Bachelor Thesis

GEOMETRIC NON-TERMINATION ARGUMENTS FOR INTEGER PROGRAMS

August 26, 2017

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

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

Acknowledgement

First, I would like to thank Prof. Dr. Jürgen Giesl for giving me the opportunity to work on a timely and relevant topic.

Secondly I would like to thank Jera Hensel, who supervised me during my thesis. I want to thank her for the many patience answers she gave my no matter how obvious the solution was 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 accepting that I often was short on time, but also strengthen my back during the whole process. Especially I want to thank my good friend Tobias Räwer, who explained many topics to me over and over again to help me pass my exams without demanding anything in return. Thanks to his selfless behaviour I got this far within 3 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 August 26, 2017

Abstract

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 that for Touring Complete programming languages the Halting-Problem is undecidable, and therefore no complete and sound method can exist, a verity of approaches to determine termination are researched and still being developed. These approaches can determine 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.
Given a tool, which can provide a sound and in many scenarios applicable mechanism to prove termination, a 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 loses of $60 billion each year in costs associated software could be reduced significantly [ZC09].

1.2 *AProVE*

One promising approach is the tool AProVE (Automated Program Verification Environment) developed at the RWTH Aachen by the Lehr- und Forschungsgebiet Informatik 2. The *AProVE*-tool (further only called AProVE) for automatic termination and complexity proving works with different programming languages of major language paradigms like Java (object oriented), Haskell (functional), Prolog (logical) as well as rewrite systems.

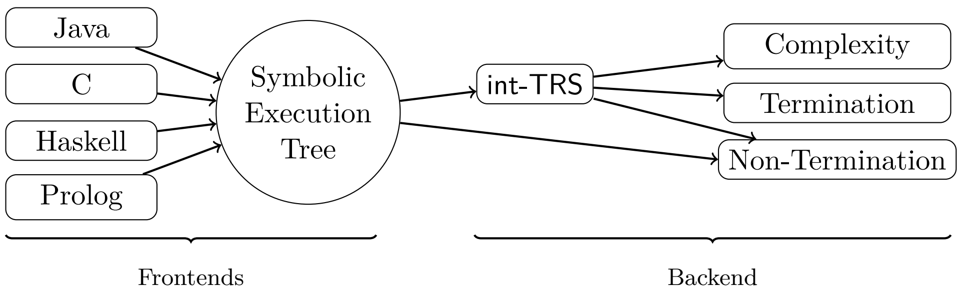


Figure 1.1: Schematic partition of the derivation process of AProVE adapted from [GAB+17]

AProVE is able to unify different languages into one structure by converting programs of specific languages like C into *Low Level Virtual Machine(llvm)*-code using the tool Clang [1]. Among others these llvm-programs can be converted into a so called *Symbolic Execution Graph*. If this graph contains lasso's, which are strongly connected components (SCC) and the corresponding path from the root to the SCC, AProVE derives so called (integer) term rewrite systems (further only called int-TRS)[2]. By adding conditions to the int-TRS rules the solution space gets restricted and therefore the int-TRS under-approximates. From that it is proven that the non-termination of (at least) one int-TRS implies non-termination of the program. A more detailed description of the process is stated in [HEF+17]
The conversion of different languages into int-TRS and subsequently applying various different approaches is what makes this tool strong in meanings of proofing [GAB+17].

1.3 Overview

This paper 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], [GTSKF03].
Within chapter 2 some preliminaries used within the paper are defined to create a well-defined base for any further argumentation and derivation. It covers the topics of basic knowledge about *Integer Term Rewrite Systems* and it's within this approach considered subset based on it's structure. Also the definition of the *geometric nontermination argument*, which builds the main constituent, and any strongly related matrices are defined. Also we define a tree-structure, which we use to handle arithmetical terms containing variables. Last we take a glimpse at the topic of SMT-solving and declare the essential parts used within the implementation of the approach.
The main chapter, which is chapter 3, deals with the derivation of the *STEM* part, for constant or variable terms, the derivation of the *LOOP* with all it's matrices and finally the derivation of the SMT-Problem, which provides a *geometric nontermination argument* if it exists.
At the end, we want to take a look at the usability of the approach itself. Also we want to point out possible adaptations and improvements of the implementation of this approach.

[1]further information: `https://clang.llvm.org/`
[2]a mathematical definition can be found within [FGP+09]

Chapter 2

Preliminaries

In order to be able to explain the solution approach we have to declare to, which programs are considered within the Geometric Nontermination. Furthermore we have to define a few structures we work on.

2.1 Non-Termination

The definition of non-termination is the most essential considering a technique proving it. Non-termination can be defined as a specific input to a program p, such that p runs in an infinite loop. Proving termination is much harder than proving non-termination, since we only have to determine one case, which fit's the condition of running into an infinite loop.
Non-termination is obviously still undecidable, since otherwise the halting problem would be decidable, nevertheless there are a large variety of possibilities to prove non-termination as we will see in this paper.

2.2 Integer Term Rewrite System(int-TRS)

In order to apply the upcoming procedure we have to define what structure the approach works on. The derivation of an int-TRS from source code is described in section 1.2. Considering the following approach we will look at a int-TRS in a more superficial way. The composition of a int-TRS considered in this paper is shown in Figure 2.1.

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

Figure 2.1: The structure of a int-TRS considered in this paper.

The considered program shown in Figure 2.1 consists of a set of structure elements necessary to be defined:

(1) A function symbol consisting of no variables is a *start function symbol* and is the first symbol to use. Further explanation in (line 1) and Figure 2.3.

(2) A function symbol denoting a current program-state

(3) The variables of a function symbol denoting the change of the variables by applying the term rewriting 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$_i$ contains only v_j $1 \leq j \leq n$ defined further in subsection 3.2.2.

(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) Such a rule is a self-looping rule considered within this approach to define further computations. Other looping rules will be presented in subsection 4.3.2.

In general a int-TRS can have rules of other forms, like $f_x(v_1, \ldots v_n) \rightarrow f_y(v_1', \ldots v_k') : | : cond$ where $n \neq k$ can occur, but these rules are for now not considered.

2.3 Geometric Nontermination Argument (GNA)

Adapted from Jan Leikes and Matthias Heizmanns paper *Geometric Nontermination Arguments* [LH14] I will define the considered programs, define the *STEM* and *LOOP* and finally state the definition of Geometric Nontermination Arguments.

2.3.1 Considered Programs

The considered programs in the Geometric Nontermination 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 to derive the so called (int-)TRS, stated in section 1.2. Because of the also stated conversion of the language into a *Symbolic Execution Graph* and further analysis the applicability of *geometric nontermination argument*'s are not bound to any programming language.
In order to define the specific conditions under which we can use the approach, we take the language Java as an example.

2.3.2 Structure

The structure of the considered programs is quite simple. They contain an optional declaration of the used variables and a *while*-loop. Even though Java would not accept this the conversion to llvm would still be sound. An example of a fulfilling Java program is shown in Figure 2.2.

- The *STEM*:
  The initialization and optional declaration of variables used within the *while*-loop. In the example line 3 and 4 are considered the *STEM*. Also only b is declared.

[1] The standard linear integer form has the following pattern: $a_1 * v_1 + \cdots + a_n * v_n + c$, where $a_i, v_i, c \in \mathbb{Z}$, $1 \leq i \leq n$. Also it is important that $a_i * v_i$ has this order and not $v_i * a_i$

- The guard:
  The guard of the *while*-loop is essential to restrict a as we will see in subsection 3.1.2. With the restriction of $a + b \geq 4$ we can prove termination for $a < 3$ without further analysis, and also to prove termination assume that $a \geq 3$.
- The linear Updates:
  The updates of the variables within the *while*-loop are the most essential part for termination, since their value determine if the guard still holds. The approach works with only linear updates of the variables, so for every variable v_i where $1 \leq i \leq n$ we can have a $v_i = a_1 * v_1 + ... + a_n * v_n$ with $n \in \mathbb{N}$. Note since we work on int-TRS it is sufficient for a_i to be 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;
9    }
10 }
```

the *STEM*

the guard

the linear update

Figure 2.2: A Java program fulfilling the conditions to be applicable

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

Through the in section 2.2 described procedure and given structure we receive the to Figure 2.2 equivalent int-TRS shown in Figure 2.3. As we can see the original program can be recognized quite easily. The first rule in line 1 denotes the *STEM*, while the second line equals the loop *LOOP*.

$$1 \quad \overbrace{f_1 \rightarrow f_2(1+3*c,2) : | : c > 2 \ \texttt{\&\&} \ 8 < 3*c}^{STEM}$$

$$2 \quad \underbrace{\underbrace{f_2(a,b) \rightarrow f_2(3*a+b, 2*b)}_{\texttt{linear update}} : | : \underbrace{3*a > 29 \ \texttt{\&\&} \ a+b > 11 \ \texttt{\&\&} \ 31 < 3*a+b \ \texttt{\&\&} \ 3 < 2*b}_{\texttt{guards}}}_{LOOP}$$

Figure 2.3: The int-TRS corresponding to the Java program in Figure 2.2

Neglecting the conditional terms for now the declaration of b is set in line 1 obviously to 2, because of the one circle the *Symbolic Execution Graph* has to compute in order to find a loop. The definition of a is more difficult and will be shown within REF section 3.1. Also the update within line 2 is the same as in Figure 2.2 line 7 and 8.

2.3.3 Necessary Definitions

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

Definition 2.3.1 (*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* int-TRS*. The values of x can be constants or defined by conditions. Examples are shown within section 3.1.*

Definition 2.3.2 (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 name , $m \in \mathbb{N}$ be the number of guards, r_i $1 \leq i \leq m$ the i-th guard, $a_{i,j} \in \mathbb{Z}$ $1 \leq i \leq n$, $1 \leq j \leq m$ 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 $g_i = c_i$.

Example 1. *The corresponding Guard Matrix to Figure 2.3 is $G = \begin{pmatrix} 3 & 0 \\ 1 & 1 \\ 3 & 1 \\ 0 & 2 \end{pmatrix}$ and the Guard Constants is $g = \begin{pmatrix} 29 \\ 11 \\ 31 \\ 3 \end{pmatrix}$*

Definition 2.3.3 (Update Matrix, Update Constants)**.** *Let $n \in \mathbb{N}$ be the number of distinct variables, 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, m_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 m_i and $c_i \in \mathbb{Z}$ $1 \leq i \leq m$ the constant term of m_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 $g_i = c_i$.

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 $u = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$.*

Definition 2.3.4 (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 **0** be a matrix of the size of G with only entry's 0 and I denote the identity matrix with the size of 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

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

Definition 2.3.5 (LOOP)**.** *The LOOP is defined as a tuple (A, b), where A is the Iteration Matrix and b the Iteration Constants of an* int-TRS*. (See: section 3.2)*

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

Definition 2.3.6 (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 nontermination argument for a program $= (STEM, LOOP)$ *with* n *variables iff all of the following statements hold:*

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

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

The usage of such a *geometric nontermination argument* is justified by the following sentence:

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

2.4 Reverse-Polish-Notation-Tree

Within the program of deriving a *geometric nontermination 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. Within our program we use this kind of notation to parse it into our own tree-structure to do further analysis. [Wik17a]

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

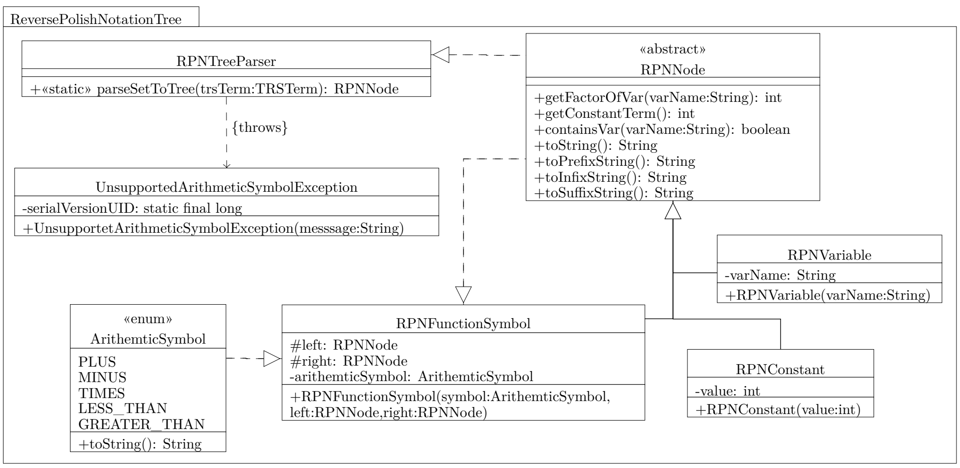


Figure 2.4: The class diagram of the *Reverse Polish Notation Tree* within the *geometric nontermination analysis*

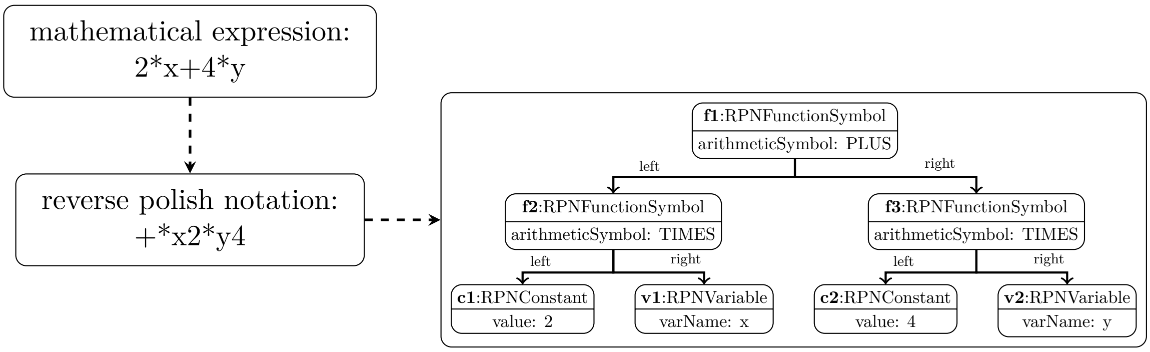


Figure 2.5: An example of the representation of the term $2*(x+4)$ as a graph using the *Reverse Polish Notation Tree* of section 2.4

2.5 SMT-Problem

Also we have to consider an *Satisfiability Modulo Theorie*-Problem (SMT-Problem, we have to solve to derive a *geometric nontermination argument* fulfilling all the criterias of definition 2.3.6. 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.

$$\underbrace{\underbrace{\underbrace{\underbrace{3}_{(1)} * \underbrace{x_1}_{(2)}}_{(3)} \;\underbrace{+}_{(4)}\; \dots}_{(5)} \;\underbrace{\leq}_{(6)}\; \underbrace{3}_{(7)}}_{(8)}$$

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) *PlainIntegerConstant*'s as coefficients

(2) *PlainIntegerVariables*'s 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*

(4) An *ArithmeticOperationType* of type *ADD*

(5) The left hand side is one big *PlainIntegerOperation* consisting of the addition(4) of the multiplication (3) of coefficient's (2) 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 <u>one</u> *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* $m_2 = \{x = 6, y = 7\}$. *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 2.3.6 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 looking how the approach works within AProVE. To find a *geometric nontermination argument* and so prove nontermination we use AProVE to generate an int-TRS of a given program. Based on the calculated int-TRS we derive the *STEM*, the *LOOP* and then generate an SMT-Problem using definition 2.3.6 and compute a *geometric nontermination argument*, which would be a prove of nontermination, or state that no *geometric nontermination argument* can exist, which does not infer termination nor nontermination.

3.1 Derivation of the *STEM*

The derivation of the *STEM* is the first step to do in order to derive a *geometric nontermination argument*. As described in subsection 2.3.2 the *STEM* defines the variables before iterating through the *LOOP*. Owned to the fact, that AProVE has to find the a loop 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:

$$f_x \rightarrow f_y(c_1, \ldots c_n) : | : TRUE$$

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.

$$f_1 \rightarrow f_2(10, 2) : | : TRUE$$

Figure 3.1: An example of a constant int-TRS rule to derive the *STEM*. The *STEM* in this case would be $\begin{pmatrix} 10 \\ 2 \end{pmatrix}$

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 subsection 3.1.1 or a variable 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 2.5, to create the assertions leading to either an assignment of x to a value or to a unsatisfiable core. Such a core would state, that the *while*-Loop would not hold after any assignment and therefore prove termination.

$$f_1 \to f_2(1 + 3 * v, 2) : | : v > 2 \;\texttt{\&\&}\; 8 < 3 * v$$

Figure 3.2: An example of a variable int-TRS rule to derive the *STEM*. In order to derive it an v fulfilling the conditions need to be found using an SMT-Solver. Since $v = 3$ is the first number in $\mathbb{Z}$ that satisfies the guards the *STEM* would be $\begin{pmatrix} 1+3*3 \\ 2 \end{pmatrix} = \begin{pmatrix} 10 \\ 2 \end{pmatrix}$

3.2 Derivation of the *LOOP*

The derivation of the *LOOP* is pretty straight forward applying definition 2.3.2, definition 2.3.3 to a looping rule and then computing *Iteration Matrix* and *Iteration Constants* using definition 2.3.4.
Let f_x be the starting function symbol given by the int-TRS and r_i be a rule, with

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

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

$$f_y(v_1, \dots, v_n) \to f_y(v_1', \dots, v_n') : | : cond_2$$

and compute the *Iteration Matrix* and *Iteration Constants* according to r_l.

3.2.1 The *Update Matrix* and *Update Constants*

The derivation of the *Update Matrix* and *Update Constants* can be achieved by applying the definition 2.3.3 to the given rule r_l. For that we create U as the coefficient matrix. The size of U can be determined by adding a column per occurring variable and rows per linear equation of every v_i'. To derive the entry's of the matrix we use the *Reverse Polish Notation Tree* of the given equation and simply perform recursive searching to derive the factor. The procedure works like the following:

Algorithm 1 Derivation of a coefficient within an *Reverse Polish Notation Tree*

```
function getCoefficient(query)
    if node == query then                              ▷ query is the tree
        return 1
    else if node does not contain query then           ▷ tree does not contain query
        return 0
    end if

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

Since we can rely on the usage of the standard linear integer form described in section 2.2 and therefore we can neglect cases for example that the *left*-child of a *RPNFunctionSymbol* with *arithmeticSymbol TIMES* is the *RPNVariable* and the *right*-child is the *RPNConstant*. An example derivation of a factor using algorithm 1 is shown in Figure 3.3.

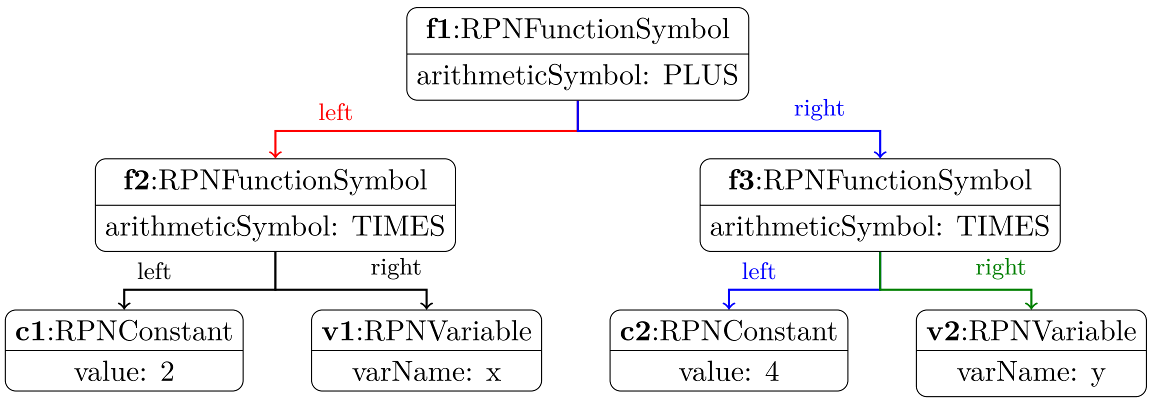


Figure 3.3: An example of deriving the coefficient of a given formula and a variable as query. This example uses the *Reverse Polish Notation Tree* of Figure 2.5 and *y* as the query.

The red-arrow stands for the neglected left 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 1, since we only have to retrieve the constant term within the tree. The corresponding derivation is given by algorithm 2.

Algorithm 2 Derivation of a constant term within an *Reverse Polish Notation Tree*

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

    flip ← 1
    if this represents MINUS then                    ▷ flip result in case of prev. negation
        flip ← −1
    end if
    if this represents sth. ≠ TIMES then
        left ← left.getConstantTerm()                ▷ recursive calls
        right ← right.getConstantTerm() ∗ flip
        return left + right
    end if
end function
```

Since a constant $c < 0$ can stored in a constellation shown in Figure 3.4 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.

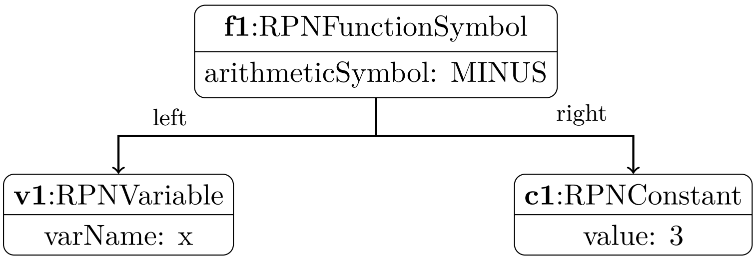


Figure 3.4: A scenario, where the $flip$ of algorithm 2 has to be used. This constellation can not be universally neglected.

Using algorithm 1 and algorithm 2 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:

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

3.2.2 The *Guard Matrix* and *Guard Constants*

The derivation of the *Guard Matrix* and *Guard Constants*, whose definition is stated in definition 2.3.2, is very similar to subsection 3.2.1, but instead of applying the algorithms to the update of the variables the algorithms have to be applied to the guards. The guards are given from the *Symbolic Execution Graph* in a standardized 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, $a_i, v_i, c \in \mathbb{Z}$, $1 \leq i \leq n$ and $\circ \in \{<, >\}$.*
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 conditions given be the *Symbolic Execution Graph* is one rule r , which represents a set G in standard guard form and

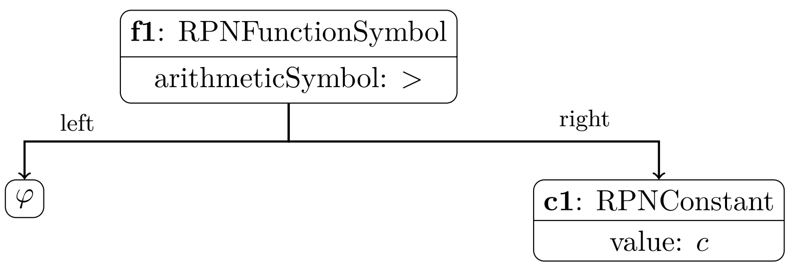
$$r = \&\&(g_1, (\&\&(\dots, (\&\&(g_{n-1}, g_n)) \dots)))$$

The easiest way to retrieve the guards g_i is by using algorithm 3.

Algorithm 3 Retrieving a set of guards G from a rule r of the form stated in subsection 3.2.2

```
function computeGuardSet(Rule r)        ▷ r has to be a rule representing a cond-term
    Stack stack ← r
    Set guards
    while !stack.isEmpty() do
        item ← stack.pop
        if item is of the form &&(x1, x2) then
            add x1 and x2 to stack
        else
            add item to guards
        end if
    end while
end function
```

So we get a set $G = \{g \mid g \text{ is in standard guard form}\}$. Not only does the *Symbolic Execution Graph* normalize the guards to only use $>$, also it provides the guards $g := \varphi > c \in G$ in the following form:



So to derive the entry's of the *Guard Matrix* we can simply use algorithm 1 of φ and to derive entry's of *Guard Constants* we have to simply get the *right*-child of the root-node.
After doing that, we got for $m = |G|$ the *Guard Matrix* $G \in \mathbb{Z}^{m \times n}$ and *Guard Constants* $g \in \mathbb{Z}^m$. In order to use it within the *Iteration Matrix* and *Iteration Constants* and further within the derivation of a *geometric nontermination argument* we transform it to the following:

$$
\begin{aligned}
G \times \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} &> g \\
\begin{pmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \dots & a_{n,n} \end{pmatrix} \times \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} &> \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} && | * -1 \\
\begin{pmatrix} -a_{1,1} & \dots & -a_{1,n} \\ \vdots & \ddots & \vdots \\ -a_{n,1} & \dots & -a_{n,n} \end{pmatrix} \times \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} &< \begin{pmatrix} -c_1 \\ \vdots \\ -c_n \end{pmatrix} && | reshape \\
\underbrace{\begin{pmatrix} -a_{1,1} & \dots & -a_{1,n} \\ \vdots & \ddots & \vdots \\ -a_{n,1} & \dots & -a_{n,n} \end{pmatrix}}_{G} \times \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} &\leq \underbrace{\begin{pmatrix} -c_1 - 1 \\ \vdots \\ -c_n - 1 \end{pmatrix}}_{g}
\end{aligned}
$$

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 2.3.4 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 variables and m the number of guards.

3.3 Derivation of the *SMT*-Problem

The existence of a *geometric nontermination argument* is checked using an SMT solver, presented in section 2.5, 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 2.3.6, 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 [1] because of performance reasons. Computation of such matrices can be very costly if programmed inefficiently. After computing the eigenvalues, we have set values for x and $\lambda_1, \dots \lambda_k$ as constant values.
Using the *SMTFactory*, which offers methods to create the within section 2.5 and Figure 2.6 stated structure, we are able to create assertions and add them to the *SMT-solver*, such that the following holds:

[1]the mentioned method can be found under [Apa]

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 $\mathbb{Z}$ such that $(x, y_1, \dots, y_k, \lambda_1, \dots, \lambda_k, \mu_1, \dots, \mu_{k-1})$ is a *geometric nontermination argument*.

3.3.1 The Domain Criteria

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

(see: definition 2.3.6)

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 set's 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*.
Therefore this criteria adds no further assertions towards the *SMT-solver*.

3.3.2 The Initiation Criteria

(init) x represents the *start term* (*STEM*)

(see: definition 2.3.6)

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 also adds no further assertions towards the *SMT-solver*.

3.3.3 The Point Criteria

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

(see: definition 2.3.6)

The *Point Criteria* is the first criteria to add assertions towards the *SMT-solver*.
The point criteria has a special role within the derivation. Since within the *Iteration Matrix A* the *Update Matrix* is contained twice with 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:

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

$$\Leftrightarrow \begin{pmatrix} & G & & 0 & \dots & 0 \\ a_{1,1} & \dots & a_{1,n} & -1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & \dots & a_{n,n} & 0 & \dots & -1 \\ -a_{1,1} & \dots & -a_{1,n} & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ -a_{n,1} & \dots & -a_{n,n} & 0 & \dots & 1 \end{pmatrix} \begin{pmatrix} x1 \\ \vdots \\ x_n \\ s_1 \\ \vdots \\ s_n \end{pmatrix} \leq \begin{pmatrix} g \\ -u_1 \\ \vdots \\ -u_n \\ u_1 \\ \vdots \\ u_n \end{pmatrix}$$

$\Rightarrow Gx \leq g$, which means that the guards have to hold, and

$$\begin{pmatrix} a_{1,1} & \dots & a_{1,n} & -1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & \dots & a_{n,n} & 0 & \dots & -1 \\ -a_{1,1} & \dots & -a_{1,n} & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ -a_{n,1} & \dots & -a_{n,n} & 0 & \dots & 1 \end{pmatrix} \begin{pmatrix} x1 \\ \vdots \\ x_n \\ s_1 \\ \vdots \\ s_n \end{pmatrix} \leq \begin{pmatrix} -u_1 \\ \vdots \\ -u_n \\ u_1 \\ \vdots \\ u_n \end{pmatrix}$$

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

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. Since variable vectors are represented as a *Reverse Polish Notation Tree* we can use a symbol 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_j is added within subsection 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 2.3.6)

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.

$i = 0$:

For $i = 0$ the second addend $\mu_{i-1}y_{i-1}$ is equal to 0, because of definition 2.3.6, 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 and create an assertion per row using the *SMTFactory*, the *IntegerRelationType LE* and as the right hand side constant a 0.

$i \neq 0$:

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.* 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 $i > 0$ 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 obvious.

3.3.5 Additional assertion

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

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

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

After the adding the *Additional assertion* from subsection 3.3.5 the *SMT-solver* contains all the restrictions to compute a *geometric nontermination argument* or an unsatisfiable core for the given program.
If a *geometric nontermination 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 nontermination argument* can be rechecked giving the *Iteration Matrix* and *Iteration Constants* if all of the four criteria of definition 2.3.6 by simply computing and checking if the conditions hold.

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 a normalized and shortened int-TRS 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.

4.2 Benchmarks

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 subsection 2.3.2. Nevertheless also this implementation has certain cases in, which it does not perform as efficient as it could. 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 nontermination 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* [1] or *International Competition on Software Verification* [2]. [Gie]
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 int-TRS program structure

As stated in subsection 2.3.2 we restrict this implementation to the form

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

which obviously is a restriction, because int-TRS's of the form

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

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

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

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.

[1] further information: `http://termination-portal.org/wiki/Termination_Competition`
[2] further information: `https://sv-comp.sosy-lab.org/2017/`

Chapter 5

related work

Bibliography

[Apa] Apache commons.math3. 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.

[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.

[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.

[Gie] Jürgen Giesl. Aprove. http://aprove.informatik.rwth-aachen.de/. Accessed: 26-08-2017.

[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. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 350–354. Springer, 2017.

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

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

[Wik17a] Wikipedia. Reverse polish notation — Wikipedia, The Free Encyclopedia, 2017. [Accessed: 17-August-2017].

[Wik17b] Wikipedia. Termination competiton 2017 — Wikipedia, The Free Encyclopedia, 2017. [Accessed: 17-August-2017].

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