Bachelor Thesis

Geometric Non-Termination Arguments for Integer Programs

14. September 2017

Timo Bergerbusch
Rheinisch Westfälisch Technische Hochschule Aachen
Lehr- und Forschungsgebiet Informatik 2

First Referee: Prof. Dr. Jürgen Giesl
Second Referee: apl. Prof. Dr. Thomas Noll
Supervisor: Jera Hensel

Acknowledgement

First, I would like to thank Prof. Dr. Jürgen Giesl for giving me the opportunity to work on an ongoing and relevant topic. Secondly, I would like to thank apl. Prof. Dr. Thomas Noll for agreeing to be the second referee of my thesis.

Thirdly, I would like to thank Jera Hensel, who supervised me during my thesis. I want to thank her for the many patient answers she gave my no matter how obvious the solution was. She did not only answer questions, but also got proactive herself and helped me creating better results by pointing out my failures and encouraging me during the whole process. Also I want to thank her for the possibility to write the underlying program the way I wanted to without any restrictions or limits regarding the way of approaching the topic.

Also I want to thank my girlfriend Nadine Vinkelau and all my friends, who encouraged me during my whole studies and not only accepted that I often was short on time, but also backed me up during the whole process. Especially I want to thank my good friend Tobias Räwer, who explained many topics to me over and over again throughout the whole Bachelor studies to help me pass my exams without demanding anything in return. Thanks to his selfless behaviour I got this far within only three years.

Finally I want to thank my parents for giving me the possibility to fulfil my desire to study at a worldwide known university. Without the financial support I would not have had this opportunity.

Erklärung Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 14. September, 2017

Timo Bergerbusch

Abstract

The topic of program termination analysis undergoes a significant importance increase owed to the expansion of software usage throughout everyday life. Since fixing problems caused by software bugs leads to an overhead of support, the initial guarantee of correctness can save time spent on fixing these problems. Therefore the research of automated assisting during the engineering of large programs is a growing field. One major point of a correct program is determined by its termination, which means that it reaches a final state after finitely many steps. Even though such a tool can never provide soundness in every condition since it would have to solve the *halting problem*, which is proven by Turing to be undecidable, a variety of tools addressing this problem exist, for example AProVE. AProVE tries to prove (non-)termination for as many programs possible, although not all programs can be handled.

In this thesis we extend the possibilities of proving non-termination using AProVE by a special set of programs based on the approach described in [LH14] by Jan Leike and Matthias Heizmann. Altering the underlying structure from linear loop programs to integer transition systems (ITS) we prove non-termination using a *geometric non-termination argument (GNA)* derived from the program itself. By the usage of linear algebra and SMT-solver we are able to prove the existence of a GNA, which results in a proof of non-termination of the integer transition system. Using this technique as an additional approach in AProVE increases its power power to prove non-termination.
As a result we will see that the implemented technique provides the desired mechanism of proving non-termination for the considered programs under certain limitations, which are reasoned by the complexity of integer transition systems in general.

One restriction is the existence of a start term within the system, which is mandatory to apply the definition of a GNA. Further the handling of newly introduced variables within the systems are only very basic, since correctness of using division on integers is not generally given.

In summary we can say that the use of GNAs is a promising approach to prove non-termination of ITSs, as it is for linear lasso programs. The restriction to only use linear updates and its consequences regarding modern programs need further investigation to evaluate the applicability in real industrial software.

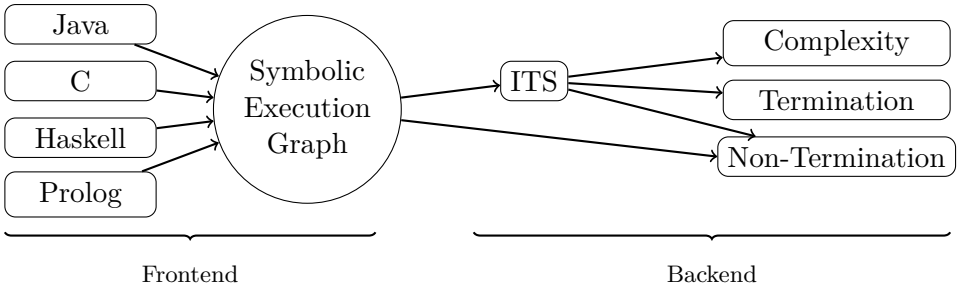
Contents

Chapter 1

Introduction

1.1 Motivation

The topic of verification and termination analysis of software increases in importance with the development of new programs. Even though for Turing complete programming languages the *halting problem* is undecidable, and therefore no complete and sound method to decide termination can exist, a variety of approaches are researched and still being developed. These approaches try to prove (non-)termination on programs, which match certain criteria in form of structure, composition or using only a closed set of operations for example only linear updates of variables. Many of these subsets are still Turing complete and therefore these approaches prove soundness but not completeness.
Given a tool which can provide a sound mechanism to prove termination, an optimized framework could analyse written code and find bugs before the actual release of the software [VDDS93]. Contemplating that automatic verification can be applied to termination proved software the estimated annual US economy loss of $60 billion each year in costs associated with software maintenance could be reduced significantly [ZC09].
One promising approach is the tool AProVE (Automated Program Verification Environment) developed at the RWTH Aachen, Lehr- und Forschungsgebiet Informatik 2. The tool (further only called AProVE) for automatic termination and complexity analysis proves termination for programs of various programming language paradigms like Java (object oriented), Haskell (functional), Prolog (logical) as well as rewrite systems.



AProVE uses Clang to compile the C program to LLVM code [CLA17]. Among others these LLVM programs can be converted into a so called Symbolic Execution Graph. This graph represents all possible computations of the input program. If this graph contains lassos, which are strongly connected components (SCC) and the corresponding path from the root to the SCC, AProVE derives so called (integer) transition systems (ITS). A mathematical definition can be found

within [FGP+09]. The construction of such an ITS fulfils the property that termination of the ITS implies termination of the C program. In special cases the same holds for non-termination. A more detailed description of the process is stated in [SGB+17] [GAB+17].

1.2 Overview

This thesis provides the introduction to the topic of termination analysis. We focus on the very basic steps because of the huge variety of possible approaches and related methods. Any further knowledge about termination analysis techniques and how they are applied within AProVE can be found in the related papers [GAB+17], [GSKT06], [SGB+17], [GTSKF03].
In Chapter 2 some preliminaries needed to understand the contributions of this thesis are defined. It covers most essential definitions such as that of non-termination, topics of basic knowledge about *Integer Transition Systems* and their structure as it is considered in this thesis. Also geometric non-termination arguments, which build the main constituent, and mathematical knowledge needed to find GNAs are defined. Finally we take a look at the topic of SMT-solving and declare the essential parts used within the implementation of the approach.
Chapter 3 deals with the derivation of the different parts resulting in a SMT-problem, which provides a geometric non-termination argument if it exists.
At the end, we want to take a look at the usability of the approach. We also want to point out possible adaptations and improvements of the implementation of this approach.

Chapter 2

Preliminaries

In order to be able to explain the new non-termination approach we have to declare, what non-termination means, which programs are considered within this approach and present the construction of geometric non-termination arguments (GNAs), which builds the core of the approach. Furthermore we have to define a few structures we work on, we have to define what an SMT-solver is and how it is used within this implementation.

2.1 Integer Transition Systems

In order to apply the upcoming procedure we have to define what structure the approach works on. As described in Section 1.1, the C program is transferred into a symbolic execution graph. Afterwards, from each lasso of the graph an ITS is constructed. Here, a cycle is a lasso together with the path from the initial state to this cycle. Then, if we know that the resulting ITSs has the property that it is equivalent to the original program in its termination behaviour, we can prove non-termination of the original program by proving non-termination of one of the ITSs. Considering the following approach we will look at ITSs of a special form shown in Figure 2.1.

$$
\begin{array}{ll}
 & \overbrace{f_x}^{(1)} \quad \rightarrow \overbrace{f_y}^{(2)}(v_1, \ldots v_n) : | : cond_1 \\
2 & f_y(\underbrace{v_1, \ldots v_n}_{(3)}) \rightarrow f_y(\underbrace{v'_1, \ldots v'_n}_{(3)}) : | : \underbrace{cond_2}_{(4)}
\end{array}
$$

Figure 2.1: The structure of an ITS considered in this thesis

The ITS shown in Figure 2.1 consists of a set of structure elements, whose definition is necessary:

(line 1) The first line is the rewriting rule the program starts with and can be seen as a declaration of initial values of some variables. An example is shown in Figure 2.3.

(line 2) A self-looping rule. Other looping rules will be presented in Section 4.3.2.

(1) The *start function symbol* is the first symbol used, consisting of a function symbol without arguments. Further explanation in (line 1) and Figure 2.3.

(2) A function symbol denoting a current program state

(3) The arguments of a function symbol showing the update of the values by applying this rule. The value of v_i' is a linear update of the variables v_j, $1 \leq j \leq n$, in standard linear integer form. [1]

(4) The conditional term of the form (in)equation$_1$ && ... && (in)equation$_m$, $m \in \mathbb{N}$, where (in)equation$_i$ does not only contain the variables v_j, $1 \leq j \leq n$, but can also introduce new variables. The form of the (in)equalities is defined in Section 3.2.1.

Definition 2.1.1 (run)**.** *A run of an integer transitioning system is the successively applying of rules, starting from a start function symbol.*

The termination of an ITS is now defined as the not existing of an infinite run, i.e. for every run we reach a state in which we can not apply any rule. Vice versa non-termination of an ITS is defined as the existence of an infinite run.

2.2 geometric non-termination arguments

Adapted from Jan Leikes and Matthias Heizmanns paper *geometric non-termination arguments (GNA)* [LH14] we will define the considered programs, define the division of these programs into STEM and LOOP and finally give the definition of *geometric non-termination arguments*.

2.2.1 Considered Programs

The considered programs are not bound to a special programming language. The paper works on so called *linear lasso programs*, which in fact are also used within AProVE. Instead of the linear lasso programs, AProVE represents them as ITSs, stated in Section 1.1. Because of the also stated conversion of the program into a Symbolic Execution Graph and because of the further analysis the applicability of GNAs is not bound to any programming language.
In order to define the specific conditions under which we can use the approach, we choose the language Java as an example.

2.2.2 Structure

The structure of the considered programs is quite simple. It contains an optional initialization of the used variables and a `while`-loop. Even though C would not accept the usage of a variable without declaration, the conversion to LLVM would still be sound. An example of a linear lasso program in C is shown in Figure 2.2.

- The STEM:
  The declaration and optional initialization of variables used within the `while`-loop. In Figure 2.2 lines 3 and 4 are considered the STEM. Only b is initialized with a value.
- The guard:
  The guard of the `while`-loop is essential to restrict the variable a as we will see in Section 3.1.2. With the restriction of $a+b \geq 4$ we can prove termination for an initial value of $a < 3$ without further analysis, and also, in order to prove non-termination, assume that initially $a \geq 3$.

[1] The standard linear integer form has the following pattern: $a_1 * v_1 + \cdots + a_n * v_n + c$, where $a_i, c \in \mathbb{Z}$, $1 \leq i \leq n$.

- The linear updates:
  The updates of the variables within the `while`-loop are the most essential part for termination, since their values determine if the guard still holds. The approach only works with linear updates of the variables, so for every variable v_i where $1 \leq i \leq n$ we can have an update of the form $v_i = a_1 * v_1 + ... + a_n * v_n + c$ with $a_i \in \mathbb{Z}$ for $1 \leq i \leq n$ and $c \in \mathbb{Z}$.

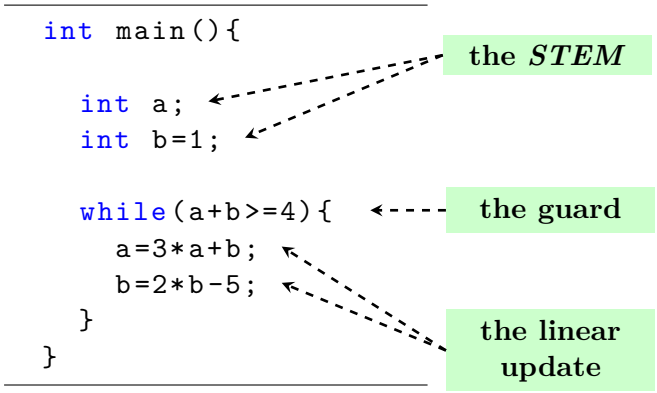


Figure 2.2: A linear lasso program fulfilling the conditions mentioned in Section 2.2.2 to be applicable

The guard and linear updates together form the so called LOOP.

The program from Figure 2.2 can be transformed into the ITS shown in Figure 2.3, which is conform to the structure described in Section 2.2.2. As we can see, the original program can be recognized quite easily. The first rule in line 1 represents the STEM, while the second line forms the LOOP.

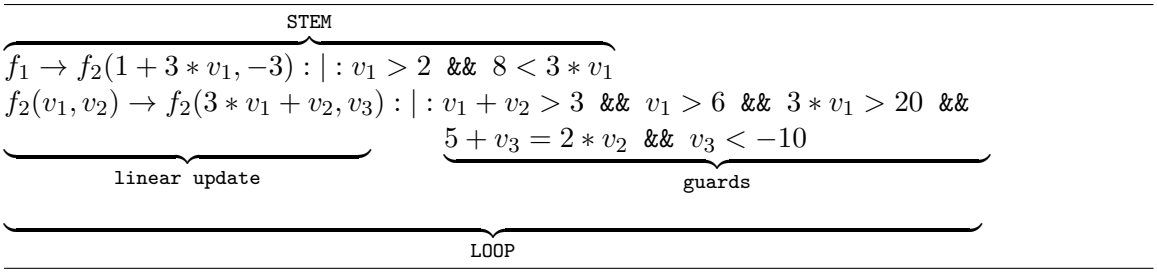


Figure 2.3: The ITS corresponding to the Java program in Figure 2.2

Neglecting the conditional terms for now, the declaration of v_2', which is the second argument of f_2, in line 1 to -3, because during the construction of the Symbolic Execution Graph, AProVE always unrolls the first iteration of the LOOP. Therefore, the STEM computed by AProVE will always contain the first execution of the LOOP. Starting with $b = 1$ one step would be the computation of $b = 2 * 1 - 5 = -3$. The definition of v_1' is more difficult and will be shown within Section 3.1. Also the update for the first argument of f_2 v_1' within line 2 is the same as in Figure 2.2 line 7. The definition of the second argument of f_2, $v_2' = v_3$, is fundamental and not as simple as v_1', since v_3 is a new variable introduced within the guards through the equality $5+v_3 = 2*v_2$. The handling of such variables will be explained in Section 3.2.1 and Section 3.2.2.

2.2.3 Preliminary Definitions

In order to be able to define the key element of this approach, the geometric non-termination argument, we have to define a number of matrices and constant vectors, which are used to derive such a geometric non-termination argument.

Definition 2.2.1 (STEM)**.** *The STEM is denoted as a vector $x \in \mathbb{Z}^n$, where n is the number of variables within the rule of the start function symbols right hand side of a ITS. The values of x can be constants or defined by conditions. Examples are shown within Section 3.1.*

Definition 2.2.2 (Guard Matrix, Guard Constants)**.** *Let $n \in \mathbb{N}$ be the number of distinct variables, v_i $1 \leq i \leq n$ the i-th distinct variable names occurring on the left hand side, $m \in \mathbb{N}$ be the number of guards not containing equality, $a_{i,j} \in \mathbb{Z}$ $1 \leq i \leq n$, $1 \leq j \leq m$ the factor of v_i in g_j and $c_i \in \mathbb{Z}$ be the constant term within r_j.*

Then the Guard Matrix $G \in \mathbb{Z}^{m \times n}$ is defined as $G_{i,j} = a_{i,j}$ and Guard Constants $g \in \mathbb{Z}^m$ are defined as $g_i = c_i$.

Newly introduced variables must not be represented by a column of the Guard Matrix, but create substitutions further used in Section 3.1.2, Section 3.2.1 and Section 3.2.2.

Example 1. *The corresponding Guard Matrix to Figure 2.3 is* $G = \begin{pmatrix} -1 & -1 \\ -1 & 0 \\ -3 & 0 \\ 0 & 2 \end{pmatrix}$ *and the Guard Constants is* $g = \begin{pmatrix} -4 \\ -7 \\ -21 \\ -6 \end{pmatrix}$

The normalization of the guards r_i to the form $a_{i,1}v_1 + \ldots a_{i,n}v_n \leq c$ transforms for example the guard r_1 in the following way

$$r_1 \Leftrightarrow v_1 + v_2 > 3 \Leftrightarrow -v_1 - v_2 < -3 \Leftrightarrow -v_1 - v_2 \leq -4$$

Definition 2.2.3 (Update Matrix, Update Constants)**.** *Let $n \in \mathbb{N}$ be the number of distinct variables of the left hand side, v_i $1 \leq i \leq n$ the i-th distinct variable name, $m \in \mathbb{N}$ the arity of the function symbol of the right hand side, v'_i $1 \leq i \leq m$ the i-th variable definition of the right hand sight's function symbol, $a_{i,j} \in \mathbb{Z}$ $1 \leq i \leq n$ $1 \leq j \leq m$ be the factor of variable v_i in variable definition v'_i and $c_i \in \mathbb{Z}$ $1 \leq i \leq m$ the constant term of v'_i.*

Then the Update Matrix $U \in \mathbb{Z}^{m \times n}$ is defined as $U_{i,j} = a_{i,j}$ and Update Constants $u \in \mathbb{Z}^m$ are defined as $g_i = c_i$.

Regarding the new variable v_3, we have to substitute in order to keep the desired size of the matrix. This procedure is further defined within Section 3.2.1 and Section 3.2.2.

Example 2. *The corresponding Update Matrix to Figure 2.3 is* $U = \begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix}$ *and the Update Constants are* $u = \begin{pmatrix} 0 \\ -5 \end{pmatrix}$.

The second row is not as obvious as the first row and will be derived in detail within Section 3.2.2.

Definition 2.2.4 (Iteration Matrix, Iteration Constants)**.** *Let G be the Guard Matrix, g the Guard Constants, U the Update Matrix, u the Update Constants, $n \in \mathbb{N}$ the number of variables and $m \in \mathbb{N}$ the number of conditional terms.*
Also let $\mathbf{0}$ be a matrix of the size of G with only entry's 0 and I denote the identity matrix having the same dimension as U.

The Iteration Matrix $A \in \mathbb{Z}^{2*n+m \times 2*n}$, which defines one complete execution of the LOOP, and the Iteration Constants $b \in \mathbb{Z}^{2*n+m}$ is defined as

$$A = \begin{pmatrix} G & \mathbf{0} \\ U & -I \\ -U & I \end{pmatrix} \text{ and } b = \begin{pmatrix} g \\ -u \\ u \end{pmatrix} \text{ [LH14]}$$

Definition 2.2.5 (LOOP)**.** *The LOOP is defined as a tuple (A, b), where A is the Iteration Matrix and b the Iteration Constants of an ITS.*

Now we can define the key element, which was originally defined for linear lasso programs.

Definition 2.2.6 (geometric non-termination argument)**.** *A tuple of the form:*

$$(x, y_1, \ldots, y_k, \lambda_1, \ldots, \lambda_k, \mu_1, \ldots, \mu_{k-1})$$

is called a geometric non-termination argument of size k for a program $= (STEM, LOOP)$ with n variables iff all of the following statements hold:

$$\begin{aligned} &\textit{(domain)} && x, y_1, \ldots, y_k \in \mathbb{R}^n,\ \lambda_1, \ldots \lambda_k, \mu_1, \ldots \mu_{k-1} \geq 0 \\ &\textit{(init)} && x \text{ represents the start term (STEM)} \\ &\textit{(point)} && A \begin{pmatrix} x \\ x + \sum_i y_i \end{pmatrix} \leq b \\ &\textit{(ray)} && A \begin{pmatrix} y_i \\ \lambda_i y_i + \mu_{i-1} y_{i-1} \end{pmatrix} \leq 0 \text{ for all } 1 \leq i \leq k \end{aligned}$$

Note that $y_0 = \mu_0 = 0$ is set for the ray instead of a case distinction. [LH14]

The usage of such a geometric non-termination argument is justified by the following sentence:

Sentence 2.1. *If a geometric non-termination argument a for a program p exists, then p does not terminate. [LH14]*

2.3 Reverse-Polish-Notation-Tree

Within the program of deriving a geometric non-termination argument it happens that we get a mathematical term in the so-called *Polish Notation* or *Reverse Polish Notation in prefix notation*, which is a special form of rewriting a, in our case linear, expression to compute the solution efficiently using a stack. [Wik17a] Within our program we use this kind of notation to parse it into our own tree-structure to do further analysis.

As shown in Figure 2.4 we have an `abstract` root, subclasses for every occurring type of element within the ITS, a `static` parsing of a given term and an exception for parsing exceptions. An example of the Reverse Polish Notation Tree's usage is shown in Figure 2.5

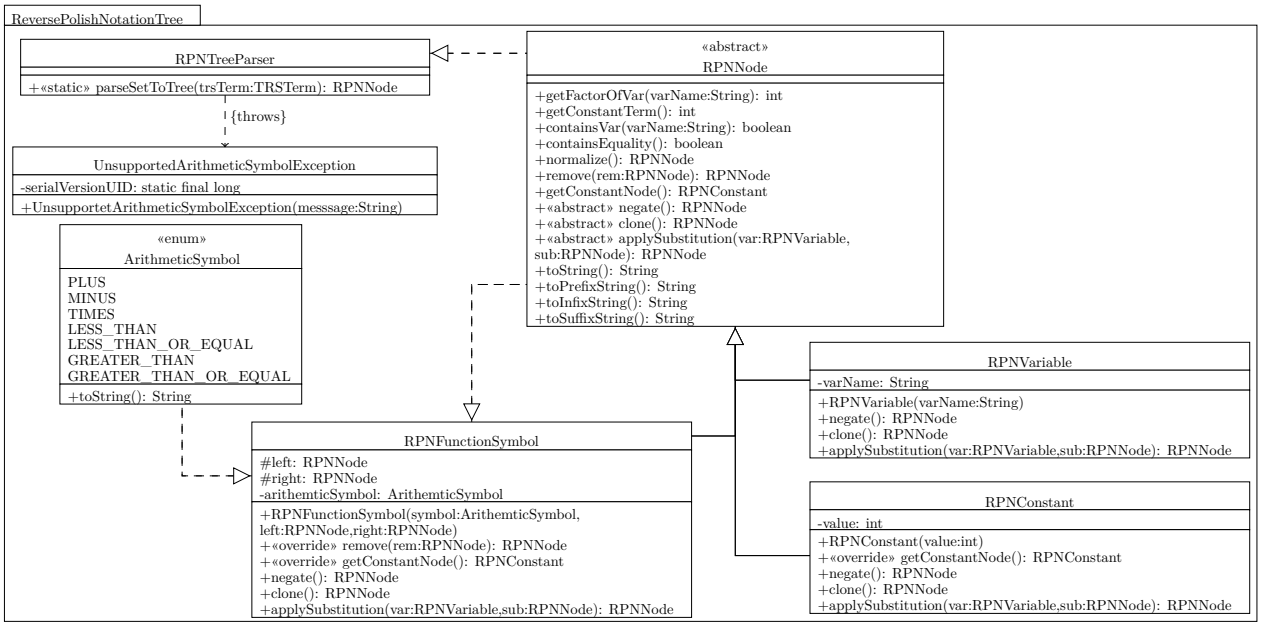


Figure 2.4: The class diagram of the Reverse Polish Notation Tree within the geometric non-termination analysis

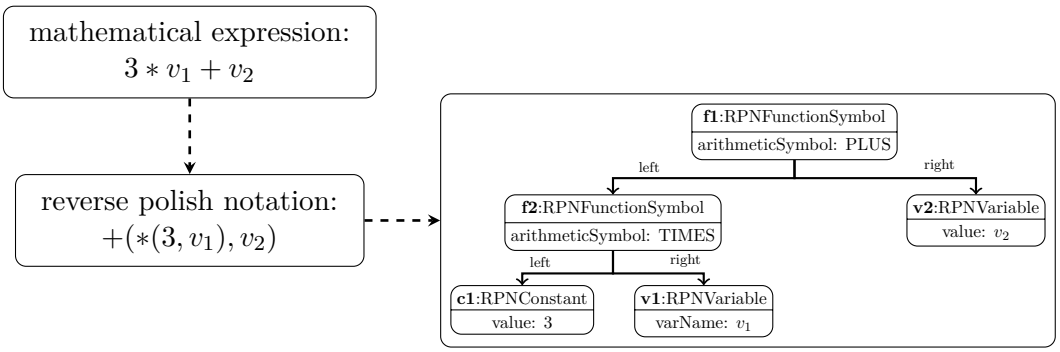


Figure 2.5: An example of the representation of the term $3 * v_1 + v_2$ as a graph using the Reverse Polish Notation Tree of Section 2.3

2.4 SMT-Problem

Also we have to consider an *Satisfiability Modulo Theory*-Problem (SMT-Problem), we have to solve to derive a geometric non-termination argument fulfilling all the criterias of Definition 2.2.6. Since SMT-Problem solving is a big research topic on it's own we only consider the very basic of SMT-Solving necessary to understand how the program solves the problem.
Within this approach we use the so called `Basic Structures` defined within AProVE to add assertions to the SMT-solver using the `SMTFactory`. An example of the structure of the assertions can be found in Figure 2.6.

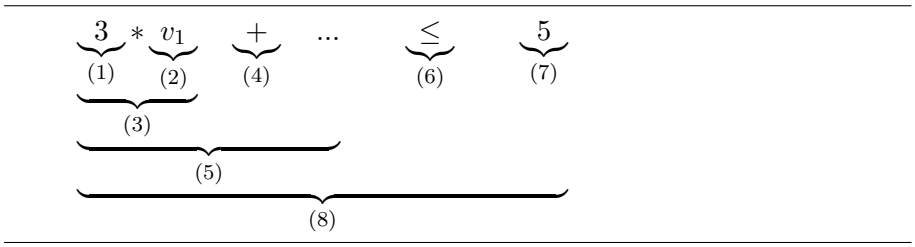


Figure 2.6: An example to show the structure of an assertion used for the SMT-solver

Such an example assertion can be split into different parts:

(1) `PlainIntegerConstants` as coefficients

(2) `PlainIntegerVariables'` as variables the SMT-solver should derive values for such that all assertions are satisfied

(3) A coefficient multiplied with a variables is represented by an `PlainIntegerOperation` with `ArithmeticOperationType` `MUL` to denote multiplication

(4) An `ArithmeticOperationType` of type `ADD` to denote addition

(5) The left hand side is one big `PlainIntegerOperation` consisting of the addition(4) of the multiplication (3) of coefficient's (1) and variables (2).

(6) The `IntegerRelationType` defining the assertion. We only use the *EQ (equal)* or *LE (less than or equal)* relations.

(7) The right hand side is only one `PlainIntegerConstant`

(8) The whole line is a `PlainIntegerRelation`, which can be transformed into the `SMTExpressionFormat` the SMT-solver uses.

We use a solver within AProVE to create a bunch of assertions restricting the possible solution space. Since we operate in integer arithmetic and use linear equations we can restrict the solver to only use *quantifier free linear integer arithmetic*. In order to solve the problem given by the assertions the solver tries to derive a model satisfying all of them or derive an unsatisfiable core. [Áb16]

Example 3. *Consider the following assertions that should hold:*

$$x \leq y \quad x > 5 \quad x + y \leq 20 \quad y \neq 10$$

Then a possible model would be $m_1 = \{x = 6, y = 6\}$*. An other model would be* $m_2 = \{x = 6, y = 7\}$*. However If we change the third rule to* $x + y \leq 10$ *there is no model to the problem and we would receive the unsatisfiable core* $c = \{x \leq y,\ x > 5,\ x + y \leq 10\}$*.*

Since for Definition 2.2.6 the existence of a model is the crucial information, the model which should be derived is arbitrary among the set of possible models.

Further knowledge about SMT-Problem solving can be gathered from the lecture "Introduction to Satisfiability Checking" or the SMT-RAT toolbox for Strategic and Parallel SMT Solving by Prof. Dr. Erika Ábrahám and her team at the *RWTH Aachen University* [CKJ+15].

Chapter 3

Geometric Non-Termination

Now that all preliminaries are stated we can start to take a look at how the approach works within AProVE. To find a geometric non-termination argument and so prove non-termination we use AProVE to generate an ITS of a given program, which is defined within Section 1.1. Based on the calculated ITS we derive the STEM, the LOOP and then generate an SMT-Problem using Definition 2.2.6 and compute a geometric non-termination argument, which would be a prove of non-termination, or state that no geometric non-termination argument can exist, which does not infer termination nor non-termination.

3.1 Derivation of the STEM

The derivation of the STEM is the first step in order to derive a geometric non-termination argument. As described in Section 2.2.2 the STEM defines the variables before iterating through the LOOP. Owed to the fact, that AProVE has to find the lasso to derive an ITS within the generated Symbolic Execution Graph one iteration through the LOOP will be calculated. Obviously this does not falsify the result. If it does not terminate i will still not terminate after one iteration and if it terminates after n iterations and we compute one it will still terminate after $n-1$ iterations.
Within the derivation of the STEM we distinguish between two cases discussed in the following sections.

3.1.1 Constant STEM

The constant stem is the easiest case to derive the STEM from. It has the form:

$$f_x \to f_y(c_1, \dots c_n) : | : TRUE$$

Here the $c_1, \dots c_n \in \mathbb{Z}$ denote constant numbers.
An example of a constant STEM is shown in Figure 3.1. The values of x can be directly read from the right hand side and need no further calculations.

1 $f_1 \to f_2(10, -3) : | : TRUE$

Figure 3.1: An example of a constant ITS rule to derive the STEM. The STEM in this case would be $\begin{pmatrix} 10 \\ -3 \end{pmatrix}$

3.1.2 Variable STEM

The more complex case is given if the start function symbol has the following form:

$$f_x \to f_y(v_1, \dots v_n) : | : cond$$

where v_i $1 \leq i \leq n$ is either a constant term like in Section 3.1.1 or a linear update defined by the *cond* term. An example for such a STEM is shown in Figure 3.2. In order to derive terms in $\mathbb{Z}$ an SMT-Problem needs to be solved. We can compute the Guard Matrix, Guard Constants, Update Matrix and Update Constants of the start function symbol and use the `SMTFactory`, which is explained within Section 2.4, to create the assertions leading to either an assignment of the STEM x to a value or to a unsatisfiable core. Such a core would state, that the `while`-Loop would not hold after any assignment. If such a constellation entails termination or if it just does not entail non-termination needs further observance not provided in this paper.
The handling of equations within the guard term is described within Section 3.2.1 and underlies the same procedure regarding the stem.

1 $f_1 \to f_2(1 + 3 * v, -3) : | : v > 2$ `&&` $8 < 3 * v$

Figure 3.2: An example of a variable ITS rule to derive the STEM from. In order to derive the STEM, an v fulfilling the conditions need to be found using an SMT-Solver. Since $v = 3$ is the first number provided by the SMT-solver in $\mathbb{Z}$ that satisfies the guards the STEM would be $\begin{pmatrix} 1 + 3 * 3 \\ -3 \end{pmatrix} = \begin{pmatrix} 10 \\ -3 \end{pmatrix}$. Note that $v = 4$ would be equally permissible.

3.2 Derivation of the LOOP

The derivation of the LOOP is pretty straight forward applying Definition 2.2.2, Definition 2.2.3 to a looping rule and then computing Iteration Matrix and Iteration Constants using Definition 2.2.4, if no guards with equalities ("=") occur. Then we have to perform previous steps to apply the mentioned definitions.
Let f_x be the starting function symbol given by the ITS and r_i be a rule, with

1 $f_x \to f_y(v_1, \dots, v_n) : | : cond_1$

then we take the in lexicographical order first rule r_l of the form

1 $f_y(v_1, \dots, v_n) \to f_y(v'_1, \dots, v'_n) : | : cond_2$

and compute the Iteration Matrix and Iteration Constants according to r_l. If we observe, that a second rule that could possibly chosen exists we encounter non-determinism, which is not supported so far.

3.2.1 The Guard Matrix and Guard Constants

The derivation of the Guard Matrix and Guard Constants can be achieved by applying the Definition 2.2.2 to the guards of the given rule r_l. For that we create G as the coefficient matrix. The size of G is determined by the arity of the function symbol of r_l and the number of guards not containing "=". In Section 3.2.1 we show the dealing with guards containing "=" with Example 4 as an example. The first step is to define the desired form of the guards. For that we introduce the *standard guard form*, the guards are given from the Symbolic Execution Graph, and the desired *strict guards form*.

Definition 3.2.1 (standard guard form)**.** *A guard g is in standard guard form iff $g := \varphi \circ c$, with φ in standard linear integer form and $\circ \in \{<, >, \leq, \geq, =\}$.*
A condition to a rule cond is in standard guard form iff

$$cond = \{g | g \text{ guard}, g \text{ is in standard guard form}\}$$

The condition given be the Symbolic Execution Graph is one rule r , which represents a set G in standard guard form as

$$r = \&\&(g_1, (\&\&(\dots, (\&\&(g_{n-1}, g_n))\dots))),$$

where $g_i \in G$.
The easiest way to retrieve the guards g_i is by using Algorithm 1.

Algorithm 1 Retrieving a set of guards G from a rule r of the form stated in Section 3.2.1

```
1:  function computeGuardSet(Rule r)        ▷ r has to be a rule representing a cond-term
2:      Stack stack ← r
3:      Set guards
4:      while !stack.isEmpty() do
5:          item ← stack.pop
6:          if item is of the form &&(x1, x2) then      ▷ break up concatenation of two guards
7:              add x1 and x2 to stack                  ▷ and add them individually
8:          else
9:              add item to guards                      ▷ if it is no concatenation it is a single guard
10:         end if
11:     end while
12:     return guards
13: end function
```

So we get a set $G = \{g \mid g \text{ is in standard guard form}\}$. Now we want to compute the desired form, the *strict guard form*, from which we can derive the Guard Matrix and Guard Constants.

Definition 3.2.2 (strict guard form)**.** *A guard g is in strict guard form iff $g := \varphi \leq c$, with φ in standard linear integer form and $c \in \mathbb{Z}$ subtracted by a possible constant c_φ previously included by φ*

To transfer a guard from standard guard form to strict guard form we have to apply the two following steps:

1. **rewrite equations**
   If the guards contain a guard with the symbol "=" we have to rewrite the "new" variable. To define, which are new variables and substitute these, we perform the following algorithm:

Algorithm 2 The algorithm to handle equalities that introduce new variables within the guards.

```
1: function filterEqualities(G)                          ▷ G is in standard guard form
2:     V_left = {v |the left hand side of the rule contains v}
3:     V_right = {v |the right hand side of the rule contains v}
4:     V_sub = V_right − V_left
5:     define substitution θ = {}
6:     while V_sub ≠ ∅ do
7:         select s ∈ V_sub
8:         select g_s ∈ {g ∈ G | g contains " = "}        ▷ Should only be one guard
9:         remove g_s from G
10:        rewrite g_s to the form s = ψ
11:        θ = θ{s/ψ}                                    ▷ update substitution
12:        for all g ∈ G do                              ▷ apply Substitution
13:            g = θg
14:        end for
15:        remove s from V_sub
16:    end while
17:    return G
18: end function
```

The result of that is a set G', which satisfy the condition of occurring variables.

2. **normalizing**
   Given the new set G' we have to normalize the inequalities to achieve strict guard form. This is done with two steps:

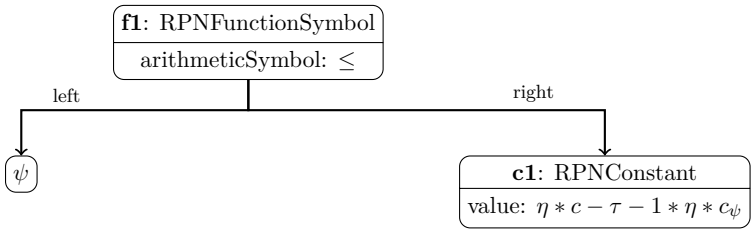
   (a) rewrite a guard g_i of the form $g_i \Leftrightarrow \psi + c_\psi \circ c$, where $\circ \in \{<, >, \leq, \geq\}$ to the form $\eta * \psi + \eta * c_\psi \leq \eta * c - \tau$ depending on $\circ$.

$\circ$	η	τ	$\eta * \psi + \eta * c_\psi \leq \eta * c - \tau$
$<$	1	1	$\psi + c_\psi \leq c - 1$
$>$	-1	1	$-\psi - c_\psi \leq -c - 1$
$\leq$	1	0	$\psi + c_\psi \leq c$
$\geq$	-1	0	$-\psi - c_\psi \leq -c$

   η is the indicator of inverting the guard to convert $\geq$ ($>$) to $\leq$ ($<$)
   τ can be seen as the subtraction of 1 to receive the $\leq$ instead of a $<$.

   (b) transfer the c_ψ to the right side to only contain one constant term located on the right side. So the final form is $\eta * \psi \leq \eta * c - \tau - 1 * \eta * c_\psi$, where $\eta * c - \tau - 1 * \eta * c_\psi$ is constant term and ψ is in standard linear integer form without a constant term.

After that every guard is in strict guard form. So all we have to do in order to now compute the Guard Matrix and the Guard Constants is to apply Algorithm 3, to determine the coefficients stored in the Guard Matrix, and determine the constant terms.
Regarding the normalization, which is implemented on the Reverse Polish Notation Tree, we could apply Algorithm 4 to compute the constant terms, but also we can use the normalized guard to compute the constant term within linear time. The implementation of the transformation guarantees the following form for a guard in strict guard form:



So the constant term can simply be read off from the right child-node of the `RPNFunctionSymbol` "$\leq$", neglecting the left ψ-term.

Example 4. *This example is based in the ITS from Figure 2.3. Regarding the ITS we have the guard-term:*

$$v_1 + v_2 > 3 \;\&\&\; v_1 > 6 \;\&\&\; 3 * v_1 > 20 \;\&\&\; 5 + v_3 = 2 * v_2 \;\&\&\; v_3 < -10$$

which lead to the set G using Algorithm 1:

$$\{v_1 + v_2 > 3,\ v_1 > 6,\ 3 * v_1 > 20,\ 5 + v_3 = 2 * v_2,\ v_3 < -10\}$$

Starting with Algorithm 2 to handle equalities:

(line 2-4) We compute $V_{left} = \{v_1, v_2\}$, $V_{right} = \{v_1, v_2, v_3\}$ so $V_{sub} = \{v_3\}$

(line 5) Begin with $\theta = \{\}$

(line 7,8) Since obviously $V_{sub} \neq \emptyset$ we select $s = v_3$ and select $g_s \Leftrightarrow 5 + v_3 = 2 * v_2$

(line 9,10) g_s rewritten to the form $s = \psi$ then follows with $v_3 = 2 * v_2 - 5$

(line 11) $\theta = \theta\{s/2 * v_2 - 5\} = \{s/2 * v_2 - 5\}$

(line 12-15) $G = \{v_1 + v_2 > 3,\ v_1 > 6,\ 3 * v_1 > 20,\ 2 * v_2 - 5 < -10\}$

(end) Since $V_{sub} = \emptyset$ return G

Starting with the normalization:
Applying the stated rule for the different inequation-signs we receive the new guards:

$$\{-1 * v_1 + -1 * v_3 \leq -4,\ -1 * v_1 \leq -7,\ -3 * v_1 \leq -21,\ 2 * v_2 \leq -6\}$$

Note that the writing of for example $-1 * v_1$ is wanted in order to be able to neglect the case of for example $-2 * -v_1$ if -2 gets multiplied.

From that we apply Algorithm 3 and receive the Guard Matrix $G = \begin{pmatrix} -1 & -1 \\ -1 & 0 \\ -3 & 0 \\ 0 & 2 \end{pmatrix}$. *Also using the stated structure details we can derive the Guard Constants* $g = \begin{pmatrix} -4 \\ -7 \\ -21 \\ -6 \end{pmatrix}$

3.2.2 The Update Matrix and Update Constants

The Update Matrix and Update Constants can be derived quite easily. The updates within the function symbol neither contain an equation nor an inequation sign. Therefore no new variables can be initialized. Since it is still possible, that some of the, within the guard part instantiated, variables appear within the update we have to apply the final set of substitutions θ from Algorithm 2 to the linear update.

Example 5. *This example is based in the ITS from Figure 2.3 in combination with the Example 4 providing V_{sub} and θ.*
Since the set of substitutions $\theta = \{v_3/2 * v_2 - 5\}$ is not empty and within the update given by
$(3 * v_1 + v_2,\ v_3)$
contains $v_3 \in V_{sub}$ we have to apply the substitution and receive:
$(3 * v_1 + v_2,\ 2 * v_2 - 5)$

After restraining the updates to only mention the desired variables we can introduce Algorithm 3 as a procedure to compute the coefficient of a given variable. It performs a recursive search on the tree and uses the standard linear integer form definition that a coefficient is always the left child of the multiplication with it's corresponding variable. The procedure works like the following:

Algorithm 3 Derivation of a coefficient within an Reverse Polish Notation Tree

```
1:  function GETCOEFFICIENT(query)
2:      if this == query then                          ▷ query is a variable name
3:          return 1
4:      else if this does not contain query then       ▷ tree does not contain query
5:          return 0
6:      end if
7:
8:      if this represents PLUS then                   ▷ Choose the subtree containing the query
9:          if left side contains query then
10:             return getCoefficient(query)
11:         else
12:             return getCoefficient(query)
13:         end if
14:     end if
15:     if this represents TIMES then                  ▷ Retrieve value
16:         if this.right == query then
17:             return this.left.value
18:         end if
19:     end if
20: end function
```

An example derivation of a factor using Algorithm 3 is shown in Figure 3.3.

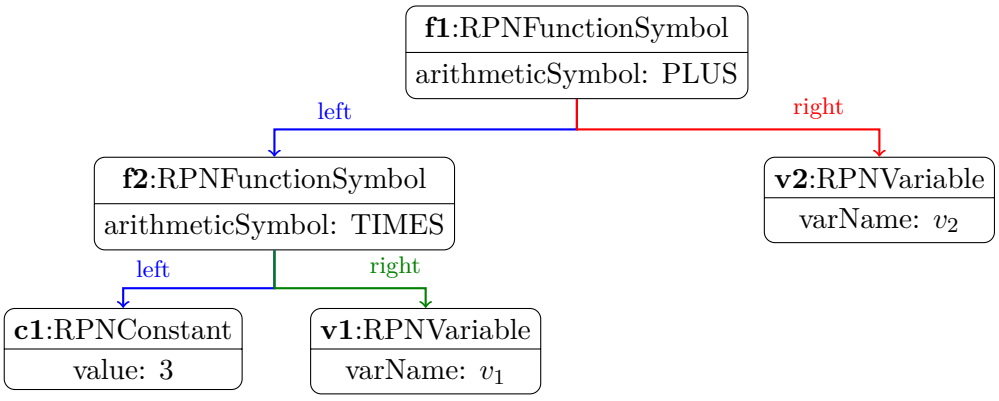


Figure 3.3: An example of deriving the coefficient of a given formula and a variable as query. This example uses the Reverse Polish Notation Tree of Figure 2.5 and variable v_1 as the query.

The red-arrow stands for the neglected right subtree of the root node, which can be neglected because the query is not contained. The blue-arrows show the path to the subtree further investigated. The green-arrow determines, that the right child node is the query so the left child node has to be the coefficient. Since the underlying update is in standard linear integer form the left subtree has to be a *RPNConstant*.

The Update Constants can be derived by an simplification of Algorithm 3, since we only have to retrieve the constant term within the tree. The corresponding derivation is given by Algorithm 4.

Algorithm 4 Derivation of a constant term within an Reverse Polish Notation Tree

```
1:  function getConstantTerm
2:      if this is a constant then
3:          return this.value
4:      end if
5:
6:      flip ← 1
7:      if this represents MINUS then                ▷ flip result in case of prev. negation
8:          flip ← −1
9:      end if
10:     if this represents sth. ≠ TIMES then
11:         left ← left.getConstantTerm()                              ▷ recursive calls
12:         right ← right.getConstantTerm() ∗ flip
13:         return left + right
14:     end if
15:     return 0
16: end function
```

An example of a constant term using Algorithm 4 can be found in Figure 3.4.

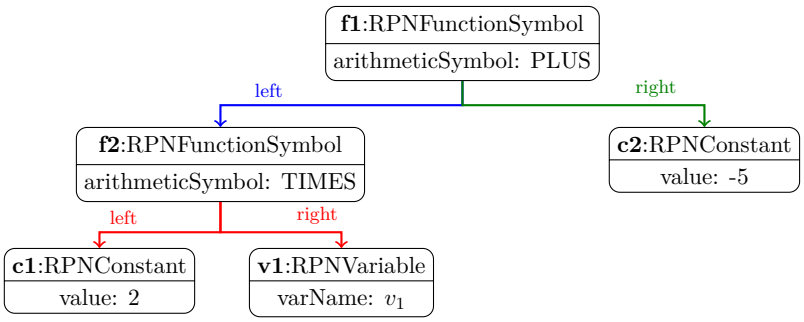


Figure 3.4: An example derivation of a constant term in the second variable update of the example in Example 5. Here red stands for neglected paths, blue stands for considered paths/recursive calls, and green stands for a found constant term.

Since a constant $c < 0$ can stored in a constellation shown in Figure 3.5 we consider a variable $flip$ to store a sign change occurring for a subtraction. Knowing that the standard linear integer form is used all occurs of a multiplication can be neglected.
Through the standard linear integer form one of the recursive calls has to be 0 since only one constant term exists.

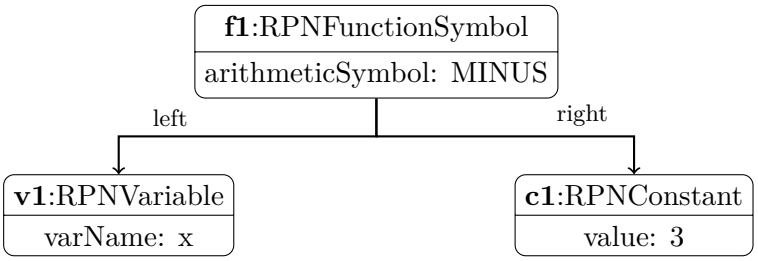


Figure 3.5: The Reverse Polish Notation Tree of the term $x - 3$, where the $flip$ of Algorithm 4 has to be used. This constellation can not be universally neglected. The value recursively found for the constant term would be $(-1) * 3 = -3$

Using Algorithm 3 and Algorithm 4 one can derive the Update Matrix $U \in \mathbb{Z}^{n \times n}$ and Update Constants $u \in \mathbb{Z}^n$ for a rule r_j of the form

$$r_j := f_y(v_1, \dots v_n) \to f_y(v'_1, \dots v'_n) : | : cond$$

so that the following holds:

$$U \times \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} + u = \begin{pmatrix} v'_1 \\ \vdots \\ v'_n \end{pmatrix}$$

3.2.3 The Iteration Matrix

The Iteration Matrix and Iteration Constants are a composition of the previously derived *Iteration-* and Guard Matrix respectively *Iteration-* and Guard Constants.
As stated in Definition 2.2.4 the Iteration Matrix and Iteration Constants can be computed as

$$A = \begin{pmatrix} G & \mathbf{0} \\ M & -I \\ -M & I \end{pmatrix} \text{ and } b = \begin{pmatrix} g \\ -u \\ u \end{pmatrix} \text{ [LH14]}$$

Given G, g, U and u computing A and b is simply inserting and creating a matrix $\mathbf{0} \in \{0\}^{m \times n}$ and identity-matrix $I \in \{0, 1\}^{n \times n}$, where n is the number of distinct not newly introduced variables and m the number of guards without equations.

3.3 Derivation of the SMT-Problem

The existence of a geometric non-termination argument is checked using an SMT solver, presented in Section 2.4, which will either give us a model satisfying the constraints or proof the non existence by giving an unsatisfiable core.
The constraints the SMT solver has to fulfil are the four criteria mentioned within Definition 2.2.6, which are non-linear. So the satisfiability of these is decidable. Since we derive the deterministic update as Update Matrix we can further compute it's eigenvalues and assign these to $\lambda_1, \dots \lambda_k$, receive linear constraints and thus can decide existence efficiently. [LH14].
So the next step in order to proof non-termination is to compute the eigenvalues of the Update

Matrix. This is done by the *Apache math3* library [Apa17] because of performance reasons. Computation of such matrices can be very costly if programmed inefficiently. After computing the eigenvalues, we have set values for the STEM x and $\lambda_1, \dots \lambda_k$ as constant values.
Using the `SMTFactory`, which offers methods to create the within Section 2.4 and Figure 2.6 stated structure, we are able to create assertions and add them to the SMT-solver, such that the following holds:

> If the SMT-solver, with assertions $a_1^p, \dots a_n^p$ created from program p,has a model m
> then m defines variables $y_1, \dots, y_k$ and μ_1, μ_{k-1} within $\mathbb{N}$ such that
> $(x, y_1, \dots, y_k, \lambda_1, \dots, \lambda_k, \mu_1, \dots, \mu_{k-1})$ is a geometric non-termination argument.

3.3.1 The Domain Criteria

(domain) $x, y_1, \dots, y_k \in \mathbb{R}^n$, $\lambda_1, \dots \lambda_k, \mu_1, \dots \mu_{k-1} \geq 0$

(see: Definition 2.2.6)

The Domain Criteria for x and $y_1, \dots y_k$ are trivial, because at no point of computation we would consider a vector $v \in \mathbb{C}$. The arity of x is set within the derivation of the STEM (see: Section 3.1) and sets the starting values for the n-variables. The arity of every y_i is determined within the assertions of the Point Criteria and the Ray Criteria.
The assertions one has to make towards the SMT-solver is to ensure that the λ's are not negative, since the proof by Leike and Heizmann is only for positive eigenvalues. Also for the μ's we have to add the assertion of being positive to fulfil the criteria.
Research on negative eigenvalues is mentioned within Chapter 5.

3.3.2 The Initiation Criteria

(init) x represents the start term (STEM)

(see: Definition 2.2.6)

The Initiation Criteria is quite trivial to mention within the SMT-solver, since we defined the STEM x within Section 3.1 to be exactly the *start term*.
So this criteria adds no further assertions towards the SMT-solver.

3.3.3 The Point Criteria

(point) $A \begin{pmatrix} x \\ x + \sum_i y_i \end{pmatrix} \leq b$

(see: Definition 2.2.6)

Since within the Iteration Matrix A the Update Matrix U is contained twice with a different sign the Iteration Matrix creates ,through the Point Criteria exactly, opposite signed rules for the last $2n$ rows. This means that, even though within the Point Criteria the relation is $\leq$, the last $2n$ have to fulfil the equality of the rows.

Let $s \in \mathbb{R}^n$ for $1 \leq i \leq n$ be $s_i = x_i + \sum_j (y_j)_i$, where $(y_j)_i$ denotes the i-th entry of y_j. Then the Point Criteria can be rewritten to:

$$A \begin{pmatrix} x \\ s \end{pmatrix} \leq b$$

$$\Leftrightarrow \begin{pmatrix} & G & & 0 & \dots & 0 \\ a_{1,1} & \dots & a_{1,n} & -1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & \dots & a_{n,n} & 0 & \dots & -1 \\ -a_{1,1} & \dots & -a_{1,n} & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ -a_{n,1} & \dots & -a_{n,n} & 0 & \dots & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ s_1 \\ \vdots \\ s_n \end{pmatrix} \leq \begin{pmatrix} g \\ -u_1 \\ \vdots \\ -u_n \\ u_1 \\ \vdots \\ u_n \end{pmatrix}$$

$\Rightarrow Gx \leq g$, which means that the guards have to hold. Note that s_1 to s_n get multiplied with 0 so don't need to be considered. Also the following has to hold:

$$\begin{pmatrix} a_{1,1} & \dots & a_{1,n} & -1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & \dots & a_{n,n} & 0 & \dots & -1 \\ -a_{1,1} & \dots & -a_{1,n} & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ -a_{n,1} & \dots & -a_{n,n} & 0 & \dots & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ s_1 \\ \vdots \\ s_n \end{pmatrix} \leq \begin{pmatrix} -u_1 \\ \vdots \\ -u_n \\ u_1 \\ \vdots \\ u_n \end{pmatrix}$$

$$= \begin{pmatrix} a_{1,1} * x_1 & \dots & a_{1,n} * x_n & -1 * s_1 & \dots & 0 * s_n \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} * x_1 & \dots & a_{n,n} * x_n & 0 * s_1 & \dots & -1 * s_n \\ -a_{1,1} * x_1 & \dots & -a_{1,n} * x_n & 1 * s_1 & \dots & 0 * s_n \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ -a_{n,1} * x_1 & \dots & -a_{n,n} * x_n & 0 * s_1 & \dots & 1 * s_n \end{pmatrix} \leq \begin{pmatrix} -u_1 \\ \vdots \\ -u_n \\ u_1 \\ \vdots \\ u_n \end{pmatrix}$$

By looking closely one can see that for every line l_i $1 \leq i \leq n$ with

$$l_i^{\text{left}} \leq l_i^{\text{right}}$$

there is a rule l_{i+n} with

$$-l_i^{\text{left}} \leq -l_i^{\text{right}},$$

which can be rewritten as n rules of the form:

$$l_i^{\text{left}} = l_i^{\text{right}}$$

So using the `SMTFactory` we create such variables s_i and add the n assertions determined above. The guards can be neglected, since if they are constant there are no guards and if there are, we derive the STEM to fulfil these, so adding them would not give any advantage.
Since variable vectors are represented as a Reverse Polish Notation Tree we can use an implemented method to calculate the multiplication, normalize the outcome and parse the Reverse Polish Notation Tree into an assertion all featured by the `SMTFactory`.
The assertion ensuring that the new variables s_i are the sum of the i-th value of the y's is added

within Section 3.3.5.

3.3.4 The Ray Criteria

$$\text{(ray)}\ A\begin{pmatrix} y_i \\ \lambda_i y_i + \mu_{i-1} y_{i-1} \end{pmatrix} \leq 0 \text{ for all } 1 \leq i \leq k$$

(see: Definition 2.2.6)

The Ray Criteria is the hardest criteria in terms of asserting, because of it's way of computation. The computation can be split into two parts on it's own.

$i = 1$**:**

For $i = 0$ the second addend $\mu_{i-1} y_{i-1}$ is equal to 0, because of Definition 2.2.6 that $y_0 = \mu_0 = 0$. So with λ_1 being the first eigenvalue of the Update Matrix we get that $A\begin{pmatrix} y_1 \\ \lambda_1 y_1 \end{pmatrix} \leq 0$.
Through A and the Domain Criteria we know, that every $y_i \in \mathbb{R}^n$ so we add n new variables $y_{1,i}$, such that $y_1 = \begin{pmatrix} y_{1,1} \\ \vdots \\ y_{1,n} \end{pmatrix}$, multiply the Update Matrix A with the new vector regarding the substitution and create an assertion per row using the `SMTFactory`, the `IntegerRelationType LE` (denotes: less than or equal) and as the right hand side constant a 0.

$i > 1$**:**

Since we don't have any concrete values for any y_i or μ_i so far the solving of the problem with the term $\mu_{i-1} y_{i-1}$ is not linear and therefore the computation has to be either performed in *quantifier free non-linear integer arithmetic* or iterated over possible entry's for the μ's.
In the implemented approach the *quantifier free non-linear integer arithmetic* is used. Even if it's generally undecidable there are implementations over finite domains semi-deciding the problems. [BDE+14] [GAB+16]
Further comment about the usage of QF-NIA can be found within Chapter 4.
With λ_i being the i-th eigenvalue of the Update Matrix we can compute the result of the multiplication as in the $i = 0$ case, but have to normalize the outcome using the basic distributive property in order to handle it within an Reverse Polish Notation Tree and correctly generate an assertion from it.
So for every step $i > 1$ we add n new variables $y_{i,n}$ such that $y_i = \begin{pmatrix} y_{i,1} \\ \vdots \\ y_{i,n} \end{pmatrix}$ and a new variable μ_{i-1} such that

$$\lambda_i y_i + \mu_{i-1} y_{i-1} \Leftrightarrow \lambda_i \begin{pmatrix} y_{i,1} \\ \vdots \\ y_{i,n} \end{pmatrix} + \mu_{i-1} \begin{pmatrix} y_{i-1,1} \\ \vdots \\ y_{i-1,n} \end{pmatrix} \Leftrightarrow \begin{pmatrix} \lambda_i y_{i,1} + \mu_{i-1} y_{i-1,1} \\ \vdots \\ \lambda_i y_{i,n} + \mu_{i-1} y_{i-1,n} \end{pmatrix}$$

As in the other case we can simply compute the multiplication with the Update Matrix A , normalize the outcome and analogously create an assertion per row with the `SMTFactory`.

Note that $y_{i-1,n}$ represent the values of the previous step and therefore not only already exist, but also create a lattice of restrictions for the $y_{i,j}$ since the values depend highly on the previous values. At this point the relation of rewriting the problem as a geometric series, like it is done in the underlying paper [LH14], is quite comprehensibly.

3.3.5 Additional assertion

The final step of asserting needs to be done, because of the restriction of the variables from the Ray Criteria in Section 3.3.4 to the sum from the Point Criteria in Section 3.3.3.
The assertion, that needs to be added has the following form:

$$s_i = y_{1,i} + \cdots + y_{n,i}$$

This ensures that the values of y sum up to the values determined in the Point Criteria.

After the adding the Additional assertion from Section 3.3.5 the SMT-solver contains all the restrictions to compute a geometric non-termination argument or an unsatisfiable core for the given program.
If a geometric non-termination argument is found it is stored as an instance of the corresponding class and given to AProVE as a proof.

3.4 Verification of the Geometric Non-Termination Argument

An instance of a geometric non-termination argument can be rechecked giving the Iteration Matrix and Iteration Constants if all of the four criteria of Definition 2.2.6 by simply computing and checking if the conditions hold. In the whole chapter we work with the example used throughout the whole thesis with it's Java-code in Figure 2.2 its corresponding ITS in Figure 2.3. The in Chapter 3 computed Iteration Matrix and Iteration Constants based on the *Guard* and *Update Matrix/Constants* and also the Eigenvalues as the λ's.

The implemented technique provides a model m from the SMT-solver and it's assertions stated in Section 3.3. From the model m the technique filters the values and stores the in the geometric non-termination argument-class, which is given to AProVE as a witness. Such an geometric non-termination argument-object stores all variable declarations needed in order to revalidate itself to a given scenario. If one has an other technique of derivation it is possible to compute the matrices and set the values for an geometric non-termination argument-object and validate it, either proving it or mentioning, which criteria does not hold.

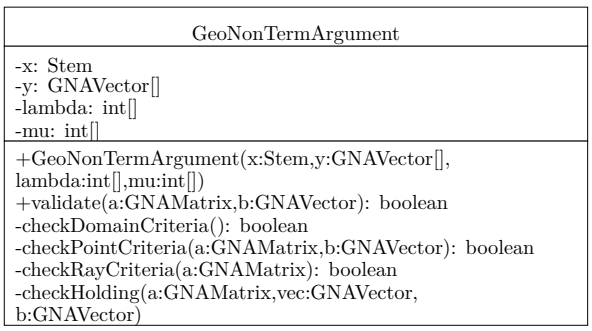


Figure 3.6: The Java-class of a geometric non-termination argument given to AProVE as a witness of non-termination

From Chapter 3 we get the following Iteration Matrix A and Iteration Constants b:

$$A = \begin{pmatrix} -1 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 3 & 1 & -1 & 0 \\ 0 & 2 & 0 & -1 \\ -3 & -1 & 1 & 0 \\ 0 & -2 & 0 & 1 \end{pmatrix} \quad b = \begin{pmatrix} -4 \\ -7 \\ -21 \\ -6 \\ 0 \\ 5 \\ 0 \\ -5 \end{pmatrix}$$

The from the technique given witness is the geometric non-termination argument of size 2 with the following values:

STEM	y_1	y_2	λ_1	λ_2	μ_1
$\begin{pmatrix} 10 \\ -3 \end{pmatrix}$	$\begin{pmatrix} 9 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 8 \\ -8 \end{pmatrix}$	3	2	0

From that we can start and validate every criteria stated in Definition 2.2.6.

3.4.1 (domain-criteria) $x, y_1, \ldots, y_k \in \mathbb{R}^n$, $\lambda_1, \ldots \lambda_k, \mu_1, \ldots \mu_{k-1} \geq 0$

The validity of this criteria is given in Section 3.4 with $n = 2$. This should not be surprising, since we defined the vectors to this dimension and the λ's and μ's to be ≥ 0 in Section 3.3.

3.4.2 (init-criteria) x represents the start term (STEM)

Also the given STEM meets the conditions as already derived in Section 3.1.2.

3.4.3 (point-criteria) $A \begin{pmatrix} x \\ x + \sum_i y_i \end{pmatrix} \leq b$

Now we have to compute the sum of the y_i's and observe if the inequation holds. So with the values given in Section 3.4 we receive:

$$A \begin{pmatrix} x \\ x + \sum_i y_i \end{pmatrix} \leq b \Leftrightarrow A \begin{pmatrix} 10 \\ -3 \\ 10 + (9 + 8) \\ -3 + (0 + (-8)) \end{pmatrix} \leq b \Leftrightarrow A \begin{pmatrix} 10 \\ -3 \\ 27 \\ -11 \end{pmatrix} \leq b \Leftrightarrow \begin{pmatrix} -7 \\ -10 \\ -30 \\ -6 \\ 0 \\ 5 \\ 0 \\ -5 \end{pmatrix} \leq \begin{pmatrix} -4 \\ -7 \\ -21 \\ -6 \\ 0 \\ 5 \\ 0 \\ -5 \end{pmatrix}$$

Not only one can see, that every row on it's own satisfies the desired inequation, we can also observe the within Section 3.3.3 stated equality hold for the last $2n = 2 * 2 = 4$ rows. The point-criteria obviously holds.

3.4.4 (ray-criteria) $A\begin{pmatrix} y_i \\ \lambda_i y_i + \mu_{i-1} y_{i-1} \end{pmatrix} \leq 0$ for all $1 \leq i \leq k$

Because the geometric non-termination argument is of size 2 we have only 2 ray-criteria subformulas that need to be tested

r_1: i=1

$$A\begin{pmatrix} y_1 \\ \lambda_1 y_1 \end{pmatrix} \leq 0 \Leftrightarrow A\begin{pmatrix} 9 \\ 0 \\ 3*9 \\ 3*0 \end{pmatrix} \leq 0 \Leftrightarrow A\begin{pmatrix} 9 \\ 0 \\ 27 \\ 0 \end{pmatrix} \leq 0 \Leftrightarrow \begin{pmatrix} -9 \\ -9 \\ -27 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

r_2: i=2

$$A\begin{pmatrix} y_2 \\ \lambda_2 y_2 + \mu_1 y_1 \end{pmatrix} \leq 0 \Leftrightarrow A\begin{pmatrix} 8 \\ -8 \\ 2*8+0*9 \\ 2*(-8)+0*0 \end{pmatrix} \leq 0 \Leftrightarrow A\begin{pmatrix} 8 \\ -8 \\ 16 \\ -16 \end{pmatrix} \leq 0 \Leftrightarrow \begin{pmatrix} 0 \\ -8 \\ -24 \\ -16 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Since all r_i $1 \leq i \leq 2$ are computed and checked that they hold also the ray-criteria holds.

Observing that all criteria hold the whole geometric non-termination argument is suitable to Definition 2.2.6 and therefore is a witness of non-termination regard the program stated in Figure 2.2.

Chapter 4

Evaluation and Benchmark

In this chapter we want to take a look at the implementation and evaluate, if the approach itself is useful in terms of applicable cases or if the approach works only on very exotic and uncommon preconditions.
Also we want to take a look at the benchmarks of the implementation, in terms of storage and computational efficiency.
Further we want to outline improvement possibilities of the implementation and problems within, where an efficient solution is not quite obvious.

4.1 Evaluation of the approach

The approach provides a sound and complete solution to specific type of programs. Given a tool like AProVE, which provides an ITS the further computation that has to be done can be solved quite efficient. Using an state of the art SMT-solver and the definition of λ_i to be the i-th eigenvalue the problem can also be resolved efficiently for given μ's. If the μ's are not given the problem is undecidable, which makes it still useful within AProVE, but not as strong as before. Since the paper itself does not mention equalities within the guards the substitution of newly introduced variables as handled within Section 3.2.1 is necessary in order to apply the definition. Such a substitution is very costly in computation time and also highly error prone.

4.2 Benchmarks

In order to discuss the possibilities of improvement regarding the implementation, we want to take a look at the performance. For that we compared AProVE, with all its techniques, to AProVE only running the geometric non-termination analysis. The set of programs is from the SV-COMP [SVC17].
Detailed results of all programs can be found in [Ben17]. For now we only focus on programs that meet the conditions of the geometric non-termination analysis. An overview is given below.

number of programs	847	∅-time for termination	+0.0078 sec
applicable programs	53	∅-time for non-termination	-1.2675 sec
false pos./neg.	0	∅-time save overall	0.76 sec

Regarding the average duration it takes AProVE to proof these programs is 4,513 seconds a save of average 0.76 seconds is a save of about 17 percent. Keeping in mind that the geometric

non-termination analysis has the objective of proving non-termination and that AProVE uses approaches in parallel the save can be even bigger. AProVE stops if any approach returns a proof. So the on average longer proof of termination does not affect the average save up.

```
1  int main(void){
2    int a = __VERIFIER_nondet_int();
3    int b = __VERIFIER_nondet_int();
4    int c = __VERIFIER_nondet_int();
5
6    while (a+b+c >= 4) {
7      a = b;
8      b = a+b;
9      c = b-1;
10   }
11   return 0;
12 }
```

Figure 4.1: A program AProVE can not prove non-termination for without the geometric non-termination analysis

Also there are programs, like the program shown in Figure 4.1,which AProVE can not proof non-termination for without using the geometric non-termination analysis. Including this technique to the set of approaches AProVE is able to prove non-termination. So the set of provable programs gets extended and improves AProVE.

4.3 Possible Improvement of the Implementation

The implementation of the approach is fully functional under the circumstances mentioned, like for example the defined structure in Section 2.2.2. Nevertheless also this implementation has certain cases in, which it does not perform as efficient as it could, or where it can be improved in terms of applicability. So here we state the possible improvements of the implementation to make it universally more useful and therefore stronger or more efficient.

4.3.1 SMT-solver logic

As already stated in Section 3.3 the problematic of the μ's can lead to a shift into undecidability, since the solvating of variable multiplication on integers (*quantifier free non-linear integer arithmetic*) is undecidable. Also mentioned in Section 3.3 there are a bunch of approaches, which lead to semi-decidability and therefore to the possibility to still use the variable multiplication within the problem if the μ's can be restricted to a finite domain.
A possible improvement could be an iteration over different values of the μ's. The number of problems, that have to be solved would be blow up, but the problem itself would always be decidable.
A reliable case-study of a large set of examples could underline the necessity of the iteration, since we wouldn't be able to derive a geometric non-termination argument using the *quantifier free non-linear integer arithmetic*. It could also lead to the overhead of computational cost using the iterative method, which can be useful if the problem does not have any time restrictions

of the deciding process, but is not suitable within the competitions AProVE participates, like the *International Competition of Termination Tools* or *International Competition on Software Verification*. [I217] [Ter17] [SVC17]
The *Termination Competition 2017*, which is organized by the *International Competition of Termination Tools*, for example has a time limit of 300 seconds and only allows 4 core usage, which makes an iterative method very costly. [Wik17b]

4.3.2 ITS program structure

As stated in Section 2.2.2 we restrict this implementation to the form

$$
\begin{array}{ll}
1 & f_x \rightarrow f_y(v_1, \dots v_n) \ : | : cond_1 \\
2 & f_y(v_1, \dots v_n) \rightarrow f_y\,(v'_1, \dots v'_n) : | : cond_2
\end{array}
$$

which obviously is a restriction, because ITSs of the form

$$
\begin{array}{ll}
1 & f_x \rightarrow f_y(v_1, \dots v_n) \ : | : cond_1 \\
2 & f_{y_1}(v_1, \dots v_n) \rightarrow f_{y_2}\,(v'_1, \dots v'_n) : | : cond_2 \\
3 & \quad \vdots \\
4 & f_{y_k}(v_1, \dots v_n) \rightarrow f_{y_1}\,(v'_1, \dots v'_n) : | : cond_{k+1}
\end{array}
$$

could possibly considered using the method k times. Analysing such ITS would make the implementation much stronger in terms of proving.
An other possible variation of the considered ITS could be like the following

$$
\begin{array}{ll}
1 & f_x \rightarrow f_y(v_1, \dots v_n) \ : | : cond_1 \\
2 & f_{y_1}(v_1, \dots v_n) \rightarrow f_{y_2}(v'_1, \dots v'_m) : | : cond_2 \\
3 & f_{y_2}(v_1, \dots v_m) \rightarrow f_{y_1}(v'_1, \dots v'_n) \ : | : cond_3
\end{array}
$$

where $m \neq n$, but the values v'_i $1 \leq i \leq m$ is computed as a linear update of the values v_j $1 \leq j \leq n$.

These are only two alternations of the considered structure, which would be also recommended to implement in order to create a more universal applicable method.

4.3.3 Reverse Polish Notation Tree

Within Section 2.3 we defined the Reverse Polish Notation Tree, on which we base the arithmetic computations and statements. AProVE also has a tree structure it handles such statements in. The Reverse Polish Notation Tree structure exists mainly because of two reasons:

1. the structure AProVE bases these statements on is much more complex, but also much more powerful, which made programming a lot more difficult. Parsing it into a tree, which can only contain elements expected to be in such expressions not only works equivalently it also prevents errors if AProVE's structure gets extended or changed. The `RPNTreeParse` handles the conversion and therefore can be seen as an adapter, which filters every ITS that must not occur in *geometric non-termination analysis* as stated in this thesis.

2. many algorithms are difficult to encode if not programmed recursively. Since coding in the already stated classes was no option, and inheritance would not work because of accessing problems, creating my own structure was a simple work-around.

The examples worked with have been small enough to not create any problems with the conversion and possible less efficient methods, but if applied to huge problems a converting of the approach to work on the structure AProVE proposes would be the better way.

Chapter 5

Related Work

The first related source that should be mentioned is the research of Jan Leike and Matthias Heizmann. These two researchers from the Australian National University and University of Freiburg wrote the geometric non-termination argument-Paper this paper relies on. Their definition and proof of the geometric non-termination argument in combination with the derivation on lasso-programs is the base of all computation explained throughout Chapter 2 and Chapter 3. [LH14]
In that context also Ultimate LassoRanker should be mentioned. A tool for termination and non-termination arguments for linear lasso programs by Jan Leike and Matthias Heizmann also containing geometric non-termination argument's. [LH17]
Further A. Tiwari considered linear loop programs not over the naturals, but over the real numbers. For such programs he proofed that they in fact are decidable in terms of termination if they met the condition of only strict guards. [Tiw04]
An other interesting related work is written by R. Rebiha et al., which contains the generalization of the eigenvalues not only to be natural but also can be within the real numbers. [RMM14]
Also J. Ouaknine generalized from the basic approach by mentioning integer lasso programs, where the corresponding Update Matrix is diagonalizable. [OPW14]
An extension of the geometric non-termination argument approach can be found within the work of C. David et al. providing a technique also applicable to non deterministic programs. It uses a constraint-based synthesis of recurrence sets, which are apart from others defined by A. Tiwari in [Tiw04]. Also it can work with second order theory for bit vectors. These can be used to find non-terminating lassos, which do not have a geometric non-termination argument. The downside of such an extension is the problem of solving an $\exists\forall\exists$-constraint. [DKL15] [LH14]

Last but not least the often mentioned tool AProVE is to be mentioned. Using a verity of techniques including lasso's within a range of proofing attempts this tool provides a series of promising techniques implemented. As stated in Section 4.2 the approach of this thesis not only improves proving speed of applicable programs it also expands the proving ability.

Bibliography

[Apa17] Apache. *Apache commons.math3 Documentation*, 2017. Available at `http://commons.apache.org/proper/commons-math/javadocs/api-3.6/org/apache/commons/math3/linear/EigenDecomposition.html`, [Accessed: 27-August-2017].

[Áb16] Prof. Dr. Erika Ábrahám. Introduction to satisfiability checking. 2016.

[BDE+14] Karsten Behrmann, Andrej Dyck, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Patrick Kabasci, Peter Schneider-Kamp, and René Thiemann. Bit-blasting for smt-nia with aprove. *Proc. SMT-COMP*, 14, 2014.

[Ben17] *Benchmark results filtered and further computed*, 2017. Available at `https://docs.google.com/spreadsheets/d/1fH0fVvRW8FPtU4I3KeELEXl33t9oDMjaSG_k_X4aUZs/edit?usp=sharing`, [Accessed: 14-September-2017].

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. Smt-rat: an open source c++ toolbox for strategic and parallel smt solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–368. Springer, 2015.

[CLA17] *clang: a C language family frontend for LLVM*, 2017. Available at `https://clang.llvm.org/`, [Accessed: 07-September-2017].

[DKL15] Cristina David, Daniel Kroening, and Matt Lewis. Unrestricted termination and non-termination arguments for bit-vector programs. In *ESOP*, pages 183–204, 2015.

[FGP+09] Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. Proving termination of integer term rewriting. In *International Conference on Rewriting Techniques and Applications*, pages 32–47. Springer, 2009.

[GAB+16] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Semi-deciding qf nia with aprove via bit-blasting. 2016.

[GAB+17] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. Analyzing program termination and complexity automatically with aprove. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

[GSKT06] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *International Joint Conference on Automated Reasoning*, pages 281–286. Springer, 2006.

[GTSKF03] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Aprove: A system for proving termination. In *Extended Abstracts of the 6th International Workshop on Termination, WST*, volume 3, pages 68–70, 2003.

[I217] Lehrstuhl Informatik I2. *AProVE*, 2017. Available at `http://aprove.informatik.rwth-aachen.de/`, [Accessed: 26-August-2017].

[LH14] Jan Leike and Matthias Heizmann. Geometric series as nontermination arguments for linear lasso programs. *arXiv preprint arXiv:1405.4413*, 2014.

[LH17] Jan Leike and Matthias Heizmann. *Ultimate - LassoRanker*, 2017. Available at `https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=lasso_ranker`, [Accessed: 30-August-2017].

[OPW14] Joël Ouaknine, João Sousa Pinto, and James Worrell. On termination of integer linear loops. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 957–969. SIAM, 2014.

[RMM14] Rachid Rebiha, N Matringe, and AV Moura. Characterization of termination for linear homogeneous programs. Technical report, Technical Report IC-14-08, Institute of Computing, University of Campinas (March 2014), 2014.

[SGB+17] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann. Automatically proving termination and memory safety for programs with pointer arithmetic. *J. Autom. Reasoning*, 58(1):33–65, 2017.

[SVC17] *Competition on Software Verification*, 2017. Available at `https://sv-comp.sosy-lab.org/2017/`, [Accessed: 14-September-2017].

[Ter17] *Termination Competition*, 2017. Available at `http://termination-portal.org/wiki/Termination_Competition`, [Accessed: 14-September-2017].

[Tiw04] Ashish Tiwari. Termination of linear programs. In *CAV*, volume 4, pages 70–82. Springer, 2004.

[VDDS93] Kristof Verschaetse, Stefaan Decorte, and Danny De Schreye. Automatic termination analysis. In *Logic Program Synthesis and Transformation*, pages 168–183. Springer, 1993.

[Wik17a] The Free Encyclopedia Wikipedia. *Reverse Polish Notation*, 2017. Available at `http://en.wikipedia.org/w/index.php?title=Reverse_Polish_notation`, [Accessed: 17-August-2017].

[Wik17b] The Free Encyclopedia Wikipedia. *Termination Competiton 2017*, 2017. Available at `http://termination-portal.org/wiki/Termination_Competition_2017`, [Accessed: 17-August-2017].

[ZC09] Michael Zhivich and Robert K Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2), 2009.