Bachelor Thesis

Geometric Non-Termination Arguments for Integer Programs

25. September 2017

Timo Bergerbusch
Rheinisch-Westfälische 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 me 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 25. 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 and completeness at the same time 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* derived from the program itself. By the usage of linear algebra and SMT solver we are able to prove the existence of a geometric non-termination argument, 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 to prove non-termination.
As a result we will see that the implemented technique provides a mechanism of proving non-termination for different programs. For some of them non-termination could not be proven before. Restricting the set of considered programs to certain criteria defined within this thesis, we are able to prove non-termination and in special cases we are able to prove termination instead.

In summary we can say that the use of geometric non-termination arguments 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.

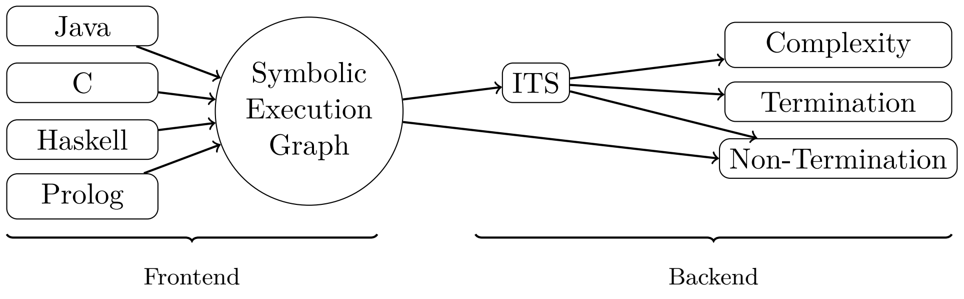
Contents

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 geometric non-termination arguments 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, 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.

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

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 4.3.3.

(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.

A run of an ITS is the successive application of rules, starting from a start function symbol. The termination of an ITS is now defined as the absence 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 geometric non-termination arguments 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

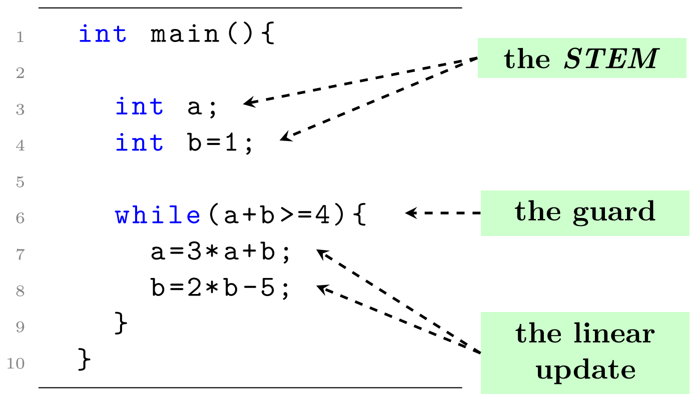
The structure of the considered programs is quite simple. It contains an optional initialization of the used variables and a `while`-loop. Even though C would not accept the usage of a variable without declaration, the conversion to LLVM would still be sound. An example of a linear lasso program in C 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: $a_1 \cdot v_1 + \cdots + a_n \cdot v_n + c$, 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 $v_i = a_1 \cdot v_1 + ... + a_n \cdot v_n + c$ with $a_i \in \mathbb{Z}$ for $1 \leq i \leq n$ and $c \in \mathbb{Z}$.



```
1  int main(){
2
3    int a;
4    int b=1;
5
6    while(a+b>=4){
7      a=3*a+b;
8      b=2*b-5;
9    }
10 }
```

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.

$$
\begin{array}{rl}
1 & \overbrace{f_1 \rightarrow f_2(1 + 3 \cdot v_1, -3) : | : v_1 > 2 \ \texttt{\&\&} \ 8 < 3 \cdot v_1}^{\texttt{STEM}} \\
2 & \underbrace{\underbrace{f_2(v_1, v_2) \rightarrow f_2(3 \cdot v_1 + v_2, v_3)}_{\texttt{linear update}} : | : \underbrace{v_1 + v_2 > 3 \ \texttt{\&\&} \ v_1 > 6 \ \texttt{\&\&} \ 3 \cdot v_1 > 20 \ \texttt{\&\&} \\
3 & \quad 5 + v_3 = 2 \cdot v_2 \ \texttt{\&\&} \ v_3 < -10}_{\texttt{guards}}}_{\texttt{LOOP}} \\
4 &
\end{array}
$$

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

Neglecting the conditional terms for now, we initialize v_2', which is the second argument of f_2, in line 1 to the value of -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 $b = 2 \cdot 1 - 5 = -3$. The definition of v_1' is more difficult and will be shown within Section 3.1. Also the update of v_1', which is the first argument of f_2, 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 $5 + v_3 = 2 \cdot v_2$. 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 arity of the function symbol at the right hand side of the rule representing the STEM of the ITS. The values of x can be constants or restricted by a (possibly empty) conjunction of conditions.*

Examples of the STEM are shown within Section 3.1.

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

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 discussed in Section 3.1.2, Section 3.2.1 and Section 3.2.2.

Example 1. *First we normalize every guard r_i to the form $a_{i,1}v_1 + \ldots a_{i,n}v_n \leq c$. For example the guard r_1 in the following way:*

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

The necessity and computation of the normalization is explained within Section 3.2.1.
Since the coefficient of v_1 is -1 and of v_2 is also -1 we can set the entries $G_{1,1} = G_{1,2} = -1$.
Iterating over all r_i we derive the corresponding Guard Matrix to Figure 2.3 is $G = \begin{pmatrix} -1 & -1 \\ -1 & 0 \\ -3 & 0 \\ 0 & 2 \end{pmatrix}$.
The constant factor of a guard r_i is the right hand side of the inequation. So the constant of $r_1 = -4$. Iterating over all r_i we derive the Guard Constants are $g = \begin{pmatrix} -4 \\ -7 \\ -21 \\ -6 \end{pmatrix}$.

Definition 2.2.3 (Update Matrix and Update Constants)**.** *Let $n \in \mathbb{N}$ be the number of distinct variables of the LOOP rules left hand side, v_j, $1 \leq j \leq n$, the j-th distinct variables of the LOOP rules left hand side, $m \in \mathbb{N}$ the arity of the function symbol of the right hand side, v'_i, $1 \leq i \leq m$, the i-th argument of the right hand side function, $a_{i,j} \in \mathbb{Z}$, $1 \leq i \leq m$ and $1 \leq j \leq n$, be the coefficient of variable v_j in 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 the Update Constants $u \in \mathbb{Z}^m$ are defined as $u_i = c_i$.

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. *Referring to the ITS from Figure 2.3 we have to substitute v_3.*

$$5 + v_3 = 2 \cdot v_2 \Leftrightarrow v_3 = 2 \cdot v_2 - 5$$

So we substitute v_3 in the second argument and receive the arguments $2 \cdot v_1 + v_2$ and $2 \cdot v_2 - 5$. The corresponding Update Matrix is therefore $U = \begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix}$ and the Update Constants are $u = \begin{pmatrix} 0 \\ -5 \end{pmatrix}$.

The detailed derivation of substitutions is explained in Section 3.2.2.

Definition 2.2.4 (Iteration Matrix and 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.*
Furthermore let $\mathbf{0}$ be a matrix of the size of G with only zero entries and I denote the identity matrix having the same dimension as U.
The Iteration Matrix $A \in \mathbb{Z}^{2n+m \times 2n}$, which computes one complete execution of the LOOP, and the Iteration Constants $b \in \mathbb{Z}^{2n+m}$ are defined as

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

Example 3. *Taking the Guard Matrix G and Guard Constants g from Example 1, and the Guard Matrix U and Update Constants u from Example 2 and computing the Iteration Matrix A and Iteration Constants b by inserting the matrices and vectors into the formula of Definition 2.2.4 we get:*

$$A = \begin{pmatrix} G & \mathbf{0} \\ U & -I \\ -U & I \end{pmatrix} \Rightarrow \begin{pmatrix} -1 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 3 & 1 & -1 & 0 \\ 0 & 2 & 0 & -1 \\ -3 & -1 & 1 & 0 \\ 0 & -2 & 0 & 1 \end{pmatrix} \textit{ and } b = \begin{pmatrix} g \\ -u \\ u \end{pmatrix} \Rightarrow \begin{pmatrix} -4 \\ -7 \\ -21 \\ -6 \\ 0 \\ 5 \\ 0 \\ -5 \end{pmatrix}$$

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:

$$\begin{aligned} \textit{(domain)}\quad & x, y_1, \ldots, y_k \in \mathbb{R}^n,\ \lambda_1, \ldots \lambda_k, \mu_1, \ldots \mu_{k-1} \geq 0 \\ \textit{(init)}\quad & x \text{ represents a valid } STEM \\ \textit{(point)}\quad & A \begin{pmatrix} x \\ x + \sum_i y_i \end{pmatrix} \leq b \end{aligned}$$

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

Here $y_0 = \mu_0 = 0$ is set for the ray instead of a case distinction for $i = 1$ [LH14].

The validity of the initiation criterion refers to the possibility of different valid STEMs, further defined in Section 3.1.2. An intuitive interpretation of the point criterion is the validity of entering the corresponding loop. If the point criterion does not hold the ITS is unable to enter the loop and therefore we can not argue about a non-termination loop. One iteration of the ray criterion is computed based on the previous iteration through the y_{i-1}. Also the stated initialization of $y_0 = 0$ is very intuitive since $i = 1$ represents the first iteration and therefore no previous iteration exists. The similarity to the geometric series used within [LH14] is quite obvious. The whole ray criterion can therefore intuitively interpreted as the computation, which keeps the program within this loop.

Example 4. *A valid geometric non-termination argument to the computed matrices of Example 1 and Example 2 is:*

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

The proof of the example being valid is written in Section 3.4. The usage of such a geometric non-termination argument is justified by the following sentence:

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

2.3 Reverse Polish Notation Tree

Within the procedure 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 [Wik17]. 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

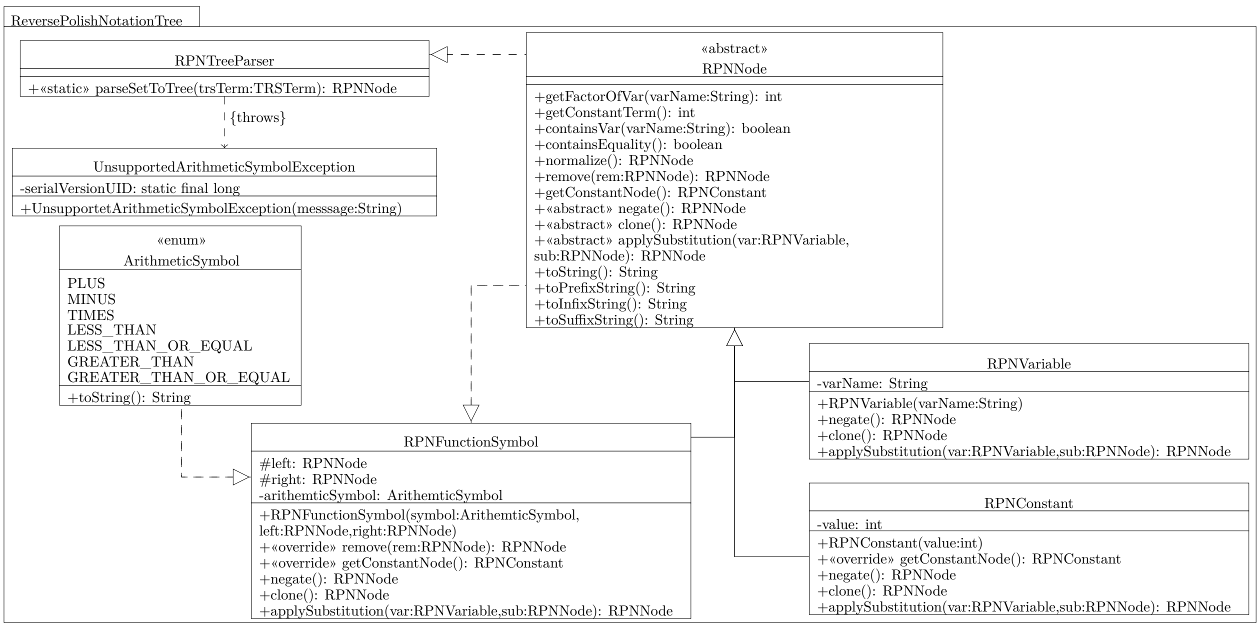

ReversePolishNotationTree
RPNTreeParser
+«static» parseSetToTree(trsTerm:TRSTerm): RPNNode
{throws}
UnsupportedArithmeticSymbolException
-serialVersionUID: static final long
+UnsupportetArithmeticSymbolException(messsage:String)
«enum»
ArithmeticSymbol
PLUS
MINUS
TIMES
LESS_THAN
LESS_THAN_OR_EQUAL
GREATER_THAN
GREATER_THAN_OR_EQUAL
+toString(): String
«abstract»
RPNNode
+getFactorOfVar(varName:String): int
+getConstantTerm(): int
+containsVar(varName:String): boolean
+containsEquality(): boolean
+normalize(): RPNNode
+remove(rem:RPNNode): RPNNode
+getConstantNode(): RPNConstant
+«abstract» negate(): RPNNode
+«abstract» clone(): RPNNode
+«abstract» applySubstitution(var:RPNVariable, sub:RPNNode): RPNNode
+toString(): String
+toPrefixString(): String
+toInfixString(): String
+toSuffixString(): String
RPNVariable
-varName: String
+RPNVariable(varName:String)
+negate(): RPNNode
+clone(): RPNNode
+applySubstitution(var:RPNVariable,sub:RPNNode): RPNNode
RPNFunctionSymbol
#left: RPNNode
#right: RPNNode
-arithemticSymbol: ArithemticSymbol
+RPNFunctionSymbol(symbol:ArithemticSymbol, left:RPNNode,right:RPNNode)
+«override» remove(rem:RPNNode): RPNNode
+«override» getConstantNode(): RPNConstant
+negate(): RPNNode
+clone(): RPNNode
+applySubstitution(var:RPNVariable,sub:RPNNode): RPNNode
RPNConstant
-value: int
+RPNConstant(value:int)
+«override» getConstantNode(): RPNConstant
+negate(): RPNNode
+clone(): RPNNode
+applySubstitution(var:RPNVariable,sub:RPNNode): RPNNode

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

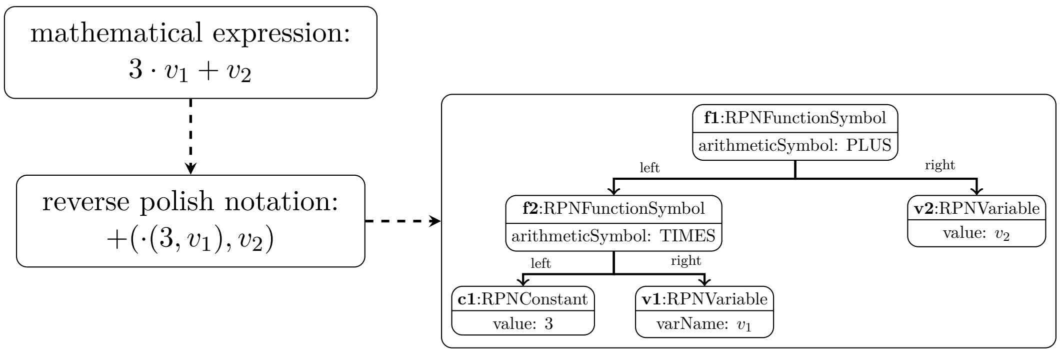


Figure 2.5: An example of the representation of the term $3 \cdot v_1 + v_2$ as a graph using the Reverse Polish Notation Tree

2.4 SMT Problem

To derive a geometric non-termination argument that satisfies all the criteria of Definition 2.2.6, we encounter a *Satisfiability Modulo Theory* problem (SMT-Problem), we have to solve to derive a geometric non-termination argument fulfilling all the criteria of Definition 2.2.6. Since SMT problem solving is a big research topic on its own, we only consider the very basics of SMT solving necessary to understand how to solve 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.

$$\underbrace{\underbrace{\underbrace{\underbrace{3}_{(1)} \cdot \underbrace{v_1}_{(2)}}_{(3)} + \ldots}_{(4)} \quad \underbrace{\leq}_{(5)} \quad \underbrace{5}_{(6)}}_{(7)}$$

Figure 2.6: 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 by a variable is represented by a `PlainIntegerOperation` with `ArithmeticOperationType MUL` to denote multiplication

(4) An `ArithmeticOperationType` of type `ADD` to denote addition. The left hand side is a `PlainIntegerOperation` consisting of the addition of the multiplication (3) of coefficients (1) and variables (2)

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

(6) The right hand side is a `PlainIntegerConstant`

(7) The whole assertion is a `PlainIntegerRelation`, which can be transformed into the `SMTExpressionFormat` the SMT solver uses

AProVE calls a solver and uses the `SMTFactory` to create a bunch of assertions added to the solver to 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 5. *Consider the following assertions that should hold:*

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

Then a valid model is $m_1 = \{x = 6, y = 6\}$. Another model is $m_2 = \{x = 6, y = 7\}$. However, if we change the third rule to $x + y \leq 10$, there is no model for the problem and we 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 valid 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 this way 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 is a proof 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 the program does not terminate, it 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. 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. Here, the STEM is $\begin{pmatrix} 10 \\ -3 \end{pmatrix}$. The values of x can be directly read from the right hand side and need no further calculations.

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

Figure 3.1: Example of a constant STEM.

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 expression, where the variables are constrained 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 to a value or to an unsatisfiable core. Such a core would state that the `while` loop would not be entered after some assignment. If such a constellation entails termination or if it just does not entail non-termination needs further observance not provided in this thesis.
The handling of equations within the guard is described within Section 3.2.1 and underlies the same procedure regarding the STEM.

```
f_1 → f_2(1 + 3 · v, −3) : | : v > 2 && 8 < 3 · v
```

Figure 3.2: Example of a variable STEM.

In order to derive the STEM, a v satisfying the conditions needs to be found using an SMT solver. Since $v = 3$ is the first number in $\mathbb{Z}$ provided by the SMT solver that satisfies the guards, the STEM is $\begin{pmatrix} 1 + 3 \cdot 3 \\ -3 \end{pmatrix} = \begin{pmatrix} 10 \\ -3 \end{pmatrix}$. Note that $v = 4$ would be equally valid.

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 the Iteration Matrix and the Iteration Constants using Definition 2.2.4, if no guards with equalities ("=") occur. Otherwise we first have to perform the following steps to apply the mentioned definitions.
Let f_x be the start function symbol given in the `ITS` and r_i be the following rule:

```
f_x → f_y(v_1, ..., v_n) : | : cond_1
```

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

```
f_y(v_1, ..., v_n) → f_y(v'_1, ..., v'_n) : | : cond_2
```

and compute the Iteration Matrix and the 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 the Guard Constants

The derivation of the Guard Matrix and the Guard Constants can be achieved by applying Definition 2.2.2 to the guards of the given rule r_l. For this 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 this section we show how to deal with guards containing "=" with Example 6 as an example. The first step is to define the desired form of the guards. For that we introduce the *standard guard form*, which is the form of the guards when they are extracted from the Symbolic Execution Graph, and the desired *strict guard 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 cond of a rule is in standard guard form iff

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

The condition extracted from the Symbolic Execution Graph is a formula ϕ , which represents a set G in standard guard form as

$$\phi = \&\&(g_1, (\&\&(\ldots, (\&\&(g_{n-1}, g_n)) \ldots))),$$

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 formula ϕ of the standard guard form

```
 1: function COMPUTEGUARDSET(formula $\phi$)    ▷ $\phi$ has to be a formula representing a cond-formula
 2:     Stack stack ← $\phi$
 3:     Set guards
 4:     while !stack.isEmpty() do
 5:         item ← stack.pop
 6:         if item is of the form $\&\&(x_1, x_2)$ then    ▷ break up concatenation of two guards
 7:             add $x_1$ and $x_2$ 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 with constant term 0, and $c \in \mathbb{Z}$.*

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 guard contains a condition 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 Handling equalities that introduce new variables within the guards

```
1: function FILTEREQUALITIES($G$)                          ▷ $G$ is in standard guard form
2:     $V_{left} = \{v \mid$the left hand side of the rule contains $v\}$
3:     $V_{right} = \{v \mid$the right hand side of the rule contains $v\}$
4:     $V_{sub} = V_{right} - V_{left}$
5:     define substitution $\theta = \{\}$
6:     while $V_{sub} \neq \emptyset$ do
7:         select $s \in V_{sub}$
8:         select $g_s \in \{g \in G \mid g$ contains " = "$\}$      ▷ should only be one guard
9:         remove $g_s$ from $G$
10:        rewrite $g_s$ to the form $s = \psi$
11:        $\theta = \{s/\psi\} \circ \theta$                          ▷ update substitution
12:        for all $g \in G$ do                                  ▷ apply substitution
13:            $g = \theta(g)$
14:        end for
15:        remove $s$ from $V_{sub}$
16:    end while
17:    for all $g \in G$ do
18:        if $g$contains" = " then              ▷ rewrite equalities of variables in $V_{left}$
19:            remove $g \Leftrightarrow l = r$ from $G$
20:            add $l - r \leq 0$ and $r - l \leq 0$ to $G$
21:        end if
22:    end for
23:    return $G$
24: end function
```

If a guard contains equality, but only mentions variable that to not have to be substituted we rewrite the equality to two inequalities to retain the bounds of the Guard Matrix. The result is a set G', which satisfies the condition of occurring variables.

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

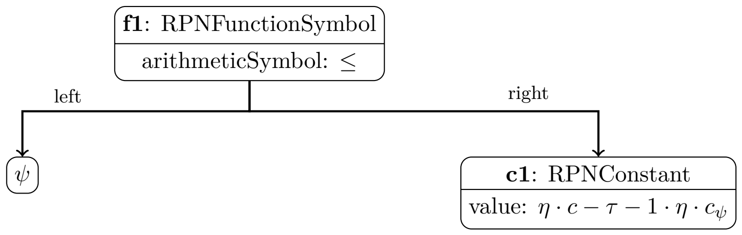
   (a) rewrite a guard g_i of the form $g_i \Leftrightarrow \psi + c_\psi \circ c$, where $\circ \in \{<, >, \leq, \geq\}$ to the form $\eta \cdot \psi + \eta \cdot c_\psi \leq \eta \cdot c - \tau$ depending on $\circ$.

$\circ$	η	τ	$\eta \cdot \psi + \eta \cdot c_\psi \leq \eta \cdot c - \tau$
$<$	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 if it is necessary to invert the guard to convert $\geq$ ($>$) to $\leq$ ($<$), τ can be seen as the subtraction of 1 to receive $\leq$ instead of $<$.

   (b) transfer the c_ψ to the right side to only obtain one constant term located on the right side. So the final form is $\eta \cdot \psi \leq \eta \cdot c - \tau - 1 \cdot \eta \cdot c_\psi$, where $\eta \cdot c - \tau - 1 \cdot \eta \cdot c_\psi$ is a 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 that is left to do in order to 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 we can also 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:


f1: RPNFunctionSymbol
arithmeticSymbol: $\leq$
left
right
ψ
c1: RPNConstant
value: $\eta \cdot c - \tau - 1 \cdot \eta \cdot c_\psi$

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

Example 6. *This example is based on the ITS from Figure 2.3. Regarding the ITS we have the guard-term:*

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

which leads to the set G using Algorithm 1:

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

We start 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* $g_s \Leftrightarrow 5 + v_3 = 2 \cdot v_2$

(line 9,10) *By rewriting* g_s *to the form* $s = \psi$, *we get* $v_3 = 2 \cdot v_2 - 5$

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

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

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

We continue with the normalization:
Applying the rules for the different inequation signs we receive the new guards:

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

Next we apply Algorithm 3 and receive the Guard Matrix $G = \begin{pmatrix} -1 & -1 \\ -1 & 0 \\ -3 & 0 \\ 0 & 2 \end{pmatrix}$. *Furthermore, we can derive the Guard Constants* $g = \begin{pmatrix} -4 \\ -7 \\ -21 \\ -6 \end{pmatrix}$

3.2.2 The Update Matrix and the Update Constants

The Update Matrix and the 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 instantiated variables appear within the update, we have to apply the final set of substitutions θ from Algorithm 2 to the linear update.

Example 7. *This example is based on the ITS from Figure 2.3 in combination with Example 6 providing V_{sub} and θ.*
Since the set of substitutions $\theta = \{v_3/2 \cdot v_2 - 5\}$ is not empty and the update given by

$$(3 \cdot v_1 + v_2,\ v_3)$$

contains $v_3 \in V_{sub}$, we have to apply the substitution and receive:

$$(3 \cdot v_1 + v_2,\ 2 \cdot v_2 - 5)$$

After restraining the updates to only contain 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 that guarantees that a coefficient is always the left child of the multiplication with its corresponding variable. The procedure works as follows:

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

```
function GETCOEFFICIENT(query)
    if this == query then                              ▷ query is a variable name
        return 1
    else if this does not contain query then           ▷ tree does not contain query
        return 0
    end if

    if this represents PLUS then        ▷ Choose the subtree containing the query
        if this.left side contains query then
            return this.left.getCoefficient(query)
        else
            return this.right.getCoefficient(query)
        end if
    end if
    if this represents TIMES then                                ▷ Retrieve value
        if this.right == query then
            return this.left.value
        end if
    end if
end function
```

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

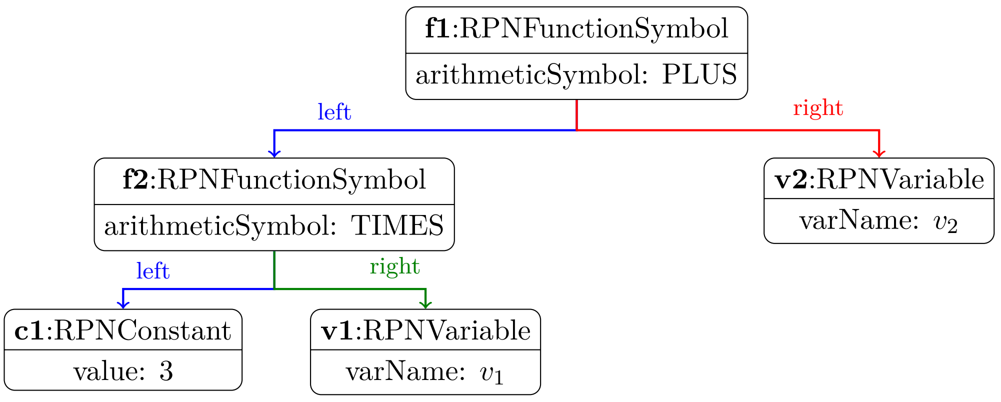


Figure 3.3: An example of deriving the coefficient of the variable v_1 in a formula given as query.

The red arrow between the right subtree and 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 a 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 a Reverse Polish Notation Tree

```
function GETCONSTANTTERM
    if this is a constant then
        return this.value
    end if

    $flip \leftarrow 1$
    if this represents MINUS then                 ▷ flip result in case of prev. negation
        $flip \leftarrow -1$
    end if
    if this represents sth. $\neq$ TIMES then
        $left \leftarrow left.getConstantTerm()$                        ▷ recursive calls
        $right \leftarrow right.getConstantTerm() \cdot flip$
        return $left + right$
    end if
    return 0
end function
```

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

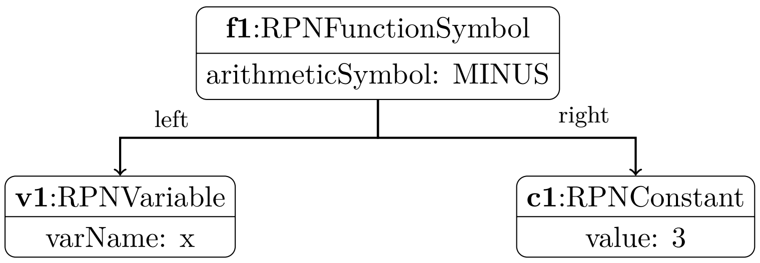


Figure 3.5: The Reverse Polish Notation Tree of the term $x - 3$, where the *flip* of Algorithm 4 has to be used. The value recursively found for the constant term is $(-1) \cdot 3 = -3$.

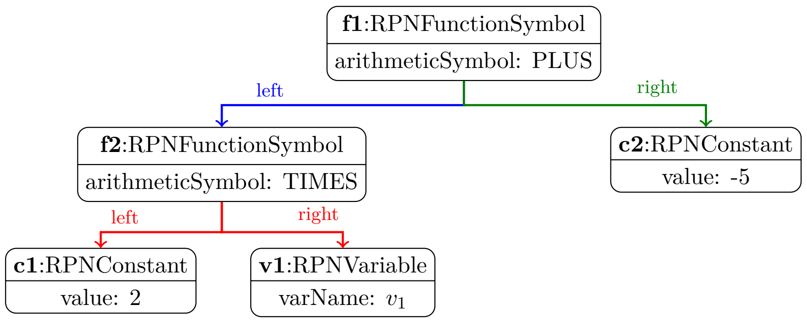


Figure 3.4: An example derivation of a constant term in the second variable update of Example 7. Here red stands for neglected paths, blue stands for considered paths/recursive calls, and green stands for a found constant term.

Since a constant $c < 0$ can be 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 occurrences 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.

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

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

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 and the Iteration Constants

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

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

Given G, g, U and u, to compute A and b we only insert the matrix $\mathbf{0} \in \{0\}^{m \times n}$ and the identity matrix $I \in \{0,1\}^{n \times n}$, where n is the number of distinct not newly introduced variables and m is 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 as presented in Section 2.4, which will either give us a model satisfying the constraints or prove the nonexistence of a model by giving an unsatisfiable core.
The constraints the SMT solver has to assert are the four criteria mentioned within Definition 2.2.6, which are non-linear. So the satisfiability of these is undecidable in general. Since we derive the deterministic update as an Update Matrix, we can further compute its eigenvalues and if we have nonnegative eigenvalues, assign these to $\lambda_1, \dots \lambda_k$, receive linear constraints and thus can decide the existence of a geometric non-termination argument efficiently [LH14].
So the next step in order to prove 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 constant values for the STEM x and $\lambda_1, \dots \lambda_k$.
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:

Lemma 1. *If the SMT solver finds a model m for the assertions created from a program p, then m defines variables $y_1, \dots, y_k$ and $\mu_1, \mu_{k-1} \in \mathbb{N}$ 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 Criterion

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

(cf. Definition 2.2.6)

The domain criterion for x and $y_1, \dots y_k$ trivially hold. The arity of x is set within the derivation of the STEM (cf. 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 criterion and the ray criterion.
The assertions one has to pass to the SMT solver ensure that the λ_i are not negative, since the proof by Leike and Heizmann only holds for nonnegative eigenvalues. Also for the μ_i we have to add the assertion of being nonnegativity to satisfy the criterion.
Related work on negative eigenvalues is mentioned in Chapter 5.

3.3.2 The Initiation Criterion

(init) x represents the start term (STEM)

(cf. Definition 2.2.6)

The initiation criterion is quite trivial to pass to the SMT solver, since we defined the STEM x within Section 3.1 to be exactly the start term. So for this criterion, we do not pass any further assertions to the SMT solver.

3.3.3 The Point Criterion

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

(cf: 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 criterion exactly opposite signed rules for the last $2n$ rows. This means that, even though within the point criterion the relation is $\le$, the last $2n$ inequalities are actually n equalities.

Let $s \in \mathbb{R}^n$ and for $1 \le i \le n$ 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 criterion can be rewritten to:

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

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

$\Rightarrow Gx \le g$, which means that the guards have to hold. Note that s_1 to s_n are multiplied by 0 so they do not need to be considered. Also the following has to hold:

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

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

One can see that for every line l_i $1 \le i \le n$ 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 a 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 they are not constant, we derive the STEM in such a way that the values satisfy these, so adding them would not give any additional knowledge.
Since the entries of variable vectors are represented by Reverse Polish Notation Trees, we can use already implemented methods to calculate the multiplication, normalize the outcome and parse the Reverse Polish Notation Trees into assertions 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_i is added within Section 3.3.5.

3.3.4 The Ray Criteria

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

(cf. Definition 2.2.6)

The ray criterion is the hardest criteria to assert because of its way of computation.
The computation can be split into two parts.

$i = 1$:

For $i = 1$ the second addend $\mu_{i-1}y_{i-1}$ is equal to 0, because of Definition 2.2.6, where $y_0 = \mu_0 = 0$. 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 criterion 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 by the new vector regarding the substitution and create an assertion for each 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 do not 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 iterating over possible entries for the μ_i.

In the implemented approach the *quantifier-free non-linear integer arithmetic* is used. Even if it is undecidable in general, there are implementations over finite domains semi-deciding the problems [BDE+14] [GAB+16]. Further comments on the usage of QF-NIA can be found in 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 $i = 1$ case but have to normalize the outcome using the basic distributive property in order to handle it within a 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 by the Update Matrix A, normalize the outcome and analogously create an assertion for each row using the `SMTFactory`.
Note that the $y_{i-1,n}$ represent the values of the previous step and therefore do 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 idea of rewriting the problem as a geometric series, like it is done in the underlying paper [LH14], is quite comprehensible.

3.3.5 Additional Assertions

The final step of assertion construction needs to be done because of the restriction of the variables from the ray criterion in Section 3.3.4 to the sum from the point criterion in Section 3.3.3.
The assertion that needs to be added has the following form:

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

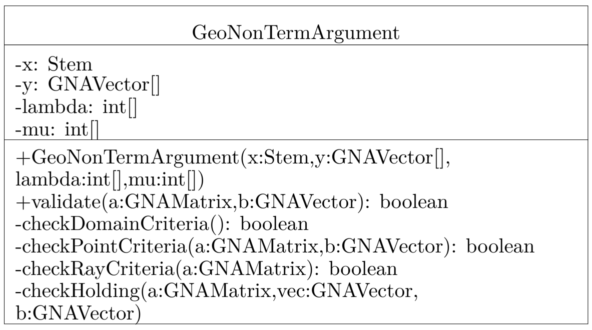
This ensures that the values of y sum up to the values determined in the point criterion.

After adding the additional assertions from Section 3.3.5, the SMT solver received all the constraints 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 Example of a 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 hold. This mechanism was mainly used for debugging purposes. In this chapter we want to show that the given model from the SMT solver holds and therefore the criteria are stated correctly.
In the whole chapter we work with the example used throughout the whole thesis with its Java

code in Figure 2.2 and its corresponding ITS in Figure 2.3. The Iteration Matrix and Iteration Constants computed in Chapter 3 are based on the Guard and Update Matrix and Constants and also the eigenvalues as the λ_i.



GeoNonTermArgument

-x: Stem
-y: GNAVector[]
-lambda: int[]
-mu: int[]

+GeoNonTermArgument(x:Stem,y:GNAVector[], lambda:int[],mu:int[])
+validate(a:GNAMatrix,b:GNAVector): boolean
-checkDomainCriteria(): boolean
-checkPointCriteria(a:GNAMatrix,b:GNAVector): boolean
-checkRayCriteria(a:GNAMatrix): boolean
-checkHolding(a:GNAMatrix,vec:GNAVector, b:GNAVector)

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

The implemented technique provides a model m from the SMT solver and its assertions stated in Section 3.3. From the model m the technique extracts the values and stores them in the class `GeoNonTermArgument`, which is given to AProVE as a witness. An object of this class stores all variable declarations needed in order to revalidate itself to a given scenario. If one has another technique of derivation, it is possible to compute the matrices and set the values for a `GeoNonTermArgument` object and validate it, either proving it or identifying, which criterion does not hold.

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 witness given by this technique 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 criterion stated in Definition 2.2.6.

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

The validity of this criterion is given in Section 3.4 with $n = 2$. This should not be surprising, since we defined the vectors for this dimension and the λ_i and μ_i to be ≥ 0 in Section 3.3.

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

Now we have to compute the sum of the y_i and observe if the inequations hold. So with the values given in Section 3.4 we receive:

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

One can not only see that every row on its own satisfies the desired inequation, we can also observe that the equalities defined in Section 3.3.3 hold for the last $2n = 2 \cdot 2 = 4$ rows. The point-criterion obviously holds.

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

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

r_2**: i=2**

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

Since all r_i, $i \in \{1, 2\}$, are computed and it is checked that they hold, also the ray criterion holds.

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

Chapter 4

Evaluation and Benchmarks

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

4.1 Evaluation of the Approach

The approach provides a sound solution to specific types of programs. Given a tool like AProVE which provides an ITS, the further computation that has to be done can be solved quite efficiently. Using a 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_i. If the μ_i are not given, the problem is undecidable, which makes it still useful within AProVE but not as strong as before.
Since the original approach 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 example programs is taken from the Termination category of the SV-COMP [SVC17].
Detailed results of all programs can be found in [Ben17]. Not all of the programs are non-terminating and also the set includes non-terminating programs, which do not meet the stated preliminaries. 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 prove these programs, which is 4.513 seconds, an average time saving of 0.76 seconds, which is 17 percent, is observed. 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 if more than one proof is found, all but the fastest proof do not affect the average saving of time.

```c
int main(void){
  int a = __VERIFIER_nondet_int();
  int b = __VERIFIER_nondet_int();
  int c = __VERIFIER_nondet_int();

  while (a+b+c >= 4) {
    a = b;
    b = a+b;
    c = b-1;
  }
  return 0;
}
```

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

Also there are programs, like the program shown in Figure 4.1, which AProVE cannot prove 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 applicable to more cases and therefore stronger or more efficient.

4.3.1 Choosing a Logical Fragment

As already stated in Section 3.3 the difficulty of the μ_i leads to a shift into undecidability, since variable multiplication on integers is non-linear. 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_i can be restricted to a finite domain.
A possible improvement could be an iteration approach over different values of the μ_i. The number of problems that have to be solved would blow up but the problems themselves would be decidable.
For the examples of the International Competition on Software Verification, we estimate the method that we chose more suitable. In general, a reliable case study could give an indication of the advantages and disadvantages of the two approaches.

4.3.2 The STEM derivation

Within this thesis we considered the STEM to be derived from the ITS without any dependence to the LOOP. Based on a specific STEM we try to derive a geometric non-termination argument. A stronger approach would be if the STEM would included in the derivation of a geometric non-termination argument. So we combine the set of assertions from Section 3.1.2 and add them to the SMT solver of Section 3.3. Not restricting the geometric non-termination argument to one particular STEM we would consider any STEM that fulfils the conditions. This could prove non-termination for programs, which have a geometric non-termination argument for a STEM that is not the possible first STEM given from the solver as used in this thesis.

4.3.3 ITS program structure

As stated in Section 2.2.2 we restricted our implementation to ITSs of the form

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

This obviously is a restriction, because ITSs of the form

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

could possibly be considered by compressing the rules into one rule using multiple equalities and concatenation of the conditional terms. Analysing such ITSs would make the implementation much stronger.
Another possible variation of the considered ITSs could be of the form

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

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

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

4.3.4 Reverse Polish Notation Tree

In Section 2.3 we defined the Reverse Polish Notation Tree, on which we base the arithmetic computations and statements. AProVE also uses a tree structure to handles such statements with. We chose Reverse Polish Notation Trees structure because of two reasons:

1. The structure AProVE uses 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, but it also prevents errors if AProVE's structure gets extended or changed. The `RPNTreeParser` 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 implement if not programmed recursively. Since extending the existing classes was no option, and inheritance would not work because of restricted visibility, creating my own structure was a simple workaround.

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

Chapter 5

Related Work

The research of Jan Leike and Matthias Heizmann within the field of geometric non-termination is the base of various different approaches. 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 of the geometric non-termination argument approach and their proof of soundness is the base of all computation explained throughout Chapter 2 and Chapter 3 [LH14].
In this context also Ultimate LassoRanker should be mentioned. It is 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 proved that termination in fact is decidable if the condition of only strict guards is met. This means, that every condition is of the form $\varphi < c$ or $\varphi > c$. [Tiw04].
Another interesting related work is written by R. Rebiha et al., which contains the relaxation of the eigenvalues to not range over the naturals but to range over the real numbers [RMM14].
Also J. Ouaknine defined an approach to decide termination of integer lasso programs in those cases where the corresponding Update Matrix is diagonalizable [OPW14].
An alternative non-termination approach is given by the work of C. David et al., a technique also applicable to non-deterministic programs. It uses a constraint-based synthesis of recurrence sets, which are defined by A. Tiwari in [Tiw04]. It also supports second-order theory for bitvectors. This approach can be used to find non-terminating lassos that do not have a geometric non-termination argument. The downside of this 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. AProVE already uses two different non-termination approaches. [HEF$^+$17] As stated in Section 4.2 the approach of this thesis not only leads to a saving of time for applicable programs, it also extends AProVE's power to prove non-termination.

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/1fHOfVvRW8FPtU4I3KeELEXl33t9oDMjaSG_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.

[HEF$^+$17] Jera Hensel, Frank Emrich, Florian Frohn, Thomas Ströder, and Jürgen Giesl. Aprove: Proving and disproving termination of memory-manipulating C programs - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, pages 350–354, 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].

[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.

[Wik17] 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].

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