Bachelor Thesis

GEOMETRIC NON-TERMINATION ARGUMENT FOR INTEGER PROGRAMS

August 17, 2017

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

First Supervisor: Jera Hensel
Second Supervisor: Prof. Dr. Thomas Noll

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 17, 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 a 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. The conversion of these different languages into (integer) term rewrite systems ((int-)TRS) and subsequently applying various different approaches is what makes this tool strong in meanings of proofing [GAB+17].

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 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.

2.1.1 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. Because of the, within the introduction stated, conversion of the language into *llvm*-code and further analysis the applicability of Geometric Nontermination Arguments are not bound to any program language.
In order to define the specific conditions under which we can use the approach, we take the language Java as an example.

2.1.2 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 2.1.

- 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.
- The guard:
  The guard of the *while*-loop is essential to restrict a as we will see in . 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 $f(v_i) = a_1 * v_1 + ... + a_n * v_n$ 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 2.1: A Java program fulfilling the conditions to be applicable

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

After ... we finally receive the equivalent int-TRS shown in Figure 2.2. 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 $\overbrace{f_1 -> f_2(1 + 3 * c, 2) :|: c > 2 \ \texttt{\&\&} \ 8 < 3 * c}^{STEM}$

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

Figure 2.2: The int-TRS corresponding to the Java program in Figure 2.1

Neglecting the conditional terms for now the declaration of b is set in line 1 obviously to 2, because of the one circle the GRAPH has to compute in order to find a loop. The definition of a is more difficult and will be shown within REF . Also the update within line 2 is the same as in Figure 2.1 line 7 and 8.

2.1.3 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 2.1.1 (*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 2.1.2 (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 2.2 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 2.1.3 (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 2.2 is $U = \begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix}$ and the Update Constants are $u = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$.*

Corollary 2.1.1. *In this approach we consider programs of the form*

1 $f_x \to f_y(v_1, \dots v_n) : | : cond_1$
2 $f_y(v_1, \dots v_n) \to f_y(v'_1, \dots v'_n) : | : cond_2$

where line 1 is the STEM and line 2 is the LOOP consisting of one function symbol looping to itself. So the number of variables stays the same and $U \in \mathbb{Z}^{n \times n}$ with $n \in \mathbb{N}$ the number of variables. The $cond_i$ are conditions of rule i of the form:

$$\text{(in)equation}_1 \ \&\& \ \dots \ \&\& \ \text{(in)equation}_m$$

Definition 2.1.4 (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 2.1.5 (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*.*

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

Definition 2.1.6 (Geometric Non Termination Argument)**.** *A tuple of the form:*

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

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

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

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

2.2 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. [Wik17]

As shown in Figure 2.3 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 2.4

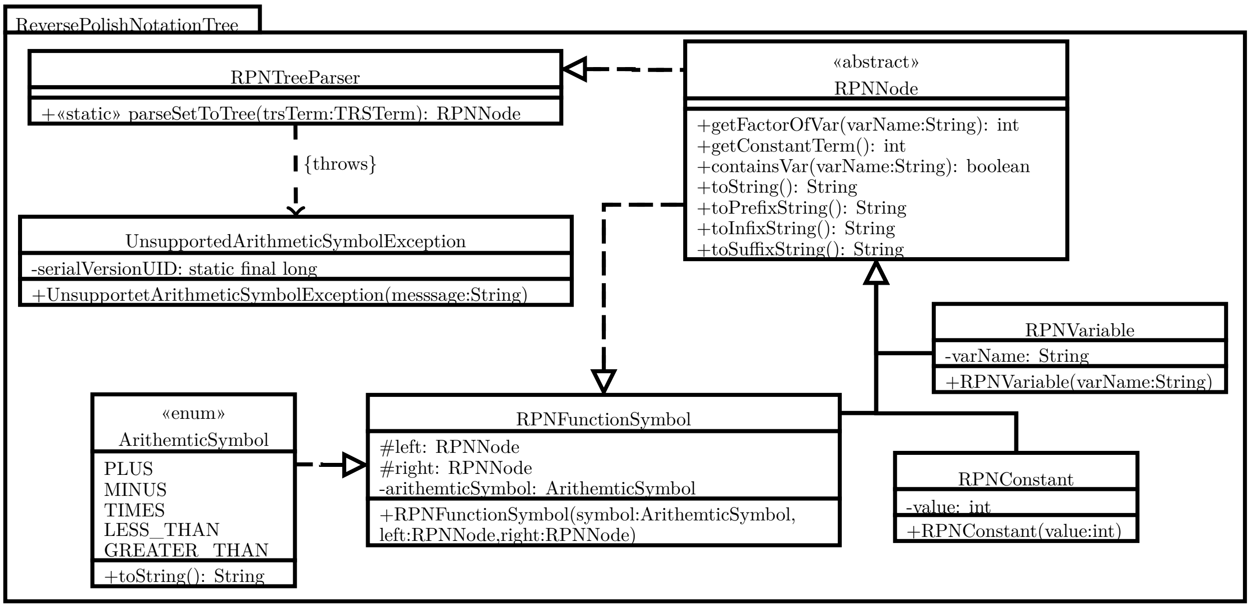

Figure 2.3: The class diagram of the *Reverse Polish Notation Tree* within the *geometric nontermination analysis*

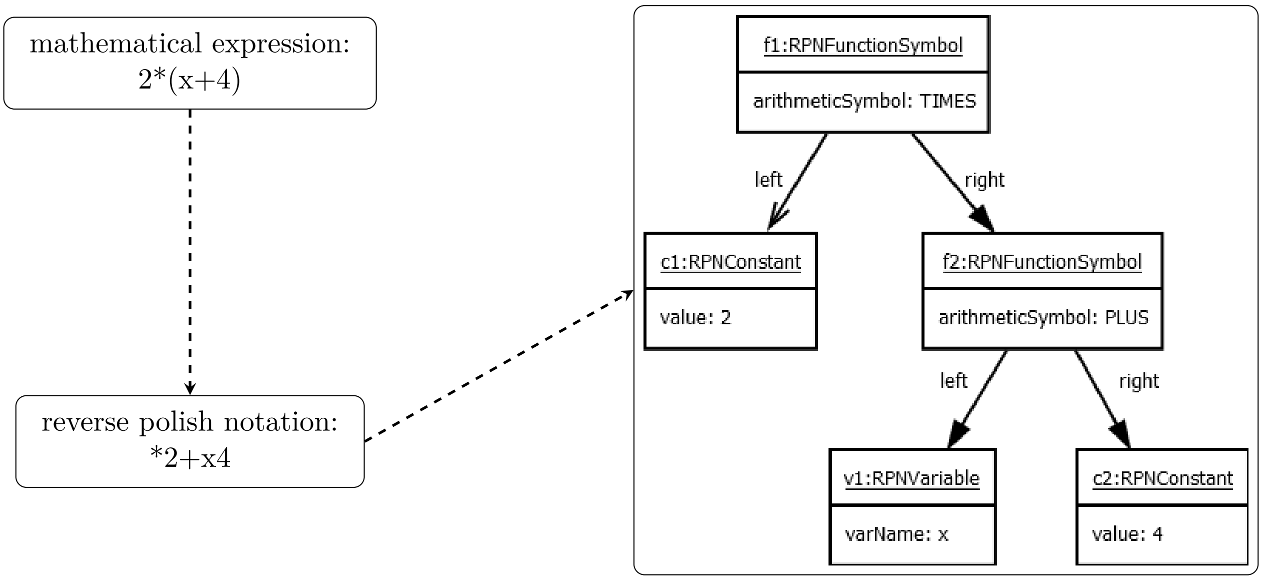

Figure 2.4: An example of the representation of the term $2*(x+4)$ as a graph using the *Reverse Polish Notation Tree* of section 2.2

2.3 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 2.1.6. 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.
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 2.1.6 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 2.1.6 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 2.1.2 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:

$$f_x \rightarrow f_y(v_1, \ldots v_n) : | : cond$$

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 we will explain in , 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.

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

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*

3.2.1 The Update Matrix

3.2.2 The Guard Matrix

3.2.3 The Iteration Matrix

3.3 Derivation of the *SMT*-Problem

3.3.1 The Domain Criteria

3.3.2 The Initiation Criteria

3.3.3 The Point Criteria

3.3.4 The Ray Criteria

3.4 Verification of the Geometric Non-Termination Argument

Chapter 4

Benchmarks

Chapter 5

related work

Bibliography

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

[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.

[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.

[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.

[Wik17] Wikipedia. Reverse polish notation — Wikipedia, The Free Encyclopedia, 2017. [Online; accessed 17-August-2017].

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