ISBN 978-981-11-3671-9
Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering (WCSE 2017)
Beijing, 25-27 June, 2017, pp. 195-203
doi: 10.18178/wcse.2017.06.034

Precision Adjustment Strategies Based on Constraint Simplification in Polyhedra Abstract Domain

Xiang Chen [+], Min Zhou, and Ming Gu

Tsinghua University, Beijing, China

Abstract. Some precision adjustment strategies based on constraint simplification are introduced to find a trade-off between precision and its cost in program analysis. In program analysis built on numerical abstract domains, a constraint (such as Linear Inequation in Polyhedra abstract domain) implies the relationships between program variables. The main idea of precision adjustment strategies in this paper is simplifying the constraints in some pivotal program locations before these constraints are fed to abstract transfer functions. The strategy at Function Start Location(FSL) replaces expressions with constraints related to their ranges. The strategy at Loop Start Location(LSL) simplifies constraints' coefficients through combining several analogous constraints to a typical constraint. The strategy at Ordinary Location(OL) decreases the quantities of variables in constraints by replacing these variables with their ranges. Our framework is implemented on the top of Apron which is dependent on Convex Polyhedra and Linear Equalities library (POLKA). According to a series of experiments on different programs, we demonstrate that the application of precision adjustment strategies can improve the efficiency of program analysis.

Keywords: program analysis, abstract domain, abstract interpretation, precision adjustment

1. Introduction

Nowadays, computer programs have been applied in various fields. To ensure the programs meet people's expectations and no abnormal state is in running programs, the technologies about program analysis and verification have been introduced. Abstract Interpretation [1] is a program analysis method to simulate the possible running program states through semantic abstraction.

Numerical Abstract Domain plays a key role in numerical program analysis based on abstract interpretation. An abstract numerical domain can describe the ranges of different numerical variables and the relationships between these variables. A simple and intuitive Interval abstract domain ($X_i \in [a_i, b_i]$) [2] can track the upper and lower bounds of each numerical variable and analyze codes with very high efficiency. Octagon abstract domain ($\pm x_i \pm x_j \leq \beta$) [3] is able to express specific constraints between two variables. Polyhedral abstract domain ($\sum_i \alpha_i X_i \leq \beta$) [4] is used to derive linear relationships between any number of variables.

In the past decades, Polyhedra abstract domain is widely applied [5, 6]. But the application of Polyhedra has been limited by its double-description [7]. The solutions aiming to find trade-offs between precision and efficiency can be divided into two categories: on the one hand, limiting the ability to express relationships between program variables, and on the other hand, improving some abstract domain operators.

In the case of restrictive expression, many weakly relationship abstract domains have been introduced, such as Octagons [3], octahedron ($\sum_i \alpha_i X_i \leq \beta, \alpha_i \in \{-1,0,1\}$) [8], Pentagons ($x \leq y \wedge a \leq x \leq b$) [9] and so on. These abstract domains can make program analysis quicker, but some relational information among program variables will be discarded. What's more, the method adding restrictions on existing abstract

[+] Corresponding author. Tel.: + 15201614913.
E-mail address: kuailezhish@gmail.com.

domains also perform well [10]. [11] puts up two methods called "linearization" and "symbolic constant propagation" to simplify numerical expressions.

Some techniques related to abstract domain operators are used to reduce the analysis time, such as making the threshold for the maximum amount of the constraints [12], selecting appropriate abstract domains for different program variables [13] have been illustrated to foster the speed of program analysis.

This paper aims to make the complexity of program analysis lower through simplifying the constraints in Polyhedra abstract domain. According to a good code style "Declare Variables At First Use" [14], we divide program variables and program locations into different categories and introduce some methods to simplify constraints according to these variables and locations. When there are some loops and functions in programs along with several symbolic variables, the precision adjustment strategies produce a good influence on improving the efficiency of program analysis.

This paper is organized as follows: First, we give the preliminaries in Sect. 2. Then in Sect. 3, the precision adjustment strategies are presented. In Sect. 4, some experiments are established to give an evaluation to the methods in this paper. The last Sect. 5 followed by conclusions and an outlook on future research.

2. Preliminaries

We briefly recall the classical Abstract Interpretation framework [1] which is designed to compute numerical invariants automatically.

2.1. Syntax of the Language

A simple imperative programming language is used to analyze programs. All the variables are assumed to be integers in Z. Suppose that the quantity of variables is finite and $V = \{v_1, \cdots, v_n\}$ represents all the variables.

Table 1 is the syntax of the simple language. And the programming language will be used in examples is C programming language.

Table 1: Syntax of the simple language.

val	$::=$	$const$		$constant$
	$\mid$	v		$variable$
$expr$	$::=$	val		$value$
	$\mid$	$!\,expr$		$negation$
	$\mid$	$expr \circ expr$	$\circ \in \{+, -, \times, /\}$	
op	$::=$	$v = expr$	$v \in V$	$assignment$
	$\mid$	$expr \rhd\lhd 0$	$\rhd\lhd \in \{=, \neq, <, \leq, >, \geq\}$	$assumption$

2.2. Program Analysis

A program is represented by a control-flow-automaton (CFA) [15], which consists of a set L meaning all the control locations, an initial location l_0 and a transfer relation set $G \subseteq L \times Operators \times L$.

2.3. Abstract Domain

Every abstract domain contains two components: domain states and domain operators. The process of program analysis can be regarded as the transformation between domain states by means of domain operators.

Domain states consist of concrete states set E and abstract states set S. Any abstract state $s \in S$ can be regarded as a set of concrete states $\{e_1, e_2, \ldots, e_i\}, \forall e_i \in E$. All the abstract states should be the elements of a semi-lattice so that different domain states can be compared and domain operators between states can be carried out.

Domain operators are operators carried on domain states. There are several types of domain operators in abstract domain.

- Preorder $\sqsubseteq \subseteq S \times S$. This operator is used to decide whether one abstract state subsumes another abstract state (partial order).
- Join Operator $\bigsqcup: S \times S \to S$. This operator aims to combine two abstract states into a new abstract state and all the concrete states represented in two abstract states will be represented in the new abstract state.
- Transfer Operator $\rightsquigarrow \subseteq S \times G \times S$. This operator $\rightsquigarrow$assigns each abstract state s a list of new abstract successor states, which forms as $s \rightsquigarrow s'$ if $(s, g, s) \in \rightsquigarrow$. Every transfer operator is associated with a certain edge $g \in G$ which connects different program locations. The type of edge g varies for different semantics. These edge types include assignment, assumption (make an assumption true or false), function call statement, return statement and so on.

2.4. Constraint

Every numerical abstract domain has its representations of constraints which illustrate the relationships between program variables. Constraints can be unary, binary or multiary, linear or nonlinear and equation or inequation. There are several typical constraints which will be used in the following part of this paper.

- Interval Expression. The expression is $\pm x_i \leq \beta (\beta \in \mathbb{Q})$ which can only make restrictions in upper bound and lower bounds of variables.
- Linear Inequation. The expression is $\sum_i \alpha_i X_i \leq \beta (\alpha_i, \beta \in \mathbb{Q})$. From the expression of the view, Linear Inequations includes all the Interval Expressions.
- Interval Linear Inequation. The expression is $\sum_i [\alpha_i, \beta_i] X_i = \beta (\alpha_i, \beta_i \in \mathbb{Q}, \alpha_i \leq \beta_i)$. This expression is used in the process of replacing variables with their ranges. This constraint expression comes from Interval Polyhedra abstract domain [16].

2.5. Invariant

Let π represents a set of constraints. If all the constraints in π can be meet in location $l \in L$, we call π is an invariant at l.

2.6. Precision

Every abstract domain can present different levels of abstraction (fine-grained or coarse-grained). The level of abstraction of current abstract state in chosen abstract domain is determined by the abstraction precision [17].

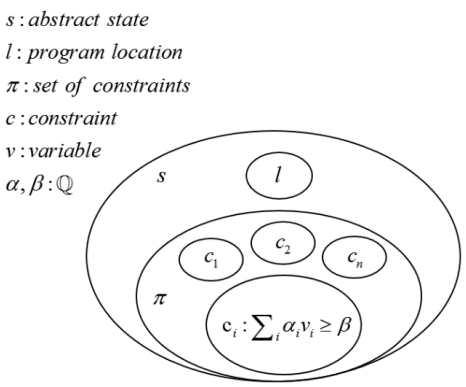

Fig. 1: Inclusive Relationship among important concepts.

In general, the higher the level of abstraction (coarse-grained), the more efficient the analysis will be. However, high-level abstraction will bring us an overwhelming number of false alarms. When the level of abstraction is lower (fine-grained), the abstract state is much closer to the concrete program state and the reported alarms will be more precise. But the analysis will be costlier and can't be scale to large programs.

In this paper, the abstract precision of program state can be represented by a set of related constraints. A set of constraints $\pi \in \Pi$ (Π is the set of all the sets of constraints) records all the constraints associated to the abstract state s.

To sum up, the inclusive relationship of abstract states, program locations, constraints and variables can be shown in Fig.1. In Fig. 1, every abstract state s has a program location l and a set of constraints π. π contains a list of constraints and every constraint c_i is a linear inequation.

3. Precision Adjustment

In this section, a sample program with location labels is given firstly. Then, some new terms will be introduced to pave the way for illustrating precision adjustment strategies. Finally, formalized description about precision adjustment strategies will be given.

Listing 1:

```
int total = 100; (L1)
void process (int x, int y, int z) { (L8)
   // some calculations
   (L9)
}
int main() { (L0)
   for (int i = 0; (L2) i <= total / 5; (L3) i++ (L14)) {
      for (int j = 0; (L4) j <= total / 3; (L5) j++ (L12)) {
         int k = total - i - j; (L6)
         if (k % 3 == 0 && i * 5 + j * 3 + k / 3 == total) { (L7)
            process(i, j, k); (L10)
         } (L11)
      } (L13)
   } (L15)
   return 0; (L16)
}
```

Listing 1 is an example with location labels. The program is a solution to solve Chinese classic problem named "One hundred yuan to buy one hundred chicken problem". Every L_i in Listing 1 is a location label which indicates the control location and will be called program location in the following part of this paper.

3.1. Program Location

Control locations have been mentioned in section 2.2. There are several program locations.

Function Start Location (FSL) Function Start Location means the first control location in the function. At FSL, some parameters are passed from parent function. In Listing 1, L_8 is a function start location of function process.

Loop Start Location (LSL) Loop Start Location is the location in CFA which will be analyzed periodically during loop analysis. What's more, LSL is the location that can be transferred to the outside of the loop. In Listing 1, L_2 and L_4 are loop start locations in function main.

Ordinary Location (OL) All the program locations except FSL and LSL are called ordinary locations including the beginning location and the ending location.

3.2. Variable Categories

In C or C-like programming language, variables include global variables and local variables. In this paper, variables are put into several categories according to program locations where they are declared.

Global Variable: the global variables in this paper means the variables declared in ordinary location and not in any independent function. In Listing 1, the variable $total$ is a global variable.

Function Start Variable: the variables defined at FSL are called function start variables. Their values are determined by the arguments passed from upper functions. In Listing 1, the variables x, y and z in function process are function start variables.

Ordinary Variable: ordinary variable is neither a global variable nor a function start variable. It is declared in the body of function and stores some information about the loop depth in the current function.

Loop Depth (the depth of loop) Loop Depth means which level is the loop where the ordinary variable declared. If the entry of function is zero layer loop, then the depth of new loop nested in another loop is equal

to the depth of its parent loop plus one.

In Listing 1, the variables i, j and k in function main are ordinary variables. Their loop depths are 0, 1, 2 respectively. What's more, the depths of all function start variables are 0.

3.3. Precision Adjustment Strategies

What the precision adjustment strategies aiming to is adjusting the constraints stored in abstract states and making them simpler.

When a transfer $s(l,\pi) \rightsquigarrow s'(l',\pi')$, $l,l' \in L$, $s,s' \in S$, $\pi,\pi' \in \Pi$ finished, the precision adjustment $\rho: \Pi \to \Pi$ will be carried out. The process of precision adjustment can be formalized as $s'(l',\pi') \twoheadrightarrow s''(l',\pi''), s'' \in S, \pi'' \in \Pi$. Here, the parentheses in $s(l,\pi)$ means inclusion relation.

Some predicates are defined to express strategies clearly in the following analysis:

- $Val(\pi, v)$: return the range $interval([low, high])$ of variable v under the set of constraints π
- $Con(v, interval)$: return the constraint $low \leq v \leq high$
- $Dep(v)$: return the loop depth of v in current function

Following are several strategies for precision adjustment $\rho(\pi') = \pi''$:

Strategy 1 (At FSL) Firstly, $\pi'' = \pi$. Then $\forall v \in V$, if v is function start variable in current function, then $\pi'' = \pi'' + Con(v, Val(\pi', v))$.

Listing 2:

```
void bar (int x, int y) {
   // euclidean algorithm
}
void foo (int x, int y) {
   if (0 < x && x < 6 && 0 < y && y < 4) {
      if (x + 2 * y > 7 && 3 * x + y < 16) {
         bar(x, y);
      }
   }
}
```

In Listing 2, before function bar is called, $\pi = \{0 < x_{foo} < 6, 0 < y_{foo} < 4, x_{foo} + 2 * y_{foo} > 7, 3 * x_{foo} + y_{foo} < 16\}$. When finished calling function bar, $\pi' = \{0 < x_{foo} < 6, 0 < y_{foo} < 4, x_{foo} + 2 * y_{foo} > 7, 3 * x_{foo} + y_{foo} < 16, x_{bar} = x_{foo}, y_{bar} = y_{foo}\}$. After precision adjustment at function bar start location, $\pi'' = \{0 < x_{foo} < 6, 0 < y_{foo} < 4, x_{foo} + 2 * y_{foo} > 7, 3 * x_{foo} + y_{foo} < 16, 1 < x_{bar} < 5, 1 < y_{bar} < 4\}$.

Strategy 2 (At LSL) In this section, we introduce two concepts Feature and Concatenation, then an algorithm about adjustment strategy at loop start location will be given.

Feature $\forall c \in \pi'$, the $feature$ of c is a set which stores some information about what the kind of c (one of $\{=, \neq, <, \leq, >, \geq\}$) and the signs of every variable coefficients in c. Each sign can be one of $\{-1, 0, 1\}$ which means negative, zero and positive. For example, when $V = \{x, y, z\}$ and a constraint cc is $2 * x - 3 * y + 1 > 0$, then the $feature$ of cc is $\{>, [1, -1, 0]\}$. In the array, the coefficient of z is 0 because z is not in cc.

Concatenation the $concatenation$ of a constraint is a linear expression which consists of all the variables and their signs. $concatenation$ can be formalized as $\sum_i sign_i * X_i$. Through the previous example, we get the $concatenation$ of cc is $x - y$.

The algorithm about adjustment strategy at LSL is illustrated in Fig. 2.

Listing 3:

```
int main() {
   int x, y;
   if (0 <= x && y <= 10)  {
      if (4 * y - x + 4 > 0 && 10 * y - x + 1 > 0 && y - 3 * x + 22 > 0 && 2 * y - 11 * x + 99 > 0)  {
         for (int i = 0; i < x + y; i++) {
            int z = x * x - y * y;
```

```
        }
      }
    }
    return 0;
}
```

Procedure 1 Precision Adjustment Algorithm at LSL

Input: Two set of constraints π and π'
Output: The adjusted set of constraints π''

1: $\pi'' = \pi'$;
2: generate a set of features F from all the constraints in π'';
3: **for all** $feature_i \in F$ **do**
4: **if** $feature_i$ appears more than once **then**
5: put all the constraints whose features are equal to $feature_i$ into a constraint set C_i
6: create a $concatenation$ from any $c \in C_i$ and the result is expressed as $expr$.
7: $\pi'' = \pi'' - C_i + Con(expr, Val(\pi', expr))$;
8: **else**
9: $\pi'' = \pi''$;
10: **end if**
11: **end for**
12: **return** π'';

Fig. 2: Precision adjustment algorithm at LSL.

In Listing 3, when the statement "int i = 0;" has been executed, $\pi' = \{0 \leq x, y \leq 10, 4 * y - x + 4 > 0, 10 * y - x + 1 > 0, y - 3 * x + 22 > 0, 2 * y - 11 * x + 99 > 0, i = 0\}$. The features of four constraints are all $\{>, [-1,1,0]\}$ and they are in same C_i. A new concatenation $expr$ generating from any constraint $c \in C_i$ is $-x + y$. Then these four constrations is dropped and a new constraint $Con(expr, Val(\pi', expr))$ is added. After carrying out precision adjustment at loop start location, π'' becomes to $\{0 \leq x, y \leq 10, -x + y + 6 \geq 0, i = 0\}$.

Strategy 3 (At OL) Let c represents the constraint related to transfer edge g. If the type of edge g is not assumption, or the type of edge g is assumption and the loop depths of all variables in c are equivalent, then $\pi'' = \pi'$, the adjustment is done. Otherwise there are several steps to express the strategy at OL.

At first step, let d represent the maximum loop depth of variables in c. At second step, $\forall v \in c$, if v is Global Variable or $Dep(v) < d$, replace these variables with their respective ranges $Val(\pi', v)$ in c to get a new constraint c' (Interval coefficient is used in this constraint). At third step, adjusting the constraints c' to the one c'' which Polyhedra abstract domain can express according to the operations from Interval Polyhedra abstract domain [16]. Lastly $\pi'' = \pi + c''$.

Listing 4:

```
int main () {
    int x;
    if (0 <= x && x <= 10) {
        for (int i = 0; i < x; i++) {
            int sum = 0;
            for (int j = 0; j < x + i; j++) {
                sum = sum + 1;
            }
        }
    }
    return 0;
}
```

In Listing 4, when the statement $j < x + i$; has been executed, the depths of variable x and i are 0, and the depth of variable j is 1. So, the ranges of x and i will be calculated and the constraint $j < x + i$ will be simplified. When the program reaches its fixpoint, we can get the adjusted constraint $-j + 18 >= 0$ instead of $j - x - i < 0$.

4. Experiment Evaluation

All the experiments are performed on Ubuntu 15.04(64-bit) system with Intel Core i7-4790 CPU(3.60GHz). The analysis are implemented in CPAchecker [18] which is a framework and tool for formal software verification, and program analysis of C programs.

4.1. Design

Numerical abstract domains are usually used to infer the relationships among numerical variables in programs. So, all the experiments are designed to find the linear invariants. There are four stages of carrying out program experiments: precision adjustment at FSL, precision adjustment at LSL, precision adjustment at OL and precision adjustment in all previous locations (also called synthesized case).

4.2. Results

There are some aspects to judge the effects of experiments: time cost, memory cost, the number of iterations to get invariants and complexity of codes (the numbers of variables, loops and functions). Table 2 shows the analysis results of 9 programs. The information shown about variables, loops and functions are quantities. S(function), S(loop), S(ordinary) means adopting different precision adjustment strategies.

Table 2: Experiment Results

Program	variables	loops	functions	S(function)	S(loop)	S(ordinary)	iterations	Memory(average)	Analysis time
prog1	8	1	2				100	38MB	72ms
prog1	8	1	2	√			50	38MB	35ms
prog2	8	1	2				--	--	--
prog2	8	1	2	√			54	35MB	35ms
prog3	4	1	1				122	37MB	97ms
prog3	4	1	1		√		122	36MB	83ms
prog4	5	3	1				15011	71MB	3433ms
prog4	5	3	1		√		2182	54MB	917ms
prog5	6	2	1				84	36MB	63ms
prog5	6	2	1			√	61	36MB	38ms
prog6	4	2	1				--	--	--
prog6	4	2	1			√	242	39MB	131ms
prog7	9	2	2				14977	75MB	4222ms
prog7	9	2	2	√			10452	56MB	2888ms
prog7	9	2	2		√		14027	77MB	3783ms
prog7	9	2	2			√	7607	50MB	2472ms
prog7	9	2	2	√	√		9766	66MB	2841ms
prog7	9	2	2	√		√	7604	59MB	2373ms
prog7	9	2	2		√	√	6980	57MB	2354ms
prog7	9	2	2	√	√	√	6977	52MB	2515ms
prog8	7	2	2				--	--	--
prog8	7	2	2		√		206	50MB	1029ms
prog8	7	2	2			√	183	38MB	125ms
prog9	72	1	2				2051	51MB	8351ms
prog9	72	1	2	√	√	√	2029	63MB	5526ms

Program 1 and Program 2 are used to evaluate strategy 1. Program 1 is the full version of Listing 2. The Euclidean algorithm is the algorithm for finding the greatest common divisor of two integers. The results show that the efficiency of analysis can be improved by strategy 1. The comparable result in program 2 shows that a program which can't terminate when calculating invariants can terminate through applying strategy 1.

Program 3 and Program 4 are used to evaluate strategy 2. Program 3 is Listing 3. Improvements in Program 3 is tiny but still useful. Program 4 is a program with three layers of loop. The result shows the

number of iteration and analysis time can be decreased in this program.

Program 5 and program 6 are used to evaluate strategy 3. Program 5 is Listing 4. Adopting strategy 3 can both improve the efficiency and make the program terminate.

Program 7, 8 and 9 are synthesized cases. Program 7 is Listing 1. The origin program adopting none strategy costs the most analysis time. Any single strategy is helpful to save analysis time. Combining several strategies is useful to improve efficiency further. Program 8 shows a program which can't terminate when calculating fixpoint can terminate by adopting any of strategy 2 and strategy 3. Program 9 is a simplified version coming from the actual project which codes about array and pointer are simplified. Program 9 has 72 variables and the analysis can be accelerated via adopting all the strategies.

4.3. Discussion

The main idea to apply our strategies is to simplify some constraints at proper circumstances. Mostly, the analysis time is decreased. The increment in memory consumption is negligible and memory consumptions of some programs are even decreased. Simplified constraints make the ranges of variables simple and over-approximate, which can result in converging faster and make some programs terminate possibly.

Due to not every program has multi-layer loops and functions and the large quantities of relationships between variables, the precision adjustment strategies are not perfect. But operations in these strategies are lightweight, which can achieve excellent improvements in the case of little additional cost.

5. Conclusion

The quantity and complexity of constraints have a significant influence on the efficiency of program analysis in Polyhedra abstract domain. We introduced precision adjustment strategies based on program locations, variable attributes and constraint types to simplify constraints in program states. According to a series of experiments on different programs, when there are some loops and functions in programs along with several symbolic variables, the precision adjustment strategies have a positive effect on improving the efficiency of program analysis through replacing variables with their ranges and combining several analogous constraints to a typical constraint.

In the future research, applying precision adjustment strategies in this paper to other numerical abstract domains is worth trying. We hope that our ideas behind precision adjustment strategies could be applied to not only in program analysis based on abstract domains but also in other program analysis fields if only program states could be adjusted through analyzing the relationships between variables and pivotal program locations.

6. References

[1] P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[2] ——, “Static determination of dynamic properties of programs,” in *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976, pp. 106–130.

[3] A. Min’e, “The octagon abstract domain,” *Higher-order and symbolic computation*, vol. 19, no. 1, pp. 31–100, 2006.

[4] P. Cousot and N. Halbwachs, “Automatic discovery of linear restraints among variables of a program,” in *Proceedings of the 5th ACM SIGACT- SIGPLAN symposium on Principles of programming languages*. ACM, 1978, pp. 84–96.

[5] D. Delmas and J. Souyris, “Astr´ee: from research to industry,” in *International Static Analysis Symposium*. Springer, 2007, pp. 437–451.

[6] R. Bagnara, P. M. Hill, and E. Zaffanella, “Applications of polyhedral computations to the analysis and verification of hardware and software systems,” *Theoretical Computer Science*, vol. 410, no. 46, pp. 4672–4691, 2009.

[7] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall, “The double description method,” 1953.

[8] R. Claris’o and J. Cortadella, “The octahedron abstract domain,” in *International Static Analysis Symposium*. Springer, 2004, pp. 312–327.

[9] F. Logozzo and M. Fahndrich, “Pentagons: a weakly relational abstract domain for the efficient validation of array accesses,” in *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 184–188.

[10] V. Laviron and F. Logozzo, “Subpolyhedra: A (more) scalable approach to infer linear inequalities,” in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2009, pp. 229–244.

[11] A. Min´ e, “Symbolic methods to enhance the precision of numerical abstract domains,” in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2006, pp. 348–363.

[12] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, “Scalable analysis of linear systems using mathematical programming,” in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2005, pp. 25–41.

[13] S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. von Rhein, “Domain types: Abstract-domain selection based on variable usage,” in *Haifa Verification Conference*. Springer, 2013, pp. 262–278.

[14] “Declare variables at first use,” http://c2.com/cgi/wiki ? DeclareVariablesAtFirstUse, accessed: 2016-08-03.

[15] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, “The software model checker blast,” *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[16] L. Chen, A. Min’e, J. Wang, and P. Cousot, “Interval polyhedra: An abstract domain to infer interval linear relationships,” in *International Static Analysis Symposium*. Springer, 2009, pp. 309–325.

[17] D. Beyer, T. A. Henzinger, and G. Th´eoduloz, “Program analysis with dynamic precision adjustment,” in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 29–38.

[18] D. Beyer and M. E. Keremoglu, “Cpachecker: A tool for configurable software verification,” in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 184–190.