corners:
$max(x - l_5, y - l_6) \geq 0$ and $max(-x + u_7, y - l_7) \geq 0$

	$y \geq h$ absent	$y \geq h$ present
$x \geq f$ absent	$(l_5 + u_7 \geq A \wedge l_7 - l_6 \leq B)$ $\vee\, l_5 + u_7 \leq A \vee l_7 - l_4 \leq B$ $\vee\, l_2 - l_7 + u_7 \geq A - B$	$l_7 \leq h + B$
$x \geq f$ present	$u_7 \geq -f + A$	$u_7 \geq -f + A \vee l_6 \leq h + B$

The constraints for two absent tests can also be used as disjuncts in the other cases.

	$y \geq h$ absent	$y \geq h$ present
$x \leq e$ absent	$(l_5 + u_7 \leq A \wedge l_6 - l_7 \leq B)$ $\vee\, l_5 + u_3 \leq A \vee l_6 - l_4 \leq B$ $\vee\, l_5 + l_6 - u_1 \geq A + B$	$l_6 \leq h + B$
$x \leq e$ present	$l_5 \leq -e + A$	$l_5 \leq -e + A \vee l_6 \leq h + B$

The constraints for two absent tests can also be used as disjuncts in the other cases.

Table 6 Contd: Parametric Constraints for assignments with sign of one variable reversed

Assignments: $x := -x + A, y := y + B$

Top left and top right corners:
$max(x - l_8, -y + u_8) \geq 0$ and $max(-x + u_5, -y + u_6) \geq 0$

	$y \leq g$ absent	$y \leq g$ present
$x \geq f$ absent	$(l_8 + u_5 \geq A \wedge u_6 - u_8 \geq B)$ $\vee\, l_3 + u_5 \geq A \vee u_6 - u_4 \geq B$ $\vee\, l_1 + u_5 + u_6 \geq A + B$	$u_6 \geq g + B$
$x \geq f$ present	$u_5 \geq -f + A$	$u_5 \leq -f + A \vee u_6 \geq g + B$

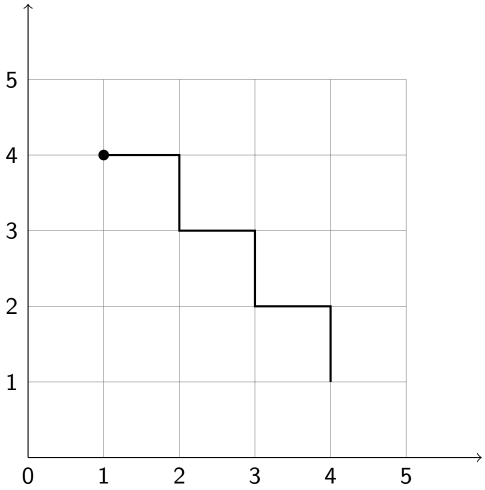
The constraints for two absent tests can also be used as disjuncts in the other cases.

	$y \leq g$ absent	$y \leq g$ present
$x \leq e$ absent	$(l_8 + u_5 \leq A \wedge u_8 - u_6 \geq B)$ $\vee\, l_8 + u_3 \leq A \vee u_8 - u_4 \geq B$ $\vee\, l_8 + u_2 - u_8 \leq A - B$	$u_8 \geq g + B$
$x \leq e$ present	$l_8 \leq -e + A$	$l_8 \leq -e + A \vee u_8 \geq g + B$

The constraints for two absent tests can also be used as disjuncts in the other cases.

Example: Stairs Program

```
x := 1; y := 4;
while (y>1) {
  if (x<2)
    x++;
  else if (y>3)
    y--;
  else if (x<3)
    x++;
  else if (y>2)
    y--;
  else if (x<4)
    x++;
  else
    y--;
}
```



Example: Stairs Program

Parametric Constraints due to the initialization `x := 1; y := 4`:

$$
\begin{aligned}
l_1 \leq -3 \leq u_1 \qquad & l_5 \leq 1 \vee l_6 \leq 4 \\
l_2 \leq 5 \leq u_2 \qquad & u_5 \geq 1 \vee u_6 \geq 4 \\
l_3 \leq 1 \leq u_3 \qquad & l_7 \leq 1 \vee u_7 \geq 4 \\
l_4 \leq 4 \leq u_4 \qquad & l_8 \geq 1 \vee u_8 \geq 4
\end{aligned}
$$

Example: Stairs Program

Parametric Constraints from Table look up for

- Program paths in which x is increasing:

$$u_1 = +\infty \qquad u_5 \geq 2$$
$$u_2 = +\infty \qquad u_5 \geq 3 \vee u_6 \geq 3$$
$$u_3 \geq 2 \qquad u_5 \geq 4 \vee u_6 \geq 2$$
$$u_3 \geq 3$$
$$u_3 \geq 4$$

- Program Paths in which y is decreasing:

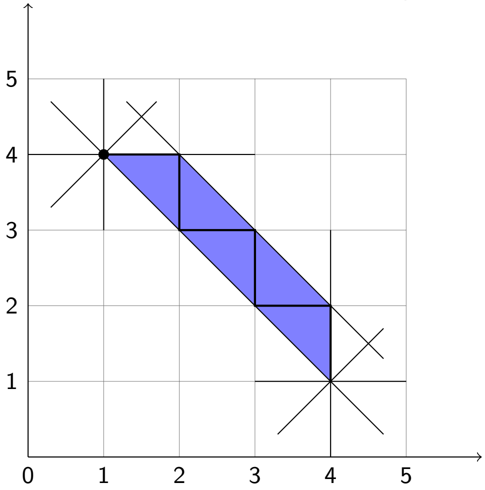
$$u_1 = +\infty \qquad l_5 \geq 2 \vee l_6 \geq 5$$
$$l_2 = -\infty \qquad l_5 \geq 3 \vee l_6 \geq 4$$
$$l_4 \leq 3 \qquad l_5 \geq 4 \vee l_6 \geq 3$$
$$l_4 \leq 2$$
$$l_4 \leq 1$$

Example: Stairs Program

Putting all parametric constraints together and deriving the strongest max invariant

(contrasted with the strongest octagonal invariant)

```
x := 1; y := 4;
while (y>1) {
  if (x<2)
    x++;
  else if (y>3)
    y--;
  else if (x<3)
    x++;
  else if (y>2)
    y--;
  else if (x<4)
    x++;
  else
    y--;
}
```

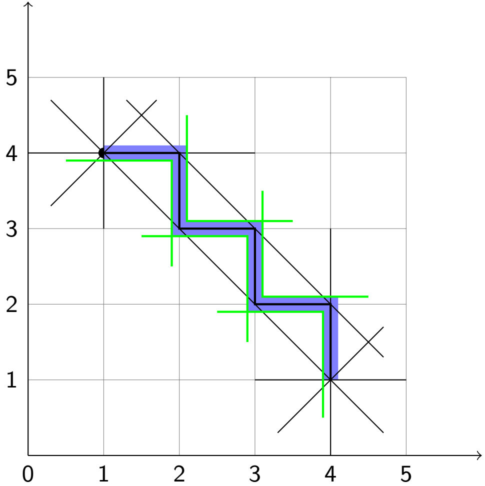


Example: Stairs Program

Putting all parametric constraints together and deriving the strongest max invariant

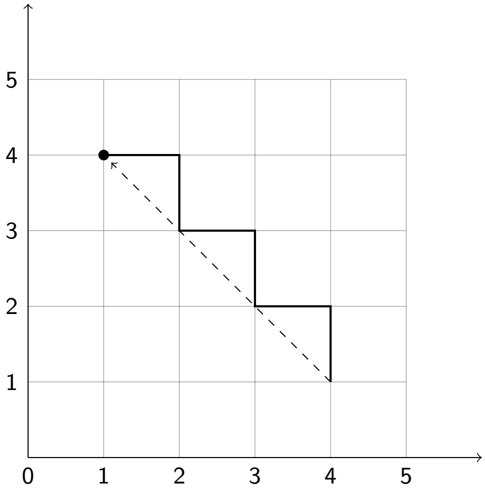
(contrasted with the strongest octagonal invariant)

```
x := 1; y := 4;
while (y>1) {
  if (x<2)
    x++;
  else if (y>3)
    y--;
  else if (x<3)
    x++;
  else if (y>2)
    y--;
  else if (x<4)
    x++;
  else
    y--;
}
```



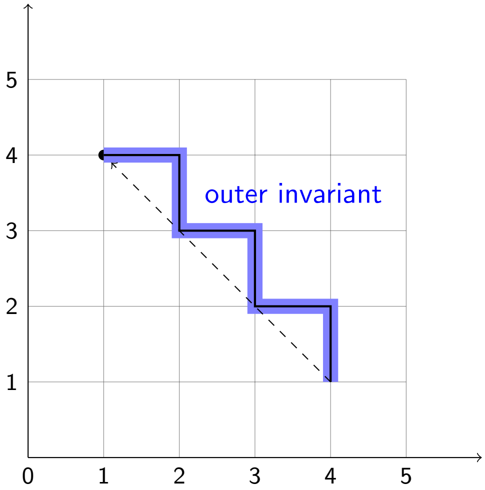
Invariants of a Program with a nested loop

```
x := 1; y := 4;
while (true) {
  if (x<2)
    x++;
  else if (y>3)
    y--;
  else if (x<3)
    x++;
  else if (y>2)
    y--;
  else if (x<4)
    x++;
  else if (y>1)
    y--;
  else
    while (x>1) {
      x--; y++;
    }
}
```



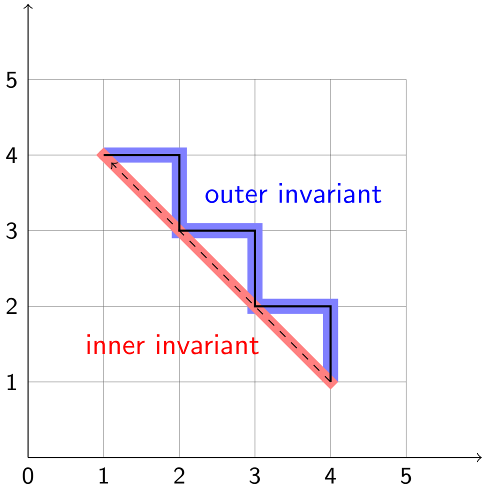
Invariants of a Program with a nested loop

```
x := 1; y := 4;
while (true) {
  if (x<2)
    x++;
  else if (y>3)
    y--;
  else if (x<3)
    x++;
  else if (y>2)
    y--;
  else if (x<4)
    x++;
  else if (y>1)
    y--;
  else
    while (x>1) {
      x--; y++;
    }
}
```



Invariants of a Program with a nested loop

```
x := 1; y := 4;
while (true) {
  if (x<2)
    x++;
  else if (y>3)
    y--;
  else if (x<3)
    x++;
  else if (y>2)
    y--;
  else if (x<4)
    x++;
  else if (y>1)
    y--;
  else
    while (x>1) {
      x--; y++;
    }
}
```



Max Invariants vs Octagonal Invariants

- 16 instead of 8 parameters per variable pair:
  $l_1, u_1, \ldots, l_4, u_4, \quad l_5, u_5, \ldots, l_8, u_8$

Max Invariants vs Octagonal Invariants

- 16 instead of 8 parameters per variable pair:
  $l_1, u_1, \ldots, l_4, u_4, \quad l_5, u_5, \ldots, l_8, u_8$
- **Octagon**: Unique optimal values for parameters.
  Max: Multiple noncomparable values for parameter tuples.
  (recall the step function invariant before)

$$\frac{max(x - l_5, y - l_6) \geq 0 \qquad max(-x + u_5, -y + u_6) \geq 0}{\begin{array}{ll} max(x-2, y-4) \geq 0 & max(-x+2, -y+3) \geq 0 \\ max(x-3, y-3) \geq 0 & max(-x+3, -y+2) \geq 0 \\ max(x-4, y-2) \geq 0 & \end{array}}$$

Max Invariants vs Octagonal Invariants

- 16 instead of 8 parameters per variable pair:
  $l_1, u_1, \ldots, l_4, u_4, \quad l_5, u_5, \ldots, l_8, u_8$
- **Octagon**: Unique optimal values for parameters.
  Max: Multiple noncomparable values for parameter tuples.
  (recall the step function invariant before)

$$\begin{array}{ll} max(x - l_5, y - l_6) \geq 0 & max(-x + u_5, -y + u_6) \geq 0 \\ \hline max(x-2, y-4) \geq 0 & max(-x+2, -y+3) \geq 0 \\ max(x-3, y-3) \geq 0 & max(-x+3, -y+2) \geq 0 \\ max(x-4, y-2) \geq 0 & \end{array}$$

- Many disjunctions in Tables.

Max Invariants vs Octagonal Invariants

- 16 instead of 8 parameters per variable pair:
  $l_1, u_1, \ldots, l_4, u_4, \quad l_5, u_5, \ldots, l_8, u_8$
- **Octagon**: Unique optimal values for parameters.
  Max: Multiple noncomparable values for parameter tuples.
  (recall the step function invariant before)

$$\frac{max(x - l_5, y - l_6) \geq 0 \qquad max(-x + u_5, -y + u_6) \geq 0}{\begin{array}{ll} max(x-2, y-4) \geq 0 & max(-x+2, -y+3) \geq 0 \\ max(x-3, y-3) \geq 0 & max(-x+3, -y+2) \geq 0 \\ max(x-4, y-2) \geq 0 & \end{array}}$$

- Many disjunctions in Tables.
  - Experimentation and heuristics for determining possibilities in disjunctions that are more useful.

Max Invariants vs Octagonal Invariants

- 16 instead of 8 parameters per variable pair:
  $l_1, u_1, \ldots, l_4, u_4, \quad l_5, u_5, \ldots, l_8, u_8$
- **Octagon**: Unique optimal values for parameters.
  Max: Multiple noncomparable values for parameter tuples.
  (recall the step function invariant before)

$$\frac{max(x - l_5, y - l_6) \geq 0 \qquad max(-x + u_5, -y + u_6) \geq 0}{\begin{array}{ll} max(x-2, y-4) \geq 0 & max(-x+2, -y+3) \geq 0 \\ max(x-3, y-3) \geq 0 & max(-x+3, -y+2) \geq 0 \\ max(x-4, y-2) \geq 0 & \end{array}}$$

- Many disjunctions in Tables.
  - Experimentation and heuristics for determining possibilities in disjunctions that are more useful.
  - Sacrificing efficiency to generate stronger invariants.

Max Invariants vs Octagonal Invariants

- 16 instead of 8 parameters per variable pair:
  $l_1, u_1, \ldots, l_4, u_4, \quad l_5, u_5, \ldots, l_8, u_8$
- **Octagon**: Unique optimal values for parameters.
  Max: Multiple noncomparable values for parameter tuples.
  (recall the step function invariant before)

$$\frac{max(x - l_5, y - l_6) \geq 0 \qquad max(-x + u_5, -y + u_6) \geq 0}{\begin{array}{ll} max(x-2, y-4) \geq 0 & max(-x+2, -y+3) \geq 0 \\ max(x-3, y-3) \geq 0 & max(-x+3, -y+2) \geq 0 \\ max(x-4, y-2) \geq 0 & \end{array}}$$

- Many disjunctions in Tables.
  - Experimentation and heuristics for determining possibilities in disjunctions that are more useful.
  - Sacrificing efficiency to generate stronger invariants.
- Same asymptotic complexity if a single parametric constraint in every table entry is selected.

Termination Analysis based on Quantifier Elimination

Ranking functions can be synthesized by hypothesizing poiynomials in program variables and unary predicates on program variable in a loop body.

Example

```
while (n>1) {
   if  n mod 2 = 0 then n := n/2
        else  n := n+1
}
```

Termination Analysis based on Quantifier Elimination

Ranking functions can be synthesized by hypothesizing poiynomials in program variables and unary predicates on program variable in a loop body.

Example

```
while (n>1) {
   if  n mod 2 = 0 then n := n/2
       else  n := n+1
}
```

Theorem There does not exist any polynomial in n that can serve as a ranking function.

Termination Analysis based on Quantifier Elimination

The synthesis of a polynomial ranking function of arbitrary degree can be hypothesized: much like verification conditions, leading to two constraints:

Termination Analysis based on Quantifier Elimination

The synthesis of a polynomial ranking function of arbitrary degree can be hypothesized: much like verification conditions, leading to two constraints:

1. $n \ mod \ 2 = 0$: easy.
2. otherwise: $n' = n + 1$, so tricky.

Termination Analysis based on Quantifier Elimination

The synthesis of a polynomial ranking function of arbitrary degree can be hypothesized: much like verification conditions, leading to two constraints:

1. $n \ mod \ 2 = 0$: easy.
2. otherwise: $n' = n + 1$, so tricky.

Must use the function $n \ mod \ 2$. Consider $n + 2(n \ mod \ 2)$ as a possible ranking function (which can be generated from $An + B(n \ mod \ 2) + C$).

1. $n \ mod \ 2 = 0$: tricky but with the loop condition $n > 1$, easy.
2. otherwise: $n' = n + 1$: easy.

Interpolant Generation using Quantifier Elimination

Craig: Given $\alpha \implies \beta$, an intermediate formula γ in common symbols of α and β exists and can be constructed such that

$$\alpha \implies \gamma \wedge \gamma \implies \beta$$

Interpolant Generation using Quantifier Elimination

Craig: Given $\alpha \implies \beta$, an intermediate formula γ in common symbols of α and β exists and can be constructed such that

$$\alpha \implies \gamma \wedge \gamma \implies \beta$$

The existence and construction typically relies on a proof.

Interpolant Generation using Quantifier Elimination

Craig: Given $\alpha \implies \beta$, an intermediate formula γ in common symbols of α and β exists and can be constructed such that

$$\alpha \implies \gamma \wedge \gamma \implies \beta$$

The existence and construction typically relies on a proof.

In Kapur et al (FSE06) we showed an obvious connection between interpolation and quantifier elimination.

Interpolant Generation using Quantifier Elimination

- Eliminate uncommon symbols from α: interpolant.

Interpolant Generation using Quantifier Elimination

- Eliminate uncommon symbols from α: interpolant.
- Similarly for β.

Interpolant Generation using Quantifier Elimination

- Eliminate uncommon symbols from α: interpolant.
- Similarly for β.
- All interpolants between α and β form a lattice using implication with the interpolant generated from α as the top element of the lattice and the one from β being the bottom element.

Interpolant Generation using Quantifier Elimination

- Eliminate uncommon symbols from α: interpolant.
- Similarly for β.
- All interpolants between α and β form a lattice using implication with the interpolant generated from α as the top element of the lattice and the one from β being the bottom element.
- The above assertions assume complete quantifier elimination.

Interpolant Generation using Quantifier Elimination

- Eliminate uncommon symbols from α: interpolant.
- Similarly for β.
- All interpolants between α and β form a lattice using implication with the interpolant generated from α as the top element of the lattice and the one from β being the bottom element.
- The above assertions assume complete quantifier elimination.
- This interpolant generated from α can serve as an interpolant all β's whose uncommon symbols with α are precisely remain invariant. Other properties of such interpolants can also be established.

Interpolants over Equality with Uninterpreted Symbols

α, a finite conjunction of equality and disequalities over constants and function symbols, with their subset UC being uncommon symbols with β's (UC may or may not include nonconstant function symbols).

- Run Kapur's congruence closure algorithm (RTA 1997) on equations with two differences.

Interpolants over Equality with Uninterpreted Symbols

α, a finite conjunction of equality and disequalities over constants and function symbols, with their subset UC being uncommon symbols with β's (UC may or may not include nonconstant function symbols).

- Run Kapur's congruence closure algorithm (RTA 1997) on equations with two differences.
  - Flatten terms by introducing new constant symbols. If a nonconstant subterm being replaced by a new constant has an outermost uncommon symbol, then the new constant is also viewed as uncommon. Otherwise, it is viewed as a common symbol.

Interpolants over Equality with Uninterpreted Symbols

α, a finite conjunction of equality and disequalities over constants and function symbols, with their subset UC being uncommon symbols with β's (UC may or may not include nonconstant function symbols).

- Run Kapur's congruence closure algorithm (RTA 1997) on equations with two differences.
  - Flatten terms by introducing new constant symbols. If a nonconstant subterm being replaced by a new constant has an outermost uncommon symbol, then the new constant is also viewed as uncommon. Otherwise, it is viewed as a common symbol.
  - Define a total ordering in which all uncommon nonconstant symbols are bigger than all uncommon constant symbols, followed by all common nonconstant symbols which are made bigger than all common constant symbols, run congruence closure which is ground completion.

Interpolants over Equality with Uninterpreted Symbols

α, a finite conjunction of equality and disequalities over constants and function symbols, with their subset UC being uncommon symbols with β's (UC may or may not include nonconstant function symbols).

- Run Kapur's congruence closure algorithm (RTA 1997) on equations with two differences.
  - Flatten terms by introducing new constant symbols. If a nonconstant subterm being replaced by a new constant has an outermost uncommon symbol, then the new constant is also viewed as uncommon. Otherwise, it is viewed as a common symbol.
  - Define a total ordering in which all uncommon nonconstant symbols are bigger than all uncommon constant symbols, followed by all common nonconstant symbols which are made bigger than all common constant symbols, run congruence closure which is ground completion.
  - This is in contrast to Kapur's algorithm in which all nonconstant symbols are bigger than constant symbols.

Interpolants over Equality with Uninterpreted Symbols

- The result is a finite set of rewrite rules of the form
  $f(c, d) \rightarrow e \quad (f, c, d$ are common $) \Longrightarrow e$is common
  $c \rightarrow e$; (c is common) implies (e is common)

Interpolants over Equality with Uninterpreted Symbols

- The result is a finite set of rewrite rules of the form $f(c, d) \to e \quad (f, c, d$ are common $) \Longrightarrow e$is common
  $c \to e$; (c is common) implies (e is common)
- **Horn clause introduction** From

$$f(a, b) \to e, \quad f(c, d) \to g,$$

$$(a = c \wedge b = d) \Longrightarrow e = g$$

where f is uncommon or least one of a, b, c, d is uncommon.

Interpolants over Equality with Uninterpreted Symbols

- The result is a finite set of rewrite rules of the form $f(c,d) \to e \quad (f, c, d$ are common $) \Longrightarrow e$is common
  $c \to e$; (c is common) implies (e is common)
- **Horn clause introduction** From

$$f(a,b) \to e, \quad f(c,d) \to g,$$

$$(a = c \wedge b = d) \Longrightarrow e = g$$

  where f is uncommon or least one of a, b, c, d is uncommon.
- **Normalize Horn clauses** Run congruence closure on the antecedent and normalize the consequent. If a Horn clause becomes trivially true, it is discarded. This is done every time a new Horn clause is generated.

Interpolants over Equality with Uninterpreted Symbols

- **Conditional Rewriting** The consequent of a Horn clause may have a uncommon symbol on its left side, which may also appear in an antecedent. That can be replaced in all such antecedents by carrying the conditions of this antecedent,

$$(c_1 = d_1 \wedge \cdots \wedge c_k = d_k) \Longrightarrow c = d$$

$$(a_1 = b_1 \wedge \cdots \wedge a_l = b_l) \Longrightarrow a = b$$

If a is some c_i or d_i, then

$$(a_1 = b_1 \wedge \cdots \wedge a_l = b_l) \wedge (c_1 = d_1 \wedge b = d_i \wedge \cdots c_k = d_k) \Longrightarrow c = d$$

DIsequalities do not play since at best they can do is to delete a Horn clause or identify unsatisfiability. But if α is assumed to be satisfiable in the input, then the result of this includes an interpolant which is all the equations and Horn clauses which only have common symbols.

An Example of Interpolant Generation on EUF

Mutually contradictory $\alpha = \{x_1 = z_1, z_2 = x_2, z_3 = f(x_1), f(x_2) = z_4, x_3 = z_5, z_6 = x_4, z_7 = f(x_3), f(x_4) = z_8\}$ and

$\beta = \{z_1 = z_2, z_5 = f(z_3), f(z_4) = z_6, y_1 = z_7, z_8 = y_2, y_1 \neq y_2\}$
Commons symbols are $\{f, z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8\}$.

An Example of Interpolant Generation on EUF

Mutually contradictory $\alpha = \{x_1 = z_1, z_2 = x_2, z_3 = f(x_1), f(x_2) = z_4, x_3 = z_5, z_6 = x_4, z_7 = f(x_3), f(x_4) = z_8\}$ and

$\beta = \{z_1 = z_2, z_5 = f(z_3), f(z_4) = z_6, y_1 = z_7, z_8 = y_2, y_1 \neq y_2\}$
Commons symbols are $\{f, z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8\}$.

Our algorithm gives:
$\{x_1 \to z_1, x_2 \to z_2, f(z_1) \to z_3, f(z_2) \to z_4, x_3 \to z_5, x_4 \to z_6, f(z_5) \to z_7, f(z_6) \to z_8\}$.

An Example of Interpolant Generation on EUF

Mutually contradictory $\alpha = \{x_1 = z_1, z_2 = x_2, z_3 = f(x_1), f(x_2) = z_4, x_3 = z_5, z_6 = x_4, z_7 = f(x_3), f(x_4) = z_8\}$ and

$\beta = \{z_1 = z_2, z_5 = f(z_3), f(z_4) = z_6, y_1 = z_7, z_8 = y_2, y_1 \neq y_2\}$
Commons symbols are $\{f, z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8\}$.

Our algorithm gives:
$\{x_1 \to z_1, x_2 \to z_2, f(z_1) \to z_3, f(z_2) \to z_4, x_3 \to z_5, x_4 \to z_6, f(z_5) \to z_7, f(z_6) \to z_8\}$.

The interpolant I_α:
$\{f(z_1) = z_3, f(z_2) = z_4, f(z_5) = z_7, f(z_6) = z_8\}$. No need to generate any Horn clauses.

An Example of Interpolant Generation on EUF

Mutually contradictory $\alpha = \{x_1 = z_1, z_2 = x_2, z_3 = f(x_1), f(x_2) = z_4, x_3 = z_5, z_6 = x_4, z_7 = f(x_3), f(x_4) = z_8\}$ and
$\beta = \{z_1 = z_2, z_5 = f(z_3), f(z_4) = z_6, y_1 = z_7, z_8 = y_2, y_1 \neq y_2\}$
Commons symbols are $\{f, z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8\}$.

Our algorithm gives:
$\{x_1 \rightarrow z_1, x_2 \rightarrow z_2, f(z_1) \rightarrow z_3, f(z_2) \rightarrow z_4, x_3 \rightarrow z_5, x_4 \rightarrow z_6, f(z_5) \rightarrow z_7, f(z_6) \rightarrow z_8\}$.

The interpolant I_α:
$\{f(z_1) = z_3, f(z_2) = z_4, f(z_5) = z_7, f(z_6) = z_8\}$. No need to generate any Horn clauses.

The interpolant reported by McMillan's algorithm is:
$(z_1 = z_2 \wedge (z_3 = z_4 \Longrightarrow z_5 = z_6)) \Longrightarrow (z_3 = z4 \wedge z_7 = z_8)$

An Example of Interpolant Generation on EUF

Mutually contradictory $\alpha = \{x_1 = z_1, z_2 = x_2, z_3 = f(x_1), f(x_2) = z_4, x_3 = z_5, z_6 = x_4, z_7 = f(x_3), f(x_4) = z_8\}$ and
$\beta = \{z_1 = z_2, z_5 = f(z_3), f(z_4) = z_6, y_1 = z_7, z_8 = y_2, y_1 \neq y_2\}$
Commons symbols are $\{f, z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8\}$.

Our algorithm gives:
$\{x_1 \to z_1, x_2 \to z_2, f(z_1) \to z_3, f(z_2) \to z_4, x_3 \to z_5, x_4 \to z_6, f(z_5) \to z_7, f(z_6) \to z_8\}$.

The interpolant I_α:
$\{f(z_1) = z_3, f(z_2) = z_4, f(z_5) = z_7, f(z_6) = z_8\}$. No need to generate any Horn clauses.

The interpolant reported by McMillan's algorithm is:
$(z_1 = z_2 \wedge (z_3 = z_4 \implies z_5 = z_6)) \implies (z_3 = z4 \wedge z_7 = z_8)$

Tinelli et al's algorithm, it is:
$(z_1 = z_2 \implies z_3 = z_4) \wedge (z_5 = z_6 \implies z_7 = z_8)$.

Interpolant Generation over Octagonal formulas

Let α to be a conjunction of $\pm x_i \leq c_i$ and $\pm x_i \pm x_j \leq c_{i,j}$, where x_i and x_j are distinct.

1. For each uncommon symbol x_i in α, consider two octagon formulas in which the sign of x_i is positive in one and negative in the other.
   x_i is eliminated by adding the two formulas. This must be done for every pair of such formulas.

The result of all uncommon symbols is an interpolant generated from α. This is illustrated below.

Interpolant Generation over Octagonal formulas

Let α to be a conjunction of $\pm x_i \leq c_i$ and $\pm x_i \pm x_j \leq c_{i,j}$, where x_i and x_j are distinct.

1. For each uncommon symbol x_i in α, consider two octagon formulas in which the sign of x_i is positive in one and negative in the other.
   x_i is eliminated by adding the two formulas. This must be done for every pair of such formulas.
2. In case a formula of the form $2x_j \leq a$ or $-2x_J \leq a$, it is normalized in the case octagonal formulas are over the integers.

The result of all uncommon symbols is an interpolant generated from α. This is illustrated below.

Interpolant Generation over Octagonal formulas

Let α to be a conjunction of $\pm x_i \leq c_i$ and $\pm x_i \pm x_j \leq c_{i,j}$, where x_i and x_j are distinct.

1. For each uncommon symbol x_i in α, consider two octagon formulas in which the sign of x_i is positive in one and negative in the other.
   x_i is eliminated by adding the two formulas. This must be done for every pair of such formulas.
2. In case a formula of the form $2x_j \leq a$ or $-2x_J \leq a$, it is normalized in the case octagonal formulas are over the integers.
3. If some uncommon symbol only appears positively or negatively, all octagonal formulas containing it can be eliminated as they do not occur in the interpolant.

The result of all uncommon symbols is an interpolant generated from α. This is illustrated below.

Griggio's example of Octagonal Formulas

Mutually contradictory
$\alpha = \{x_1 - x_2 \geq -4, \quad -x_2 - x_3 \geq 5, \quad x_2 + x_6 \geq 4, \quad x_2 + x_5 \geq -3\}$,
$\beta = \{ -x_1 + x_3 \geq -2, \quad -x_4 - x_6 \geq 0, \quad -x_5 + x_4 \geq 0\}$
Uncommon symbols: $\{x_2\}$.

Griggio's example of Octagonal Formulas

Mutually contradictory
$\alpha = \{x_1 - x_2 \geq -4, \quad -x_2 - x_3 \geq 5, \quad x_2 + x_6 \geq 4, \quad x_2 + x_5 \geq -3\}$,
$\beta = \{ -x_1 + x_3 \geq -2, \quad -x_4 - x_6 \geq 0, \quad -x_5 + x_4 \geq 0\}$
Uncommon symbols: $\{x_2\}$.

1. Eliminate x_2:
   $\{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7, \quad -x_3 + x_6 \geq 9\}$.

Griggio's example of Octagonal Formulas

Mutually contradictory
$\alpha = \{x_1 - x_2 \geq -4, \quad -x_2 - x_3 \geq 5, \quad x_2 + x_6 \geq 4, \quad x_2 + x_5 \geq -3\}$,
$\beta = \{ -x_1 + x_3 \geq -2, \quad -x_4 - x_6 \geq 0, \quad -x_5 + x_4 \geq 0\}$
Uncommon symbols: $\{x_2\}$.

1. Eliminate x_2:
   $\{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7, \quad -x_3 + x_6 \geq 9\}$.
2. No literal from α is included.

Griggio's example of Octagonal Formulas

Mutually contradictory

$\alpha = \{x_1 - x_2 \geq -4, \quad -x_2 - x_3 \geq 5, \quad x_2 + x_6 \geq 4, \quad x_2 + x_5 \geq -3\}$,

$\beta = \{ -x_1 + x_3 \geq -2, \quad -x_4 - x_6 \geq 0, \quad -x_5 + x_4 \geq 0\}$

Uncommon symbols: $\{x_2\}$.

1. Eliminate x_2:
   $\{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7, \quad -x_3 + x_6 \geq 9\}$.
2. No literal from α is included.
3. Interpolant:
   $I_\alpha = \{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7, -x_3 + x_6 \geq 9\}$.

Griggio's example of Octagonal Formulas

Mutually contradictory
$\alpha = \{x_1 - x_2 \geq -4, \quad -x_2 - x_3 \geq 5, \quad x_2 + x_6 \geq 4, \quad x_2 + x_5 \geq -3\}$,
$\beta = \{\ -x_1 + x_3 \geq -2, \quad -x_4 - x_6 \geq 0, \quad -x_5 + x_4 \geq 0\}$
Uncommon symbols: $\{x_2\}$.

1. Eliminate x_2:
   $\{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7, \quad -x_3 + x_6 \geq 9\}$.
2. No literal from α is included.
3. Interpolant:
   $I_\alpha = \{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7, -x_3 + x_6 \geq 9\}$.

Griggio's example of Octagonal Formulas

Mutually contradictory
$\alpha = \{x_1 - x_2 \geq -4, \quad -x_2 - x_3 \geq 5, \quad x_2 + x_6 \geq 4, \quad x_2 + x_5 \geq -3\}$,
$\beta = \{\ -x_1 + x_3 \geq -2, \quad -x_4 - x_6 \geq 0, \quad -x_5 + x_4 \geq 0\}$
Uncommon symbols: $\{x_2\}$.

1. Eliminate x_2:
   $\{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7, \quad -x_3 + x_6 \geq 9\}$.
2. No literal from α is included.
3. Interpolant:
   $I_\alpha = \{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7, -x_3 + x_6 \geq 9\}$.

Griggio's algorithm gives the conditional interpolant
$(-x_6 - x_5 \geq 0) \Longrightarrow (x_1 - x + 3 \geq 3)$

Griggio's example of Octagonal Formulas

Mutually contradictory
$\alpha = \{x_1 - x_2 \geq -4, \quad -x_2 - x_3 \geq 5, \quad x_2 + x_6 \geq 4, \quad x_2 + x_5 \geq -3\}$,
$\beta = \{\ -x_1 + x_3 \geq -2, \quad -x_4 - x_6 \geq 0, \quad -x_5 + x_4 \geq 0\}$
Uncommon symbols: $\{x_2\}$.

1. Eliminate x_2:
   $\{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7, \quad -x_3 + x_6 \geq 9\}$.
2. No literal from α is included.
3. Interpolant:
   $I_\alpha = \{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7, -x_3 + x_6 \geq 9\}$.

Griggio's algorithm gives the conditional interpolant
$(-x_6 - x_5 \geq 0) \Longrightarrow (x_1 - x + 3 \geq 3)$
The strongest interpolant is an octagonal formula and is generated by our algorithm.

Saturation based Invariant Strengthening and Abductor Generation

- Given a formula claimed to be an invariant I (or a post condition a la IC3)

Saturation based Invariant Strengthening and Abductor Generation

- Given a formula claimed to be an invariant I (or a post condition a la IC3)
- attempt to prove it inductive incrementally for every path: $(I \wedge cond) \implies I'$.

Saturation based Invariant Strengthening and Abductor Generation

- Given a formula claimed to be an invariant I (or a post condition a la IC3)
- attempt to prove it inductive incrementally for every path: $(I \wedge cond) \implies I'$.
  - lucky and success: great.

Saturation based Invariant Strengthening and Abductor Generation

- Given a formula claimed to be an invariant I (or a post condition a la IC3)
- attempt to prove it inductive incrementally for every path: $(I \wedge cond) \Longrightarrow I'$.
  - lucky and success: great.
  - unsuccessful: attempt to strengthen I using ψ such that $(I \wedge cond \wedge \psi) \Longrightarrow (I' \wedge \psi')$

Saturation based Invariant Strengthening and Abductor Generation

- Given a formula claimed to be an invariant I (or a post condition a la IC3)
- attempt to prove it inductive incrementally for every path: $(I \wedge cond) \implies I'$.
  - lucky and success: great.
  - unsuccessful: attempt to strengthen I using ψ such that $(I \wedge cond \wedge \psi) \implies (I' \wedge \psi')$
- Repeat this process until a counter-example is found or success.

Saturation based Invariant Strengthening and Abductor Generation

- Given a formula claimed to be an invariant I (or a post condition a la IC3)
- attempt to prove it inductive incrementally for every path: $(I \wedge cond) \implies I'$.
  - lucky and success: great.
  - unsuccessful: attempt to strengthen I using ψ such that $(I \wedge cond \wedge \psi) \implies (I' \wedge \psi')$
- Repeat this process until a counter-example is found or success.
- How to obtain ψ?

Saturation based Invariant Strengthening and Abductor Generation

- Given a formula claimed to be an invariant I (or a post condition a la IC3)
- attempt to prove it inductive incrementally for every path: $(I \wedge cond) \implies I'$.
  - lucky and success: great.
  - unsuccessful: attempt to strengthen I using ψ such that $(I \wedge cond \wedge \psi) \implies (I' \wedge \psi')$
- Repeat this process until a counter-example is found or success.
- How to obtain ψ?
- Approximate ψ to be $(I \wedge cond \wedge \psi) \implies I'$

Saturation based Invariant Strengthening and Abductor Generation

- Given a formula claimed to be an invariant I (or a post condition a la IC3)
- attempt to prove it inductive incrementally for every path: $(I \wedge cond) \implies I'$.
  - lucky and success: great.
  - unsuccessful: attempt to strengthen I using ψ such that $(I \wedge cond \wedge \psi) \implies (I' \wedge \psi')$
- Repeat this process until a counter-example is found or success.
- How to obtain ψ?
- Approximate ψ to be $(I \wedge cond \wedge \psi) \implies I'$
- ψ is an abductor for $(I \wedge cond, I')$.

Saturation based Invariant Strengthening and Abductor Generation

Example

var x, y, z: integer **end var**

$x := 0, \quad y := 0, \quad z := 9;$

while $x \leq N$ **do**

$x := x + 1; \quad y := y + 1; \quad z := z + x - y;$

end while

Goal: $z \leq 0$ is a loop invariant.

Saturation based Invariant Strengthening and Abductor Generation

Example

var x, y, z: integer **end var**
$x := 0, \quad y := 0, \quad z := 9;$
while $x \leq N$ **do**
$x := x + 1; \quad y := y + 1; \quad z := z + x - y;$
end while

Goal: $z \leq 0$ is a loop invariant.
$z \leq 0 \not\Longrightarrow z + x - y \leq 0.$

Saturation based Invariant Strengthening and Abductor Generation

Example

var x, y, z: integer **end var**
$x := 0, \quad y := 0, \quad z := 9;$
while $x \leq N$ **do**
$x := x + 1; \quad y := y + 1; \quad z := z + x - y;$
end while

Goal: $z \leq 0$ is a loop invariant.
$z \leq 0 \not\Longrightarrow z + x - y \leq 0.$
$(z \leq 0 \wedge z + x - y \leq 0) \not\Longrightarrow (z + x - y \leq 0 \wedge z + 2x - 2y \leq 0).$

Saturation based Invariant Strengthening and Abductor Generation

Example

var x, y, z: integer **end var**
$x := 0, \quad y := 0, \quad z := 9;$
while $x \leq N$ **do**
$x := x + 1; \quad y := y + 1; \quad z := z + x - y;$
end while

Goal: $z \leq 0$ is a loop invariant.
$z \leq 0 \not\Rightarrow z + x - y \leq 0$.
$(z \leq 0 \wedge z + x - y \leq 0) \not\Rightarrow (z + x - y \leq 0 \wedge z + 2x - 2y \leq 0)$.
Strengthen it to $z \leq 0 \wedge x - y \leq 0$

Saturation based Invariant Strengthening and Abductor Generation

Example

var x, y, z: integer **end var**
$x := 0, \quad y := 0, \quad z := 9;$
while $x \leq N$ **do**
$x := x + 1; \quad y := y + 1; \quad z := z + x - y;$
end while

Goal: $z \leq 0$ is a loop invariant.
$z \leq 0 \not\Rightarrow z + x - y \leq 0.$
$(z \leq 0 \wedge z + x - y \leq 0) \not\Rightarrow (z + x - y \leq 0 \wedge z + 2x - 2y \leq 0).$
Strengthen it to $z \leq 0 \wedge x - y \leq 0$
Quantifier elimination comes to the rescue

Salient Points

- Quantifier-elimination is ubiquitous.

Salient Points

- Quantifier-elimination is ubiquitous.
- Since general (complete) QE methods are very expensive and their outputs are hard to decipher, it is better to consider special cases, sacrificing completeness as well as generality.

Salient Points

- Quantifier-elimination is ubiquitous.
- Since general (complete) QE methods are very expensive and their outputs are hard to decipher, it is better to consider special cases, sacrificing completeness as well as generality.
- There is a real trade-off between resources/efficiency and precision/incompleteness.

Salient Points

- Quantifier-elimination is ubiquitous.
- Since general (complete) QE methods are very expensive and their outputs are hard to decipher, it is better to consider special cases, sacrificing completeness as well as generality.
- There is a real trade-off between resources/efficiency and precision/incompleteness.
- Let us call a spade a spade.

Challenges

- Can we develop specialized quantifier elimination algorithms/heuristics for various fragments of real and complex arithmetic?

Challenges

- Can we develop specialized quantifier elimination algorithms/heuristics for various fragments of real and complex arithmetic?
- Outputs generated by them need not be complete but must be useful for SMT solvers and theorem provers/verification systems.

Challenges

- Can we develop specialized quantifier elimination algorithms/heuristics for various fragments of real and complex arithmetic?
- Outputs generated by them need not be complete but must be useful for SMT solvers and theorem provers/verification systems.
- How can propositional reasoning, first-order and equational reasoning, redundancy checks, and preprocessing be exploited in general quantifier elimination methods to make them more effective?

Challenges

- Can we develop specialized quantifier elimination algorithms/heuristics for various fragments of real and complex arithmetic?
- Outputs generated by them need not be complete but must be useful for SMT solvers and theorem provers/verification systems.
- How can propositional reasoning, first-order and equational reasoning, redundancy checks, and preprocessing be exploited in general quantifier elimination methods to make them more effective?
- Can we have **better, effective** interfaces between a computer algebra system and a theorem prover?

Challenges

- Can we develop specialized quantifier elimination algorithms/heuristics for various fragments of real and complex arithmetic?
- Outputs generated by them need not be complete but must be useful for SMT solvers and theorem provers/verification systems.
- How can propositional reasoning, first-order and equational reasoning, redundancy checks, and preprocessing be exploited in general quantifier elimination methods to make them more effective?
- Can we have **better, effective** interfaces between a computer algebra system and a theorem prover?
- Can certificates be generated for outputs computed by a symbolic computation algorithm so that a theorem prover/SMT solver can trust it?

Parametric Gröbner Computations in Quantifier Elimination over the reals

- Gröbner basis computations are being widely used in many application domains, especially for equational solving

Parametric Gröbner Computations in Quantifier Elimination over the reals

- Gröbner basis computations are being widely used in many application domains, especially for equational solving
- Given the success of Gröbner basis computations for handling many problems in algebraic geometry, polynomial equation solving and program analysis, as well our good experience in computing comprehensive Gröbner systems, we are encouraged to build a practical incomplete heuristic for the theory of real closed field:

Parametric Gröbner Computations in Quantifier Elimination over the reals

- Gröbner basis computations are being widely used in many application domains, especially for equational solving
- Given the success of Gröbner basis computations for handling many problems in algebraic geometry, polynomial equation solving and program analysis, as well our good experience in computing comprehensive Gröbner systems, we are encouraged to build a practical incomplete heuristic for the theory of real closed field:
  1. draw upon extensive experience from SAT/SMT solvers, first-order reasoning to handle "nonalgebraic" reasoning.

Parametric Gröbner Computations in Quantifier Elimination over the reals

- Gröbner basis computations are being widely used in many application domains, especially for equational solving
- Given the success of Gröbner basis computations for handling many problems in algebraic geometry, polynomial equation solving and program analysis, as well our good experience in computing comprehensive Gröbner systems, we are encouraged to build a practical incomplete heuristic for the theory of real closed field:
  1. draw upon extensive experience from SAT/SMT solvers, first-order reasoning to handle "nonalgebraic" reasoning.
  2. use sum of squares heuristics (using completing square strategy): $\Sigma_{u=1}^{k} p_u^2 = 0 \Longrightarrow p_i = 0$.

Parametric Gröbner Computations in Quantifier Elimination over the reals

- Gröbner basis computations are being widely used in many application domains, especially for equational solving
- Given the success of Gröbner basis computations for handling many problems in algebraic geometry, polynomial equation solving and program analysis, as well our good experience in computing comprehensive Gröbner systems, we are encouraged to build a practical incomplete heuristic for the theory of real closed field:
  1. draw upon extensive experience from SAT/SMT solvers, first-order reasoning to handle “nonalgebraic” reasoning.
  2. use sum of squares heuristics (using completing square strategy): $\Sigma_{u=1}^{k} p_u^2 = 0 \Longrightarrow p_i = 0$.
  3. perhaps positive NullstellensatzNullstellensatzNullstellensatz (some aspects)

Parametric Gröbner Computations in Quantifier Elimination over the reals

- Gröbner basis computations are being widely used in many application domains, especially for equational solving
- Given the success of Gröbner basis computations for handling many problems in algebraic geometry, polynomial equation solving and program analysis, as well our good experience in computing comprehensive Gröbner systems, we are encouraged to build a practical incomplete heuristic for the theory of real closed field:
  1. draw upon extensive experience from SAT/SMT solvers, first-order reasoning to handle "nonalgebraic" reasoning.
  2. use sum of squares heuristics (using completing square strategy): $\Sigma_{u=1}^{k} p_u^2 = 0 \Longrightarrow p_i = 0$.
  3. perhaps positive NullstellensatzNullstellensatzNullstellensatz (some aspects)
  4. semi-definite Programming

Parametric Gröbner Computations in Quantifier Elimination over the reals

- Gröbner basis computations are being widely used in many application domains, especially for equational solving
- Given the success of Gröbner basis computations for handling many problems in algebraic geometry, polynomial equation solving and program analysis, as well our good experience in computing comprehensive Gröbner systems, we are encouraged to build a practical incomplete heuristic for the theory of real closed field:
  1. draw upon extensive experience from SAT/SMT solvers, first-order reasoning to handle "nonalgebraic" reasoning.
  2. use sum of squares heuristics (using completing square strategy): $\Sigma_{u=1}^{k} p_u^2 = 0 \Longrightarrow p_i = 0$.
  3. perhaps positive NullstellensatzNullstellensatzNullstellensatz (some aspects)
  4. semi-definite Programming
  5. a small step: interpolant generation for concave quadratic polynomial inequalities (over EUF).